Introduction aux bases de données

Langage de requête

Table des matières

1. Préambule

La structure de la base de données qui va nous servir pour ce cours est la suivante :

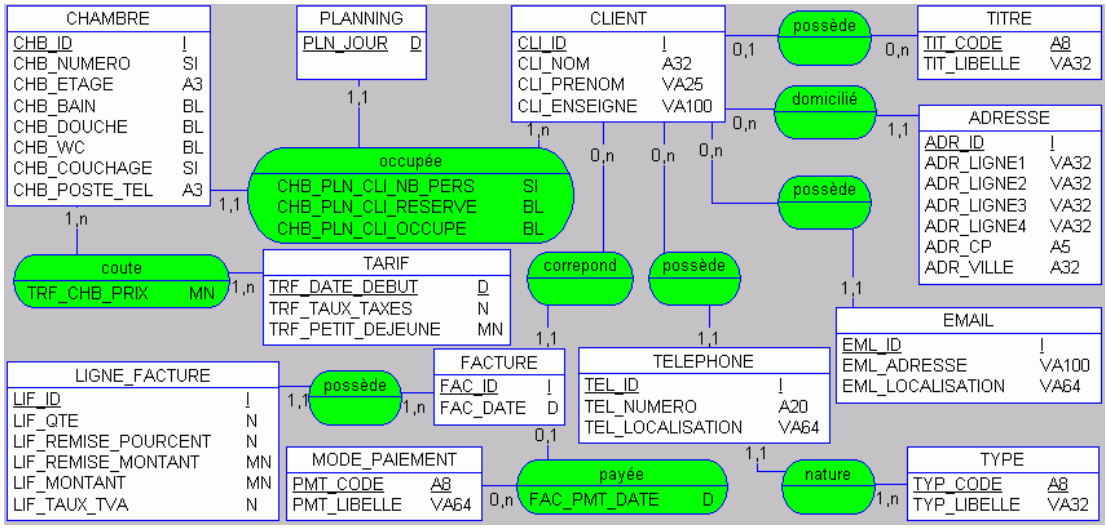

Illustration 1: MCD de la base exemple

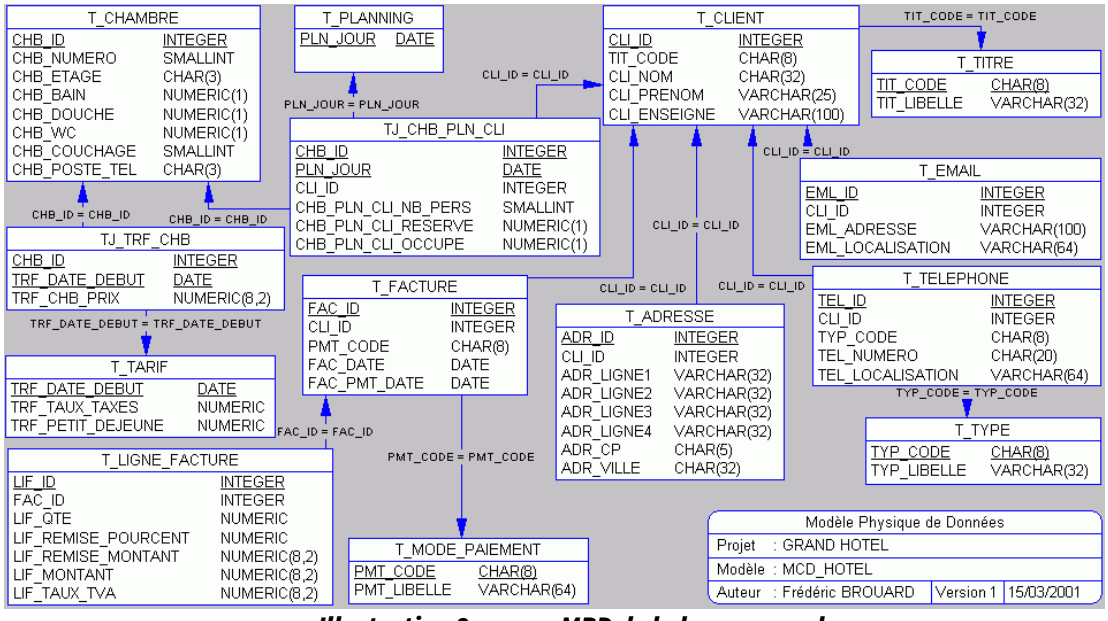

Illustration 2: MPD de la base exemple

Si vous souhaitez importer cette base de données dans votre SGBDR, utilisez le fichier disponible ici :

https://www.tala-informatique.fr/wiki/images/f/fe/Hotel.txt

2. Différence entre les requêtes

Il existe une différence fondamentale entre l'ordre de manipulation des données qu'est le **SELECT** et les différents ordres de mise à jour de données :

- Parce qu'en dehors de problèmes de syntaxe ou de typage, le **SELECT** est une opération dont l'aboutissement est en théorie toujours garantie (sauf à indiquer des noms d'objets inexistants) alors que les opérations de mise à jour des données sont susceptibles d'être rejetées, notamment si elles violent des contraintes ;
- Parce qu'autant il est possible d'effectuer des extractions de données via un ordre **SELECT** sur plusieurs tables, autant il est impossible d'insérer, de modifier ou de supprimer dans plusieurs tables simultanément ;
- Parce qu'une requête de mise à jour (**INSERT**, **UPDATE** ou **DELETE**) est une transaction en elle-même et que tout doit être réalisé, sinon rien ne se passe (en particulier si une seule donnée viole une contrainte, alors aucune insertion, mise à jour ou suppression n'est effective dans le cas d'un ordre concernant la mise à jour de plusieurs lignes simultanément) ;
- Parce qu'à moins de gérer une transaction et notamment à l'aide du mot clef **ROLLBACK**, il est impossible de revenir sur un ordre de mise à jour si l'on s'est trompé. Dans le cas d'un **SELECT** ceci n'a pas d'importance puisque les données ne sont pas modifiées. =

3. Récupérer des données grâce à SELECT

La syntaxe de base de l'ordre SQL SELECT de données dans une table est la suivante :

```
SELECT [DISTINCT ou ALL] * ou liste_de_colonnes FROM nom_des_tables_ou_des_vues
```

Exemple 1: syntaxe du SELECT

a) Jointure

Les jointures permettent d'exploiter pleinement le modèle relationnel des tables d'une base de données et sont faites pour mettre en relation deux (ou plusieurs) tables concourant à rechercher la réponse à des interrogations.

Une jointure permet donc de combiner les colonnes de plusieurs tables.

Il existe en fait différentes natures de jointures et retenez que la plupart des jointures entre tables s'effectuent en imposant l'égalité des valeurs d'une colonne d'une table à une colonne d'une autre table.

On parle alors de jointure naturelle ou équi-jointure.

Mais on trouve aussi des jointures d'une table sur elle-même et on parle alors d'auto-jointure.

De même, il arrive que l'on doive procéder à des jointures externe, c'est-à-dire joindre une table à une autre, même si la valeur de liaison est absente dans une table ou l'autre.

Enfin, dans quelques cas, on peut procéder à des jointures hétérogènes, c'est-à-dire que l'on remplace le critère d'égalité par un critère d'inégalité ou de différence.

Une jointure entre tables peut être mise en œuvre, soit à l'aide des éléments de syntaxe SQL que nous avons déjà vu, soit à l'aide d'une clause spécifique du SQL, la clause **JOIN**.

Premier essai

Essayons de récupérer les n° de téléphones associés aux clients.

```
SELECT CLI_NOM, TEL_NUMERO
FROM T_CLIENT, T_TELEPHONE
```

Exemple 2: Ordre de sélection des numéros de téléphone

On obtient le résultat suivant :

```
CLI_NOM  TEL_NUMERO
-------  --------------
DUPONT   01-45-42-56-63
DUPONT   01-44-28-52-52
DUPONT   01-44-28-52-50
DUPONT   06-11-86-78-89
DUPONT   02-41-58-89-52
DUPONT   01-51-58-52-50
DUPONT   01-54-11-43-21
DUPONT   06-55-41-42-95
DUPONT   01-48-98-92-21
DUPONT   01-44-22-56-21
...
```

Cette requête ne possède pas de critère de jointure entre une table et l'autre. Dans ce cas, le compilateur SQL calcule le produit cartésien des deux ensembles, c'est-à-dire qu'à chaque ligne de la première table, il accole l'ensemble des lignes de la seconde.

Nous verrons qu'il existe une autre manière, normalisée cette fois, de générer ce produit cartésien et la requête précédente est à **proscrire**.

Dans notre exemple elle génère 17 400 lignes!

Il faut donc définir absolument un critère de jointure.

Dans le cas présent, ce critère est la correspondance entre les colonnes contenant la référence de l'identifiant du client (*CLI_ID*).

```
SELECT CLI_NOM, TEL_NUMERO
FROM T_CLIENT, T_TELEPHONE
WHERE CLI_ID = CLI_ID
```

Exemple 3: Utilisation d'une jointure

Cependant, nous n'avons pas fait mieux, car nous avons créé une clause toujours vraie, un peu à la manière de 1 = 1 !

Deuxième essai

En fait il nous manque une précision : il s'agit de déterminer de quelles tables proviennent les colonnes *CLI_ID* de droite et de gauche. Cela se précise à l'aide d'une notation pointée en donnant le nom de la table.

Il est donc nécessaire d'indiquer au compilateur la provenance de chacune des colonnes *CLI_ID* et donc d'opérer une distinction entre l'une et l'autre colonne. Ainsi, chaque colonne devra être précédée du nom de la table, suivi d'un point.

```
SELECT CLI_NOM, TEL_NUMERO
FROM T_CLIENT, T_TELEPHONE
WHERE T_CLIENT.CLI_ID = T_TELEPHONE.CLI_ID
```

Exemple 4: Utilisation d'une jointure entre deux tables

On obtient 174 enregistrements !

Il existe une autre façon de faire, plus simple encore. On utilise la technique du "**sur-nommage**", c'est-à-dire que l'on attribue un surnom à chacune des tables présente dans la partie **FROM** du **SELECT** :

```
SELECT CLI_NOM, TEL_NUMERO
FROM T_CLIENT C, T_TELEPHONE T
WHERE C.CLI_ID = T.CLI_ID
```

Exemple 5: SELECT avec sur-nommage

La table *T_CLIENT* a été surnommée "C" et la table *T_TELEPHONE* "T".

Bien entendu, et comme dans les requêtes mono-tabulaires on peut poser des conditions supplémentaires de filtrage dans la clause **WHERE**.

Cherchons par exemple les clients dont les numéros de téléphone correspondent à un fax :

```
SELECT CLI_NOM, TEL_NUMERO
FROM T_CLIENT C, T_TELEPHONE T
WHERE C.CLI_ID = T.CLI_ID AND TYP_CODE = 'FAX'
```

Exemple 6: SELECT avec clause WHERE à deux conditions

Le fait de placer comme critère de jointure entre les tables, l'opérateur logique "égal" donne ce que l'on appelle une "**équi-jointure**".

Nota Bene :

On peut aussi utiliser les surnoms dans la partie qui suit immédiatement le mot clef **SELECT**. Ainsi l'exemple 6, peut aussi s'écrire :

```
SELECT C.CLI_ID, C.CLI_NOM, T.TEL_NUMERO
FROM T_CLIENT C, T_TELEPHONE T
WHERE C.CLI_ID = T.CLI_ID
  AND T.TYP_CODE = 'FAX'
```

Exemple 7: Sur-nommage après le mot clé SELECT

b) Les différents types de jointure

En utilisant la jointure entre clefs primaires et clefs secondaires basée sur l'égalité des valeurs des colonnes nous exécutons ce que l'on appelle une **jointure naturelle**.

Il est aussi possible de faire des **équi-jointures** qui ne sont pas naturelles, soit par accident (une erreur !), soit par nécessité.

Il est aussi possible de faire des **non équi-jointures**, c'est-à-dire des jointures basée sur un critère différent de l'égalité, mais aussi des auto-jointures, c'est-à-dire de joindre la table sur elle-même.

Le cas le plus délicat à comprendre est celui des **jointures externes**, c'est-à-dire exiger que le résultat comprenne toutes les lignes des tables (ou d'au moins une des tables de la jointure), même s'il n'y a pas correspondance des lignes entre les différentes tables mise en œuvre dans la jointure.

La **jointure d'union** consiste à ajouter toutes les données des deux tables à condition qu'elles soient compatibles dans leurs structures.

La **jointure croisée** permet de faire le produit cartésien des tables.

Dans la mesure du possible, **utilisez toujours** un opérateur de jointure normalisé Sql2 **(JOIN)**.

En effet :

- les jointures faites dans la clause **WHERE** ne permettent pas de faire la distinction de prime abord entre ce qui relève du filtrage et ce qui relève de la jointure ;
- il est à priori absurde de vouloir filtrer dans le **WHERE** (ce qui restreint les données du résultat) et de vouloir "élargir" ce résultat par une jointure dans la même clause **WHERE** de filtrage.
- la lisibilité des requêtes est plus grande en utilisant la syntaxe à base de **JOIN**, en isolant ce qui est filtrage et jointure, mais aussi en isolant avec clarté chaque condition de jointures entre chaque couples de table ;
- l'optimisation d'exécution de la requête est souvent plus pointue du fait de l'utilisation du **JOIN** ;
- lorsque l'on supprime la clause **WHERE** à des fins de tests, le moteur SQL réalise le produit cartésiens des tables ce qui revient la plupart du temps à mettre à genoux le serveur !

c) Syntaxe normalisée des jointures

Les jointures normalisées s'expriment à l'aide du mot clef **JOIN** dans la clause **FROM** et suivant la nature de la jointure, on devra préciser sur quels critères se base la jointure.

Ci-dessous un récapitulatif des différents types de jointures normalisées :

```
SELECT ...
FROM   <table gauche>
       NATURAL JOIN <table droite>
               [USING <noms de colonnes>]
```

Exemple 8: Syntaxe de la jointure naturelle

```
SELECT ...
FROM   <table gauche>
       [INNER]JOIN <table droite>
       ON <condition de jointure>
```

Exemple 9: Syntaxe de la jointure interne

```
SELECT ...
FROM   <table gauche>
       LEFT | RIGHT | FULL OUTER JOIN <table droite>
               ON condition de jointure
```

Exemple 10: Syntaxe de la jointure externe

```
SELECT ...
FROM   <table gauche>
       CROSS JOIN <table droite>
```

Exemple 11: Syntaxe de la jointure croisée

```
SELECT ...
FROM   <table gauche>
       UNION JOIN <table droite>
```

Exemple 12: Syntaxe de la jointure d'union

d) Jointure naturelle

L'opérateur **NATURAL JOIN** permet d'éviter de préciser les colonnes concernées par la jointure. Dans ce cas, le compilateur SQL va rechercher dans les 2 tables, les colonnes dont le nom est identique et, bien entendu, le type de données doit être le même !

Nota Bene : on veillera au niveau de la modélisation et notamment au niveau du MPD (Modèle Physique de Données) que les noms des colonnes de clefs en relation avec d'autres tables par l'intermédiaire des clefs étrangères soient strictement identiques.

```
SELECT CLI_NOM, TEL_NUMERO
FROM T_CLIENT NATURAL JOIN T_TELEPHONE
```

Exemple 13: Jointure naturelle

Cependant cette syntaxe est rarement acceptée par les moteurs SQL actuels et la partie optionnelle **USING** , qui permet de restreindre les colonnes concernées lorsque plusieurs colonnes servent à définir la jointure naturelle, doit être utilisée :

```
SELECT CLI_NOM, TEL_NUMERO
FROM   T_CLIENT
       NATURAL JOIN T_TELEPHONE
               USING (CLI_ID)
```

Exemple 14: Jointure naturelle restreinte

e) Jointure interne

Comme il s'agit de la plus commune des jointures c'est celle qui s'exerce par défaut si on ne précise pas le type de jointure. Après le mot clef **ON** on doit préciser le critère de jointure.

```
SELECT CLI_NOM, TEL_NUMERO
FROM   T_CLIENT C
       INNER JOIN T_TELEPHONE T
             ON C.CLI_ID = T.CLI_ID
```

Exemple 15: Jointure interne

Le mot clef **INNER** est facultatif car c'est le type de jointure par défaut. Ainsi on peut reformuler le requête ci-dessus en :

```
SELECT CLI_NOM, TEL_NUMERO
FROM   T_CLIENT C
       JOIN T_TELEPHONE T
             ON C.CLI_ID = T.CLI_ID
```

Exemple 16: Jointure interne par défaut

f) Jointure externe

Jointure externe simple

Les jointures externes sont extrêmement pratiques pour rapatrier le maximum d'informations disponible, même si des lignes de table ne sont pas renseignées entre les différentes tables jointes.

Procédons à l'aide d'un exemple pour mieux comprendre la différence entre une jointure interne et une jointure externe. Nous avons vu à l'exemple 2 que seul les clients dotés d'un numéro de téléphone étaient répertoriés dans la réponse.

```
SELECT CLI_NOM, TEL_NUMERO
FROM   T_CLIENT C
       LEFT OUTER JOIN T_TELEPHONE T
            ON C.CLI_ID = T.CLI_ID AND TYP_CODE IS NULL
```

Exemple 17: Jointure externe permettant d'afficher les clients sans téléphone

```
CLI_NOM   TEL_NUMERO
-------   --------------
DUPONT    01-44-28-52-50
DUPONT    05-59-45-72-42
MARTIN    01-47-66-29-55
BOUVIER   NULL
DUBOIS    04-66-62-95-64
DREYFUS   04-92-19-18-58
FAURE     NULL
LACOMBE   NULL
DUHAMEL   01-54-11-43-89
DUHAMEL   01-55-60-93-81
...
```

Les différents type de jointure externe

```
SELECT colonnes
FROM TGauche LEFT OUTER JOIN TDroite ON condition de jointure
```

On recherche toutes les valeurs satisfaisant la condition de jointure précisée dans prédicat, puis on rajoute toutes les lignes de la table ***TGauche*** qui n'ont pas été prises en compte au titre de la satisfaction du critère.

```
SELECT colonnes
FROM TGauche RIGHT OUTER JOIN TDroite ON condition de jointure
```

On recherche toutes les valeurs satisfaisant la condition de jointure précisée dans prédicat, puis on rajoute toutes les lignes de la table ***TDroite*** qui n'ont pas été prises en compte au titre de la satisfaction du critère.

```
SELECT colonnes
FROM TGauche FULL OUTER JOIN TDroite ON condition de jointure
```

On recherche toutes les valeurs satisfaisant la condition de jointure précisée dans prédicat, puis on rajoute toutes les lignes de la table ***TGauche*** et ***TDroite*** qui n'ont pas été prises en compte au titre de la satisfaction du critère.

Utilisation des expressions logiques

il existe des équivalences entre différentes expressions logiques à base de jointures externes :

- la jointure externe droite peut être obtenue par une jointure externe gauche dans laquelle on inverse l'ordre des tables ;
- la jointure externe bilatérale peut être obtenue par la combinaison de deux jointures externes unilatérales avec l'opérateur ensemblistes **UNION**.

Prenons la requête suivante :

```
SELECT CLI_NOM, TEL_NUMERO
FROM   T_CLIENT C
       FULL OUTER JOIN T_TELEPHONE T
            ON C.CLI_ID = T.CLI_ID
WHERE  TYP_CODE = 'FAX'
```

Exemple 18: Jointure externe complète

Elle peut être réécrite en :

```
SELECT CLI_NOM, TEL_NUMERO
FROM   T_CLIENT C
       LEFT OUTER JOIN T_TELEPHONE T
            ON C.CLI_ID = T.CLI_ID
WHERE  TYP_CODE = 'FAX'
UNION
SELECT CLI_NOM, TEL_NUMERO
FROM   T_CLIENT C
       RIGHT OUTER JOIN T_TELEPHONE T
            ON C.CLI_ID = T.CLI_ID
WHERE  TYP_CODE = 'FAX'
```

Exemple 19: Double jointures externes unilatérales

Ou avec une double jointure externe gauche :

```
SELECT CLI_NOM, TEL_NUMERO
FROM   T_CLIENT C
       LEFT OUTER JOIN T_TELEPHONE T
            ON C.CLI_ID = T.CLI_ID
WHERE  TYP_CODE = 'FAX'
UNION
SELECT CLI_NOM, TEL_NUMERO
FROM   T_TELEPHONE T
       LEFT OUTER JOIN T_CLIENT C
            ON C.CLI_ID = T.CLI_ID
WHERE  TYP_CODE = 'FAX'
```

Exemple 20: Double jointures externe unilatérales gauches

g) Différence entre jointure interne et externe

L'hypothèse du monde clos

Les jointures externes sont extrêmement pratiques pour rapatrier le maximum d'informations disponible, même si des lignes de table ne sont pas renseignées entre les différentes tables jointes. Sans le savoir, nous faisons assez systématiquement l'hypothèse du monde clos. c'est-à-dire que nous considérons que l'absence d'information, n'est pas une information.

Si vous demandez à une secrétaire de vous communiquer les coordonnées des clients qui sont domiciliés à Paris, elle vous donnera une liste où figurera autant de fois le nom "Montpellier" qu'il y a de clients dans la liste. Sauf que, comme l'aurait fait tout un chacun, votre secrétaire a fait l'hypothèse du monde clos sans le savoir en présumant que les clients pour lesquels l'adresse n'est pas renseignée ne sont pas domiciliés à Montpellier !

C'est cela l'hypothèse du monde clos : considérer que l'absence d'information doit être synonyme de critère de discrimination...

La jointure externe permet de contrer l'hypothèse du monde clos en considérant qu'en cas d'absence de jointure entre une table et l'autre, on ne supprime par pour autant l'information.

Mécanisme en jeu

Lorsqu'une ligne d'une table figurant dans une jointure n'a pas de correspondance dans les autres tables, le critère d'équi-jointure n'est pas satisfait et la ligne est rejetée : c'est la jointure interne.

Au contraire, la jointure externe permet de faire figurer dans le résultat les lignes satisfaisant la condition d'équi-jointure ainsi que celles n'ayant pas de correspondances.

Ainsi, si je veux contacter tous mes clients, quelque soit le mode de contact que je veux utiliser dans le cadre d'une campagne publicitaire. J'ai intérêt à obtenir une réponse contenant **tous** les clients, même ceux qui n'ont pas de téléphone, ou d'e-mail ou d'adresse.

```
SELECT C.CLI_ID, C.CLI_NOM, T.TYP_CODE || ' : ' || T.TEL_NUMERO AS TEL_CONTACT,
E.EML_ADRESSE, A.ADR_VILLE
FROM T_CLIENT C
   LEFT OUTER JOIN T_TELEPHONE T
        ON C.CLI_ID = T.CLI_ID
   LEFT OUTER JOIN T_EMAIL E
        ON  C.CLI_ID = E.CLI_ID
   LEFT OUTER JOIN T_ADRESSE A
        ON  C.CLI_ID = A.CLI_ID
```

Exemple 21: `Jointure externe multiple`

On peut remarquer que certain clients sortent plusieurs fois. Utilisez ***DISTINCT*** après le mot clé ***SELECT*** !

h) La jointure croisée

La jointure croisée n'est autre que le produit cartésien de deux tables. Rappelons que le produit cartésien de deux ensembles n'est autre que la multiplication généralisée. Dans le cas des tables, c'est le fait d'associer à chacune des lignes de la première table, toutes les lignes de la seconde.

Ainsi, si la première table compte 267 lignes et la seconde 1214, on se retrouve avec un résultat contenant 324 138 lignes. Bien entendu, s'il n'y a aucun doublon dans les tables, toutes les lignes du résultat seront aussi uniques.

Elle peut s'écrire de deux manières différentes :

```
SELECT colonnes
FROM table_1 CROSS JOIN table_2
```

Exemple 22: Jointure croisée avec opérateur normalisé

```
SELECT colonnes
FROM table_1, table_2
```

Exemple 23: Jointure croisée avec clause FROM

Pour savoir si notre table des tarifs de chambre (***TJ_TRF_CHB***) est complète, c'est-à-dire si l'on a bien toutes les chambres (***T_CHAMBRE***) pour toutes les dates de début de tarif (***T_TARIF***) :

```
SELECT CHB_ID, TRF_DATE_DEBUT, 0 AS TRF_CHB_PRIX
FROM   T_TARIF, T_CHAMBRE
ORDER  BY CHB_ID, TRF_DATE_DEBUT
```

Exemple 24: Jointure croisée simple

i) Jointure d'union

La jointure d'union, permet de faire l'union de deux tables de structures quelconque et ne s'exprime qu'à l'aide de l'opérateur normalisé suivant :

```
SELECT colonnes
FROM table_1 UNION JOIN table_2
```

Exemple 25: Jointure d'union simple

Effectuons par exemple la jointure d'union entre les tables ***T_TITRE*** ("M.", "Mme.", "Melle.") et ***T_TYPE*** ("FAX", "GSM", "TEL") :

```
SELECT *
FROM T_TITRE UNION JOIN T_TYPE
```

```
TIT_CODE TIT_LIBELLE                      TYP_CODE TYP_LIBELLE
-------- -------------------------------- -------- ------------------
M.       Monsieur                         NULL     NULL
Melle.   Mademoiselle                     NULL     NULL
Mme.     Madame                           NULL     NULL
NULL     NULL                             FAX      Télécopie
NULL     NULL                             GSM      Téléphone portable
NULL     NULL                             TEL      Téléphone fixe
```

C'est comme si l'on avait listé la première table, puis la seconde en évitant toute colonne commune et compléter les espaces vides des valeurs **NULL**.

j) Nature des conditions de jointures

Équi-jointure

L'équi-jointure consiste à opérer une jointure avec une condition d'égalité. Cette condition d'égalité dans la jointure peut ne pas porter nécessairement sur les clefs (primaires et étrangères).

Recherchons les clients dont le nom est celui d'une ville contenu dans la table des adresses :

```
SELECT DISTINCT C.CLI_ID, C.CLI_NOM, A.ADR_VILLE
FROM   T_CLIENT C
       INNER JOIN T_ADRESSE A
             ON C.CLI_NOM = A.ADR_VILLE
```

Exemple 26: Équi-jointure

Le mot clef **INNER** n'étant pas obligatoire, mais voulant s'opposer aux mot clefs **OUTER**, **UNION** et **CROSS**.

Non équi-jointure

Il s'agit là d'utiliser n'importe quelle condition de jointure entre deux tables, exceptée la stricte égalité. Ce peuvent être les conditions suivantes :

>	supérieur
>=	supérieur ou égal
<	inférieur
<=	inférieur ou égal
<>	différent de
IN	dans un ensemble
LIKE	correspondance partielle
BETWEEN ... AND ...	entre deux valeurs
EXISTS	dans une table

Nous voulons obtenir les factures qui ont été émises avant que le prix des petits déjeuners n'atteigne 6 €.

```
SELECT F.*
FROM   T_FACTURE F
       INNER JOIN  T_TARIF T
             ON F.FAC_DATE < T.TRF_DATE_DEBUT
WHERE  TRF_PETIT_DEJEUNE >= 6
```

Exemple 27: `Non équi-jointure`

Nota Bene : pour récupérer toutes les colonnes d'une table, on peut utiliser l'opérateur * suffixé par le nom de table.

Auto-jointure

Le problème consiste à joindre une table à elle-même. Il est assez fréquent que l'on ait besoin de telles auto-jointures car elle permettent notamment de modéliser des structures de données complexes comme des arbres. Voici quelques exemples de relation nécessitant une auto-jointure de tables :

- dans une table des employées, connaître le supérieur hiérarchique de tout employé ;
- dans une table de nomenclature savoir quels sont les composants nécessaires à la réalisation d'un module, ou les modules nécessaires à la réalisation d'un appareil ;
- dans une table de personnes, retrouver l'autre moitié d'un couple marié.

La représentation d'une telle jointure dans le modèle de données, se fait par le rajout d'une colonne contenant une pseudo clef étrangère basée sur le clef de la table.

Dans ce cas, une syntaxe possible pour l'auto-jointure est la suivante :

```
SELECT [DISTINCT ou ALL] * ou liste de colonnes
FROM   laTable t1
       INNER JOIN laTable t2
       ON t1.laClef = t2.laPseudoClefEtrangère
```

Exemple 28: `Syntaxe de l'auto-jointure`

Modélisons le fait qu'une chambre puisse communiquer avec une autre (par une porte). Dès lors, le challenge est de trouver quelles sont les chambres qui communiquent entre elles par exemple pour réaliser une sorte de suite. Pour ce faire, nous allons ajouter à notre table des chambres une colonne de clef étrangère basée sur la clef de la table.

Dans ce cas, cette colonne doit obligatoirement accepter des valeurs nulles !

Rajoutons la colonne ***CHB_COMMUNIQUE*** dans la table ***T_CHAMBRE*** :

```
ALTER TABLE T_CHAMBRE ADD CHB_COMMUNIQUE INTEGER
```

Ajoutons quelques valeurs exemples en considérant que la 7 communique avec la 9 et la 12 avec la 14 :

```
UPDATE T_CHAMBRE SET CHB_COMMUNIQUE = 9 WHERE CHB_ID = 7
UPDATE T_CHAMBRE SET CHB_COMMUNIQUE = 7 WHERE CHB_ID = 9
UPDATE T_CHAMBRE SET CHB_COMMUNIQUE = 12 WHERE CHB_ID = 14
UPDATE T_CHAMBRE SET CHB_COMMUNIQUE = 14 WHERE CHB_ID = 12
```

Pour formuler la recherche de chambres communicantes, il suffit de faire la requête suivante où la table ***T_CHAMBRE*** figure deux fois par l'entremise de deux sur-nommages différents ***c1*** et ***c2***.

```
SELECT DISTINCT c1.CHB_ID, c1.CHB_COMMUNIQUE
FROM   T_CHAMBRE c1
       INNER JOIN T_CHAMBRE c2
             ON c1.CHB_ID = c2.CHB_COMMUNIQUE
```

Exemple 29: Auto-jointure

```
CHB_ID  CHB_COMMUNIQUE
------  --------------
     7               9
     9               7
    12              14
    14              12
```

k) Clauses d'agrégation

GROUP BY

La clause ***GROUP BY*** est un élément optionnel des requêtes ***SELECT***. Syntaxiquement, elle est constituée de la séquence de mots clefs ***GROUP BY***, suivie d'une liste d'expressions (scalaires) séparées par des virgules.

```
SELECT [DISTINCT ou ALL] * ou liste_de_colonnes FROM nom_des_tables_ou_des_vues
GROUP BY liste_de_colonnes
```

Exemple 30: syntaxe du SELECT avec clause GROUP BY

Si une requête ***SELECT*** contient une clause ***GROUP BY***, celle-ci doit figurer après la clause WHERE sauf si la clause ***WHERE*** est omise, la clause ***GROUP BY*** suit immédiatement la clause FROM).

Quand elle est présente, la clause ***GROUP BY*** signifie que les données issu d'une requête doivent être divisées selon certains groupes, ne retournant qu'une seule ligne pour chacun de ces groupes. La liste des expressions précisées dans le ***GROUP BY*** définit la façon dont le regroupement doit se faire.

Toutes les lignes présentant la même combinaison de valeurs pour toutes les expressions précisées dans le ***GROUP BY*** sont dans le même groupe.

Par exemple, pour afficher tous les étages de l'hôtel on fait :

```
SELECT CHB_ETAGE FROM T_CHAMBRE GROUP BY CHB_ETAGE
```

Exemple 31: Exemple de clause GROUP BY

Calcul sur les agrégats

Une requête avec ***GROUP BY*** permet l'utilisation de fonctions d'agrégation sur une série de lignes réunies dans chaque groupe par la clause ***GROUP BY***. Une fonction d'agrégation peut traiter des expressions pour chaque ligne dans un groupe de lignes afin de retourner une seule valeur.

Quelques fonctions d'agrégation standard bien connues : ***COUNT***, ***MIN***, ***MAX*** et ***SUM.***

Les fonctions d'agrégation peuvent également être employées sans clause ***GROUP BY***, auquel cas le jeu de données intermédiaire est traité comme un seul grand groupe. Essayez d'imaginer l'effet d'une opération ***GROUP BY*** avec une liste ***GROUP BY*** vide : la requête ne va retourner qu'une seule ligne qui résume toutes les lignes du jeu de données intermédiaire.

Pour compléter notre exemple précédent avec la clause ***GROUP BY***, l'exemple suivant montre l'effet de quelques fonctions d'agrégation :

```
SELECT CHB_ETAGE AS ETAGE, GROUP_CONCAT( CONVERT( CHB_NUMERO, CHAR( 2 ) ) ) AS
NUMERO_CHAMBRE, COUNT( * ) AS NOMBRE_CHAMBRE, MIN( CHB_COUCHAGE ) MIN_COUCHAGE,
MAX( CHB_COUCHAGE ) MAX_COUCHAGE
FROM T_CHAMBRE
GROUP BY CHB_ETAGE
```

Exemple 32: *Utilisation de fonctions d'agrégation*

ETAGE	NUMERO_CHAMBRE	NOMBRE_CHAMBRE	MIN_COUCHAGE	MAX_COUCHAGE
1er	11,12,10,9,8,7,6,5	8	2	5
2e	18,19,17,20,16,15,14,21	8	2	5
RDC	4,3,2,1	4	2	3

La fonction ***GROUP_CONCAT*** a été appliquée à la colonne ***CHB_ETAGE***. Pour chaque étage dans notre table ***T_CHAMBRE***, il peut y avoir plusieurs numéro, et ***GROUP_CONCAT*** concatène ces numéros, les séparant par défaut par une virgule. Ainsi, dans cet exemple, l'expression ***GROUP_CONCAT*** fournit la constitution de chaque groupe de numéro d'un même étage ;

La fonction ***COUNT*** est employée avec un joker *, lui demandant de compter le nombre de lignes associé à chaque groupe. On peut vérifier cela avec la colonne précédente et l'on voit immédiatement que le nombre de lignes au sein de chaque groupe est cohérent avec le nombre de numéro concaténés par la fonction ***GROUP_CONCAT*** ;

Les fonctions ***MIN*** et ***MAX*** sont appliquées à la colonne ***CHB_COUCHAGE*** et ramène respectivement, pour chaque chambre, le nombre de couchage minimal (***MIN(CHB_COUCHAGE)***) et celui maximal (***MAX(CHB_COUCHAGE)***).

l) Utilisation des dates

Un peu plus compliqué, cherchons à savoir quel a été le nombre de nuitées pour chaque chmabre au cours de l'année 1999 (une nuitée étant une personne passant une nuit dans une chambre. Si deux personnes occupent la même chambre cela fait deux nuitées) :

```
SELECT SUM(CHB_PLN_CLI_NB_PERS), C.CHB_ID
FROM   T_CHAMBRE C
       JOIN TJ_CHB_PLN_CLI P
            ON C.CHB_ID = P.CHB_ID
WHERE  PLN_JOUR BETWEEN '1999-01-01' AND '1999-12-31'
GROUP  BY C.CHB_ID
```

Exemple 33: Utilisation des dates

```
PERSONNE    CHB_ID
----------- -----------
558         1
385         2
322         3
367         4
445         5
720         6
390         7
456         8
362         9
364         10
529         11
468         12
347         13
360         14
484         15
720         16
491         17
372         18
504         19
369         20
```

m) Sous-requête

On peut rechercher le taux d'occupation de chaque chambre dans cette période.

Tout d'abord on cherche le nombre maximal de nuitée qui est le nombre de couchage de chaque chambre multiplié par toutes les dates de l'année, ce calcul s'obtient par :

```
SELECT SUM(CHB_COUCHAGE) AS MAX_OCCUPATION, CHB_ID
FROM   T_CHAMBRE C
       CROSS JOIN T_PLANNING P
WHERE  PLN_JOUR BETWEEN '1999-01-01' AND '1999-12-31'
GROUP  BY C.CHB_ID
```

Exemple 34: Calcul du nombre de nuitée

Il ne suffit plus que de "raccorder" les requêtes des exemples 33 et 34 :

```
SELECT (SUM( CHB_PLN_CLI_NB_PERS ) / SUM( CHB_COUCHAGE )) *100 AS
TAUX_OCCUPATION_POURCENT, C.CHB_ID
FROM T_CHAMBRE C
JOIN TJ_CHB_PLN_CLI CPC ON C.CHB_ID = CPC.CHB_ID
CROSS JOIN T_PLANNING P
WHERE P.PLN_JOUR
BETWEEN '1999-01-01'
AND '1999-12-31'
GROUP BY C.CHB_ID
ORDER BY TAUX_OCCUPATION_POURCENT DESC
```

Exemple 35: Calcul du taux d'occupation avec sous-requête

```
TAUX_OCCUPATION_POURCENT   CHB_ID
-------------------------- ------
77.7630     18
76.6252     20
75.9209     13
75.8784     9
75.6812     10
74.7593     7
74.6603     4
74.6449     3
74.6066     2
74.5957     14
67.7155     11
67.3433     15
67.1920     8
66.8913     17
65.9009     19
65.3013     5
65.2930     1
65.2095     12
61.1622     16
60.2797     6
```

n) La clause HAVING

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

Recherchons un étage de l'hôtel capable de coucher au moins 20 personnes. Nous serions tenter d'écrire :

```
SELECT SUM(CHB_COUCHAGE) AS NOMBRE, CHB_ETAGE
FROM   T_CHAMBRE
WHERE  SUM(CHB_COUCHAGE) >= 20
GROUP  BY CHB_ETAGE
```

Exemple 36: *agrégation impossible dans la clause WHERE*

Mais cette requête va immanquablement provoquer une erreur avant exécution du fait de la clause WHERE. En effet, souvenons nous que **le filtre WHERE agit sur les données des tables** et permet de filtrer ligne après ligne.

Or le filtrage ne porte plus sur la notion de lignes, mais sur une notion de sous ensemble de la table. En d'autre termes, le filtre, ici, doit porter sur chacun des groupes. C'est pourquoi SQL introduit le filtre **HAVING** qui porte, non pas sur les données, mais sur les calculs résultants des regroupements.

En l'occurrence, dans notre exemple, nous devons déporter le filtre WHERE dans la clause HAVING et l'opération deviendra possible :

```
SELECT SUM(CHB_COUCHAGE) AS NOMBRE, CHB_ETAGE
FROM   T_CHAMBRE
GROUP  BY CHB_ETAGE
HAVING SUM(CHB_COUCHAGE) >= 20
```

Exemple 37: *Utilisation de HAVING pour filtrer des agrégats*

```
NOMBRE      CHB_ETAGE
----------- ---------
23          1er
22          2e
```

Partant de la requête vue à l'exemple 35, essayons de ne retenir que les chambres occupées à plus de 2/3, soit 66.666666... % ?

```
SELECT (SUM(CHB_PLN_CLI_NB_PERS) /
         SUM(CHB_COUCHAGE)) * 100 AS
         TAUX_OCCUPATION_POURCENT, C.CHB_ID
FROM   T_CHAMBRE C
        JOIN TJ_CHB_PLN_CLI CPC
              ON C.CHB_ID = CPC.CHB_ID
        CROSS JOIN T_PLANNING P
WHERE  P.PLN_JOUR BETWEEN '1999-01-01' AND '1999-12-31'
GROUP  BY C.CHB_ID
HAVING (SUM(CHB_PLN_CLI_NB_PERS) /
       SUM(CHB_COUCHAGE)) >= 2.0/3.0
```

Exemple 38: Utilisation de HAVING pour filtrer des agrégats

o) Les opérateurs ensemblistes

C'est une des parties les plus simples de l'ordre ***SELECT*** à la fois par sa syntaxe mais aussi par sa compréhension.

Il s'agit, ni plus ni moins que de réaliser des opérations sur les ensembles représentés par des tables ou des extraits de table. Les opérations ensemblistes du SQL sont l'union, l'intersection et le différence.

L'union

Pour faire une union, il suffit de disposer de deux ensembles de données compatibles et d'utiliser le mot clef ***UNION*** :

```
SELECT …
UNION
SELECT …
```

Exemple 39: Syntaxe de l'UNION

Bien entendu il est indispensable que les deux ordres SELECT :

- produisent un même nombre de colonnes ;
- que les types de données de chaque paires ordonnées de colonnes soient de même type (ou d'un type équivalent).

L'intersection

La démarche est la même pour faire une intersection, que pour le mécanisme de l'union. La syntaxe utilisant le mot clef ***INTERSECT*** :

```
SELECT …
INTERSECT
SELECT …
```

Exemple 40: *Syntaxe de INTERSECT*

La différence

La différence de deux ensembles s'obtient de la même manière, en utilisant le mot clef ***EXCEPT*** :

```
SELECT …
EXCEPT
SELECT …
```

Exemple 41: *Syntaxe de EXCEPT*

4. Modification des données grâce à INSERT

La syntaxe de base de l'ordre SQL d'insertion de données dans une table est la suivante :

```
INSERT [INTO] nom_de_la_table_cible [(liste_des_colonnes_visées)]
{VALUES (liste_des_valeurs) | requête_select | DEFAULT VALUES }
```

Exemple 42: syntaxe d'INSERT

Nota Bene :

1. la liste des colonnes visées peut être omise à condition que l'ordre d'insertion concerne toutes les colonnes de la table.
2. la liste des valeurs peut être remplacée par un constructeur de lignes valuées pour une insertion de plusieurs lignes en un seul ordre, mais rares sont les SGBDR à l'accepter (Oracle est l'un des rares SGBDR à accepter cette syntaxe).

L'ordre des valeurs de la liste doit être le même que l'ordre des colonnes visées et cela, même si la liste des colonnes visées est omise.

Du fait de sa syntaxe, l'ordre **INSERT** se décompose en trois ordres assez différents :

- Insertion simple explicite ;
- Insertion multiple explicite (à l'aide du constructeur de lignes valuées) ;
- Insertion à base de sous-requête **SELECT** ;
- Insertion des valeurs par défaut.

Nous allons maintenant détailler ces différentes déclinaisons de l'ordre **INSERT**.

a) Insertion simple explicite

Cette syntaxe porte sur l'insertion d'une ligne unique au sein de la table. Il s'agit de préciser les valeurs à insérer explicitement :

```
INSERT INTO T_MODE_PAIEMENT(PMT_CODE,PMT_LIBELLE) VALUES('CB','Carte bancaire')
```

Exemple 43: Insert simple explicite

Cet exemple propose d'insérer dans la table ***T_MODE_PAIEMENT*** une ligne comportant les valeurs "CB" et "Carte bancaire" dans les colonnes respectives ***PMT_CODE*** et ***PMT_LIBELLE***.

Compte tenu du fait que cette table ne possède que deux colonnes, on aurait pu omettre de préciser les colonnes. Dans ce cas, la requête devient :

```
INSERT INTO T_MODE_PAIEMENT VALUES('CB', 'Carte bancaire')
```

Exemple 44: Insert simple explicite sans nommage des colonnes

Lorsqu'une valeur n'est pas connue, il est possible de préciser le mot clef **NULL** (marqueur) qui laisse la colonne vide.

```
INSERT INTO T_MODE_PAIEMENT VALUES('X' , NULL)
```

Exemple 45: Insert simple avec utilisation de NULL

Qui insère un mode de paiement de code "X" et dont le libellé n'est pas renseigné.

b) Insertion multiple explicite à l'aide du constructeur de lignes valuées

Cette syntaxe porte sur l'insertion de multiples lignes au sein de la table cible. Il faut préciser les lignes de valeurs à insérer explicitement.

```
INSERT T_TITRE (TIT_CODE, TIT_LIBELLE)
       VALUES  ('M.'    , 'Monsieur',
                'Mlle.' , 'Mademoiselle'
                'Mme.'  , 'Madame')
```

Exemple 46: Insert multiple

Cet exemple propose d'insérer dans la table ***T_TITRE*** trois lignes de valeurs dans les colonnes ***TIT_CODE*** et ***TIT_LIBELLE***.

De même que dans notre précédent exemple, cette table ne possédant que deux colonnes, on aurait pu omettre de préciser les colonnes. Dans ce cas, la requête devient :

```
INSERT T_TITRE
       VALUES  ('M.'    , 'Monsieur',
                'Mlle.' , 'Mademoiselle'
                'Mme.'  , 'Madame')
```

Exemple 47: Insert multiple sans nommage des colonnes

c) Insertion partiellement explicite avec le mot clef DEFAULT

Si la définition de la table possède une ou plusieurs valeurs par défaut, alors il est possible de les y insérer en utilisant le mot-clé **DEFAULT** en lieu et place de la valeur.

Supposons que nous créions une table permettant de "pister" les connexions à la base de données, de la manière suivante :

```
CREATE TABLE T_SUIVI_CONNEXION
(CNX_USER        VARCHAR(128)  NOT NULL  DEFAULT 'Administrateur',
 CNX_DATE_HEURE  TIMESTAMP     NOT NULL  DEFAULT CURRENT_TIMESTAMP)
```

Exemple 48: Création d'une table avec valeur par défaut

Alors nous pouvons insérer l'heure et la date par défaut sans en préciser la valeur, à l'aide de la requête :

```
INSERT INTO T_SUIVI_CONNEXION (CNX_USER, CNX_DATE_HEURE)
       VALUES                 ('Dupont', DEFAULT)
```

Exemple 49: Insert explicite avec valeur par défaut

Qui insère automatiquement la valeur par défaut dans la colonne ***CNX_DATE_HEURE***.

Bien entendu on peut omettre la liste des noms de colonnes puisque, à nouveau, toutes les colonnes sont concernées par l'ordre d'insertion :

```
INSERT INTO T_SUIVI_CONNEXION VALUES ('Dupont',DEFAULT)
```

Exemple 50: Insert avec valeur par défaut sans nommage des colonnes

Qui donne le même résultat si l'ordre est exécuté au même moment !

On peut aussi ne donner qu'une liste partielle des colonnes visées, les autres dotées d'une valeur par défaut seront automatiquement alimentées :

```
INSERT INTO T_SUIVI_CONNEXION (CNX_USER) VALUES('Dupont')
```

Qui donne encore le même résultat si l'ordre est exécuté au même moment !

d) Insertion totalement implicite avec l'expression DEFAULT VALUES

Si chacune des colonnes de la définition de la table possède des valeurs par défaut, on peut demander l'insertion de toutes les valeurs par défaut en utilisant l'expression-clé **DEFAULT VALUES**. Dans ce cas il ne faut pas préciser les noms des colonnes :

```
INSERT INTO T_SUIVI_CONNEXION DEFAULT VALUES
```

Exemple 51: Insert implicite avec valeurs par défaut

Qui insérera l'utilisateur "Administrateur" avec la date et l'heure courante.

e) Insertion multiple à base de sous requête SELECT

On peut insérer une ou plusieurs lignes dans une table en utilisant une sous-requête de type **SELECT**. Dans ce cas, les colonnes retournées par l'ordre **SELECT** doivent avoir les contraintes suivantes :

- Être en nombre identique aux colonnes précisées dans la liste ou en l'absence de précision de cette liste, le même nombre de colonnes que la table ;
- Avoir le même ordre que l'ordre des noms de colonnes de la liste, ou bien, le même ordre que les colonnes de la table si l'on omet cette liste ;
- Avoir des types correspondants ;
- Répondre à toutes les contraintes et dans le cas où au moins une valeur viole une contrainte, aucune ligne n'est insérée

Nous allons décrire différents cas et le comportement du SGBDR correspondant.

Pour ce faire, je vous propose de créer de toutes pièces une nouvelle table dans notre base de données exemple, la table des prospects, et d'y insérer explicitement quelques données :

```
CREATE TABLE T_PROSPECT
(PRP_ID               INTEGER       NOT NULL,
 PRP_CODE_TITRE       CHAR(4)       NULL    ,
 PRP_NOM              CHAR(25)      NOT NULL,
 PRP_PRENOM           VARCHAR(16)   NULL    ,
 PRP_ENSEIGNE         VARCHAR(60)   NULL    ,
 PRP_DATE_SAISIE      TIMESTAMP     NOT NULL  DEFAULT CURRENT_DATETIME,
 CONSTRAINT PK_T_PROSPECT PRIMARY KEY (PRP_ID))
```

Exemple 52: Création de la table T_PROSPECT

```
INSERT INTO T_PROSPECT (PRP_ID, PRP_CODE_TITRE, PRP_NOM, PRP_PRENOM,
PRP_ENSEIGNE, PRP_DATE_SAISIE)
       VALUES          (1,      'M.',          'Dupont','Alain',    NULL,
DEFAULT        )
INSERT INTO T_PROSPECT
       VALUES          (2,      'Mme.',        'Durand','Aline',    'SNCF',
'2000-12-25'   )
INSERT INTO T_PROSPECT (PRP_ID, PRP_CODE_TITRE, PRP_NOM )
       VALUES          (3,      'M.',          'Dubois')
```

Exemple 53: Insertion de valeurs dans la table T_PROSPECT

La table T_PROSPECT contient donc :

```
PRP_ID  PRP_CODE_TITRE PRP_NOM    PRP_PRENOM   PRP_ENSEIGNE   PRP_DATE_SAISIE
------- -------------- ---------- ------------ --------------
-----------------------
1       M.             Dupont     Alain        NULL           2002-03-16
00:00:00.000
2       Mme.           Durand     Aline        SNCF           2000-12-25
00:00:00.000
3       M.             Dubois     NULL         NULL           2002-03-16
00:00:00.000
```

Exemple 54: Contenu de la table T_PROSPECT

Un premier exemple, que nous allons décortiquer, va nous permettre de comprendre comment fonctionne l'insertion avec une sous-requête **SELECT** :

```
INSERT INTO   T_CLIENT (CLI_ID, TIT_CODE,       CLI_NOM, CLI_PRENOM)
       SELECT           PRP_ID, PRP_CODE_TITRE, PRP_NOM, PRP_PRENOM
       FROM   T_PROSPECT
```

Exemple 55: Insert avec sous-requête SELECT

Première étape, exécution de la sous-requête **SELECT** :

```
SELECT PRP_ID, PRP_CODE_TITRE, PRP_NOM, PRP_PRENOM
FROM   T_PROSPECT
```

Exemple 56: Sous-requête SELECT

Qui donne le résultat :

```
PRP_ID  PRP_CODE_TITRE PRP_NOM    PRP_PRENOM
------- -------------- ---------- ----------
1       M.             Dupont     Alain
2       Mme.           Durand     Aline
3       M.             Dubois     NULL
```

Exemple 57: Résultat du SELECT

Seconde étape, Il faut maintenant insérer ces données dans les colonnes ***CLI_ID***, ***TIT_CODE***, ***CLI_NOM***, ***CLI_PRENOM*** correspondantes de la table ***T_CLIENT***. Celle-ci contient :

```
SELECT CLI_ID, TIT_CODE, CLI_NOM, CLI_PRENOM
FROM T_CLIENT

Qui donne :

CLI_ID  TIT_CODE CLI_NOM   CLI_PRENOM
------- -------- --------- ----------
1       M.       DUPONT    Alain
2       M.       MARTIN    Marc
3       M.       BOUVIER   Alain
4       M.       DUBOIS    Paul
5       M.       DREYFUS   Jean
6       M.       FAURE     Alain
7       M.       LACOMBE   Paul
8       Melle.   DUHAMEL   Evelyne
9       Mme.     BOYER     Martine
…
```

Exemple 58: Contenu de la table T_CLIENT

L'exécution de la requête donne :

```
Serveur: Msg 2627, Niveau 14, État 1, Ligne 1
Violation de la contrainte PRIMARY KEY 'PK_T_CLIENT'.
Impossible d'insérer une clé en double dans l'objet 'T_CLIENT'.
L'instruction a été arrêtée.
```

Exemple 59: Violation de contrainte

Et oui, car on ne peut insérer des clefs en double, or nous tentons d'insérer un deuxième client portant la clef 1, un autre portant le clef 2, etc...

En l'occurrence aucune ligne n'est donc insérée car toute requête de mise à jour est une transaction et fonctionne en "**tout ou rien**", c'est-à-dire qu'aucune ligne n'est insérée si au moins une contrainte n'est pas vérifiée.

La violation de la contrainte de clef primaire étant ici évidente, rien n'est inséré dans la table ***T_CLIENT***.

Nous savons à la lecture des données de la table des clients, que la clef la plus haute possède la valeur 100 → Il est donc possible d'insérer des clients **uniquement si** la clef est **supérieure** à cette valeur.

Il faut rajouter cette valeur à la valeur de clé de la table ***T_PROSPECT*** récupérée à l'aide de la requête **SELECT**, par une simple addition :

Et là, nos trois prospects ont été insérés :

```
CLI_ID  TIT_CODE CLI_NOM  CLI_PRENOM  CLI_ENSEIGNE
------- -------- -------- ----------- ------------
101     M.       Dupont   Alain       NULL
102     Mme.     Durand   Aline       NULL
103     M.       Dubois   NULL        NULL
```

Exemple 60: Insertions effectives

Il semble facile de remplacer la valeur 100 par une autre sous-requête renvoyant la valeur maximale de la clef de la table ***T_CLIENT*** afin de l'ajouter à la valeur de la clef de la table ***T_PROSPECT*** :

```
INSERT INTO   T_CLIENT (CLI_ID, TIT_CODE, CLI_NOM, CLI_PRENOM)
            SELECT PRP_ID + (SELECT MAX(CLI_ID) FROM T_CLIENT),
            PRP_CODE_TITRE,
            PRP_NOM,
            PRP_PRENOM
            FROM T_PROSPECT
```

Exemple 61: Insert avec sous-requête récupérant l'identifiant

Ce qui s'exécute parfaitement et donne :

```
CLI_ID  TIT_CODE CLI_NOM  CLI_PRENOM  CLI_ENSEIGNE
------- -------- -------- ----------- ------------
104     M.       Dupont   Alain       NULL
105     Mme.     Durand   Aline       NULL
106     M.       Dubois   NULL        NULL
```

Exemple 62: Insertions effectives avec sous-requête récupérant l'identifiant

L'explication est simple : la sous-requête (**SELECT MAX**[...]) n'est exécutée qu'une seule fois puisque qu'elle n'est pas corrélée avec la table cible de l'insertion.

f) Insertion multiple et conditionnelle à base de sous-requêtes SELECT corrélées

Il est possible de corréler la sous-requête d'insertion avec la requête cible. Dans ce cas il faut préciser une seconde fois la table cible dans la clause **FROM** de la sous-requête **SELECT**.

Nous voudrions insérer en tant que clients les prospects dont le couple de valeurs nom/prénom est différent. Dans ce cas, les prospects 1 et 2 ne doivent pas être insérés, mais le prospect 3 (Dubois) doit être inséré.

Première phase, trouvons la requête qui va récupérer les clients qui sont des prospects :

```
SELECT PRP_ID, PRP_CODE_TITRE, PRP_NOM, PRP_PRENOM
FROM   T_PROSPECT P
JOIN T_CLIENT C
ON C.CLI_NOM = P.PRP_NOM
```

Exemple 63: Insertion conditionnelle

Qui donne :

```
PRP_ID      PRP_CODE_TITRE PRP_NOM                          PRP_PRENOM
----------- -------------- -------------------------------- ----------
1           M.             Dupont                           Alain
3           M.             Dubois                           NULL
```

Exemple 64: Résultat de l'insertion conditionnelle

Nota Bene : Si vous avez inséré les données des exemples précédents, veuillez procéder à la suppression des prospects insérés à l'aide de la requête :

```
DELETE FROM T_CLIENT WHERE CLI_ID > 100
```

Exemple 65: Suppression conditionnelle

Pour trouver l'inverse, c'est-à-dire les prospects qui ne sont pas des clients, il suffit de réaliser une exclusion. On peut alors utiliser l'ordre **EXCEPT** :

```
SELECT PRP_ID, PRP_CODE_TITRE, PRP_NOM, PRP_PRENOM
FROM   T_PROSPECT P
EXCEPT
SELECT CLI_ID, TIT_CODE, CLI_NOM, CLI_PRENOM
FROM   T_CLIENT
```

Exemple 66: exclusion conditionnelle

Ou encore avec des jointures externes :

```
SELECT PRP_ID, PRP_CODE_TITRE, PRP_NOM, PRP_PRENOM
FROM   T_PROSPECT P
       LEFT OUTER JOIN T_CLIENT C
            ON C.CLI_NOM <> P.PRP_NOM
            AND C.CLI_PRENOM <> P.PRP_PRENOM
WHERE CLI_ID IS NULL
```

```
PRP_ID      PRP_CODE_TITRE PRP_NOM                          PRP_PRENOM
----------- -------------- -------------------------------- ----------
3           M.             Dubois                           NULL
```

Exemple 67: Résultat de la suppression conditionnelle

Dès lors, l'insertion devient simple si l'on oublie pas de rajouter à la clef la valeur maximale de la clef de ***T_CLIENT***.

```
INSERT INTO   T_CLIENT (CLI_ID,
                        TIT_CODE,       CLI_NOM, CLI_PRENOM)
       SELECT           PRP_ID + (SELECT MAX(CLI_ID)
                                  FROM T_CLIENT),
                        PRP_CODE_TITRE, PRP_NOM, PRP_PRENOM
       FROM   T_PROSPECT P
              LEFT OUTER JOIN T_CLIENT C
                   ON C.CLI_NOM <> P.PRP_NOM
                      AND C.CLI_PRENOM <> P.PRP_PRENOM
       WHERE  CLI_ID IS NULL
```

Exemple 68: Insertion conditionnelle avec jointure

g) Insertion en auto-référence

L'insertion en auto-référence consiste à ajouter à une table une ou plusieurs nouvelles lignes calculées d'après les lignes existantes de la table cible.

Par exemple nous voulons insérer dans la tables des tarifs une nouvelle ligne avec comme date d'application du tarif, le premier janvier 2013, le même taux de taxe que le tarif précédent et un prix de petit déjeuner de 10% de plus que le précédent tarif.

Obtenir le précédent tarif consiste à trouver la ligne de la table dont la valeur de la date est maximale. Cela peut s'effectuer à l'aide de la requête suivante :

```
SELECT *
FROM   T_TARIF
WHERE  TRF_DATE_DEBUT = (SELECT MAX(TRF_DATE_DEBUT)
                         FROM T_TARIF)
```

Exemple 69: requête sélectionnant la ligne où la date est maximale

Cela donne :

```
TRF_DATE_DEBUT TRF_TAUX_TAXES TRF_PETIT_DEJEUNE
-------------- -------------- -----------------
2013-01-01     20.6000        69.5750
```

Dès lors on peut utiliser cet ordre SELECT en le modifiant un peu de manière à lui faire insérer la nouvelle ligne tarifaire :

```
INSERT INTO T_TARIF (TRF_DATE_DEBUT, TRF_TAUX_TAXES, TRF_PETIT_DEJEUNE)
       SELECT        '2013-01-01',   TRF_TAUX_TAXES, TRF_PETIT_DEJEUNE * 1.1
       FROM   T_TARIF
       WHERE  TRF_DATE_DEBUT = (SELECT MAX(TRF_DATE_DEBUT)
                                FROM T_TARIF)
```

Exemple 70: insertion du nouveau tarif avec 10% de marge

Voici les principaux cas pour lesquels un ordre d'insertion ne peut aboutir :

- Violation de clef (index primaire) ;
- Violation de contrainte d'index secondaire unique ;
- Violation de contrainte de données (colonne **NOT NULL**) ;
- Violation d'intégrité référentielle ;
- Violation de contrainte de contrôle de validité (min, max, étendue, domaine, etc.).

5. Suppression à l'aide de DELETE

La syntaxe de base de l'ordre SQL de suppression de données dans une table est la suivante :

```
DELETE [FROM] nom_table_cible
[WHERE condition]
```

Exemple 71: Syntaxe du DELETE

Le seul cas pour lequel cet ordre peut ne pas aboutir est lorsque la suppression viole la contrainte d'intégrité référentielle. Il est en effet absurde de vouloir supprimer un client si les factures relatives à ce client n'ont pas été préalablement supprimées.

Dans certains cas, il se peut que la suppression d'une ligne entraîne la suppressions d'autres lignes dans d'autres tables lorsqu'il existe des intégrités référentielles de suppression en cascade.

a) Suppression de toutes les lignes d'une table

C'est la forme la plus simple de l'ordre **DELETE** puisqu'il suffit d'omettre la clause **WHERE** :

```
DELETE FROM T_PROSPECT
```

Exemple 72: DELETE simple

Cette requête supprime tous les prospects.

b) Suppression conditionnelle

Il suffit de rajouter la clause **WHERE** dotée d'un prédicat.

```
DELETE FROM T_PROSPECT
WHERE  PRP_PRENOM LIKE '%d'
```

Exemple 73: DELETE avec clause WHERE

Cette requête supprime tous les prospects dont le nom se termine par la lettre 'd'.

c) Suppression avec sous requête conditionnelle

Il est possible d'utiliser une sous-requête conditionnelle dans la clause **WHERE** d'un ordre **DELETE**.

Supprimons les prospects dont le couple de valeurs nom/prénom se trouve dans la table des clients. Procédons pour cela par étape.

Pour obtenir la liste des prospects qui figurent en tant que client, nous pouvons faire la requête suivante :

```
SELECT PRP_ID, PRP_NOM, PRP_PRENOM
FROM   T_PROSPECT P
       JOIN T_CLIENT C
            ON C.CLI_NOM = P.PRP_NOM
               AND C.CLI_PRENOM = P.PRP_PRENOM

Ce qui donne :

PRP_ID      PRP_NOM         PRP_PRENOM
----------- --------------- ----------
1           Dupont          Alain
```

Exemple 74: Liste des prospects qui sont également clients

Dès lors il suffit de supprimer les prospects dont l'identifiant est récupéré par la sous-requête :

```
DELETE FROM T_PROSPECT
WHERE  PRP_ID = (SELECT PRP_ID
                 FROM   T_PROSPECT P
                        JOIN T_CLIENT C
                             ON C.CLI_NOM = P.PRP_NOM
                            AND C.CLI_PRENOM = P.PRP_PRENOM)
```

Exemple 75: DELETE avec sous-requête

On peut procéder aussi à l'aide du constructeur de lignes valuées, si votre SGBDR le supporte, ce qui simplifie l'écriture de la requête :

```
DELETE FROM T_PROSPECT
WHERE  (CLI_NOM, CLI_PRENOM) = (SELECT PRP_NOM, PRP_PRENOM
                                FROM   T_CLIENT)
```

Exemple 76: DELETE valué

Bien entendu la construction de sous-requêtes dans un ordre **DELETE** peut être compliqué à souhait afin d'effectuer la suppression conditionnelle désirée.

6. Modification à l'aide d'UPDATE

La syntaxe de base de l'ordre SQL de modification de données dans une table est la suivante :

```
UPDATE nom_table_cible
SET colonne = valeur [, colonne2 = valeur2 ...]
[WHERE condition]
```

Exemple 77: Syntaxe de l'UPDATE

a) Mise à jour d'une colonne unique sans condition

C'est la forme la plus simple de l'ordre **UPDATE**. Nous voulons par exemple fixer à 8€ les tarifs de nos petits déjeuners dans la table T_TARIF :

```
UPDATE T_TARIF
SET    TRF_PETIT_DEJEUNE = 8
```

Exemple 78: UPDATE simple

b) Mise à jour d'une colonne unique avec reprise de valeur (auto référence)

On peut aussi reprendre la valeur de la colonne (ou d'une autre colonne de la table cible). Par exemple, nous pouvons demander une augmentation de 15% des tarifs des petits déjeuners :

```
UPDATE T_TARIF
SET    TRF_PETIT_DEJEUNE = TRF_PETIT_DEJEUNE * 1.15
```

Exemple 79: UPDATE auto-référencé

c) Mise à jour d'une colonne unique avec filtrage

On peut ajouter une clause de filtrage **WHERE** dans une requête de mise à jour. Par exemple nous pouvons décider de n'augmenter de 15% que les tarifs des petits déjeuners des périodes postérieures à 2013.

```
UPDATE T_TARIF
SET    TRF_PETIT_DEJEUNE = TRF_PETIT_DEJEUNE * 1.15
WHERE  EXTRACT(YEAR FROM TRF_DATE_DEBUT) > 2013
```

Exemple 80: UPDATE avec clause WHERE

d) Mise à jour de plusieurs colonnes simultanément

Pour mettre à jour simultanément plusieurs colonnes, il suffit de répéter autant de fois que nécessaire le contenu de la clause **SET**, à raison d'un couple colonne/valeur par colonne visées par la mise à jour.

```
UPDATE T_CLIENT
SET    CLI_NOM      = UPPER(CLI_NOM),
       CLI_PRENOM   = UPPER(CLI_PRENOM)
       CLI_ENSEIGNE = UPPER(CLI_ENSEIGNE)
```

Exemple 81: UPDATE simultané

Dans ce cas, la nullité de l'exécution de modification d'une valeur dans une colonne possédant le marqueur **NULL**, n'entraîne pas la nullité de l'exécution des mises à jour des autres colonnes, chaque modification de colonne étant évaluées séparément.

e) Mise à jour avec sous-requête

Comme dans les ordres **INSERT** et **DELETE**, il est possible d'utiliser une sous-requête dans la clause **WHERE** de l'ordre **UPDATE** afin de filtrer de manière plus complète.

Par exemple, afin d'éviter de confondre des prospects qui ont le même nom et prénom que certains clients, on désire ajouter le mot "bis" aux prospects homonymes :

```
UPDATE T_PROSPECT
SET    PRP_NOM = TRIM(RIGHT, PRP_NOM) || ' bis'
WHERE  PRP_ID = (SELECT  PRP_ID
                 FROM   T_PROSPECT P
                    JOIN T_CLIENT C
                         ON C.CLI_NOM = P.PRP_NOM
                        AND C.CLI_PRENOM = P.PRP_PRENOM)
```

Exemple 82: UPDATE avec sous-requête

f) Mise à jour de valeurs particulières (défaut et marqueur NULL)

Il est possible de mettre à jour une colonne à sa valeur par défaut si elle possède une telle spécificité élaborée dans la création de la table.

En reprenant la définition des tables de connexion, donnons à la colonne ***CNX_USER*** sa valeur par défaut pour toutes les lignes de la table :

```
UPDATE  T_SUIVI_CONNEXION
SET     CNX_USER = DEFAULT
```

Exemple 83: UPDATE avec valeur par défaut

Il est aussi possible de supprimer le contenu d'une colonne (ou de plusieurs) en y plaçant le marqueur **NULL** :

```
UPDATE  T_CLIENT
SET     CLI_ENSEIGNE = NULL
```

Exemple 84: UPDATE avec valeur NULL

Cette requête vide la colonne ***CLI_ENSEIGNE*** de la table des clients en y plaçant **NULL**.

Nota Bene : c'est le seul cas ou l'on trouvera le mot-clé **NULL** associé au signe égal, car dans ce cas le signe égal est un opérateur d'affectation.

De manière syntaxique il aurait mieux valu une construction du type :

```
UPDATE  T_CLIENT
SET     CLI_ENSEIGNE AS NULL
```

Exemple 85: Utilisation du AS

Une mise à jour peut échouer si elle viole les contraintes. Voici les principaux cas pour lesquels un ordre de modification ne peut aboutir :

- Violation de clef (index primaire) ;
- Violation de contrainte d'index secondaire unique ;
- Violation de contrainte de données (colonne **NOT NULL**) ;
- Violation d'intégrité référentielle ;
- Violation de contrainte de contrôle de validité (min, max, étendue, domaine, etc.).

V. Valeurs ambiguës

Il arrive lors des insertions et des mise à jour que la valeur passée en argument soit ambiguë car son format ou son type ne peut être exprimé que par l'intermédiaire du jeu de caractère ordinaire.

Comment donc savoir si la chaîne "AF12" est une chaîne de caractères ou un code hexadécimal représentant 4 octets soit 2 caractères ?

Pour lever cette ambiguïté, on doit utiliser une lettre de préfixage :

N → Unicode (National)
B → Bit
X → Hexadécimal

Exemple 86: préfixage

```
CREATE TABLE T_NOMECLATURE
(NMC_REFERENCE NATIONAL CHAR(13) NOT NULL PRIMARY KEY,
  NMC_COMPOSANT BIT(1) NOT NULL DEFAULT 0,
  NMC_ADRESSE_MEMOIRE VARBINARY (4))

INSERT INTO T_NOMECLATURE (NMC_REFERENCE, NMC_COMPOSANT, NMC_ADRESSE_MEMOIRE)
VALUES (N'747 XWC-3344', B'0', X'AF12')
```

Exemple 87: Exemple de préfixage

Annexe

7. Index des illustrations

Index des illustrations

8. Index des exemples

Index des exemples