

Chapitre 2 : SQL - recherche de données

L'objectif de ce cours est d'approfondir notre connaissance des possibilités du langage SQL en matière de recherche de données dans une base.

Table des matières

Introduction.....	2
Recherche mono-table.....	2
Projection : choix des colonnes.....	2
Elimination des doublons.....	3
Restriction : filtrage sur les données.....	3
Fonctions sur les valeurs.....	4
Tri du résultat.....	4
Fonctions statistiques.....	5
Regroupements.....	5
Filtrage sur les regroupements.....	6
Recherche multi-tables.....	7
Jointures.....	7
Jointure interne.....	7
Jointure naturelle.....	8
Autojointure.....	9
Jointure externe.....	9
Requêtes imbriquées.....	10
Sous-requêtes simples.....	10
Sous-requêtes corrélées.....	11
Opérateurs ensemblistes.....	12
Union.....	12
Intersection.....	13
Différence.....	13
Application.....	13
Annexe : base de données d'exemple.....	14

Introduction

La recherche d'informations dans une base est l'opération la plus courante. Elle s'effectue à l'aide de la commande SQL **SELECT**.

Remarques :

- ✓ Ce chapitre suppose une connaissance basique des possibilités du **SELECT**, et se concentre sur des points délicats ou particuliers.
- ✓ Il s'appuie sur la base de données d'exemple présentée en annexe.

La syntaxe générale de cette instruction est :

```
SELECT [DISTINCT] <liste d'expressions>
FROM <liste de tables>
[ WHERE <liste de conditions> ]
[ GROUP BY <liste de regroupements> ]
[ HAVING <liste de conditions sur les regroupements> ]
[ ORDER BY <liste de tris> ] ;
```

Remarques :

- ✓ Les différentes sous-parties d'un **SELECT** (**FROM**, **WHERE**, etc) sont appelées des **clauses**.
- ✓ L'ordre de déclaration des clauses ci-dessus doit être respecté.
- ✓ Toutes les clauses ne sont pas systématiquement présentes. C'est le sens des crochets [] qui les entourent.

Recherche mono-table

Projection : choix des colonnes

La clause **SELECT** de l'ordre SQL de sélection permet de choisir les attributs de l'information à extraire. Cette clause peut comprendre :

- Le caractère **étoile (*)** qui spécifie que toutes les colonnes de la **table** (ou de la **vue**) doivent être prises en compte.
- Des colonnes spécifiées par leur nom et éventuellement renommées.
- Des fonctions ou des opérateurs algébriques.

Exemples :

- **SELECT * FROM T_UTILISATEUR;**
- **SELECT USR_NOM, USR_PRENOM FROM T_UTILISATEUR;**
- **SELECT USR_ID AS "Identifiant de l'utilisateur" FROM T_UTILISATEUR;**

Remarque : le mot-clé **AS**, qui permet de renommer une colonne résultat, est optionnel. Le nouveau nom est à placer entre guillemets (" ").

Elimination des doublons

Le mot-clé **DISTINCT**, optionnel, permet d'éliminer les doublons dans les lignes du résultat.

Exemple :

```
SELECT DISTINCT USR_ORGANISATION FROM T_UTILISATEUR;
```

Restriction : filtrage sur les données

Le plus souvent, on ne s'intéresse qu'à une partie des données d'une table. C'est la clause **WHERE** qui réalise cette opération de restriction. Son algorithme est le suivant.

```
i <- 1
TantQue i <= nombre de lignes de la table
  Si la ligne i satisfait à la clause WHERE
    Ajouter la ligne i au résultat de la requête
  Fsi
  i++
Ftq
```

La clause **WHERE** peut contenir divers opérateurs :

- les opérateurs de comparaison algébrique =, <>, <, <=, >, >=,
- l'opérateur d'encadrement **BETWEEN**,
- l'opérateur de comparaison approchée **LIKE**,
- l'opérateur de comparaison avec une liste **IN**,
- l'opérateur de test d'absence de valeur **IS NULL**.

De plus, on peut employer dans une clause WHERE les éléments logiques suivants :

- le mot-clé **NOT** pour inverser le sens logique de l'expression qui suit,
- les mots-clés **AND** et **OR** pour combiner des expressions logiques,
- des **parenthèses** pour hiérarchiser des expressions (attention à leur placement !).

Exemples :

- **SELECT * FROM T_UTILISATEUR WHERE USR_ORGANISATION = 'IBM';**
- **SELECT USR_NOM, USR_PRENOM, USR_ORGANISATION FROM T_UTILISATEUR WHERE USR_ORGANISATION IS NOT NULL AND (USR_PRENOM BETWEEN 'A' AND 'G' OR USR_NOM LIKE '%R%');**

Remarques :

- ✓ Les apostrophes (') permettent de délimiter certaines valeurs de données (chaînes, dates, etc), mais pas les valeurs numériques.
- ✓ La sensibilité à la casse dépend de la configuration du SGBDR.
- ✓ Un **BETWEEN** réalise une inclusion non stricte (les bornes sont comprises).
- ✓ Dans un **LIKE**, le caractère spécial **_** permet de remplacer un caractère et le caractère spécial **%** permet de remplacer **0 ou plusieurs** caractères.

Que renvoie la 2ème requête ci-dessus ?

Les utilisateurs (nom, prénom, organisation) qui vérifient les deux conditions suivantes :

- ils sont affectés à une organisation
- leur prénom commence par une lettre entre A et G ou leur nom comporte la lettre R.

Fonctions sur les valeurs

SQL propose de nombreuses fonctions sur les valeurs. Voici les plus fréquemment utilisées :

- **UPPER**(c) renvoie la chaîne c en **majuscules**.
- **LOWER**(c) renvoie la chaîne c en **minuscules**.
- **CURRENT_DATE** renvoie la date courante.
- **EXTRACT**({ **YEAR** | **MONTH** | **DAY** | **HOUR** | **MINUTE** | **SECOND** } **FROM** t) permet d'extraire une partie d'une valeur temporelle et renvoie un entier.

Exemples :

- **SELECT** USR_NOM **AS** Nom, **UPPER**(USR_NOM) **AS** "Nom en majuscules"
FROM T_UTILISATEUR;
- **SELECT** * **FROM** T_FORUM
WHERE **EXTRACT**(**YEAR FROM** FRM_DATE_CREATION) =
EXTRACT(**YEAR FROM** CURRENT_DATE);

Que renvoie la 2ème requête ?

Les forums dont l'année de création est égale à l'année courante.

Tri du résultat

La clause **ORDER BY** permet de trier les résultats d'une requête de sélection. Les mots-clés **ASC** et **DESC** définissent l'ordre de tri : **croissant** pour **ASC** et **décroissant** pour **DESC**. Par défaut, c'est l'ordre de tri croissant qui est utilisé.

Exemple :

```
SELECT USR_ID, USR_NOM, USR_ORGANISATION  
FROM T_UTILISATEUR  
ORDER BY USR_ORGANISATION, 2 DESC;
```

Comment la requête précédente est-elle triée ?

D'abord sur l'organisation par tri croissant, puis sur le nom de l'utilisateur (2ème colonne du SELECT) par tri décroissant.



Si elle existe, la clause **ORDER BY** doit toujours figurer en dernière position dans un ordre SQL.

Fonctions statistiques

SQL permet d'effectuer des calculs statistiques portant sur des ensembles de données. Les fonctions associées sont parfois appelées fonctions d'agrégation.

Fonction	Descriptions
COUNT(*)	Compte le nombre de lignes
COUNT	Compte le nombre d'occurrences
MAX	Renvoie le m a x i m u m
MIN	Renvoie le minimum
SUM	Effectue une s o m m e
AVG	Calcule la moyenne

Toutes ces fonctions, à l'exception du **COUNT(*)** :

- prennent en paramètre une colonne (ou une expression utilisant une colonne),
- ne tiennent pas compte des NULL (absence de valeur),
- acceptent le mot-clé **DISTINCT** pour éliminer les doublons *avant* le calcul.

Exemples:

- **SELECT SUM(FIC_TAILLE_KO) AS "Taille totale en Ko" FROM T_FICHER**
WHERE FIC_NOM LIKE '%.doc%';
- **SELECT COUNT(*) AS LIGNES,**
COUNT(USR_ORGANISATION) AS NB_ORGAS,
COUNT(DISTINCT USR_ORGANISATION) AS NB_ORGAS_DIST
FROM T_UTILISATEUR;



Il faut savoir utiliser à bon escient **COUNT(*)**, **COUNT(...)**, **COUNT(DISTINCT ...)** et **SUM**.

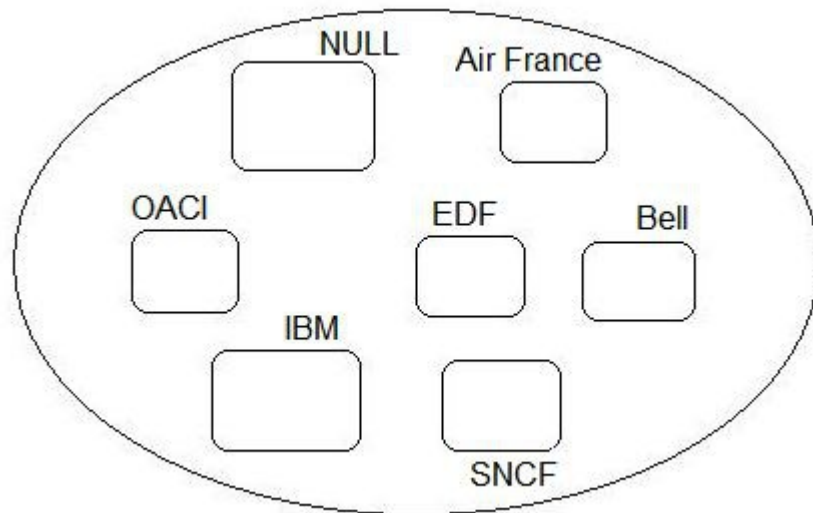
Regroupements

Il n'est pas toujours pertinent de faire des calculs statistiques sur toute une table. SQL permet de créer à la volée des **sous-ensembles** sur lesquels porteront les calculs. La clause **GROUP BY** permet de réaliser ce mécanisme.

Exemple :

```
SELECT USR_ORGANISATION FROM T_UTILISATEUR  
GROUP BY USR_ORGANISATION;
```

La requête ci-dessus crée dynamiquement autant de sous-tables qu'il existe d'organisations différentes dans la table T_UTILISATEUR.



Regroupement de la table T_UTILISATEUR selon l'organisation

On peut réaliser des calculs statistiques sur les sous-ensembles définis.

Exemple :

```
SELECT USR_ORGANISATION, COUNT(*) FROM T_UTILISATEUR
GROUP BY USR_ORGANISATION;
```

Que fait la requête précédente ?

Elle calcule le nombre d'utilisateurs pour chaque organisation.



Les colonnes utilisées dans la clause **GROUP BY** doivent figurer dans la clause **SELECT**. Une bonne règle est de considérer qu'en présence d'un **GROUP BY**, les seuls éléments autorisés dans le **SELECT** sont les colonnes du **GROUP BY** et les fonctions statistiques (**COUNT**, **SUM**, etc).

Filtrage sur les regroupements

Il est souvent intéressant de pouvoir extraire des informations fondées sur des valeurs de groupage à l'aide d'un filtre. Par exemple, on souhaiterait connaître les organisations qui comptent plus de deux utilisateurs. Pour obtenir cette information, il faudrait filtrer sur le résultat du calcul statistique (le **COUNT(*)** associé au **GROUP BY** de la requête précédente). Or, il n'est pas possible d'utiliser le filtre **WHERE** puisque les résultats du calcul statistique ne sont pas présentes dans la table, et que le **WHERE** opère sur chaque ligne de la table.

Pour y parvenir, il est nécessaire d'ajouter une clause **HAVING** après le **GROUP BY**.



Retenez que :

- Le filtre **WHERE** porte sur les lignes : il opère **avant** le groupage.
- Le filtre **HAVING** porte sur les groupes : il opère **après** le groupage.

On pourrait écrire l'algorithme du **HAVING** comme ci-dessous.

```

i <- 1
TantQue i <= nombre de lignes de la table
  Si la ligne i satisfait à la clause WHERE
    Ajouter la ligne i au résultat du WHERE
  Fsi
  i++
Ftq
<Regroupement des lignes du résultat du WHERE selon le GROUP BY>
j <- 1
TantQue j < nombre de lignes du résultat du GROUP BY
  Si la ligne j satisfait à la clause HAVING
    Ajouter la ligne j au résultat final
  Fsi
  j++
Ftq

```

Ecrivez la requête SQL qui renvoie les organisations comptant plus d'un utilisateur.

```

S E L E C T USR_ORGANISATION,      C O U N T ( * )      F R O M
T _ U T I L I S A T E U R
G R O U P B Y USR_ORGANISATION
H A V I N G C O U N T ( * ) > = 2 ;

```



- La clause **HAVING** implique l'utilisation d'un **GROUP BY**.
- La clause **HAVING**, si elle existe, se place toujours **a p r è s** le **GROUP BY**.

Recherche multi-tables

Jointures

Le modèle relationnel mis en oeuvre par les SGBDR préconise le stockage des données sous la forme de tables. L'information se trouve donc "éclatée" entre les différentes tables qui composent une base de données. Les **jointures** permettent de mettre en relation deux (ou plusieurs) tables afin de combiner les données qu'elles contiennent.

SQL propose différents types de jointure. Nous allons étudier les plus courantes, qui s'effectuent en imposant l'égalité des valeurs des colonnes d'une table à une colonne d'une autre table.

Jointure interne

La syntaxe SQL92 de la jointure interne, qui est le type de jointure par défaut, est la suivante :

```

SELECT <liste d'expressions>
FROM table1 [ INNER ] JOIN table2 ON table1.champ1 = table2.champ2 ;

```

Pour mémoire, il est également possible d'écrire cette requête en employant l'ancienne syntaxe SQL86. Dans ce cas, le critère de jointure est écrit dans la clause **WHERE** :

```
SELECT <liste d'expressions>  
FROM table1, table2 WHERE table1.champ1 = table2.champ2 ;
```

Exemples :

- **SELECT** U.USR_ID, USR_NOM, NEW_ID, NEW_TITRE
FROM T_UTILISATEUR U **JOIN** T_NEWS N **ON** U.USR_ID = N.USR_ID;
- **SELECT** U.USR_ID, USR_NOM, NEW_ID, NEW_TITRE
FROM T_UTILISATEUR U, T_NEWS N
WHERE U.USR_ID = N.USR_ID;

Remarques :

- La création d'**alias** pour les noms de tables (ici **U** pour T_UTILISATEUR et **N** pour T_NEWS) permet de rendre la requête plus courte et plus lisible, à condition de choisir des alias cohérents par rapport aux noms des tables.
- Comme indiqué par les [] plus haut, le mot-clé **INNER** est optionnel.

La suite de ce chapitre utilise exclusivement la syntaxe SQL92. Bien entendu, on peut utiliser les autres clauses d'un **SELECT**, (**WHERE**, **GROUP BY**, etc) dans une requête avec jointure interne. On peut également joindre plusieurs tables entre elles.

Ecrivez la requête SQL qui renvoie les *news* (titre) postées par les utilisateurs (nom, prénom) sur les forums (nom).

```
SELECT U.USR_NOM, U.USR_PRENOM, N.NEW_TITRE,  
F.FRM_NOM  
FROM T_UTILISATEUR U JOIN T_NEWS N  
    ON U.USR_ID = N.USR_ID  
    JOIN T_FORUM F  
    ON N.FRM_ID = F.FRM_ID;
```

Jointure naturelle

La jointure naturelle est un cas particulier de jointure interne dans laquelle les noms de colonnes ne sont pas explicitement mentionnés. Le SGBDR réalise la jointure en se basant sur des colonnes de noms et de types identiques dans les deux tables et n'extrait que l'une des deux.

Exemple :

```
SELECT USR_ID, USR_NOM, NEW_ID, NEW_TITRE  
FROM T_UTILISATEUR NATURAL JOIN T_NEWS ;
```



La jointure naturelle suppose que le SGBDR puisse déterminer sans ambiguïté les colonnes pivots de la jointure. Elle peut poser problème en cas de renommage de champs dans la base.

Autojointure

L'autojointure se définit comme la jointure interne d'une table sur elle-même. Elle permet classiquement de représenter une hiérarchie (relation chef/subordonné) ou une arborescence (relation père/fils). L'utilisation d'alias est indispensable pour écrire une autojointure.

Ecrivez la requête SQL qui renvoie les *news* (titre) et leurs réponses (titre)

```
SELECT N1.NEW_TITRE AS QUESTION, N2.NEW_TITRE
AS REPONSE
FROM T_NEWS N1 JOIN T_NEWS N2 ON N1.NEW_ID =
N2.NEW_ID_PERE;
```

Jointure externe

Lorsqu'on utilise une jointure interne pour extraire les données de deux tables, seuls les enregistrements qui satisfont au critère de jointure sont renvoyés. Par exemple, supposons que nous souhaitions récupérer les utilisateurs et leurs messages postés. La requête associée pourrait s'écrire :

```
SELECT USR_NOM, USR_PRENOM, NEW_TITRE
FROM T_UTILISATEUR U JOIN T_NEWS N ON U.USR_ID = N.USR_ID ;
```

Les utilisateurs qui n'ont posté aucune *news* sont exclus de la réponse, puisqu'ils ne satisfont pas au critère `U.USR_ID = N.USR_ID`. Pour extraire tous les utilisateurs et les news de ceux qui en ont posté, il faut utiliser un autre type de jointure : la jointure **externe**. Voici sa syntaxe :

```
SELECT <liste d'expressions>
FROM table1 { LEFT | RIGHT } [ OUTER ] JOIN table2
ON table1.champ1 = table2.champ2 ;
```

Exemple :

```
SELECT USR_NOM, USR_PRENOM, NEW_TITRE
FROM T_UTILISATEUR U LEFT JOIN T_NEWS N ON U.USR_ID = N.USR_ID ;
```

Au moment de l'exécution, le SGBDR recherche tous les enregistrements qui satisfont au critère de jointure (ici `U.USR_ID = N.USR_ID`). Il ajoute ensuite toutes les lignes de la table désignée par le mot-clé **LEFT** (ici `T_UTILISATEUR`) qui n'ont pas été prises en compte par le critère précédent. Les mots-clés **LEFT** et **RIGHT** servent donc à désigner la table dont les enregistrements seront tous affichés, même ceux qui n'auraient pas été sélectionnées par une jointure interne.

Ecrivez la requête SQL qui calcule le nombre de news pour chaque utilisateur, *en incluant ceux qui n'ont rien posté*.

```
SELECT USR_NOM, USR_PRENOM, COUNT(NEW_ID)
FROM T_UTILISATEUR U LEFT JOIN T_NEWS N
ON U.USR_ID = N.USR_ID
GROUP BY USR_NOM, USR_PRENOM;
```

Requêtes imbriquées

Une requête imbriquée (ou sous-requête) est une requête SQL, le plus souvent un ordre **SELECT**, incluse entre parenthèses dans une autre requête. Le résultat d'une sous-requête servira dans le calcul du résultat de la requête de niveau supérieur.



Suivant le type de résultat qu'une sous-requête offre, on peut la placer dans différentes clauses de la requête principale.

Sous-requêtes simples

Une sous-requête est dite simple si elle s'exécute indépendamment de la requête qui l'inclus. Pour savoir où placer une sous-requête, il faut analyser les différents types de résultat produits par un ordre **SELECT**. Nous allons nous intéresser aux sous-requêtes produisant :

- Une valeur unique.
- Une liste de valeurs (une colonne).

Exemple :

```
SELECT FIC_NOM, FIC_TAILLE_KO FROM T_FICHIER
WHERE FIC_TAILLE_KO = (SELECT MAX(FIC_TAILLE_KO) FROM T_FICHIER);
```

Que renvoie la requête ci-dessus ?

Le nom et la taille du plus volumineux fichier posté avec un news.

La sous-requête utilisée ici renvoie une seule valeur, ce qui est toujours le cas quand on utilise les opérateurs statistiques (**COUNT**, **SUM**...) *sans regroupement*. Elle peut donc être imbriquée et son résultat comparé à la valeur d'une colonne.

Les sous-requêtes produisant une liste de valeurs s'utilisent généralement avec les opérateurs **IN**, **ALL** et **ANY**. Les deux derniers permettent des comparaisons d'inégalité.

Une expression comportant **ALL** vaut vrai si la comparaison est vraie pour tous les éléments de la liste :

- 123 < **ALL** (121, 122) => **faux**
- 123 < **ALL** (122, 123) => **faux**
- 123 < **ALL** (123, 124) => **faux**
- 123 < **ALL** (124, 125) => **vrai**

Une expression comportant **ANY** vaut vrai si la comparaison est vraie pour au moins un élément de la liste :

- 123 < **ANY** (121, 122) => **faux**
- 123 < **ANY** (122, 123) => **faux**
- 123 < **ANY** (123, 124) => **vrai**
- 123 < **ANY** (124, 125) => **vrai**

Réécrivez la requête renvoyant le nom et la taille du plus volumineux fichier posté avec une *news* en utilisant ALL ou ANY.

```
SELECT FIC_NOM, FIC_TAILLE_KO FROM T_FICHIER
WHERE FIC_TAILLE_KO >= ALL
(SELECT FIC_TAILLE_KO FROM T_FICHIER) ;
```

Ecrivez la requête renvoyant le nom et la taille de tous les fichiers, sauf le plus volumineux.

```
SELECT FIC_NOM, FIC_TAILLE_KO FROM T_FICHIER
WHERE FIC_TAILLE_KO < ANY
(SELECT FIC_TAILLE_KO FROM T_FICHIER) ;
```

Sous-requêtes corrélées

Une sous-requête corrélée est une sous-requête qui s'exécute pour chaque ligne de la requête principale et non une fois pour toute. Pour arriver à un tel résultat, il suffit de faire varier une condition en rappelant dans la sous-requête la valeur d'une colonne de la requête principale. Une sous-requête corrélée réalise l'équivalent d'une double boucle imbriquée.

Par exemple, cherchons à trouver les utilisateurs qui ont le même prénom. Une première solution serait :

```
SELECT USR_NOM, USR_PRENOM FROM T_UTILISATEUR
WHERE USR_PRENOM IN (SELECT USR_PRENOM FROM T_UTILISATEUR);
```

Répond-elle au besoin exprimé ?

Non, elle renvoie tous les prénoms : un utilisateur porte le même prénom que lui-même !

Pour obtenir un résultat correct, il faut *corrél*er la sous-requête à la requête principale afin exclure de la sous-requête l'utilisateur courant de la requête principale :

```
SELECT USR_NOM, USR_PRENOM FROM T_UTILISATEUR U
WHERE USR_PRENOM IN (SELECT USR_PRENOM FROM T_UTILISATEUR
WHERE USR_ID <> U.USR_ID);
```

Remarque : comme pour une autojointure, il est nécessaire d'ajouter des alias pour employer des requêtes corrélées.

L'opérateur **EXISTS** est souvent employé avec des sous-requêtes corrélées. Il renvoie vrai si la sous-requête produit un résultat.

Exemple :

```
SELECT USR_NOM, USR_PRENOM FROM T_UTILISATEUR U1
WHERE EXISTS (SELECT * FROM T_UTILISATEUR U2
WHERE U2.USR_ID <> U1.USR_ID
AND U2.USR_PRENOM = U1.USR_PRENOM);
```

Que renvoie la requête ci-dessus ?

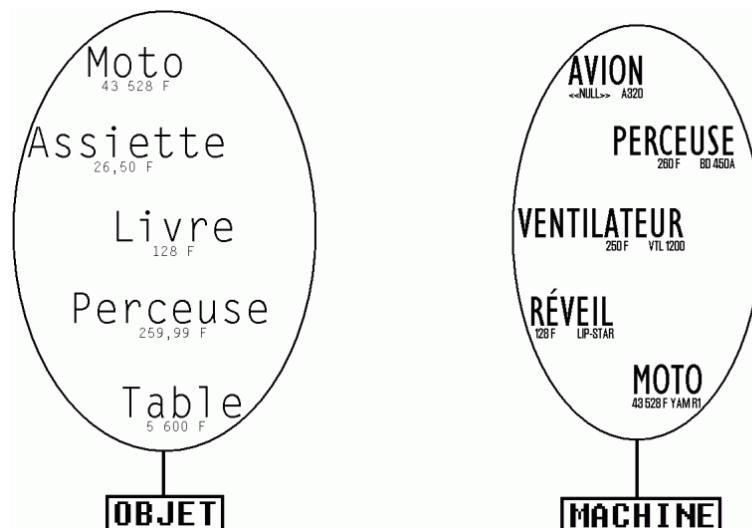
Les utilisateurs qui ont le même prénom.

Opérateurs ensemblistes

Les opérateurs ensemblistes du langage SQL permettent de travailler sur les ensembles représentés par des tables. Ils s'utilisent généralement pour rapprocher des données semblables issues de tables hétérogènes. Ils ne sont pas supportés par tous les SGBDR, même récents.

Dans la suite de ce paragraphe, nous nous baserons sur le schéma relationnel ci-dessous (source : F. Brouard) :

T_OBJET(OBJ_NOM, OBJ_PRIX)
T_MACHINE(MAC_NOM, MAC_PRIX, MAC_REF)



Union

On souhaite obtenir les éléments qui sont soit des objets, soit des machines, autrement dit réaliser l'**union** des deux ensembles :

```
SELECT OBJ_NOM FROM T_OBJET  
UNION SELECT MAC_NOM FROM T_MACHINE;
```

L'opérateur **UNION** renvoie les éléments appartenant soit à l'un, soit à l'autre des ensembles.

Remarques :

- Pour qu'une requête **UNION** fonctionne, les deux ordres **SELECT** doivent produire un même nombre de colonne de types compatibles.
- Les noms des colonnes de résultats sont ceux du premier ordre **SELECT**.
- L'opérateur **UNION** est commutatif (*select1 UNION select2* équivaut à *select2 UNION select1*).

Intersection

L'intersection est la récupération des données communes à deux ensembles. Elle se traduit en SQL de manière similaire à l'union grâce à l'opérateur **INTERSECT**.

Ecrivez la requête qui renvoie les éléments qui sont à la fois des objets et des machines.

```
SELECT OBJ_NOM FROM T_OBJET
INTERSECT
SELECT MAC_NOM FROM T_MACHINE;
```

Les remarques sur l'opérateur **UNION** (nécessité de types compatibles, commutativité) sont également valables pour **INTERSECT**.

Différence

La différence entre deux ensembles (éléments présents dans l'un mais pas dans l'autre) s'obtient à l'aide de l'opérateur **EXCEPT** (appelé **MINUS** dans certains SGBDR).

Exemple :

```
SELECT OBJ_NOM FROM T_OBJET
EXCEPT
SELECT MAC_NOM FROM T_MACHINE;
```

Que renvoie la requête ci-dessous ?

```
Les objets qui ne sont pas des machines.
```



Contrairement à **UNION** et **INTERSECT**, **EXCEPT** (ou **MINUS**) n'est pas commutatif :
<select1> **EXCEPT** <select2> n'est pas équivalent à <select2> **EXCEPT** <select1>.

Application

On souhaite récupérer les identifiants des utilisateurs n'ayant posté aucune *news*.

Ecrivez trois requêtes SQL qui répondent à ce besoin.

```
Avec une sous-requête et un NOT IN :
SELECT USR_ID FROM T_UTILISATEUR
WHERE USR_ID NOT IN
      (SELECT USR_ID FROM T_NEWS);
```

```
Avec une sous-requête corrélée et un EXISTS :
SELECT USR_ID FROM T_UTILISATEUR U
WHERE NOT EXISTS
```

```
(SELECT * FROM T_NEWS N
WHERE N.USR_ID = U.USR_ID);
```

Avec une différence ensembliste :

```
SELECT USR_ID FROM T_UTILISATEUR
EXCEPT
SELECT USR_ID FROM T_NEWS;
```

Possible : <> ALL, LEFT JOIN + WHERE IS NULL

Annexe : base de données d'exemple

Cette base (adaptée d'une source de F. Brouard) décrit la gestion d'un forum de *news*. Chaque utilisateur peut poster une *news* contenant éventuellement des fichiers joints. Plusieurs réponses (sous la forme d'une *news*) peuvent concerner une *news*. Une *news* est abritée dans un forum.

