SQL Avancé
Les Anti-jointures

CLIENTS (idClient, nomClient, prenomClient, villeClient)

C1	Terrieur	Alain	Montpellier
C2	Terrieur	Alex	Nimes
C3	Bricot	Judas	Montpellier

ACHATS (idAchat, montantAchat, idClient#)

A1	100	C1
A2	150	C1
A3	90	NULL
A4	110	C3

Cas simple : Quels sont les achats qui n'ont pas de client ?

On cherche les achats qui disparaitraient si on faisait une jointure interne entre ACHATS et CLIENTS. Il s'agit donc d'une anti-jointure. Un achat ne concerne qu'un seul client donc idClient est dans la table Achats. Dans ce cas là, l'anti-jointure est facile à faire. Il suffit de tester si idClient a une valeur `NULL` dans la table ACHATS :

```
SELECT idAchat
FROM Achats
WHERE idClient IS NULL;
```

Cas plus complexe : Afficher le nom des clients qui n'ont pas effectué d'achat.

Résultat attendu :

```
nomClient
----------
Terrieur
```

Ici aussi, il s'agit d'une anti-jointure. On veut les clients qui disparaitraient si on faisait une jointure interne entre CLIENTS et ACHATS. Mais cette fois, il n'est pas possible de faire la requête avec un simple `IS NULL`. Il faut trouver les clients qui n'ont pas effectué d'achat, c'est à dire les clients de la table CLIENTS qui ne se trouvent pas dans la table ACHATS.
Pour réaliser cette anti-jointure, nous allons voir ici trois solutions : utiliser l'opérateur ensembliste **`MINUS`**, réaliser une requête imbriquée avec le prédicat **`NOT IN`** et enfin utiliser le prédicat **`NOT EXISTS`** avec une requête imbriquée corrélée (on pourrait aussi réaliser une anti-jointure en utilisant une jointure externe).

1. Opérateur ensembliste MINUS

On fait ici la différence entre tous les clients de la table clients et les clients qui ont fait des achats (de la table ACHATS).

Pour obtenir le bon résultat, il faut appliquer le `MINUS` sur idClient.

```
SELECT nomClient
FROM Clients
WHERE idClient IN (SELECT idClient
                   FROM Clients
                   MINUS
                   SELECT idClient
                   FROM Achats);
```

```
C1
C2
C3
MINUS
C1
C1
NULL
C3
```

Attention : ne pas appliquer un MINUS sur des attributs pouvant contenir des homonymes

```
SELECT nomClient
FROM Clients
MINUS
SELECT nomClient
FROM Clients c
JOIN Achats a ON c.idClient = a.idClient;
```

```
Terrieur
Terrieur
Bricot
MINUS
Terrieur
Terrieur
Bricot
```

Cette requête ne retourne pas le bon résultat. Elle ne retourne aucun client alors qu'on attend le client Terrieur (C2) car on applique le `MINUS` sur le nomClient. Or il existe deux clients qui s'appellent Terrieur, le client C1 qui a effectué des achats et le client C2 qui n'a effectué aucun achat.

2. Requête imbriquée avec NOT IN

On va récupérer ici tous les clients de la table CLIENTS dont le numéro ne se trouve pas dans la table ACHATS.

```
SELECT idClient, nomClient
FROM Clients
WHERE idClient NOT IN (SELECT idClient
                       FROM Achats);
```

Attention : Il se peut que le `NOT IN` ne fonctionne pas dans certains cas ! Dans notre exemple, s'il n'y avait pas l'achat A3 (qui n'a pas de client) dans la table ACHATS, cette requête fonctionnerait. Mais avec l'achat A3 dans la table ACHATS, la requête imbriquée va retourner une valeur `NULL` et la requête principale ne marchera pas.

> Comme pour le `IN`, la condition du `WHERE` est remplacée par le résultat de la sous-requête. Ici étant donné qu'on utilise un `NOT IN`, la condition de la sous requête est remplacée par plusieurs tests d'inégalités séparés par des `AND`.
>
> ```
> SELECT idClient, nomClient
> FROM Clients
> WHERE idClient != 'C1' AND idClient != 'C1' AND idClient != 'C3'
>       AND idClient != NULL ;
> ```
>
> Le problème dans la requête précédente est que la condition du `WHERE` ne retourne jamais vrai à cause de `idClient != NULL`.

Pour contourner le problème, il faut ici éliminer les valeurs `NULL` dans la requête imbriquée :

```
SELECT idClient, nomClient
FROM Clients
WHERE idClient NOT IN (SELECT idClient
                       FROM Achats
                       WHERE idClient IS NOT NULL);
```

L'utilisation du `NOT IN` est donc dangereuse. Il ne marchera pas si la requête imbriquée peut retourner des `NULL`. De plus, comme vous le verrez en deuxième année, avec beaucoup de SGBD les requêtes qui utilisent un `NOT IN` ne sont pas très performantes.
Pour ces deux raisons, il est conseillé de plutôt utiliser le `NOT EXISTS`.

3. Le prédicat NOT EXISTS

On va chercher ici les clients de la table CLIENTS pour lesquels il n'existe pas de jointure avec la table ACHATS.

Le `NOT EXISTS` fonctionne comme le `EXISTS` : comme la requête imbriquée est corrélée, elle est exécutée pour chaque ligne de la requête principale. Mais contrairement à l'`EXISTS`, ici on retourne les clients dont la requête imbriquée ne retourne pas de lignes. Si la requête imbriquée ne contient pas des lignes, la réponse est `TRUE`, sinon la réponse est `FALSE`.

```
SELECT nomClient
FROM Clients c
WHERE NOT EXISTS
          (SELECT *
           FROM Achats a
           WHERE a.idClient = c.idClient);
```

```
C1    Terrieur
            FALSE
C2    Terrieur
            TRUE
C3    Bricot
            FALSE
C4    Nanas
            FALSE
```

On rappelle qu'avec le `NOT EXISTS` (comme avec le `EXISTS`) il faut absolument que la requête imbriquée soit corrélée avec la requête principale.
Il est conseillé de mettre `*` dans le `SELECT` de la requête imbriquée et de ne pas mettre de nom d'attributs. En effet le SGBD va simplement vérifier si la requête imbriquée retourne ou non des lignes ; les colonnes de ces lignes importe peu.