

# Dossier 1

## Compléments sur les requêtes SQL

---

### 1 Sous-requêtes multilignes : les quantificateurs ALL et ANY

Les quantificateurs ALL et ANY permettent, avec une requête imbriquée, d'utiliser des opérateurs de comparaison (=, >=, >, <=, <, >>) même si la sous-requête retourne plusieurs lignes.

Exemple :

*PERSONNES (codePersonne, nomPersonne, prenomPersonne, agePersonne, villePersonne)*

Question : R1 : Personnes qui sont plus vieilles que toutes les personnes qui vivent à 'Montpellier.'

```
SELECT *
FROM Personnes
WHERE agePersonne > ALL (SELECT agePersonne
                           FROM Personnes
                           WHERE villePersonne = 'Montpellier') ;
```

Question : R2 : Personnes qui sont plus vieilles qu'une des personnes qui vivent à 'Montpellier.'

```
SELECT *
FROM Personnes
WHERE agePersonne > ANY (SELECT agePersonne
                           FROM Personnes
                           WHERE villePersonne = 'Montpellier') ;
```

Il est à noter qu'on peut toujours se passer de ces quantificateurs et utiliser à la place les fonctions MIN, MAX ou le prédictat IN. Et comme nous le verrons plus tard, les quantificateurs ALL et ANY sont à utiliser avec précaution lorsqu'on a affaire à des valeurs NULL.

> ALL	> MAX()
> ANY	> MIN()
< ALL	< MIN()
< ANY	< MAX()
= ALL	= MIN() AND = MAX()
= ANY	IN

Par exemple, il est possible de réaliser la requête 1 avec un '> MAX' (et la 2 avec un '> MIN')

```
SELECT *
FROM Personnes
WHERE agePersonne > (SELECT MAX(agePersonne)
                      FROM Personnes
                      WHERE villePersonne = 'Montpellier') ;
```

---

### 2 Les expressions de table (CTE)

Une expression de table (ou CTE : Common Table Expression) est une sous-requête SELECT temporaire exprimée à l'intérieur d'une requête principale, et dont le résultat est nommé puis utilisé comme une table par la requête principale. Il est inutile d'utiliser des expressions de table pour faire des requêtes simples, mais dans le cas où on a affaire à des requêtes complexes, elles peuvent être utiles pour simplifier le code. On peut également utiliser une vue pour cela, mais contrairement à une vue qui est un objet persistant de la base de données, l'expression de table est créée temporairement par la requête principale et n'est pas accessible à l'extérieur de celle-ci. Nous allons voir ici deux façons de créer une expression de table : la clause SELECT dans le FROM et l'expression WITH.

#### 2.1 La sous requête issue d'un SELECT dans le FROM (ou dans le JOIN)

Depuis SQL2, il est possible de construire dynamiquement une table dans la clause FROM de la requête principale. En effet, le FROM manipule des colonnes et des lignes qui peuvent être matérialisées par une table, une vue ou bien le résultat d'une sous-requête.

exemple : R3 : La ville pour laquelle la moyenne d'âge des personnes est la plus élevée.

Sous Oracle, il n'est pas indispensable d'utiliser une expression de table. On peut écrire la requête comme suit :

```
SELECT villePersonne
FROM Personnes
GROUP BY villePersonne
HAVING AVG(agePersonne) = (SELECT MAX(AVG(agePersonne))
                            FROM Personnes
                            GROUP BY villePersonne);
```

Mais en théorie, dans la norme SQL, il n'est pas possible d'imbriquer plusieurs fonctions d'ensemble. Et comme vous l'avez vu l'an dernier cela n'est pas possible non plus avec PostgreSQL. Pour remédier à cela on pourrait utiliser un quantificateur ALL, ou bien utiliser une expression de table dans la requête imbriquée.

```
SELECT villePersonne
FROM Personnes
GROUP BY villePersonne
HAVING AVG(agePersonne) = (SELECT MAX(moyenneAge)
                             FROM (SELECT AVG(agePersonne) AS moyenneAge
                                   FROM Personnes
                                   GROUP BY villePersonne) cte);
```

On pourrait aussi utiliser une expression de table pour remplacer le HAVING de la requête externe par un WHERE.

```
SELECT ville
FROM (SELECT villePersonne AS ville, AVG(agePersonne) AS ageMoyen
      FROM Personnes
      GROUP BY villePersonne) cte
WHERE ageMoyen = (SELECT MAX(moyenneAge)
                  FROM (SELECT AVG(agePersonne) AS moyenneAge
                        FROM Personnes
                        GROUP BY villePersonne) cte);
```

Toutefois, lorsqu'on fait des expressions de tables avec des SELECT dans le FROM il n'est possible de réutiliser la même expression de table dans la requête externe et dans la requête imbriquée. Nous verrons dans le paragraphe suivant qu'il est possible de remédier à cela en utilisant des expressions de table WITH.

Il est possible d'utiliser plusieurs expressions de table dans une même requête et les joindre avec des jointures internes, des jointures externes ou bien des produits cartésiens.

Question : R4 : Pourcentage de clients qui habitent à 'Montpellier.'

```
SELECT cteMontp.nbMontp / cteTot.nbTot * 100 AS pourcentageMontpelliens
FROM (SELECT COUNT(*) AS nbMontp
      FROM Personnes
      WHERE villePersonne = 'Montpellier') cteMontp
CROSS JOIN (SELECT COUNT(*) AS nbTot
            FROM Personnes) cteTot;
```

Remarque : Une sous-requête dans le FROM peut retourner plusieurs lignes et plusieurs colonnes.

Il est également possible de faire des sous-requêtes à l'intérieur d'un SELECT. Mais il faut absolument que cette sous-requête retourne une valeur scalaire (une seule ligne et un seul colonne). Toutefois, ce genre de sous-requête scalaire dans le SELECT dégrade rapidement les performances notamment lorsque les sous requêtes sont corrélées.

exemple : R5 : Pour chaque personne, afficher le nom, prénom et différence avec la moyenne d'âge.

```
SELECT nomPersonne, prenomPersonne,
       ABS(agePersonne - (SELECT AVG(agePersonne) FROM Personnes))
FROM Personnes
```

## 2.2 Expression WITH

L'expression WITH permet d'écrire une requête dont le résultat sera utilisé comme une table dans la requête principale qui suit l'expression. L'expression WITH est plus puissante que le SELECT dans le FROM car la (ou les) table issue de l'expression de table peut être utilisée non seulement dans la requête principale mais aussi dans les sous-requêtes imbriquées de la requête principale (ce qui n'est pas possible avec le SELECT dans le FROM). Même si l'expression WITH est un standard de l'ISO/ANSI certains SGBD ne l'implémentent pas encore.

exemple : R3 : La ville pour laquelle la moyenne d'âge des personnes est la plus élevée.

```
WITH AgeMoyenParVille AS
      (SELECT villePersonne AS ville, AVG(agePersonne) AS ageMoyen
       FROM Personnes
       GROUP BY villePersonne)
SELECT ville
FROM AgeMoyenParVille
WHERE ageMoyen = (SELECT MAX(ageMoyen)
                  FROM AgeMoyenParVille);
```

Il est aussi possible de déclarer plusieurs expressions de table dans le WITH. Il est à noter qu'une expression de table pourrait éventuellement utiliser dans le FROM de sa requête une autre expression de table déclarée préalablement dans le même WITH.

exemple : R4 : Pourcentage de clients qui habitent à ‘Montpellier’.

```
WITH cteMontp AS
    (SELECT COUNT(*) AS nbMontp
     FROM Personnes
     WHERE villePersonne = 'Montpellier'),
cteTot AS
    (SELECT COUNT(*) AS nbTot
     FROM Personnes)
SELECT cteMontp.nbMontp / cteTot.nbTot * 100 AS pourcentageMontpelliens
FROM cteMontp
CROSS JOIN cteTot;
```

## 3 Une nouvelle façon de faire la division

---

### 3.1 Rappel sur la division

En première année nous avons vu deux façons de faire la division. La méthode ‘Force Brute’ ou il suffit de compter. Et une méthode plus ingénieuse (et moins connue en pratique) avec un NOT EXISTS et un MINUS.

Exemple :

PERSONNES (codePersonne, nomPersonne, prenomPersonne)  
VILLES (codeVille, nomVille, departementVille)  
AIMER (codePersonne #, codeVille #)

Question : R6 : On cherche le code et le nom des personnes qui aiment **toutes** les villes ...

#### 1<sup>er</sup> Méthode : La plus simple (solution ‘force brute’)

Chercher les personnes qui aiment un nombre de villes égal au nombre total de villes (par exemple si j’aime 3 villes et qu’il n’y a que 3 villes dans la base de données eh bien j’aime toutes les villes de la base de données).

```
SELECT p.codePersonne, nomPersonne
FROM Personnes p
JOIN Aimer a ON p.codePersonne = a.codePersonne
GROUP BY p.codePersonne, nomPersonne
HAVING COUNT(codeVille) = (SELECT COUNT(codeVille)
                           FROM Villes) ;
```

Remarque : ici, on pourrait remplacer COUNT(codeVille) par COUNT(\*) car codeVille ne peut pas être NULL (il fait partie de la clé primaire)

#### 2<sup>nd</sup> Méthode : Ingénieuse

Chercher les personnes pour lesquelles la différence entre les villes de la base de données et les villes aimées par la personne est nulle.

En d’autres termes, chercher les personnes pour lesquelles la différence entre les villes de la base de données et les villes aimées par la personne n’existe pas (NOT EXISTS).

```
SELECT codePersonne, nomPersonne
FROM Personnes p
WHERE NOT EXISTS (SELECT codeVille
                   FROM Villes
                   MINUS
                   SELECT codeVille
                   FROM Aimer a
                   WHERE a.codePersonne = p.codePersonne) ;
```

#### 3.2 Division élégante avec double NOT EXISTS

Nous allons voir cette année la division avec un double NOT EXISTS qui a été popularisée par Christopher J. Date. Cette division, qui peut paraître compliquée de prime abord, a l’avantage de toujours rester de difficulté constante, même pour les divisions plus complexes.

#### 3<sup>rd</sup> Méthode : Elégante (même si elle peut paraître compliquée de prime abord)

Chercher les personnes pour lesquelles il n’existe que des villes qu’ils aiment.

En d’autres termes, quelles sont les personnes pour lesquelles il n’existe pas de villes qu’ils n’aiment pas.

Ou encore, quelles sont les personnes pour lesquelles il n’existe pas de villes pour lesquelles il n’existe pas d’amour ... Ceci est bien compliqué mais peut être réalisé à l’aide d’un double NOT EXISTS.

```

SELECT codePersonne, nomPersonne
FROM Personnes p1
WHERE NOT EXISTS (SELECT *
                  FROM Villes v2
                  WHERE NOT EXISTS (SELECT *
                                     FROM Aimer a3
                                     WHERE a3.codePersonne = p1.codePersonne
                                         AND a3.codeVille = v2.codeVille)) ;

```

L'avantage de cette méthode est que, une fois qu'on a bien compris comment elle fonctionne, sa structure ne change jamais. Si on est amené à faire une division plus complexe (de type 2, 3 ou 4 – voir cours de 1<sup>ère</sup> année), seule la première requête imbriquée (au centre) change. La requête externe (en haut à gauche) ne change jamais. Et la plus imbriquée (en bas à droite) non plus.

- R7 : Le code et le nom des personnes qui aiment toutes les villes de l'Hérault.

On cherche les personnes pour lesquelles il n'existe pas de villes **de l'Hérault** qu'ils n'aiment pas.

```

SELECT codePersonne, nomPersonne
FROM Personnes p1
WHERE NOT EXISTS (SELECT *
                  FROM Villes v2
                  WHERE departementVille = 'Hérault'
                  AND NOT EXISTS (SELECT *
                                   FROM Aimer a3
                                   WHERE a3.codePersonne = p1.codePersonne
                                       AND a3.codeVille = v2.codeVille)) ;

```

- R8 : Le code et le nom des personnes qui aiment toutes les villes aimées par 'Xavier Palléja'.

Cette division de type 4 est relativement simple avec cette méthode. On cherche les personnes pour lesquelles il n'existe pas de villes **aimées par Xavier Palléja** qu'ils n'aiment pas.

```

SELECT codePersonne, nomPersonne
FROM Personnes p1
WHERE NOT EXISTS (SELECT *
                  FROM Personnes p2
                  JOIN Aimer a2 ON p2.codePersonne = a2.codePersonne
                  WHERE nomPersonne = 'Palléja'
                      AND prenomPersonne = 'Xavier'
                  AND NOT EXISTS (SELECT *
                                  FROM Aimer a3
                                  WHERE a3.codePersonne = p1.codePersonne
                                      AND a3.codeVille = a2.codeVille)) ;

```

Remarque : Pour rappel, avec la solution 'force brute' cette division de type 4 est beaucoup plus complexe. En effet, pour chaque personne, il faut compter le nombre de villes aimées – parmi celles qui sont également aimées par Xavier Palléja. Ensuite on ne garde que les personnes pour qui ce nombre est égal au nombre de villes aimées par Xavier Palléja. Cela donne la solution suivante :

```

SELECT p.codePersonne, nomPersonne
FROM Personnes p
JOIN Aimer a ON p.codePersonne = a.codePersonne
WHERE codeVille IN (SELECT codeVille
                    FROM Personnes p
                    JOIN Aimer a ON p.codePersonne = a.codePersonne
                    WHERE nomPersonne = 'Palleja'
                        AND prenomPersonne = 'Xavier')
GROUP BY p.codePersonne, nomPersonne
HAVING COUNT(codeVille) = (SELECT COUNT(codeVille)
                            FROM Personnes p
                            JOIN Aimer a ON p.codePersonne = a.codePersonne
                            WHERE nomPersonne = 'Palleja'
                                AND prenomPersonne = 'Xavier');

```

## 4 Différents types de jointures

---

### 4.1 Les inéquijoointures :

En général les jointures sont réalisées entre des clés primaires et des clés étrangères. Dans ces cas-là, la condition de la jointure est toujours l'égalité (=) afin de s'assurer que la valeur de la clé primaire d'une table est égale à la valeur de la clé étrangère d'une autre table. On parle alors d'équijointure. Toutefois dans certains cas on peut vouloir utiliser une condition d'inégalité pour réaliser la jointure. On parle d'inéquijointure. La jointure se fait alors avec les opérateurs suivants : >, <, >=, <=, <>, BETWEEN, LIKE ... et porte rarement sur des clés primaires.

Exemple :

*EMPLOYES (codeEmploye, nomEmploye, prenomEmploye, ageEmploye)  
CLIENTS (codeClient, nomClient, prenomClient, ageClient)*

Question : R9 : Le code et le nom des employés plus vieux que le client n°1.

```
SELECT codeEmploye, nomEmploye
FROM Employes
JOIN Clients ON ageEmploye > ageClient
WHERE codeClient = '1' ;
```

## 4.2 Les jointures naturelles

Lorsqu'on souhaite réaliser une équijointure et que les attributs sur lesquels porte la jointure ont les mêmes noms dans les deux tables, il est possible de faire une jointure naturelle (NATURAL JOIN) ou d'utiliser un USING. Cette façon de faire évite d'indiquer explicitement la condition de la jointure, et réalise automatiquement une projection afin d'éliminer les colonnes redondantes (qui ont le même nom). Il n'est donc pas utile d'utiliser des alias de table pour réaliser la jointure, même si on souhaite utiliser la colonne sur laquelle porte la jointure dans le SELECT.

Exemple :

PERSONNES (codePersonne, nomPersonne, prenomPersonne, codeEntreprise)				ENTREPRISES (codeEntreprise, nomEntreprise)	
P1	Némard	Jean	E1	E1	Cap Gemini
P2	Palleja	Nathalie		E2	Fnac

Question : R10 : Pour toutes les personnes qui ont une entreprise, afficher toutes les informations sur la personne et sur son entreprise.

```
SELECT *
FROM Personnes
NATURAL JOIN Entreprises ;
```

```
SELECT *
FROM Personnes
JOIN Entreprises USING (codeEntreprise) ;
```

## 4.3 Les jointures externes

Elles permettent d'extraire des données qui ne répondent pas aux critères de jointure. Il existe trois sortes de jointures externes : les jointures externes *gauches*, *droites* et *complètes*. Les jointures externes complètes sont rarement utiles (en général une gauche ou une droite suffit).

Dans l'exemple précédent, avec une jointure interne ou une jointure naturelle, on aura uniquement dans le résultat la personne P1 avec l'entreprise E1. La personne P2 qui n'a pas d'entreprise, et l'entreprise E2 qui n'a pas de salarié vont disparaître du résultat.

### 4.3.1 Jointure externe gauche

R11 : Le nom, le prénom et l'entreprise de toutes les personnes, même les personnes qui n'ont pas d'entreprise (c'est à dire même la personne P2).

```
SELECT nomPersonne, prenomPersonne, nomEntreprise
FROM Personnes p
LEFT OUTER JOIN Entreprises e ON p.codeEntreprise = e.codeEntreprise ;
```

### 4.3.2 Jointure externe droite

R12 : Le nom, le prénom et l'entreprise de toutes les personnes, même les entreprises qui n'ont pas de salariés (c'est à dire même l'entreprise E2).

```
SELECT nomPersonne, prenomPersonne, nomEntreprise
FROM Personnes p
RIGHT OUTER JOIN Entreprises e ON p.codeEntreprise = e.codeEntreprise ;
```

Cette requête peut également être réalisée à l'aide d'une jointure externe gauche :

```
SELECT nomPersonne, prenomPersonne, nomEntreprise
FROM Entreprises e
LEFT OUTER JOIN Personnes p ON p.codeEntreprise = e.codeEntreprise ;
```

### 4.3.3 Jointure externe complète

R13 : Le nom, le prénom et l'entreprise de toutes les personnes, même les personnes qui n'ont pas d'entreprise et même les entreprises qui n'ont pas de salariés (i.e. même P2 et E2).

```
SELECT nomPersonne, prenomPersonne, nomEntreprise
FROM Personnes p
FULL OUTER JOIN Entreprises e ON p.codeEntreprise = e.codeEntreprise ;
```

Remarque : On peut réaliser des jointures externes naturelles avec NATURAL LEFT/RIGHT/FULL OUTER JOIN

Remarque : On peut également utiliser la jointure externe pour faire des antijointures (à la place d'un NOT EXISTS, d'un NOT IN ou d'un MINUS)

R14 : Le nom des entreprises qui n'ont pas d'employé

```
SELECT nomEntreprise
FROM Entreprises e
LEFT OUTER JOIN Personnes p ON p.codeEntreprise = e.codeEntreprise
WHERE codePersonne IS NULL;
```

## 5 Requêtes récursives

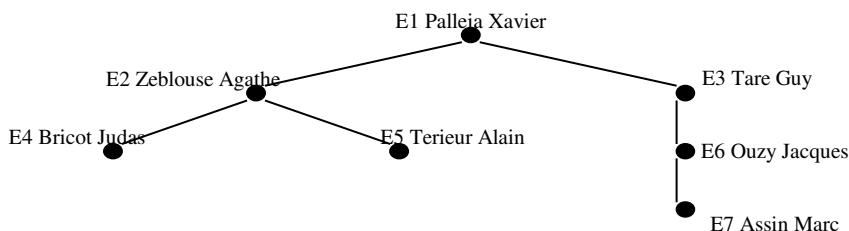
Les requêtes récursives (parfois appelées requêtes hiérarchiques) permettent d'extraire des données provenant d'une structure arborescente. Pendant de nombreuses années, pour réaliser ce type de requêtes, Oracle ne permettait pas d'utiliser la syntaxe préconisée par la norme SQL:99, et proposait uniquement l'instruction **CONNECT BY PRIOR**. Mais depuis sa version 11gR2, Oracle respecte la norme SQL et permet également de réaliser des requêtes récursives à l'aide des expressions de table récursives **WITH**.

Exemple :

**EMPLOYES** (*codeEmploye*, *nomEmploye*, *prenomEmploye*, *codeEmployeChef#*)

EMPLOYES(*codeEmploye*, *nomEmploye*, *prenomEmploye*, *codeEmployeChef#*)

E1	Palleja	Xavier	/
E2	Zeblouse	Agathe	E1
E3	Tare	Guy	E1
E4	Bricot	Judas	E2
E5	Terieur	Alain	E2
E6	Ouzy	Jacques	E3
E7	Assin	Marc	E6



### 5.1 L'instruction CONNECT BY PRIOR spécifique à Oracle

L'instruction Oracle **CONNECT BY PRIOR** permet de faire un parcours **en profondeur**. Le point de départ de la recherche, qui n'est pas forcément la racine de l'arbre, est spécifié avec **START WITH**. Si cette directive est omise, le parcours est réalisé à partir de tous les noeuds de l'arbre. La clause **CONNECT BY PRIOR** permet d'indiquer s'il s'agit d'une descente ou d'une remontée.

Il est à noter que s'il y a un cycle dans la recherche récursive (graphe avec circuit), il est indispensable d'indiquer le mot clé **NOCYCLE** juste après le **CONNECT BY**.

Question : R15 : Le code et le nom de tous les subordonnés de l'employé n°1.

```
SELECT codeEmploye, nomEmploye
FROM Employes
START WITH codeEmploye = 'E1'          -- on part de l'employé n°1
CONNECT BY PRIOR codeEmploye = codeEmployeChef; -- et on réalise une descente
```

Il est également possible d'obtenir le niveau du noeud dans la hiérarchie (à partir du noeud de départ de la recherche qui est numéroté à 1) grâce au mot clé **LEVEL**.

```
SELECT LEVEL, codeEmploye, nomEmploye
FROM Employes
START WITH codeEmploye = 'E1'
CONNECT BY PRIOR codeEmploye = codeEmployeChef;
```

On peut ne pas afficher certains noeuds grâce à une clause **WHERE** (avant le **START WITH**). Mais le noeud sera quand même traversé par le parcours. Si on souhaite réellement élaguer l'arbre d'une branche, il faut utiliser un **AND** dans la clause **CONNECT BY PRIOR**.

Question : R16 : Le code et le nom de tous les subordonnés de l'employé n°1 à l'exception de l'employé n°3.

```
SELECT codeEmploye, nomEmploye
FROM Employes
WHERE codeEmploye <> 'E3'
START WITH codeEmploye = 'E1'
CONNECT BY PRIOR codeEmploye = codeEmployeChef ;
```

Question : R17 : Le code et le nom de tous les subordonnés de l'employé n°1 à l'exception de l'employé n°3 et des subalternes de celui-ci.

```
SELECT codeEmploye, nomEmploye
FROM Employes
START WITH codeEmploye = 'E1'
CONNECT BY PRIOR codeEmploye = codeEmployeChef
AND codeEmploye <> 'E3' ;
```

Enfin, il est possible de parcourir l'arbre de bas en haut. Pour cela il faut inverser le **PRIOR** dans les colonnes de jointure de la clause **CONNECT BY**.

Question : R18 : Le code et le nom de tous les supérieurs de l'employé n°7.

```
SELECT codeEmploye, nomEmploye
FROM Employes
START WITH codeEmploye = 'E7'
CONNECT BY PRIOR codeEmployeChef = codeEmploye ;
-- ou CONNECT BY codeEmploye = PRIOR codeEmployeChef
```

## 5.2 Requêtes récursives avec l'expression de table WITH de la norme SQL

Depuis la version 11gR2, il est possible de respecter la syntaxe préconisée par la norme SQL en utilisant une expression de table récursive **WITH**. Une expression de table récursive se compose de deux requêtes : une première requête (dite *anchor*) qui indique le nœud de départ de la recherche récursive ; et une seconde requête (dite *member*) qui assure la récursivité en référençant obligatoirement (une et une seule fois) l'expression de table dans laquelle elle se trouve. Ces deux requêtes doivent être reliées par l'opérateur ensembliste **UNION ALL**. Par défaut, le parcours est réalisé en largeur (mais il est possible de forcer un parcours en profondeur avec la directive **DEPTH FIRST BY**).

Question : R15 : Le code, le nom et le niveau dans la hiérarchie de tous les subordonnés de l'employé n°1.

```
WITH A (codeEmploye, codeChef, niveau)
      AS (SELECT codeEmploye, codeEmployeChef, 1
           FROM Employes
           WHERE codeEmploye = 'E1'
      UNION ALL
           SELECT sub.codeEmploye, sub.codeEmployeChef, niveau + 1
           FROM A chef
           JOIN Employes sub ON chef.codeEmploye = sub.codeEmployeChef)
SELECT niveau, e.codeEmploye, nomEmploye
FROM A a
JOIN Employes e ON a.codeEmploye = e.codeEmploye;
```

Question : R16 : Le code, le nom de tous les subordonnés de l'employé n°1 à l'exception de l'employé n°3.

```
WITH A (codeEmploye, codeChef)
      AS (SELECT codeEmploye, codeEmployeChef
           FROM Employes
           WHERE codeEmploye = 'E1'
      UNION ALL
           SELECT sub.codeEmploye, sub.codeEmployeChef
           FROM A chef
           JOIN Employes sub ON chef.codeEmploye = sub.codeEmployeChef)
SELECT e.codeEmploye, nomEmploye
FROM A a
JOIN Employes e ON a.codeEmploye = e.codeEmploye
WHERE e.codeEmploye <> 'E3';
```

Question : R17 : Le code et le nom de tous les subordonnés de l'employé n°1 à l'exception de l'employé n°3 et des subalternes de celui-ci.

```
WITH A (codeEmploye, codeChef)
      AS (SELECT codeEmploye, codeEmployeChef
           FROM Employes
           WHERE codeEmploye = 'E1'
      UNION ALL
           SELECT sub.codeEmploye, sub.codeEmployeChef
           FROM A chef
           JOIN Employes sub ON chef.codeEmploye = sub.codeEmployeChef
           WHERE sub.codeEmploye <> 'E3')
SELECT e.codeEmploye, nomEmploye
FROM A a
JOIN Employes e ON a.codeEmploye = e.codeEmploye;
```

Question : R18 : Le code et le nom de tous les supérieurs de l'employé n°7.

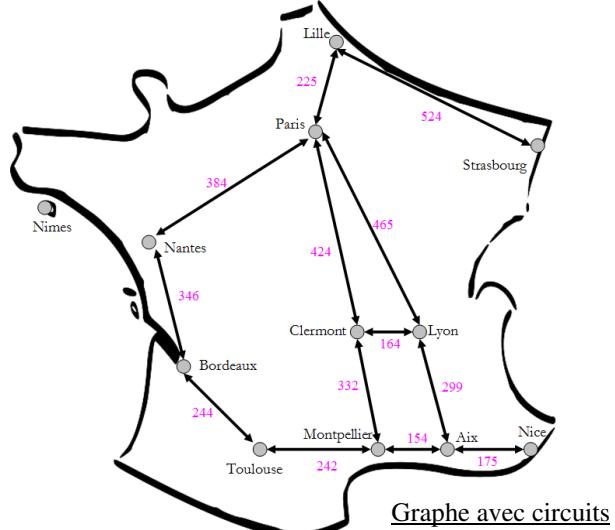
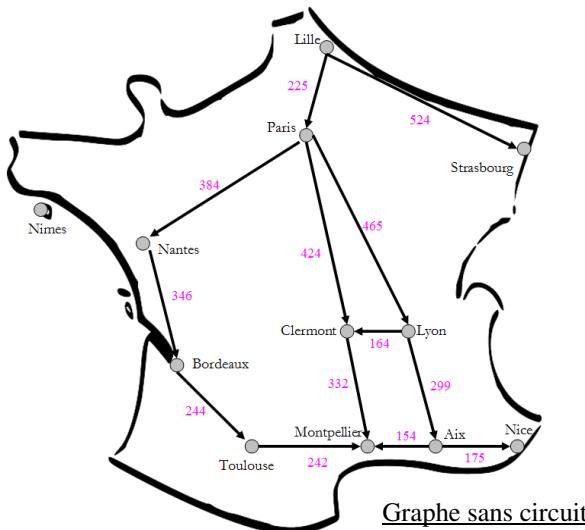
```
WITH A (codeEmploye, codeChef)
      AS (SELECT codeEmploye, codeEmployeChef
           FROM Employes
           WHERE codeEmploye = 'E7'
      UNION ALL
           SELECT chef.codeEmploye, chef.codeEmployeChef
           FROM A sub
           JOIN Employes chef ON sub.codeChef = chef.codeEmploye)
SELECT e.codeEmploye, nomEmploye
FROM A a
JOIN Employes e ON a.codeEmploye = e.codeEmploye;
```

### 5.3 Quelle instruction utiliser pour réaliser des requêtes récursives sous Oracle ?

L'instruction CONNECT BY PRIOR a l'avantage d'être plus simple et on la rencontre souvent dans du code car elle était la seule instruction disponible dans les anciennes versions d'Oracle. Toutefois, il est quand même conseillé d'utiliser la notation avec le WITH qui permet d'avoir un code qui respecte la norme (et qui sera donc portable sur les autres SGBD qui respectent la norme) et qui permet de faire des requêtes récursives plus complexes.

Exemple :

ROUTES (villeDep, villeArr, nbKm)



Question : R19 : Recherche de tous les trajets qui permettent d'aller de Lille à Montpellier, avec pour chaque trajet le nombre de kilomètres ainsi que la liste des villes traversées,

```

WITH A (villeDep, villeArr, trajet, distance)
AS (SELECT villeDep, villeArr, villeDep || '-' || villeArr, nbKm
     FROM Routes
     WHERE villeDep = 'Lille'
UNION ALL
     SELECT r.villeDep, r.villeArr, a.trajet || '-' || r.villeArr,
            a.distance + r.nbKm
     FROM A a
     JOIN Routes r ON a.villeArr = r.villeDep)
SELECT trajet, distance
FROM A
WHERE villeArr = 'Montpellier'
ORDER BY distance;

TRAJET                               DISTANCE
-----                               -----
Lille-Paris-Clermont-Montpellier      981
Lille-Paris-Lyon-Aix-Montpellier     1143
Lille-Paris-Lyon-Clermont-Montpellier 1186
Lille-Paris-Nantes-Bordeaux-Toulouse-Montpellier 1441

```

S'il y a des circuits (voir seconde figure), la requête précédente ne fonctionne pas (le message 'cycle détecté lors de l'exécution de la requête WITH récursive' sera indiqué). Pour que cela fonctionne, il ne faut pas parcourir de cycle, et donc ne pas mettre dans un trajet les villes qui ont déjà été traversées par le trajet en question. Cela peut être fait grâce à une condition WHERE dans la requête member (on trouvera en plus, ici, le trajet *Lille-Paris-Clermont-Lyon-Aix-Montpellier de 1266 km*)

```

WITH A (villeDep, villeArr, trajet, distance)
AS (SELECT villeDep, villeArr, villeDep || '-' || villeArr, nbKm
     FROM Routes
     WHERE villeDep = 'Lille'
UNION ALL
     SELECT r.villeDep, r.villeArr, a.trajet || '-' || r.villeArr,
            a.distance + r.nbKm
     FROM A a
     JOIN Routes r ON a.villeArr = r.villeDep
     WHERE a.trajet NOT LIKE '%' || r.villeArr || '%')
SELECT trajet, distance
FROM A
WHERE villeArr = 'Montpellier'
ORDER BY distance;

```

## 6 Requêtes imbriquées corrélées

Nous avons vu en première année que lorsqu'on utilise le prédictat `EXISTS` (ou `NOT EXISTS`) la requête imbriquée est toujours corrélée avec la requête externe grâce à une condition dans le `WHERE` de la requête imbriquée. En fait, il est aussi possible de réaliser des requêtes imbriquées corrélées sans utiliser le prédictat `EXISTS`, avec une requête imbriquée par un = ou un `IN`.

Comme avec le prédictat `EXISTS` la requête imbriquée va alors être recalculée pour chacune des lignes de la requête externe.

Exemple :

```
CLIENTS (codeClient, nomClient, prenomClient, ageClient, codeCategorie#)
CATEGORIES (codeCategorie, nomCategorie, nbClientsCategorie)
```

R20 : Le code, le nom et le prénom des clients qui ont un homonyme.

```
SELECT codeClient, nomClient, prenomClient
FROM Clients c1
WHERE nomClient IN (SELECT nomClient
                     FROM Clients c2
                     WHERE c1.codeClient <> c2.codeClient) ;
```

En pratique il est assez rare d'utiliser des requêtes imbriquées corrélées sans le prédictat `EXISTS`. Mais lorsqu'on est amené à faire des requêtes complexes (par exemple les requêtes de Ninjas de 1<sup>ère</sup> année) cela peut être une aide précieuse.

R21 : Le code, le nom et le prénom des clients qui sont les plus âgés de leur catégorie.

```
SELECT codeClient, nomClient, prenomClient
FROM Clients c1
WHERE ageClient = (SELECT MAX(ageClient)
                    FROM Clients c2
                    WHERE c1.codeCategorie = c2.codeCategorie) ;
```

Remarque : On aurait aussi pu réaliser cette requête en indiquant plusieurs attributs dans le `WHERE` ou bien en utilisant le prédictat `NOT EXISTS` (la requête est là aussi corrélée – et même doublement corrélée).

```
SELECT codeClient, nomClient, prenomClient
FROM Clients
WHERE (codeCategorie, ageClient) IN (SELECT codeCategorie, MAX(ageClient)
                                       FROM Clients
                                       GROUP BY codeCategorie);

SELECT codeClient, nomClient, prenomClient
FROM Clients c1
WHERE NOT EXISTS (SELECT *
                   FROM Clients c2
                   WHERE c2.ageClient > c1.ageClient
                   AND c2.codeCategorie = c1.codeCategorie);
```

Les requêtes imbriquées corrélées peuvent également être utilisées lors des mises à jours avec une instruction `UPDATE`

R22 : On veut mettre à jour l'attribut calculé `nbClientsCategorie` de toutes les lignes de la table `Categories` avec les données qu'il y a dans la table `Clients`.

```
UPDATE Categories c1
SET nbClientsCategorie = (SELECT COUNT(*)
                           FROM Clients c2
                           WHERE c2.codeCategorie = c1.codeCategorie);
```

On peut également utiliser des requêtes corrélées lorsqu'on fait des requêtes scalaires dans le `SELECT`. Toutefois lorsqu'on a affaire à de gros volumes de données, cette façon de faire dégrade rapidement les performances et il vaut mieux privilégier l'utilisation de `GROUP BY`.

R23 : Pour chaque catégorie, le code et le nom de la catégorie ainsi que la moyenne d'âge des personnes de la catégorie.

```
SELECT codeCategorie,
       nomCategorie,
       (SELECT AVG(agePersonne) FROM Personnes p WHERE p.codeCategorie = c.codeCategorie)
  FROM Categories c;
```

## 7 Les différents types de fonctions

La norme SQL propose un certain nombre de fonctions. Toutefois toutes ces fonctions ne sont pas implémentées par les différents SGBD. Et certains SGBD implémentent d'autres fonctions qui ne sont pas dans la norme. Il est donc conseillé de se référer à la documentation de son SGBD lorsqu'on a besoin d'utiliser des fonctions.

Si pour réaliser une requête particulière on a le choix entre une fonction de la norme et une fonction propriétaire du SGBD il est conseillé d'utiliser la fonction de la norme afin d'obtenir un code portable qui continuera à fonctionner même si on change de SGBD.

## 7.1 Les fonctions d'ensemble (ou fonctions d'agrégation)

Les fonctions d'ensemble les plus connues sont les fonctions `MIN()`, `MAX()`, `SUM()`, `AVG()` et `COUNT()`. Ces fonctions d'ensemble font un calcul sur un ensemble (ou un groupe) de lignes et retournent un seul résultat. Dans un `SELECT`, elles ne peuvent être accompagnées d'attributs que si la requête comporte un `GROUP BY`. Il est à noter que ces fonctions ne peuvent donc pas être utilisées dans un `WHERE` (qui travaille sur des lignes). Et il y a normalement toujours une de ces fonctions dans le `HAVING` (qui travaille sur des groupes).

Exemple :

`PARTIELS (idEtudiant, idMatiere, note)`

**R24** : L'identifiant et la moyenne des étudiants qui ont plus de 3 notes.

```
SELECT idEtudiant, AVG(note)
FROM Partiels
GROUP BY idEtudiant
HAVING COUNT(note) > 3 ;
```

Depuis sa version de 2003, SQL propose les fonctions statistiques d'ensemble `STDDEV_POP` (écart type d'une population), `STDDEV_SAMP` (écart type d'un échantillon), `VAR_POP` (variance population), `VAR_SAMP` (variance échantillon), ...

## 7.2 Les fonctions de lignes

Contrairement aux fonctions d'ensemble, les fonctions de lignes sont appliquées sur chacune des lignes de la table. Elles peuvent donc être dans le `SELECT` d'une requête même s'il y a également un attribut dans ce `SELECT`. On peut les retrouver dans un `WHERE`, et il n'y a aucune raison pour qu'elles soient dans un `HAVING`.

Sous Oracle, il existe des fonctions de lignes (SQL ou propriétaires) qui permettent de manipuler des chaînes (`SUBSTRING`, `UPPER`, `LOWER`, `INITCAP`, `TRIM`, `CHARACTER_LENGTH`, `LENGTH`, ...), les valeurs numériques (`MOD`, `ROUND`, `POWER`, `SQRT`, `LOG`, `ABS`, ...), les dates/heures (`SYSDATE`, `TO_CHAR`, `NEXT_DAY`, `EXTRACT`, `MONTHS_BETWEEN`, ...), etc.

**R25** : Pour toutes les notes de la matière 'Maths' (écrite en minuscule ou en majuscule) indiquer l'identifiant de l'étudiant et la note arrondie sans virgule.

```
SELECT idEtudiant, ROUND(note, 0)
FROM Partiels
WHERE LOWER(idMatiere) = 'maths';
```

## 7.3 Les fonctions de fenêtrage

Le fenêtrage permet de réaliser des groupements dans un `SELECT` sans avoir à utiliser l'instruction `GROUP BY`. Cela permet d'obtenir un code qui peut être plus lisible, mais surtout cela permet de pouvoir faire différents groupements dans une même requête puis d'appliquer une fonction d'ensemble sur chacun de ces groupements (pour réaliser cela sans le fenêtrage il faudrait réaliser plusieurs requêtes, ou bien faire une jointure entre plusieurs expressions de table qui réalisent des groupements différents ou enfin écrire plusieurs requêtes scalaires corrélées dans le même `SELECT` d'une requête).

Les groupements différents peuvent donc être réalisés sur chaque fonction qui est appelée dans le `SELECT` grâce à la clause `OVER (PARTITION)`. Si aucune partition n'est précisée, la fonction est appelée sur l'ensemble des lignes de la table (comme s'il n'y avait pas de groupement).

**R26** : Pour chaque note, indiquer la valeur de la note, le code de l'étudiant qui a obtenu la note, la moyenne générale de cet étudiant (toutes matières confondues), le code de la matière, la moyenne générale de cette matière (tous étudiants confondu), la plus faible note qu'il y a eu dans cette matière, le nombre d'étudiants qui ont eu une note dans cette matière et la moyenne générale de toutes les notes de la table.

```
SELECT note,
       idEtudiant,
       AVG(note) OVER (PARTITION BY idEtudiant) AS moyenneEtudiant,
       idMatiere,
       AVG(note) OVER (PARTITION BY idMatiere) AS moyenneMatiere,
       MIN(note) OVER (PARTITION BY idMatiere) AS noteMinMatiere,
       COUNT(note) OVER (PARTITION BY idMatiere) AS nbNotesMatiere,
       AVG(note) OVER () AS moyenneToutesNotes
FROM Partiels ;
```

Lorsqu'on utilise des fenêtrages, en plus des fonctions d'ensemble classiques, il est possible d'utiliser des fonctions de fenêtrage dont les principales sont : ROW\_NUMBER (numéro de la ligne sans gestion des ex-aequo), RANK (classement des ex-aequo avec trous), DENSE\_RANK (classement des ex-aequo sans trous), CUME\_DIST (distribution cumulative), NTILE (n-tile : décile, centile), LAG (valeur de la ligne précédente), LEAD (valeur de la ligne suivante), FIRST\_VALUE (valeur de la première ligne de la fenêtre), etc. Il est possible de borner la partition avec ROWS, PRECEDING, FOLLOWING, BETWEEN, etc. Pour certaines de ces fonctions, il peut être intéressant que la partition soit ordonnée dans un ordre précis avec l'instruction ORDER BY.

R27 : Pour toutes les notes classées par ordre de mérite, la valeur de la note, le classement de la note, le classement de la note dans la matière, la valeur de la note suivante et la moyenne entre la note et la note suivante.

```
SELECT note,
       ROW_NUMBER() OVER (ORDER BY note DESC) AS classementNote,
       ROW_NUMBER() OVER (PARTITION BY idMatiere ORDER BY note DESC)
                           AS classementNoteMatiere,
       LEAD(note) OVER(ORDER BY note DESC) AS ValeurNoteSuivante,
       AVG(note) OVER (ORDER BY note DESC ROWS BETWEEN CURRENT ROW AND 1 FOLLOWING)
                           AS MoyenneAvecNoteSuivante
FROM Partiels
ORDER BY note DESC;
```

Les fonctions de fenêtrage ne peuvent pas être utilisées dans un WHERE. Si on souhaite faire une sélection à partir de la valeur renournée par une fonction de fenêtrage il faut utiliser la fonction de fenêtrage dans une expression de table (par exemple un SELECT dans un FROM) puis faire la sélection dans le WHERE de la requête principale.

R28 : L'idEtudiant, l'idMatière et les notes des étudiants qui ont la moyenne générale.

```
SELECT idEtudiant, idMatiere, note
FROM (SELECT idEtudiant, idMatiere, note,
            AVG(note) OVER (PARTITION BY idEtudiant) AS moyenneEtudiant
         FROM Partiels) notes
WHERE moyenneEtudiant > 10
```

## 8 Structures conditionnelles

---

Il est possible d'utiliser des structures conditionnelles CASE ... WHEN ... THEN ... ELSE dans une instruction SQL. On retrouve généralement ces structures conditionnelles dans le SELECT d'une requête, mais elles peuvent aussi être utilisées dans les UPDATE, DELETE, WHERE, ORDER BY ou HAVING.

Exemple :

COMMANDES (idCommande, typeCommande, montantCommande, remiseCommande)

R29 : L'identifiant et le type des commandes.

```
SELECT idCommande,
       CASE typeCommande
           WHEN 1 THEN 'Premium'
           WHEN 2 THEN 'Urgente'
           ELSE 'Autre'
       END
FROM Commandes ;

SELECT idCommande,
       CASE
           WHEN typeCommande = 1 THEN 'Premium'
           WHEN typeCommande = 2 THEN 'Urgente'
           ELSE 'Autre'
       END
FROM Commandes ;
```

R30 : Mettre une remise de 10% à toutes les commandes de type 1.

```
UPDATE Commandes
SET remiseCommande =
    CASE typeCommande
        WHEN 1 THEN montantCommande * 0.1
        ELSE 0
    END
);
```

## 9 Problèmes liés aux **NULL** dans les requêtes d'interrogation

Le **NULL**, ou la valeur manquante, est un des problèmes les plus sensibles du langage SQL ; et il est à la source d'un grand nombre d'erreurs chez les programmeurs débutants (et même chez les expérimentés). Pourtant, pour éviter toutes confusions, le créateur du modèle relationnel E. F. Codd avait proposé deux types de valeurs manquantes : "inconnu" et "non applicable". Toutefois ses recommandations n'ont pas été retenues et, jusqu'à l'heure actuelle, SQL représente tous les types de valeurs manquantes par un seul et unique mot clé qui est le **NULL**. Dans ces conditions, C. J. Date (qui a popularisé la division avec un double **NOT EXISTS**) propose tout bonnement de ne pas utiliser les **NULL**. A ce sujet il a écrit en 1992 : « *Une opinion répandue parmi les auteurs est qu'on accorde beaucoup trop d'importance aux NULL, du moins tels qu'ils sont actuellement définis et implémentés en SQL, et qu'ils devraient être évités ; ils affichent un comportement très étrange et incohérent et peuvent être une source importante d'erreurs et de confusions* ».

Dans cette partie, nous allons commencer par faire un rappel sur ce qui a été vu lors des deux premières années du BUT à propos du traitement des **NULL** dans les requêtes d'extraction de données. Ensuite nous verrons en quoi les **NULL** peuvent poser des problèmes puis nous énumérerons les différentes solutions que propose SQL pour résoudre ces problèmes. Enfin, nous terminerons par un bref panorama des principaux cas pratiques où les **NULL** peuvent engendrer des complications et où les programmeurs devront redoubler de vigilance.

### 9.1 Rappels sur le traitement des **NULL**

Le **NULL** n'est pas une valeur mais une absence de valeur. On ne peut donc pas dire que deux **NULL** ont la même valeur. Ainsi, dans une table, un attribut sur lequel on a déclaré une contrainte **UNIQUE** pourra tout naturellement stocker plusieurs valeurs de tuples **NULL**. Pourtant, comme nous l'avons vu en première année, certaines instructions SQL (comme **DISTINCT** et **GROUP BY**) regroupent tous les **NULL** ensemble ; ce qui n'est pas forcément très logique comme nous le verrons dans le paragraphe suivant. Quant aux fonctions de groupe (**COUNT**, **AVG**, **SUM** ...) et aux opérateurs ensemblistes (**MINUS**, **UNION**, **UNION ALL**, **INTERSECT**), ils traitent les **NULL** de façon singulière.

Tous les exemples qui suivent prennent appui sur le SR qui suit :

**ETUDIANTS (numEtudiant, nomEtudiant, prenomEtudiant, ageEtudiant, classeEtudiant)**

ETUDIANTS (numEtudiant, nomEtudiant, prenomEtudiant, ageEtudiant, classeEtudiant)

E1	Zétofrais	Mélanie	22	S1
E2	Bricot	Judas	26	NULL
E3	Palleja	Xavier	NULL	S1
E4	Zeblouse	Agathe	NULL	NULL
E5	Ouzy	Jacques	22	S2
E6	Tarembois	NULL	34	S1

#### 9.1.1 Clause **DISTINCT** et les **NULL**.

La clause **DISTINCT** enlève du résultat toutes les lignes dupliquées. Les **NULL** dupliqués sont également exclus du résultat (pour le **DISTINCT**, tous les **NULL** sont identiques !).

**R1 : La liste des classes :**

```
SELECT DISTINCT classeEtudiant
FROM Etudiants ;
CLASSES
-----
S1
S2
NULL
```

#### 9.1.2 **GROUP BY** et les **NULL**.

Tout comme le **DISTINCT**, le **GROUP BY** regroupe tous les **NULL** ensembles.

**R2 : Le nombre d'étudiants de chacune des classes :**

```
SELECT classeEtudiant, COUNT(*)
FROM Etudiants
GROUP BY classeEtudiant ;
CLASSE      COUNT(*)
-----
S1           3
S2           1
NULL         2
```

#### 9.1.3 **ORDER BY** et les **NULL**.

Avec un **ORDER BY** les valeurs à **NULL** peuvent être considérées comme toutes supérieures ou toutes inférieures aux valeurs non **NULL**, cela dépendra du SGBD ; la norme SQL ne dit rien là-dessus (sous Oracle, les **NULL** sont considérés comme supérieurs aux non **NULL**).

Pour cette raison, la norme SQL a ensuite rajouté les indicateurs `NULLS FIRST` et `NULLS LAST` à la clause `ORDER BY` qui permet de forcer l'endroit où on veut que les `NULL` soient (par exemple `ORDER BY classeEtudiant ASC NULLS FIRST;`).

#### R3 : Nom et classe des étudiants :

SELECT nomEtudiant, classeEtudiant	NOMETUDIANT	CL
FROM Etudiants	-----	--
ORDER BY classeEtudiant ASC ;	Zetofrais	S1
	Palleja	S1
	Tarembois	S1
	Ouzy	S2
	Bricot	NULL
	Zeblouse	NULL

#### **9.1.4 Fonctions élémentaires et les `NULL`.**

Les fonctions `AVG`, `MIN`, `MAX`, `SUM` ne prennent pas en compte les `NULL`. Mais ces fonctions peuvent retourner `NULL` : une table vide a donc un `AVG` égal à `NULL`. La fonction `COUNT(x)` ne prend pas en compte les valeurs `NULL`. Par contre `COUNT(*)` qui compte les lignes va également compter les lignes où il y a des `NULL`. Mais la fonction `COUNT()` ne retourne jamais `NULL` : une table vide a naturellement un `COUNT` égal à 0.

#### R4 : Age moyen des étudiants dont on connaît l'âge :

SELECT AVG(ageEtudiant)	AVG(AGEETUDIANT)
FROM Etudiants;	----- 26

#### R5 : Nombre total d'étudiants :

SELECT COUNT(*)	COUNT(*)
FROM Etudiants;	----- 6

#### R6 : Nombre d'étudiants dont on connaît la classe :

SELECT COUNT(classeEtudiant)	COUNT(CLASSEETUDIANT)
FROM Etudiants;	----- 4

#### R7 : Nombre de classes connues :

SELECT COUNT(DISTINCT classeEtudiant)	COUNT(DISTINCTCLASSEETUDIANT)
FROM Etudiants;	----- 2

#### **9.1.5 Opérateurs ensemblistes et les `NULL`.**

Les opérateurs ensemblistes traitent les `NULL` comme les autres items. Il est à noter que si le mot clé `ALL` n'est pas utilisé un `DISTINCT` est réalisé et il ne peut pas y avoir plusieurs `NULL` dans le résultat. Par contre, avec `UNION ALL` les doublons sont gardés même s'ils sont `NULL`.

#### R8 : Ages d'étudiants de la classe S1 mais qu'on ne retrouve pas dans la classe S2 ... :

SELECT ageEtudiant	AGEETUDIANT
FROM Etudiants	-----
WHERE classeEtudiant = 'S1'	34
MINUS	NULL
SELECT ageEtudiant	
FROM Etudiants	
WHERE classeEtudiant = 'S2';	

## 9.2 Problèmes causés par les `NULL`

### **9.2.1 Valeur logique `UNKNOWN`**

Comme nous l'avons dit précédemment, `NULL` n'est pas une valeur mais une absence de valeur. Donc, si on essaye de comparer un `NULL`, c'est-à-dire une valeur manquante, avec une valeur quelconque en utilisant un opérateur de comparaison (`=`, `<`, `<=`, `>`, `>=`, `!=`) on obtient systématiquement un résultat inconnu (`UNKNOWN`) qui n'est ni vrai (`TRUE`) ni faux (`FALSE`). De ce fait, les requêtes qui effectuent des comparaisons sur des champs susceptibles d'être `NULL` peuvent donner des résultats étranges ; des résultats pas forcément faux mais qui peuvent dérouter plus d'un programmeur.

Par exemple, si on souhaite obtenir le numéro des étudiants qui se prénomment Xavier, on réalisera la requête R9 suivante et on obtiendra en résultat {'E3'}

```
SELECT numEtudiant FROM Etudiants WHERE prenomEtudiant = 'Xavier';
```

Si maintenant, on souhaite obtenir le numéro des étudiants qui ne se prénomment pas Xavier, on réalisera la requête R10 suivante et on obtiendra en résultat {'E1', 'E2', 'E4', 'E5'}

```
SELECT numEtudiant FROM Etudiants WHERE prenomEtudiant <> 'Xavier';
```

Ce résultat peut paraître surprenant mais il n'est pas faux. En effet, il n'est pas possible d'affirmer que l'étudiant 'E6' (dont on ne connaît pas le prénom) ne se prénomme pas Xavier (tout comme il n'était pas possible d'affirmer que l'étudiant 'E6' se prénomme Xavier).

A cause de cela, comme le faisait remarquer C. J. Date, certains résultats peuvent paraître très étranges voire incohérents. Regardons par exemple les requêtes suivantes :

#### R9 UNION 10 :

```
SELECT numEtudiant FROM Etudiants WHERE prenomEtudiant = 'Xavier';
UNION
SELECT numEtudiant FROM Etudiants WHERE prenomEtudiant <> 'Xavier';
L'étudiant E6 n'apparaîtra pas dans le résultat. En effet on ne peut pas dire qu'il s'appelle Xavier. Mais on ne peut pas dire non plus qu'il ne s'appelle pas Xavier.
```

#### R11 : Numéro des étudiants qui n'ont pas de prénom

```
SELECT numEtudiant
FROM Etudiants
WHERE prenomEtudiant = NULL;
Cette requête ne pourra jamais rien retourner car prenomEtudiant = NULL vaut toujours UNKNOWN !
```

#### R12 : Le nombre d'étudiants qui ne sont pas dans la classe S1

```
SELECT COUNT(*)
FROM Etudiants
WHERE classeEtudiant != 'S1';
Cette requête retournera 1, alors qu'un être humain attendrait probablement le résultat 3.
```

### 9.2.2 Logique à trois valeurs

Comme nous venons de le voir, en SQL, une expression logique peut donc avoir trois valeurs : TRUE, FALSE et UNKNOWN. Cette logique à trois valeurs se différencie donc de l'algèbre booléenne proposée en 1854 par Boole. On peut établir les tables de vérité avec les trois opérateurs logiques que l'on retrouve en SQL

	<b>NOT</b>
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

	<b>AND</b>	TRUE	UNKNOWN	FALSE
TRUE	TRUE	UNKNOWN	FALSE	
UNKNOWN	UNKNOWN	UNKNOWN	FALSE	
FALSE	FALSE	FALSE	FALSE	

	<b>OR</b>	TRUE	UNKNOWN	FALSE
TRUE	TRUE	TRUE	TRUE	
UNKNOWN	TRUE	UNKNOWN	UNKNOWN	
FALSE	TRUE	UNKNOWN	FALSE	

On remarque que le UNKNOWN se propage avec les expressions UNKNOWN AND TRUE et UNKNOWN OR FALSE. Il faudra donc en tenir compte lorsqu'on écrira des requêtes contenant plusieurs expressions logiques.

Par exemple, si on écrit la requête R13 suivante, on ne récupérera jamais aucune ligne (quel que soit le jeu de données) :

```
SELECT numEtudiant
FROM Etudiants
WHERE prenomEtudiant = NULL
AND classeEtudiant = 'S1'
AND ...
AND ...
```

Remarque : UNKNOWN se propage aussi avec les opérateurs mathématiques +, -, \*, /, MOD ...

### 9.3 Solutions proposées par SQL

Pour ne pas avoir de problème avec les NULL, la solution la plus simple est de ne pas les utiliser. Ainsi, un certain nombre de spécialistes de SQL conseillent, dans la mesure du possible, d'assortir les colonnes de la base de données (y compris les clés étrangères !?) d'une contrainte NOT NULL. Pour gérer tout de même les valeurs manquantes, on peut par exemple représenter les noms à l'aide de chaînes fictives (comme 'sans nom#1', 'sans nom#2', ...) qui contrairement aux NULL seront différenciables. Pour gérer les codes tels que le sexe on peut utiliser la classification ISO (0 = inconnu, 1 = masculin, 2 = féminin, 9 = pas applicable). Toutefois pour certains champs il va tout de même être difficile de se passer du NULL pour représenter les valeurs manquantes. Par exemple, pour les quantités, le zéro n'est généralement pas approprié et les valeurs négatives risquent de complexifier les calculs. Pour les dates, on peut difficilement se passer du NULL quand une date DEFAULT n'a pas de sens.

Quoi qu'il en soit, qu'il y ait ou non des NULL dans les tables des bases de données, le concepteur des requêtes d'interrogation doit être en mesure de gérer les NULL car les requêtes elles-mêmes sont susceptibles de produire des NULL (par le biais des jointures externes, des fonctions d'agrégation, des expressions arithmétiques ou des opérateurs ensemblistes). Pour gérer cela, SQL met à disposition du programmeur deux solutions : le prédictat IS NULL et les fonctions de conversion des NULL.

### 9.3.1 Prédicat IS [NOT] NULL

Le prédicat **IS NULL** (ou **IS NOT NULL**) permet de savoir si une colonne est renseignée ou non.

R11 : Numéro des étudiants qui n'ont pas de prénom

```
SELECT numEtudiant
FROM Etudiants
WHERE prenomEtudiant IS NULL;
```

R12 : Le nombre d'étudiants qui ne sont pas dans la classe S1

```
SELECT COUNT(*)
FROM Etudiants
WHERE classeEtudiant != 'S1'
OR classeEtudiant IS NULL;
```

### 9.3.2 Fonctions de conversion des NULL

SQL propose également un certain nombre de fonctions qui permettent de gérer les **NULL**. La fonction **COALESCE(v1, v2, v3, ... vn)** parcourt la liste des paramètres de gauche à droite et renvoie le premier paramètre qui n'est pas **NULL**. La fonction **NVL(v1, v2)** retourne **v1** si celui-ci n'est pas **NULL** ; sinon, retourne **v2**. La fonction inverse de **NVL** est **NULLIF(v1, v2)** qui renvoie **NULL** quand le premier paramètre est égal au second ; sinon elle renvoie **v1**.

R11 : Numéro des étudiants qui n'ont pas de prénom

```
SELECT numEtudiant
FROM Etudiants
WHERE COALESCE(prenomEtudiant, 'sans prenom') = 'sans prenom';
```

R12 : Le nombre d'étudiants qui ne sont pas dans la classe S1

```
SELECT COUNT(*)
FROM Etudiants
WHERE COALESCE(classeEtudiant, 'classe inconnue') != 'S1';
```

## 9.4 Quelques exemples concrets

Maintenant que nous savons comment gérer les **NULL** dans des requêtes simples, nous allons voir rapidement quelques-uns des principaux cas pratiques où les **NULL** peuvent poser des problèmes et où les développeurs doivent faire très attention.

### 9.4.1 Le NOT IN

Le développeur SQL doit être très vigilant avec les **NOT IN**. Tout comme les **IN**, ceux-ci peuvent renvoyer **TRUE**, **FALSE** ou **UNKNOWN**, mais ils ont la particularité de ne jamais renvoyer **TRUE** si un élément de la liste produite par la sous-requête vaut **NULL** (ils ne peuvent alors renvoyer que **UNKNOWN** ou **FALSE** - car **NOT IN** est équivalent à plusieurs tests d'inégalité séparés par des **AND** ; or nous avons vu précédemment que dans ce cas-là, il suffit qu'un seul des tests retourne **UNKNOWN** pour que le résultat entier soit **UNKNOWN**).

exemple :

```
ETUDIANTS (numEtudiant, nomEtudiant, prenomEtudiant, dateEtudiant, classeEtudiant#)
CLASSES (nomClasse, numEtudiantDelegue#)
```

ETUDIANTS (numEtudiant, nomEtudiant, prenomEtudiant, ageEtudiant, classeEtudiant#)					
E1	Zétofrais	Mélanie		22	S1
E2	Bricot	Judas		26	NULL
E3	Palleja	Xavier		NULL	S1
E4	Zeblouse	Agathe		NULL	NULL
E5	Ouzy	Jacques		22	S2
E6	Tarembois	NULL		34	S1

CLASSES (nomClasse, numEtudiantDelegue#)

S1	E1
S2	E5
S3	NULL

R14 : Numéro, nom et prénom des étudiants qui ne sont pas délégués

Si on ne fait pas attention on peut écrire la requête suivante qui ne renverra rien à cause du dernier tuple de la table *Classes*.

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants
WHERE numEtudiant NOT IN (SELECT numEtudiantDelegue
                           FROM Classes);
```

En utilisant le prédicat **IS NULL** ou bien la fonction **COALESCE** on obtiendra le résultat attendu

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants
WHERE numEtudiant NOT IN (SELECT numEtudiantDelegue
                           FROM Classes
                           WHERE numEtudiantDelegue IS NOT NULL);
```

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants
WHERE numEtudiant NOT IN (SELECT COALESCE (numEtudiantDelegue, 'rien')
                           FROM Classes);
```

On peut aussi utiliser un NOT EXISTS à la place du NOT IN. Au-delà de toutes considérations sur les performances, cela permet d'éviter tous les problèmes relatifs aux NULL (en effet, contrairement au IN, le EXISTS ne peut retourner que TRUE ou FALSE ; jamais UNKNOWN). En général les Ninjas SQL utilisent beaucoup les NOT EXISTS.

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants e
WHERE NOT EXISTS (SELECT *
                  FROM Classes c
                  WHERE e.numEtudiant = c.numEtudiantDelegue);
```

Enfin, voici une solution originale qui utilise une jointure externe et le prédictat IS NULL ...

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants
LEFT OUTER JOIN Classes ON numEtudiantDelegue = numEtudiant
WHERE numEtudiantDelegue IS NULL;
```

#### 9.4.2 Le quantificateur ALL

Tout comme les NOT IN, les quantificateurs ALL doivent attirer l'attention du programmeur. En effet il est équivalent à plusieurs tests d'égalité séparés par des AND. Et donc si une seule ligne de la sous-requête renvoie UNKNOWN, le résultat ne peut plus être TRUE (il ne peut plus être que FALSE ou UNKNOWN).

R15 : Numéro, nom et prénom de l'étudiant le plus âgé

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants
WHERE ageEtudiant >= ALL (SELECT ageEtudiant
                           FROM Etudiants);
```

La requête précédente ne retourne rien car certains étudiants n'ont pas d'âge. Là aussi on peut s'en sortir en utilisant le prédictat IS NULL ou la fonction COALESCE.

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants
WHERE ageEtudiant >= ALL (SELECT ageEtudiant
                           FROM Etudiants
                           WHERE ageEtudiant IS NOT NULL);
```

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants
WHERE ageEtudiant >= ALL (SELECT COALESCE (ageEtudiant, 0)
                           FROM Etudiants);
```

Ou bien, on peut tout simplement ne pas avoir recours au quantificateur ALL

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants
WHERE ageEtudiant = (SELECT MAX (ageEtudiant)
                      FROM Etudiants);
```

#### 9.4.3 La division

Nous avons vu trois façons différentes de réaliser une division (la solution ‘force brute’, l’élégante’ avec double NOT EXISTS et l’ingénieuse’ avec un MINUS). Chacune de ces trois solutions peut être affectée par les NULL.

Exemple :

ETUDIANTS (numEtudiant, nomEtudiant, prenomEtudiant, dateEtudiant, classeEtudiant#),  
LIVRES (codeLivre, titreLivre, categorieLivre, numEtudiantEmprunteur#)

ETUDIANTS (numEtudiant, nomEtudiant, prenomEtudiant, ageEtudiant, classeEtudiant#)

E1	Zétofrais	Mélanie	22	S1
E2	Bricot	Judas	26	NULL
E3	Palleja	Xavier	NULL	S1
...	...	...	...	...

LIVRES (codeLivre, titreLivre, categorieLivre, numEtudiantEmprunteur#)

L1	UML en action	Analyse	E1
L2	Merise pour les nuls	Analyse	E1
L3	Java et UML	Prog	E1
L4	Design Patterns	Analyse	E2
L5	UP	Analyse	E2
L6	eXtreme Programming	Analyse	NULL
L7	Livre sans thème	NULL	E2

R16 : Numéro, nom et prénom des étudiants qui possèdent des livres de toutes les catégories :

- La solution ‘force brute’ est celle qui est la moins sensible aux NULL. Il suffit simplement de ne pas compter les NULL en utilisant un COUNT (x) plutôt qu’un COUNT (\*).

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants
JOIN Livres ON numEtudiant = numEtudiantEmprunteur
GROUP BY numEtudiant, nomEtudiant
HAVING COUNT (DISTINCT categorieLivre) = (SELECT COUNT (DISTINCT categorieLivre)
FROM Livres);
```

- La solution élégante avec un double NOT EXISTS, qui a été popularisée par Date, doit être légèrement adaptée car avec les données que l’on a dans les tables, la requête suivante ne va rien retourner.

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants e1
WHERE NOT EXISTS (SELECT *
                  FROM Livres l2
                  WHERE NOT EXISTS (SELECT *
                                    FROM Livres l3
                                    WHERE l3.categorieLivre = l2.categorieLivre
                                      AND l3.numEtudiantEmprunteur = e1.numEtudiant));
```

En effet, pour tous les étudiants de la table *Etudiants*, y compris pour l’étudiante ‘E1’, il va exister dans la table des *Livres* au moins une catégorie de livre (en l’occurrence celle du livre ‘L7’ qui est NULL) dont la catégorie n’est pas égale aux catégories des livres que l’étudiant a empruntées.

On peut remarquer que même si l’étudiante ‘E1’ avait emprunté un autre livre sans catégorie (par exemple le livre ‘L8’ -- INSERT INTO Livres VALUES ('08', 'Livre sans thème', NULL, 'E1')) la requête ne retournerait quand même aucune ligne. En effet la catégorie du livre ‘L7’ n’est pas égale à la catégorie du livre ‘L8’ (on rappelle que NULL = NULL donne UNKNOWN !).

La solution consiste donc à chercher les étudiants pour lesquels il n’existe pas de catégories *non nulles* pour lesquelles ils n’ont pas réalisé d'emprunt.

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants e1
WHERE NOT EXISTS (SELECT *
                  FROM Livres l2
                  WHERE NOT EXISTS (SELECT *
                                    FROM Livres l3
                                    WHERE l3.categorieLivre = l2.categorieLivre
                                      AND l3.numEtudiantEmprunteur = e1.numEtudiant)
                    AND categorieLivre IS NOT NULL);
```

- La solution ‘ingénieuse’ avec le MINUS doit elle aussi être légèrement adaptée car avec les données que l’on a dans les tables, la requête suivante ne va, elle non plus, rien retourner.

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants e
WHERE NOT EXISTS (SELECT categorieLivre
                  FROM Livres
                  MINUS
                  SELECT categorieLivre
                  FROM Livres l
                  WHERE l.numEtudiantEmprunteur = e.numEtudiant) ;
```

Ici même pour l’étudiant ‘E1’ la sous requête va trouver quelque chose. En effet, si on fait la différence (MINUS) entre les catégories de la table *Livres* {‘Analyse’, ‘Prog’, NULL} et les catégories de livres empruntés par l’étudiant ‘E1’ {‘Analyse’, ‘Prog’} on obtient le résultat {NULL} ; qui n’est pas l’ensemble vide et donc qui existe !

On peut remarquer qu’ici, si l’étudiante ‘E1’ avait emprunté un autre livre sans catégorie (toujours le livre ‘L8’) alors la requête aurait retourné l’étudiant ‘E1’. En effet le MINUS ne fait pas de distinction entre les différents NULL et {‘Analyse’, ‘Prog’, NULL} MINUS {‘Analyse’, ‘Prog’, NULL} aurait donné l’ensemble vide !

Pour avoir une solution qui marche toujours, il faut faire la différence entre toutes les catégories *non nulles* de la table *Livres* et les catégories des livres empruntés par les différents étudiants.

```
SELECT numEtudiant, nomEtudiant, prenomEtudiant
FROM Etudiants e
WHERE NOT EXISTS (SELECT categorieLivre
                  FROM Livres
                  WHERE categorieLivre IS NOT NULL
                  MINUS
                  SELECT categorieLivre
                  FROM Livres l
                  WHERE l.numEtudiantEmprunteur = e.numEtudiant) ;
```