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 prédicat `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 fonction 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 pourcentageMontpellierains
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 seule colonne). Toutefois, ce genre de sous-requête scalaires 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 pourcentageMontpellierains
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 connues 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 Nom des personnes qui aiment **toutes** les villes …

1er 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(*)
                           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)

2nd 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 paraitre compliquée de prime abord, a l'avantage de toujours rester de difficulté constante, même pour les divisions plus complexes.

3nd 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 1ère 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 des personnes qui aiment toutes les villes de l'Hérault.

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

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

• R8 : 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 ville **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 inéquijointures :

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'équijointures. 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.

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)

P1	Némard	Jean	E1
P2	Palleja	Nathalie	

ENTREPRISES (codeEntreprise, nomEntreprise)

E1	Cap Gemini
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 disparaitre 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 de travail (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 de travail 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 également utiliser la jointure externe pour faire des antijointures (à la place d'un `NOT EXITS`, 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;
```