Les jointures SQL.

Les **bases de données relationnelles** sont structurées autour de **plusieurs tables** qui stockent des **données liées entre elles**. Lorsqu'il s'agit d'extraire des informations à partir de ces bases de données, il est souvent nécessaire de rassembler des données provenant de plusieurs tables à la fois. C'est là que les jointures SQL deviennent cruciales.

Les jointures permettent aux **ingénieurs, scientifiques et analystes de données** de fusionner des ensembles de données en fonction d'attributs. Que ce soit pour récupérer des informations client associées à leurs transactions, ou pour examiner les produits vendus dans des régions spécifiques, les jointures fournissent un mécanisme puissant pour naviguer à travers les relations complexes entre les données.

En utilisant des clauses comme **INNER JOIN**, **LEFT JOIN**, **RIGHT JOIN**, et d'autres, les analystes peuvent spécifier comment les lignes de différentes tables doivent être combinées - en fonction d'une ou plusieurs variables et de critères spécifiques.

Les avantages des jointures SQL ne se limitent pas à la simple combinaison de données, mais s'étendent également à l'**optimisation des requêtes**, à la simplification du code SQL en évitant les requêtes multiples, et à l'amélioration de la précision et de la cohérence des résultats obtenus.

En explorant comment et quand **utiliser efficacement les jointures SQL**, nous pouvons exploiter pleinement la richesse des informations stockées dans nos bases de données relationnelles, facilitant ainsi **une analyse de qualité et une prise de décision éclairée**.

Cette note explore les différents types de jointures sql. Nous fournissons une explication logique de chacune d'entre elles ainsi qu'un exemple contextuel. Tous les exemples s'appuient sur la base de données *employee_department* que j'ai créée ici. Voici un aperçu de cette base de données :

```sql
SELECT * FROM employees;

SELECT * FROM departments;
```

```
# Employee DB
 employee_id |  employee_name  | department_id | manager_id | salary
-------------+-----------------+---------------+------------+--------
           1 | John Dupont     |             1 |          3 |   5000
           2 | Anthony Smith   |             2 |          3 |   6000
           3 | Michelle Paris  |             1 |          4 |   4500
           4 | Emily Blanc     |             3 |       NULL |   5500
           5 | Camille Johnson |             2 |          4 |   5200

# Departments DB
 department_id | department_name | location
---------------+-----------------+----------
             1 | Sales           | New York
```

```
           2 | Marketing       | Paris
           3 | Finance         | Tokyo
```

Jointure interne.

Les jointures internes, spécifiées avec la clause **INNER JOIN** en SQL, permettent de retourner uniquement **les lignes des tables qui ont au moins une correspondance dans l'autre table** en fonction d'une **condition spécifiée dans la clause 'ON'**. Cette condition est définie par une colonne ou un ensemble de colonnes partagées entre les deux tables, tel que des clés primaires et étrangères.

Dans un premier temps une jointure interne va nous permettre de **joindre nos deux tables en fonction d'une condition**. Seront joints les éléments des deux tables vérifiant la condition. Les éléments des tables qui ne vérifient pas la condition seront laissés de côté. Supposons que l'on souhaite regrouper les informations de la table *employees* et *departments* en fonction de l'identifiant du département. Cela va nous permettre de disposer d'une vision global des relations entre les employés et leur département d'activité.

```sql
SELECT *
FROM employees
INNER JOIN departments
    ON employees.department_id = departments.department_id ;
```

```
 employee_id |  employee_name  | department_id | manager_id | salary |
department_id | department_name | location
-------------+-----------------+---------------+------------+--------+-----
----------+-----------------+----------
           1 | John Dupont     |             1 |          3 |   5000 |
1 | Sales           | New York
           2 | Anthony Smith   |             2 |          3 |   6000 |
2 | Marketing       | Paris
           3 | Michelle Paris  |             1 |          4 |   4500 |
1 | Sales           | New York
           4 | Emily Blanc     |             3 |            |   5500 |
3 | Finance         | Tokyo
           5 | Camille Johnson |             2 |          4 |   5200 |
2 | Marketing       | Paris
(5 rows)
```

On obtient bien l'ensemble des données de la table *employees* et de la table *departments* dès lors qu'il existe une correspondance pour la colonne *department_id*. Cependant, la logique d'omission des entrées qui n'ont pas de correspondance n'est pas très claire. Voyons ce qui se passe lorsqu'une entrée de la table *employees* n'a pas de valeur pour l'attribut *department_id*.

```sql
UPDATE employees SET department_id = NULL WHERE employee_id=1;
```

On modifie la table *employees* de sorte qu'il manque à *John Dupont* son département d'activité.

> Puisque l'identifiant du département est ici une clé secondaire, il ne peut être modifié par une valeur non présente dans la table *departments* sinon par la valeur nulle.

Supposons qu'on souhaite simplement obtenir le nom des employées et le nom de leur département d'activité. On peut alors spécifier les colonnes à sélectionner et observer que le résultat ne contient que les entrées ayant une correspondance dans chacune des deux tables.

```
SELECT
    emp.employee_name
    , dpt.department_name
FROM employees AS emp
  INNER JOIN departments AS dpt
    ON emp.department_id = dpt.department_id;
```

```
  employee_name   | department_name
------------------+-----------------
 Anthony Smith    | Marketing
 Michelle Paris   | Sales
 Emily Blanc      | Finance
 Camille Johnson | Marketing
(4 rows)
```

> Vous notez la présence de la clause 'AS', c'est un alias qui nous permet de renommer les attributs et tables afin de faciliter leur référencement dans le code ; le code est moins lourd à la lecture.

Nous avons joint le nom du département d'activité au nom des employés ayant bien renseigné leur département d'activité.

Joitures externes.

Les **jointures externes** permettent de récupérer non seulement les **lignes correspondantes entre deux tables**, mais aussi les **lignes d'une table même si elles n'ont pas de correspondance** dans l'autre table. Ainsi, les lignes n'ayant **pas de correspondance dans la seconde table** seront affectées de la **valeur nulle**. Attention, cette valeur nulle sera affectée au niveau des attributs de la seconde table ; les éléments de la première table restant inchangés. Qu'est-ce qui permet de caractériser la "première" et la "seconde" table ? Il existe plusieurs types de jointures externes :

- **LEFT JOIN** : Retourne toutes les lignes de la table de gauche (première table mentionnée) et les lignes correspondantes de la table de droite (deuxième table mentionnée). Si aucune correspondance n'est trouvée dans la table de droite, les colonnes de cette table seront nulles.
- **RIGHT JOIN** : Retourne toutes les lignes de la table de droite et les lignes correspondantes de la table de gauche. Si aucune correspondance n'est trouvée dans la table de gauche, les colonnes de cette table seront nulles.

- **FULL JOIN** : Retourne toutes les lignes lorsqu'il y a une correspondance dans l'une des tables. Les colonnes des tables sans correspondance sont nulles. Cela inclut les résultats des deux tables, même si aucune correspondance n'est trouvée.

Joiture externe complète.

Voyons cela en pratique. Commençons par observer le résultat d'une **joiture externe complète**. Pour cela nous allons joindre la table *departments* (deuxième table) à la table *employees* conditionnellement à l'attribut *department_id*.

```
SELECT *
FROM employees
  FULL JOIN departments
    ON employees.department_id = departments.department_id;
```

```
 employee_id |  employee_name  | department_id | manager_id | salary |
department_id | department_name | location
-------------+-----------------+---------------+------------+--------+-----
----------+-----------------+----------
           1 | John Dupont     |             1 |          3 |   5000 |
1 | Sales           | New York
           2 | Anthony Smith   |             2 |          3 |   6000 |
2 | Marketing       | Paris
           3 | Michelle Paris  |             1 |          4 |   4500 |
1 | Sales           | New York
           4 | Emily Blanc     |             3 |            |   5500 |
3 | Finance         | Tokyo
           5 | Camille Johnson |             2 |          4 |   5200 |
2 | Marketing       | Paris
(5 rows)
```

Cette commande a permis de combiner les 2 tables, de les associer entre elles grâce à une condition. Si le département d'activité n'était pas renseigné pour l'un des employés il en résulterait un ensemble de valeurs nulles pour les attributs de la seconde table. Voyez plutôt le résultat.

```
UPDATE employees SET department_id = NULL  WHERE employee_id=1 ;

SELECT *
FROM employees
  FULL JOIN departments
    ON employees.department_id = departments.department_id;
```

```
 employee_id |  employee_name  | department_id | manager_id | salary |
department_id | department_name | location
-------------+-----------------+---------------+------------+--------+-----
```

```
----------+-----------------+----------
           2 | Anthony Smith   |             2 |          3 |   6000 |
2 | Marketing       | Paris
           3 | Michelle Paris  |             1 |          4 |   4500 |
1 | Sales           | New York
           4 | Emily Blanc     |             3 |            |   5500 |
3 | Finance         | Tokyo
           5 | Camille Johnson |             2 |          4 |   5200 |
2 | Marketing       | Paris
           1 | John Dupont     |               |          3 |   5000 |
|                 |
(5 rows)
```

Effectivement, pour l'employé *John Dupont*, toutes les valeurs de la table *departments* sont *NULL*. On obtient bien la jointure de l'ensemble des données de la table *employees* et *departments*.

Joiture externe à gauche.

Passons maintenant à l'étude des **jointures externes à gauche**. Comme précédemment, nous allons utiliser la table dans laquelle *John Dupont* n'a pas renseigné son département d'activité. Nous allons également ajouter un attribut *speciality* à la table *employees*.

Ajoutons la variable *speciality* à la table *employees* :

```sql
ALTER TABLE employees ADD COLUMN speciality VARCHAR(15);

UPDATE employees SET speciality = 'Data mining' WHERE employee_id = 1;
UPDATE employees SET speciality = 'Marketing' WHERE employee_id = 2;
UPDATE employees SET speciality = 'SQL' WHERE employee_id = 3;
UPDATE employees SET speciality = 'Finance' WHERE employee_id = 4;
UPDATE employees SET speciality = 'Clustering' WHERE employee_id = 5;

SELECT * FROM employees ;
```

Voici le résultat de notre modification.

```
 employee_id |  employee_name  | department_id | manager_id | salary |
speciality
-------------+-----------------+---------------+------------+--------+-----
--------
           1 | John Dupont     |               |          3 |   5000 | Data
mining
           2 | Anthony Smith   |             2 |          3 |   6000 |
Marketing
           3 | Michelle Paris  |             1 |          4 |   4500 | SQL
           4 | Emily Blanc     |             3 |            |   5500 |
Finance
           5 | Camille Johnson |             2 |          4 |   5200 |
```

```
Clustering
(5 rows)
```

Nous allons procéder à la jointure externe à gauche. La joiture sera conditionnelle aux attributs *speciality* et *department_name*. Autrement dit, nous voulons **joindre les données du département d'activité aux seuls employés dont la spécialité correspond à l'activité de leur département**.

Compte tenu des modifications apportées, le résultat devrait être le suivant :

- les employés 2 et 4 présenterons des données concernant leur département d'activité ;
- en ce qui concerne les autres employées, les données relatives à leur département d'activité seront affectées de la valeur nulle.

```
SELECT *
FROM employees
  LEFT JOIN departments
    ON employees.speciality = departments.department_name;
```

```
 employee_id |  employee_name  | department_id | manager_id | salary |
speciality  | department_id | department_name | location
------------+-----------------+---------------+------------+--------+-----
--------+---------------+-----------------+----------
           2 | Anthony Smith   |             2 |          3 |   6000 |
Marketing   |             2 | Marketing       | Paris
           4 | Emily Blanc     |             3 |            |   5500 |
Finance     |             3 | Finance         | Tokyo
           5 | Camille Johnson |             2 |          4 |   5200 |
Clustering  |               |                 |
           1 | John Dupont     |               |          3 |   5000 | Data
mining |               |                 |
           3 | Michelle Paris  |             1 |          4 |   4500 | SQL
|               |                 |
(5 rows)
```

Le résultat est bien celui que nous attendions.

Jointure externe à droite.

Voyons le comportement d'une **jointure externe à droite**. Supposons que nous ayons accès à la ville d'origine des employés et qu'on souhaite **obtenir les informations salariales des individus exerçants dans leur ville d'origine**. Une jointure externe à droite va nous permettre de joindre aux données des départements d'activité celles des employées qui travaillent dans leur ville de naissance ; les autres seront affectés de la valeur nulle.

```
ALTER TABLE employees ADD COLUMN hometown VARCHAR(15);
```

```
UPDATE employees SET hometown = 'Nice' WHERE employee_id = 1;
UPDATE employees SET hometown = 'Londres' WHERE employee_id = 2;
UPDATE employees SET hometown = 'New-York' WHERE employee_id = 3;
UPDATE employees SET hometown = 'Paris' WHERE employee_id = 4;
UPDATE employees SET hometown = 'Paris' WHERE employee_id = 5;

SELECT * FROM employees ;
```

Voici notre table *employee* agrémentée d'un attribut *hometown*.

```
 employee_id |  employee_name  | department_id | manager_id | salary |
speciality  | hometown
-------------+-----------------+---------------+------------+--------+-----
--------+----------
           1 | John Dupont     |               |          3 |   5000 | Data
mining | Nice
           2 | Anthony Smith   |             2 |          3 |   6000 |
Marketing   | Londres
           3 | Michelle Paris  |             1 |          4 |   4500 | SQL
| New-York
           4 | Emily Blanc     |             3 |            |   5500 |
Finance     | Paris
           5 | Camille Johnson |             2 |          4 |   5200 |
Clustering  | Paris
(5 rows)
```

Procédons à la jointure externe à droite **conditionnellement à la ville de naissance et au lieu de travail**.

```
SELECT *
FROM employees
  RIGHT JOIN departments
    ON employees.hometown = departments.location ;
```

```
 employee_id |  employee_name  | department_id | manager_id | salary |
speciality | hometown | department_id | department_name | location
-------------+-----------------+---------------+------------+--------+-----
-------+----------+---------------+-----------------+----------
            |                 |               |            |        |
|          |             1 | Sales           | New York
           5 | Camille Johnson |             2 |          4 |   5200 |
Clustering | Paris    |             2 | Marketing       | Paris
           4 | Emily Blanc     |             3 |            |   5500 |
Finance    | Paris    |             2 | Marketing       | Paris
            |                 |               |            |        |
|          |             3 | Finance         | Tokyo
(4 rows)
```

Le résultat obtenu est bien celui escompté. Les employées dont la ville de naissance correspond au lieu de travail ont été joint aux données *departments*.

Que se passerait-il si nous effectuions une jointure externe à gauche ? Les éléments des départements auraient été joint aux données des employées travaillant dans leur ville de naissance. Observez plutôt.

```
SELECT *
FROM employees
  LEFT JOIN departments
    ON employees.hometown = departments.location;
```

```
 employee_id |  employee_name  | department_id | manager_id | salary |
speciality  | hometown | department_id | department_name | location
-------------+-----------------+---------------+------------+--------+-----
--------+----------+---------------+-----------------+----------
           5 | Camille Johnson |             2 |          4 |   5200 |
Clustering  | Paris    |             2 | Marketing       | Paris
           4 | Emily Blanc     |             3 |            |   5500 |
Finance     | Paris    |             2 | Marketing       | Paris
           3 | Michelle Paris  |             1 |          4 |   4500 | SQL
| New-York |               |               |
           2 | Anthony Smith   |             2 |          3 |   6000 |
Marketing   | Londres  |               |               |
           1 | John Dupont     |               |          3 |   5000 | Data
mining | Nice     |               |               |
(5 rows)
```

Les employés 5 et 4 ont été augmentés des données concernant leur département d'activité.

Jointure propre (i.e. self join).

Nous l'avons vu, les jointures sont une technique puissante pour manipuler des bases de données relationnelles et agrémenter nos analyses de données diverses et variées. L'une d'entre elles a particulièrement retenue mon attention : le fameux **self-join**. Cette jointure est peu présente, à mon avis, sur les différents réseau. Pourtant elle peut s'avérer très intéressante. Par exemple, nous disposons pour chaque employé de l'identifiant de son manager. Vous vous demandez peut-être comment regrouper ces informations pour **obtenir, pour chaque employée, le nom de son manager** ? Cela peut sembler compliquer mais le sefl-join est là pour nous aider.

Une **jointure propre** (je ne sais pas trop comment qualifier ce type de jointure en français ; jointure entre-soi ?) va nous permettre selon une condition de **joindre une table avec elle-même**. Pour l'écrire autrement, cela va nous permettre de **comparer les lignes de la table avec d'autres lignes de la même table**. Notamment, c'est très utile pour **trouver des relations hiérarchiques** dans les données -e.g. employé~manager- ou comparer une ligne avec la ligne inférieure/supérieure -e.g. peu se substituer aux fonctions de fenêtrage LAG() et LEAD() ; nous aborderons un cas d'usage à ce sujet.

Un self join fonctionne en **utilisant des alias** pour la table afin de la référencer deux fois dans la même requête. Cela permet de traiter la table **comme si elle était deux tables distinctes**. Comme pour n'importe qu'elle jointure vous pouvez fixer une ou plusieurs condition. **Les conditions agirons comme des filtres** sur votre base de données puisque le self join est une forme de **jointure interne**.

Analysons les relations employé~manager.

```
SELECT
     emp.employee_name AS employee,
     mng.employee_name AS manager
FROM employees AS emp
JOIN employees AS mng
     ON emp.manager_id = mng.employee_id
;
```

```
  employee       |  manager
-----------------+----------------
 John Dupont     | Michelle Paris
 Anthony Smith   | Michelle Paris
 Michelle Paris  | Emily Blanc
 Camille Johnson | Emily Blanc
(4 rows)
```

Le résultat est conforme à nos attentes. L'employée Emily Blanc n'est pas incluse dans l'ensemble des employés puisqu'elle n'est pas surbordonnée à un autre employé.

Self join, cas d'usage avancé.

Le service RH de votre entreprise est confronté à une erreur comptable grave : certains salaires ont été envoyé deux fois. Toutefois, ce même service à effectué le même jour plusieurs virement au même employée : l'un pour son salaire, d'autre pour diverses primes de fin d'année. Qui plus est, il se peut que certains salaires soient d'un montant égale à la prime. Aussi, on suppose que le salaire à été viré deux fois si deux virements d'un même montant sont effectué à moins de 25 minutes d'intervalles. Trouvez le nombre de virements accidentels, les montants et les employés concernés.

Vous disposez d'une table *transactions* dont les attributs sont :

- employee_id
- employee_name
- services
- amount
- date_time

L'attribut *service* correspond au service ayant effectué le virement -e.g. Sales, RH. *amount* et *date_time* représentent, respectivement, le montant et la date et l'heure du virement. Cette table contient 12 erreurs

comptables. Voyons comment les retrouver avec un self-join puis avec des requêtes sql avancées - **CTEs** et **fonction de fenêtrage**.

Ce premier exemple permet d'entrevoir le fonctionnement d'un self join. Notamment, on obsevre bien que les conditions agissent comme des filtres puisqu'il s'agit d'une jointure interne.

```sql
SELECT
    *
FROM transactions t1
JOIN transactions t2
     ON t1.employee_id = t1.employee_id
        AND t1.service = t2.service
        AND t1.amount = t2.amount
        AND t1.date_time < t2.date_time
        AND (t2.date_time - t1.date_time) <= INTERVAL '20 minutes';
```

```
 transaction_id | employee_id | employee_name | service | amount  |
date_time      | transaction_id | employee_id | employee_name | service |
amount  |      date_time
----------------+-------------+---------------+---------+---------+--------
-------------+----------------+-------------+---------------+---------+----
-----+---------------------
              1 |           1 | Employee 1    | RH      | 2926.32 | 2023-
01-01 08:00:00 |             61 |           1 | Employee 1    | RH      |
2926.32 | 2023-01-01 08:10:00
              6 |           6 | Employee 6    | RH      | 2143.80 | 2023-
01-06 08:00:00 |             62 |           6 | Employee 6    | RH      |
2143.80 | 2023-01-06 08:10:00
             11 |          11 | Employee 11   | RH      | 3368.36 | 2023-
01-11 08:00:00 |             63 |          11 | Employee 11   | RH      |
3368.36 | 2023-01-11 08:10:00
             16 |          16 | Employee 16   | RH      | 2347.38 | 2023-
01-16 08:00:00 |             64 |          16 | Employee 16   | RH      |
2347.38 | 2023-01-16 08:10:00
             21 |          21 | Employee 21   | RH      | 4604.72 | 2023-
01-21 08:00:00 |             65 |          21 | Employee 21   | RH      |
4604.72 | 2023-01-21 08:10:00
             26 |          26 | Employee 26   | RH      | 2931.13 | 2023-
01-26 08:00:00 |             66 |          26 | Employee 26   | RH      |
2931.13 | 2023-01-26 08:10:00
             31 |          31 | Employee 31   | RH      | 2709.83 | 2023-
01-01 08:00:00 |             67 |          31 | Employee 31   | RH      |
2709.83 | 2023-01-01 08:10:00
             36 |          36 | Employee 36   | RH      | 2162.53 | 2023-
01-06 08:00:00 |             68 |          36 | Employee 36   | RH      |
2162.53 | 2023-01-06 08:10:00
             41 |          41 | Employee 41   | RH      | 3161.06 | 2023-
01-11 08:00:00 |             69 |          41 | Employee 41   | RH      |
3161.06 | 2023-01-11 08:10:00
             46 |          46 | Employee 46   | RH      | 4553.37 | 2023-
```

```
01-16 08:00:00 |             70 |          46 | Employee 46   | RH      |
4553.37 | 2023-01-16 08:10:00
             51 |          51 | Employee 51   | RH      | 3848.02 | 2023-
01-21 08:00:00 |             71 |          51 | Employee 51   | RH      |
3848.02 | 2023-01-21 08:10:00
             56 |          56 | Employee 56   | RH      | 4236.88 | 2023-
01-26 08:00:00 |             72 |          56 | Employee 56   | RH      |
4236.88 | 2023-01-26 08:10:00
(12 rows)
```

On peut déjà remarquer que 12 virements ont été réalisés deux fois - le résultat compte 12 lignes. Notre intérêt porte sur les attributs *employee_name* et *amount*, on doit donc spécifier ces attributs. À ce sujet, pensez à toujours indexer les attributs par leur table de provenance sinon une erreur sera levée ; par exemple, **"column reference "employee_name" is ambiguous"**.

```sql
SELECT
    t1.employee_name,
    t1.amount
FROM transactions t1
JOIN transactions t2
     ON t1.employee_id = t1.employee_id
        AND t1.service = t2.service
        AND t1.amount = t2.amount
        AND t1.date_time < t2.date_time
        AND (t2.date_time - t1.date_time) <= INTERVAL '20 minutes';
```

```
 employee_name | amount
---------------+---------
 Employee 1    | 2926.32
 Employee 6    | 2143.80
 Employee 11   | 3368.36
 Employee 16   | 2347.38
 Employee 21   | 4604.72
 Employee 26   | 2931.13
 Employee 31   | 2709.83
 Employee 36   | 2162.53
 Employee 41   | 3161.06
 Employee 46   | 4553.37
 Employee 51   | 3848.02
 Employee 56   | 4236.88
(12 rows)
```

Nous avons bien **12 employés concernés par l'erreur du service comptable**.

Une autre façon d'obtenir le même résultat aurait été d'utiliser une **CTEs** et une **fonction de fenêtrage** telle que **LAG()**. Cet exemple vous permettra suûrement de comprendre la logique qui se cache derrière les

jointures propres et/ou les CTEs et fonctions de fenêtrage (si vous n'êtes pas familier avec ces deux derniers concepts, c'est par ici).

```
WITH ma_ctes AS (
SELECT
  employee_name,
  amount,
  date_time - LAG(date_time)
                          OVER(
                          PARTITION BY employee_name, service, amount
                          ORDER BY date_time) AS time_lead
FROM transactions
)

SELECT
  employee_name,
  amount
FROM ma_ctes
WHERE time_lead <= INTERVAL '10 MINUTES' ;
```

```
 employee_name | amount
---------------+---------
 Employee 1    | 2926.32
 Employee 11   | 3368.36
 Employee 16   | 2347.38
 Employee 21   | 4604.72
 Employee 26   | 2931.13
 Employee 31   | 2709.83
 Employee 36   | 2162.53
 Employee 41   | 3161.06
 Employee 46   | 4553.37
 Employee 51   | 3848.02
 Employee 56   | 4236.88
 Employee 6    | 2143.80
(12 rows)
```

Nous obtenons bien le même résultat.

Jointure croisée.

Je voulais vous présenter un dernier type de jointure qui présente un intérêt particulier, notamment pour retourner chaque ligne d'une table avec chaque ligne d'une autre table : **la jointure croisée**. Celle-ci effectue **le produit cartésien de deux tables**. Par exemple, si on effectue le produit cartésien de nos deux tables *employees* et *departments* on obtiendrait l'ensemble de tous les couples dont la première composante appartient à *employees* et la seconde à *departments*. La jointure croisée renvoie donc une table de grande dimension. Si la première table compte n lignes et la seconde m lignes, on obtiendra $n \times m$ lignes.

Voyons ce que cela donne en pratique.

```
SELECT *
FROM employees
  CROSS JOIN departments ;
```

```
 employee_id |  employee_name  | department_id | manager_id | salary |
speciality  | hometown | department_id | department_name | location
-------------+-----------------+---------------+------------+--------+-----
--------+----------+---------------+-----------------+----------
           1 | John Dupont     |               |          3 |   5000 | Data
mining | Nice     |             1 | Sales           | New York
           2 | Anthony Smith   |             2 |          3 |   6000 |
Marketing   | Londres  |             1 | Sales           | New York
           3 | Michelle Paris  |             1 |          4 |   4500 | SQL
| New-York |             1 | Sales           | New York
           4 | Emily Blanc     |             3 |            |   5500 |
Finance     | Paris    |             1 | Sales           | New York
           5 | Camille Johnson |             2 |          4 |   5200 |
Clustering  | Paris    |             1 | Sales           | New York
           1 | John Dupont     |               |          3 |   5000 | Data
mining | Nice     |             2 | Marketing       | Paris
           2 | Anthony Smith   |             2 |          3 |   6000 |
Marketing   | Londres  |             2 | Marketing       | Paris
           3 | Michelle Paris  |             1 |          4 |   4500 | SQL
| New-York |             2 | Marketing       | Paris
           4 | Emily Blanc     |             3 |            |   5500 |
Finance     | Paris    |             2 | Marketing       | Paris
           5 | Camille Johnson |             2 |          4 |   5200 |
Clustering  | Paris    |             2 | Marketing       | Paris
           1 | John Dupont     |               |          3 |   5000 | Data
mining | Nice     |             3 | Finance         | Tokyo
           2 | Anthony Smith   |             2 |          3 |   6000 |
Marketing   | Londres  |             3 | Finance         | Tokyo
           3 | Michelle Paris  |             1 |          4 |   4500 | SQL
| New-York |             3 | Finance         | Tokyo
           4 | Emily Blanc     |             3 |            |   5500 |
Finance     | Paris    |             3 | Finance         | Tokyo
           5 | Camille Johnson |             2 |          4 |   5200 |
Clustering  | Paris    |             3 | Finance         | Tokyo
(15 rows)
```

On obtient une table de 15 lignes et 10 colonnes. En effet, la table *employees* compte 5 lignes et la table *departments* 3. Très souvent il est d'usage de compléter la jointure croisée d'une clause *where* afin de filtrer le résultat.

Supposons qu'on souhaite mettre en place un groupe de travail pour fournir des avancés techniques majeur dans chacun des services. On veut connaitre l'ensemble des compétences qui pourrait être employé

dans chaque service. Néanmoins, les compétences identique au service doivent être exclues puisque spécifiques à ce département. Voyons combien de binome nous pouvons obtenir.

```
SELECT
    employee_name
    , speciality
    , department_name
FROM employees
CROSS JOIN departments
WHERE speciality <> department_name ;
```

```
  employee_name  | speciality  | department_name
-----------------+-------------+-----------------
 John Dupont     | Data mining | Sales
 Anthony Smith   | Marketing   | Sales
 Michelle Paris  | SQL         | Sales
 Emily Blanc     | Finance     | Sales
 Camille Johnson | Clustering  | Sales
 John Dupont     | Data mining | Marketing
 Michelle Paris  | SQL         | Marketing
 Emily Blanc     | Finance     | Marketing
 Camille Johnson | Clustering  | Marketing
 John Dupont     | Data mining | Finance
 Anthony Smith   | Marketing   | Finance
 Michelle Paris  | SQL         | Finance
 Camille Johnson | Clustering  | Finance
(13 rows)
```

Nous savons à présent que le département des ventes (*sales*) peut s'appuyer sur les compétences de 5 employées (data mining, marketing, sql, finance, clustering). Le département de marketing peut proposer des groupes de travail en analyse de données (data mining, sql et clustering) et en finance. Enfin, le département de finance peut quant à lui mettre en place un groupe de travail de conception d'outils d'aide à la décision en marketing (segmentation, ciblage), par exemple.

> Notez que la requête n'utilise pas d'alias puisqu'il est clair que les attributs filtrés ou sélectionnés sont unique dans l'une ou l'autre des tables. Toutefois, si nous avions filtré les individus en fonction de leur département -e.g. *WHERE department_id =1*- SQL aurait renvoyé une erreur due à l'ambiguité que présente cette colonne. En effet, elle est présente dans les deux tables. Il est de bonne pratique de toujours utiliser des alias pour référencer les attributs des tables.

Pour les connaisseurs, une façon simple de faire une jointure croisée :

```
SELECT *
FROM employees AS emp, departments AS dpt
WHERE emp.department_id = 1;
```

```
 employee_id | employee_name  | department_id | manager_id | salary |
speciality | hometown | department_id | department_name | location
-------------+----------------+---------------+------------+--------+------
------+----------+---------------+-----------------+----------
           3 | Michelle Paris |             1 |          4 |   4500 | SQL
| New-York |             1 | Sales           | New York
           3 | Michelle Paris |             1 |          4 |   4500 | SQL
| New-York |             2 | Marketing       | Paris
           3 | Michelle Paris |             1 |          4 |   4500 | SQL
| New-York |             3 | Finance         | Tokyo
(3 rows)
```

Si on veut un employée du département 1, seule la compétence *SQL* est disponible pour compléter les groupes de travail des différents départements.

Jointure naturelle.

Parfois, même souvent, deux (ou plus) de nos tables ont un attribut commun. Dans notre cas, les tables *employees* et *departments* partagent l'attribut *department_id*. La **jointure naturelle** va alors nous permettre de joindre nos deux tables en laissant notre RDBMS chercher un **attribut commun pour joindre deux tables**. La jointure se fera sous condition qu'il y ai **un attribut du même nom ET du même type** entre chaque colonne. Dans notre cas, c'est possible puisque l'attribut *department_id* (table *employees*) est une clé étrangère qui référence la colonne du même nom dans la table *departments*.

```
SELECT *
FROM employees
NATURAL JOIN departments ;
```

```
 department_id | employee_id |  employee_name  | manager_id | salary |
speciality | hometown | department_name | location
---------------+-------------+-----------------+------------+--------+-----
-------+----------+-----------------+----------
             2 |           2 | Anthony Smith   |          3 |   6000 |
Marketing  | Londres  | Marketing       | Paris
             1 |           3 | Michelle Paris  |          4 |   4500 | SQL
| New-York | Sales           | New York
             3 |           4 | Emily Blanc     |            |   5500 |
Finance    | Paris    | Finance         | Tokyo
             2 |           5 | Camille Johnson |          4 |   5200 |
Clustering | Paris    | Marketing       | Paris
(4 rows)
```

La jointure a bien été faite sur la colonne *department_id*.

Usage avancé des jointures.

Cette section examine et présente des solutions à certains cas d'usage très connus - cf. DataLemure. Notamment, nous proposons des cas d'école qui nécessitent l'emploi de techniques avancées (CTEs, window functions) couplées à l'usage de jointures.

Etude de cas 1 : Spotify.

Assume there are three Spotify tables: artists, songs, and global_song_rank, which contain information about the artists, songs, and music charts, respectively.

Write a query to find the top 5 artists whose songs appear most frequently in the Top 10 of the global_song_rank table. Display the top 5 artist names in ascending order, along with their song appearance ranking.

If two or more artists have the same number of song appearances, they should be assigned the same ranking, and the rank numbers should be continuous (i.e. 1, 2, 2, 3, 4, 5). If you've never seen a rank order like this before, do the rank window function tutorial.

```
WITH frequence AS (SELECT
  artists.artist_name,
  DENSE_RANK() OVER(ORDER BY COUNT(songs.song_id) DESC) as artist_rank
FROM artists
INNER JOIN songs
  ON artists.artist_id = songs.artist_id
INNER JOIN global_song_rank
  ON songs.song_id = global_song_rank.song_id
WHERE global_song_rank.rank <= 10
GROUP BY artists.artist_name)

SELECT
  artist_name,
  artist_rank
FROM frequence
WHERE artist_rank <= 5
;
```

```
 artist_name  | artist_rank
--------------+-------------
 Taylor Swift |           1
 Drake        |           2
 Bad Bunny    |           2
 Ed Sheeran   |           3
 Adele        |           3
 Lady Gaga    |           4
 Katy Perry   |           5
(7 rows)
```

Etude de cas 2 : Microsoft.

A Microsoft Azure Supercloud customer is defined as a company that purchases at least one product from each product category.

Write a query that effectively identifies the company ID of such Supercloud customers.

As of 5 Dec 2022, data in the customer_contracts and products tables were updated.

MA SOLUTION :

```
WITH ma_ctes AS (
SELECT
  cc.customer_id,
  RANK() OVER(PARTITION BY cc.customer_id ORDER BY prod.product_category)
as nbr_product
FROM customer_contracts as cc
INNER JOIN products as prod
  ON cc.product_id = prod.product_id
GROUP BY cc.customer_id, prod.product_category)

SELECT
  customer_id
FROM ma_ctes
GROUP BY customer_id
HAVING COUNT(nbr_product) = 3;
```

```
 customer_id
-------------
           7
(1 row)
```

Voici une autre solution.

```
SELECT
  customer_contracts.customer_id
FROM customer_contracts
INNER JOIN products
  ON customer_contracts.product_id = products.product_id
GROUP BY customer_contracts.customer_id
HAVING COUNT(DISTINCT products.product_category) = (SELECT
                                                      COUNT(DISTINCT
product_category)
                                                    FROM products)
;
```

```
 customer_id
-------------
```

```
             7
(1 row)
```