# 07 – Introduction aux bases de données

Une base de données (ou *database* en anglais), usuellement abrégée en BD ou BDD, est un ensemble structuré et organisé permettant le stockage de grandes quantités d'informations afin d'en faciliter l'exploitation (ajout, mise à jour, recherche de données).
Nous utilisons tous les jours des bases de données, voici quelques exemples :

- Lorsque vous consultez vos notes sur Pronote : les informations vous concernant sont enregistrées sur une base de données.
- Lorsque vous allez sur un site d'achat sur internet tel que Amazon, vous pouvez accéder à une liste de catégorie de produits (livres, CD, informatique…), consulter les différentes rubriques de livres, consulter la description du livre, commander un livre et réaliser le paiement en ligne avec vos informations bancaires, consulter vos précédentes commandes.
- Lorsque vous écoutez de la musique sur une application type Deezer ou Spotify.

*Remarque : nous travaillerons avec des ensembles de données de taille raisonnable cette année, mais vous avez sans doute déjà entendu parler des **Big Data**, ensembles de données tellement gigantesques qu'ils deviennent ingérables avec les quelques outils que nous allons découvrir ensemble. (les géants comme Google et Amazon ont développé leur propre système de gestion de base de données)*

## I) Concept des bases de données relationnelles

### 1) Bases de données simples (à une table)

Pour stocker et organiser de l'information sur un sujet donné, on peut la représenter sous forme d'un tableau à deux dimensions.
Exemples :

*Base de données sur les régions de France :*

| Nom | Préfecture | Population | Superficie |
|---|---|---|---|
| Auvergne-Rhône-Alpes | Lyon | 7 820 96 | 69 711 |
| Bourgogne-Franche-Comté | Dijon | 2 820 623 | 47 784 |
| … | … | … | … |

*Base de données sur les pays :*

| Nom | Continent | Capitale | Indépendance | Population | Superficie | PIB/hab. |
|---|---|---|---|---|---|---|
| Afghanistan | Asie | Kaboul | 1919 | 34 859 568 | 647 500 | 585 |
| Afrique du Sud | Afrique | Pretoria | 1910 | 55 653 654 | 1 219 912 | 6 160 |
| … | … | … | … | … | … | … |

## 2) Bases de données complexes (à plusieurs tables)

Pour éviter d'éventuelles redondances dans de tels tableaux, on peut être amené à stocker l'information voulue dans plusieurs tables, entre lesquelles on établit alors des liens.

Exemple : Base de données sur les régions et villes de France

*Régions :*

| Numéro | Nom | Préfecture | Population | Superficie |
|---|---|---|---|---|
| 1 | Auvergne-Rhône-Alpes | 25 | 7 820 96 | 69 711 |
| … | … | … | … | … |

*Villes :*

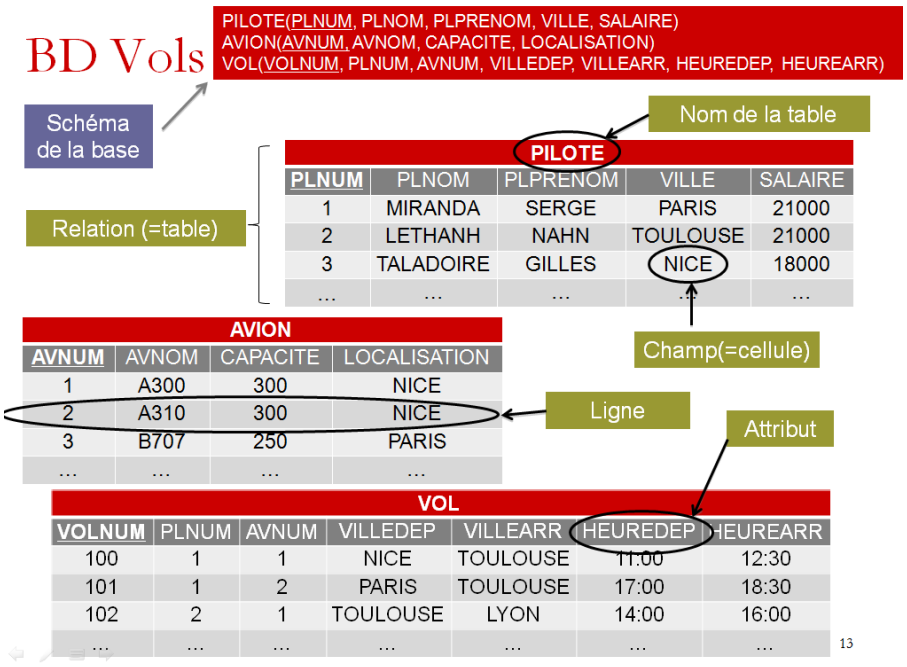
| Numéro | Nom | Code Postal | Région | Population |
|---|---|---|---|---|
| 1 | Agen | 47000 | 10 | 33 988 |
| … | … | … | … | … |

## 3) Vocabulaire

### a) Généralités

- Une base de données est un **ensemble de tables** (une table s'appelle aussi **une relation** de la base de données).
- Chaque table est composée de colonnes, le nom de chaque colonne s'appelle un **attribut**.
- A chaque attribut est associé son **domaine**, c'est-à-dire un ensemble auquel appartiennent les entrées de la colonne (chaine de caractères, entier, réel…).
- Chaque ligne d'une table (la ligne du nom des colonnes étant exclue) s'appelle un **multiplet** (**tuple** en anglais), ou encore **un enregistrement**.
- Un **champ** est l'intersection d'une ligne et d'une colonne.
- On appelle **schéma relationnel** le descriptif qui associe à chaque table la liste de ses attributs.

Exemple : Base de données d'une compagnie aérienne




*Remarque : un champ non renseigné sera codé **NULL**.*

### b) Clef primaire/clef étrangère

Les lignes (multiplets) d'une relation sont toujours deux à deux distinctes.

On appelle **clef primaire** d'une relation tout sous ensemble minimal de ses attributs permettant **d'identifier de façon unique** ses multiplets. Il faut être vigilant car dans l'exemple ci-dessus, dans la table «PILOTE», l'attribut « PLNOM » ne peut pas servir de clef primaire (plusieurs pilotes peuvent avoir le même nom), et même le couple (« nom », « prénom ») ne peut pas être une clef primaire. C'est pour ça qu'un numéro a été attribué à chaque pilote.

Dans le cas extrême, la clef primaire sera l'ensemble des attributs, c'est-à-dire seule la donnée de toute la ligne permettra d'identifier cette ligne de façon unique ; il est déconseillé d'arriver à ce cas de figure.

Dans la réalité, on utilise donc plutôt une numération des lignes, comme dans l'exemple de la base données « Vols » ci-dessus, et c'est cet identifiant unique qui sera la clef primaire, ceci pour chacune des tables.

Le lien entre les différentes tables d'une base de données se fait grâce à une **clef étrangère**.

Il s'agit des valeurs d'un ou des attributs d'une table, qui **fait référence à la clef primaire d'une autre table**.

N.B.: une clef étrangère d'une table est toujours la clef primaire d'une autre table.

Exemple :

BD Vols

PILOTE(PLNUM, PLNOM, PLPRENOM, VILLE, SALAIRE)
AVION(AVNUM, AVNOM, CAPACITE, LOCALISATION)
VOL(VOLNUM, PLNUM, AVNUM, VILLEDEP, VILLEARR, HEUREDEP, HEUREARR)

**PILOTE**

| PLNUM | PLNOM | PLPRENOM | VILLE | SALAIRE |
|---|---|---|---|---|
| 1 | MIRANDA | SERGE | PARIS | 21000 |
| 2 | LETHANH | NAHN | TOULOUSE | 21000 |
| … | TALADOIRE | GILLES | NICE | 18000 |
| … | … | … | … | … |

Clé primaire (unique, non nulle)

**AVION**

| AVNUM | AVNOM | CAPACITE | LOCALISATION |
|---|---|---|---|
| 1 | A300 | 300 | NICE |
| 2 | A310 | 300 | NICE |
| 3 | B707 | 250 | PARIS |
| … | … | … | … |

**VOL**

| VOLNUM | PLNUM | AVNUM | VILLEDEP | VILLEARR | HEUREDEP | HEUREARR |
|---|---|---|---|---|---|---|
| 100 | 1 | 1 | NICE | TOULOUSE | 11:00 | 12:30 |
| 101 | 1 | 2 | PARIS | TOULOUSE | 17:00 | 18:30 |
| 102 | 2 | 1 | TOULOUSE | LYON | 14:00 | 16:00 |
| … | … | … | | | | … |

Clés étrangères (référence)

14

*Pour qu'une base de données serve à quelque chose, on doit être en mesure d'accéder à la BDD **en consultation** ou **en écriture**. Pour cela, il faut écrire un ou plusieurs programmes qui seront chargés de lancer les requêtes nécessaires. Ces programmes seront lancés notamment lorsque l'utilisateur lance une recherche (par exemple lorsque sur une appli de musique, vous voulez trouver toutes les versions disponibles du titre « Allez les verts »).*

*Dans ce cours nous allons donc apprendre à **lancer des requêtes** sur une base de données déjà créée. L'écriture dans une base de données n'est pas au programme.*

## II) Requêtes dans une base de données comportant une seule table

Le langage utilisé pour faire des requêtes dans une base de données relationnelle est le **SQL**, pour ***Structured Query Language*** (langage de requête structuré).
Les logiciels permettant (via SQL) la gestion et l'interrogation de bases de données s'appellent des systèmes de gestions de bases de données (SGBD).

Cette année, nous utiliserons ***https://sqliteonline.com,*** qui est (comme son nom l'indique) une façon de travailler en ligne avec une base de données située sur votre ordinateur.

### 1) La commande SELECT…FROM… et tous les mot-clefs associés

Toutes les requêtes dans une base de données commencent par la commande SELECT… FROM…, à laquelle on peut ajouter des commandes optionnelles permettant de sélectionner/trier les résultats voulus.

#### a) Syntaxe de base

Une requête SELECT dans une base de données se fait selon la syntaxe suivante :

```
SELECT attribut FROM table
```

ou

```
SELECT liste d'attributs FROM table
```

ou

```
SELECT * FROM table
```

- L'astérisque * signifie l'ensemble des attributs : on récupère donc la totalité de la table.
- La commande SELECT peut potentiellement afficher des lignes en double ; pour éviter des redondances dans le résultat, il faut utiliser **SELECT DISTINCT** . Dans ce cas les lignes qui font doublons ne seront pas affichées.

Exemple sur la base de données vols :
Pour obtenir la liste de tous les noms d'avions en activité :

```
SELECT AVNOM FROM AVION
```

mais il y aura des redondances si plusieurs avions du même nom sont en circulation (s'il y a deux A320 par exemple) ; on demandera donc plutôt :

```
SELECT DISTINCT AVNOM FROM AVION
```

#### b) L'option WHERE

L'option WHERE permet de garder, parmi les multiplets de la relation considérée, seulement ceux qui vérifient une condition donnée, selon la syntaxe suivante

```
SELECT attribut, ou liste d'attributs, ou * FROM table WHERE condition
```

- La condition est typiquement de la forme « attribut=valeur », où = peut être remplacé par != (différent de) et les comparaisons >, <, >= et <=.
- La condition peut contenir des liens logiques **AND, OR** ou **NOT**, voire une combinaison des trois (attention à la place des parenthèses)

Exemple : Voici le début d'une table « Pays »

| Nom | Continent | Capitale | Indépendance | Population | Superficie | PIB/hab. |
|---|---|---|---|---|---|---|
| Afghanistan | Asie | Kaboul | 1919 | 34 859 568 | 647 500 | 585 |
| Afrique du Sud | Afrique | Pretoria | 1910 | 55 653 654 | 1 219 912 | 6 160 |
| … | … | … | … | … | … | … |

Pour obtenir toutes les informations sur les pays d'Europe, on utilisera :

```
SELECT * FROM Pays WHERE Continent=« Europe »
```

Pour obtenir le nom des pays d'Europe de moins de 10 millions d'habitants, on utilisera :

```
SELECT Nom FROM Pays WHERE Continent=« Europe » AND Population<=10000000
```

Exercice : Ecrire la requête pour obtenir le nom des pays qui ne sont pas en Afrique et qui ont une superficie supérieure ou égale à un million de km2.

On peut aussi faire une recherche suivant un modèle sur les valeurs d'un attribut, grâce à l'opérateur **LIKE**, associé au joker **%**, qui représente aucun, un ou plusieurs caractères.

Exemple :

```
SELECT Nom, Capitale FROM Pays WHERE Capitale LIKE «K% »
```

*va retourner le nom et la capitale de tous les pays dont la capitale commence par la lettre K.*

On peut enfin spécifier l'appartenance à un groupe de valeurs grâce aux instructions **IN** et **NOT IN**

Exemple :

```
SELECT Nom FROM Pays WHERE continent IN (« Europe », « Asie »)
```

### c) Opérations élémentaires sur les données

Une requête SELECT peut porter sur des opérations algébriques élémentaires (c'est-à-dire sommes, produits, différences, quotients) entre attributs, lorsque ceux-ci sont de type numérique.

Pour améliorer l'affichage du résultat, on peut procéder au renommage du calcul effectué, via la syntaxe :

```
SELECT calcul_sur_attributs AS nouveau_nom FROM table
```

Exemple : Avec la table « Pays »

| Nom | Continent | Capitale | Indépendance | Population | Superficie | PIB/hab. |
|---|---|---|---|---|---|---|
| Afghanistan | Asie | Kaboul | 1919 | 34 859 568 | 647 500 | 585 |
| Afrique du Sud | Afrique | Pretoria | 1910 | 55 653 654 | 1 219 912 | 6 160 |
| … | … | … | … | … | … | … |

Qu'effectue la requête suivante ?

```
SELECT Nom, Population/Superficie AS 'densité' FROM Pays
```

Remarque : Le renommage des attributs sans calcul est également possible.

### d) Fonctions d'agrégation

Pour faire des statistiques élémentaires sur les bases de données, le SQL dispose des fonctions d'agrégation MIN, MAX, SUM, AVG (moyenne, *average* en anglais) et COUNT.
On parle de fonctions d'agrégations car à partir des champs de plusieurs lignes, on obtient **une seule valeur**.
L'option de renommage est possible pour ces fonctions.
Exemple : Avec la table « Pays », qu'effectue la requête suivante ?

```
SELECT MIN(Superficie), MAX(Superficie) FROM Pays WHERE Continent='Asie'
```

Plus de précisions sur la fonction COUNT :

- COUNT compte le nombre de lignes que donne la requête.
- Si on veut le nombre total de lignes COUNT(*) le fait.
- Si l'on précise un attribut, par exemple Continent, COUNT(Continent) donne le nombre de lignes tel que ce champ n'est pas NULL.
  Remarquons que le résultat ici ne sera pas intéressant, puisque les continents seront comptés plusieurs fois ; pour éviter ceci, on utilise le mot clef **DISTINCT** :
- Qu'effectue la requête suivante ? `SELECT COUNT(DISTINCT Continent) FROM Pays`

Exemple : Ecrire une requête permettant de connaître le nombre de pays européen présents dans cette table.

Groupements : la fonction GROUP BY
On peut ***appliquer les fonctions d'agrégation à des groupements*** (et non pas à toutes les valeurs d'un attribut), pour cela on utilise l'instruction GROUP BY.
Par exemple si on veut récupérer le nombre total d'habitants par continents :

```
SELECT Continent , SUM(Population) FROM Pays GROUP BY Continent
```

Filtrage des agrégats : la fonction HAVING permet de filtrer les résultats que l'on a obtenus en appliquant une fonction d'agrégation. (car on ne peut pas utiliser un WHERE pour une condition qui porte sur le résultat d'une fonction d'agrégation)

```
SELECT colonne1, SUM(colonne2) AS Total FROM table
  GROUP BY colonne1   HAVING condition sur Total
```

Exemple : Avec la table « Pays »

| Nom | Continent | Capitale | Indépendance | Population | Superficie | PIB/hab. |
|---|---|---|---|---|---|---|
| Afghanistan | Asie | Kaboul | 1919 | 34 859 568 | 647 500 | 585 |
| Afrique du Sud | Afrique | Pretoria | 1910 | 55 653 654 | 1 219 912 | 6 160 |
| … | … | … | … | … | … | … |

Ecrire une requête permettant de faire afficher chaque continent et sa population totale, uniquement pour les continents qui ont une population totale supérieure ou égale à un milliard d'habitants :

Si on ne veut pas afficher la colonne sur laquelle opère le HAVING, on peut faire figurer la fonction d'agrégation seulement dans le HAVING (et pas dans le SELECT), avec la syntaxe suivante :

```
SELECT colonne1 FROM table
GROUP BY colonne1
HAVING condition sur SUM(colonne2)
```

Ecrire une requête permettant de faire afficher le nom des continents ayant une population totale supérieure ou égale à un milliard d'habitants :

### e) Sous-requêtes

On peut réutiliser le résultat d'une requête, sachant que le résultat d'une requête SQL qui ne fournit qu'un champ peut être identifié avec son unique valeur et utilisé comme tel.

Exemple : Avec la table « Pays », qu'effectue la requête suivante ?

```
SELECT Nom FROM Pays WHERE PNB>=(SELECT AVG(PNB) FROM Pays)
```

## 2) Affichage des résultats d'une requête

- La commande ORDER BY permet de trier les lignes dans un résultat d'une requête SQL. Il est possible de trier les données sur une ou plusieurs colonnes, par ordre ascendant ou descendant.

  ```
  SELECT colonne1, colonne2 FROM table ORDER BY colonne1
  ```

  Par défaut les résultats sont classés par ordre ascendant (=croissant), toutefois il est possible d'inverser l'ordre en utilisant le suffixe DESC après le nom de la colonne. Par ailleurs, il est possible de trier sur plusieurs colonnes en les séparant par une **virgule**. Une requête plus élaborée ressemblerait à cela :

  ```
  SELECT colonne1, colonne2, colonne3 FROM table
  ORDER BY colonne1 DESC, colonne2 ASC
  ```

- La clause LIMIT est à utiliser dans une requête SQL pour spécifier le nombre maximum de résultats que l'on souhaite obtenir.

  ```
  SELECT * FROM table LIMIT 10
  ```

  Cette requête permet de récupérer seulement les 10 premiers résultats d'une table. Bien entendu, si la table contient moins de 10 résultats, alors la requête retournera toutes les lignes.
  La bonne pratique lorsque l'on utilise LIMIT consiste à utiliser également la clause ORDER BY pour s'assurer que ce sont toujours les bonnes données qui sont présentées. En effet, sans instruction spécifique, l'ordre d'affichage des résultats est imprévisible.

- L'utilisation de OFFSET permet d'ignorer certains résultats : avec `OFFSET m` on ignore les m premiers résultats. La syntaxe pour utiliser une limite et un offset est la suivante :

  ```
  SELECT colonne1 FROM table ORDER BY colonne1 LIMIT 10 OFFSET 1
  ```

  Cette requête permet d'ignorer la première ligne et d'afficher 10 lignes.

  Ecrire `OFFSET 0` revient au même que de ne rien écrire.

### 3) Remarques sur le langage SQL

- A la différence du langage de programmation Python, le SQL n'est sensible ni à la casse, ni à l'indentation. L'usage veut que les commandes soient écrites en majuscule.
- Notons que SQL est un ***langage déclaratif***, c'est-à-dire que l'***on dit au SGBD ce que l'on veut faire***, mais on ne lui dit pas (vraiment) *comment* il doit le faire. En interne, l'interpréteur va s'arranger pour optimiser la requête pour obtenir le résultat en un minimum de temps.

## III) Requêtes dans une base de données comportant plusieurs tables

Il s'agit d'obtenir des informations portant sur plusieurs relations d'une même base de données.

### 1) Union, intersection, différence

#### a) Commande UNION

L'**union** de deux tables est une table contenant chaque ligne de la première table *et* chaque ligne de la seconde table.

Le nombre des colonnes sélectionnées dans chacune des deux tables doit être le même, mais les champs que l'on fait correspondre dans les deux tables n'ont pas besoin de porter les mêmes noms ni de se présenter dans le même ordre. (Cependant attention aux résultats qui n'ont aucun sens)

On peut également sélectionner seulement certaines lignes de chacune des tables grâce à la commande WHERE.

C'est donc une commande qui permet de **concaténer les résultats de deux requêtes** (ou plus).

Remarque : par défaut, les lignes exactement identiques ne seront pas répétées dans les résultats. Pour effectuer une union dans laquelle même les lignes dupliquées sont affichées il faut utiliser la commande UNION ALL.

Syntaxe :

```
SELECT attribut, ou liste d'attributs, ou * FROM table1 WHERE conditions
                                UNION
SELECT attribut, ou liste d'attributs, ou * FROM table2 WHERE conditions
```

Exemple : Imaginons une entreprise qui possède deux magasins et dans chacun de ces magasins il y a une table qui liste les clients.

La table du magasin n°1 s'appelle « mag1_client » et contient les données suivantes :

| prenom | nom | ville | date_naissance |
|---|---|---|---|
| Léon | Dupuis | Paris | 1983-03-06 |
| Marie | Bernard | Paris | 1993-07-03 |
| Sophie | Dupond | Marseille | 1986-02-22 |
| Marcel | Duron | Paris | 1976-11-24 |

La table du magasin n°2 s'appelle « mag2_client » et contient les données suivantes :

| prenom | nom | ville | date_naissance |
|---|---|---|---|
| Marion | Leroy | Lyon | 1982-10-27 |
| Paul | Moreau | Lyon | 1976-04-19 |
| Marie | Bernard | Paris | 1993-07-03 |
| Marcel | Duron | Paris | 1976-11-24 |

```
SELECT * FROM mag1_client
UNION
SELECT * FROM mag2_client
```

→

| prenom | nom | ville | date_naissance |
|---|---|---|---|
| Léon | Dupuis | Paris | 1983-03-06 |
| Marie | Bernard | Paris | 1993-07-03 |
| Sophie | Dupond | Marseille | 1986-02-22 |
| Marcel | Duron | Paris | 1976-11-24 |
| Marion | Leroy | Lyon | 1982-10-27 |
| Paul | Moreau | Lyon | 1976-04-19 |

Le résultat de cette requête montre bien que les enregistrements des 2 requêtes sont mis à la suite des uns des autres mais sans inclure plusieurs fois les mêmes lignes.

*Traduire avec une phrase la requête suivante. Combien de lignes comporte le résultat ?*

```
SELECT prenom,nom FROM mag1_client WHERE date_naissance>=1980-01-01
UNION
SELECT prenom,nom FROM mag2_client WHERE date_naissance>=1980-01-01
```

### b) Commande INTERSECT

La commande INTERSECT permet d'obtenir l'intersection des résultats de deux requêtes.
Cette commande permet donc de récupérer les lignes communes à deux requêtes.
Cela peut s'avérer utile lorsqu'il faut trouver s'il y a des données similaires sur deux tables distinctes.
Pour l'utiliser il est nécessaire que les deux requêtes retournent le même nombre de colonnes.
Syntaxe :

```
SELECT attribut, ou liste d'attributs, ou * FROM table1
                    INTERSECT
SELECT attribut, ou liste d'attributs, ou * FROM table2
```

Exemple :*Traduire avec une phrase la requête suivante. Combien de lignes comporte le résultat ?*

```
SELECT * FROM mag1_client
INTERSECT
SELECT * FROM mag2_client
```

### c) Commande EXCEPT

La commande EXCEPT permet de récupérer les lignes de la première requête sans inclure les lignes de la deuxième. (d'un point de vue ensembliste il s'agit de la ***différence entre deux ensembles***)

Pour l'utiliser il est nécessaire que les deux requêtes retournent le même nombre de colonnes.
Syntaxe :

```
SELECT attribut, ou liste d'attributs, ou * FROM table1
                    EXCEPT
SELECT attribut, ou liste d'attributs, ou * FROM table2
```

Exemple :*Traduire avec une phrase la requête suivante. Combien de lignes comporte le résultat ?*

```
SELECT * FROM mag1_client
EXCEPT
SELECT * FROM mag2_client
```

## 2) Produit cartésien et jointure

Nous allons ici utiliser **une même requête SELECT sur plusieurs tables en même temps**.

***Le produit cartésien de deux tables*** est une table obtenue en accolant à chaque ligne de la première table l'ensemble des lignes de la seconde.
Si chacune des deux tables contient un grand nombre d'enregistrements, le résultat du produit est encore plus grand (il aura un nombre de lignes égal au produit du nombre de lignes de la première table par le nombre de lignes de la deuxième), et généralement dénué de sens.
La figure ci-dessous illustre l'opération de produit cartésien :

**VOL**

| VOLNUM | PLNUM | AVNUM | VILLEDEP | VILLEARR | HEUREDEP | HEUREARR |
|---|---|---|---|---|---|---|
| 100 | 1 | 1 | NICE | TOULOUSE | 11:00 | 12:30 |
| 101 | 1 | 2 | PARIS | TOULOUSE | 17:00 | 18:30 |
| 102 | 2 | 1 | TOULOUSE | LYON | 14:00 | 16:00 |
| … | … | … | … | … | … | … |

**AVION**

| AVNUM | AVNOM | CAPACITE | LOCALISATION |
|---|---|---|---|
| 1 | A300 | 300 | NICE |
| 2 | A310 | 300 | NICE |
| 3 | B707 | 250 | PARIS |
| … | … | … | … |

Ce produit cartésien aura un résultat qui ressemble à ça :

| VOLNUM | PLNUM | AVNUM | VILLEDEP | VILLEARR | HEUREDEP | HEUREARR | AVNUM | AVNOM | CAPACITE | LOCALISATION |
|---|---|---|---|---|---|---|---|---|---|---|
| 100 | 1 | 1 | NICE | TOULOUSE | 11:00 | 12:30 | 1 | A300 | 300 | NICE |
| 100 | 1 | 1 | NICE | TOULOUSE | 11:00 | 12:30 | 2 | A310 | 300 | NICE |
| 100 | 1 | 1 | NICE | TOULOUSE | 11:00 | 12:30 | 3 | B707 | 250 | PARIS |
| 101 | 1 | 2 | PARIS | TOULOUSE | 17:00 | 18:30 | 1 | A300 | 300 | NICE |
| 101 | 1 | 2 | PARIS | TOULOUSE | 17:00 | 18:30 | 2 | A310 | 300 | NICE |
| 101 | 1 | 2 | PARIS | TOULOUSE | 17:00 | 18:30 | 3 | B707 | 250 | PARIS |
| 102 | 2 | 1 | TOULOUSE | LYON | 14:00 | 16:00 | 1 | A300 | 300 | NICE |
| 102 | 2 | 1 | TOULOUSE | LYON | 14:00 | 16:00 | 2 | A310 | 300 | NICE |
| 102 | 2 | 1 | TOULOUSE | LYON | 14:00 | 16:00 | 3 | B707 | 250 | PARIS |

Remarquons que ce tableau n'a aucun intérêt ainsi.
Pour que le résultat renvoyé ait du sens, il faut seulement garder les lignes où les numéros d'avions correspondent, c'est-à-dire là où ***la clef étrangère*** AVNUM de la table VOL est la même que ***la clef primaire*** AVNUM de la table AVION
Ceci est possible grâce à la syntaxe suivante

```
SELECT attribut(s) FROM table1 JOIN table2 ON condition
```

Remarques :

- La condition qui vient après **ON** est généralement une égalité du type
  *table1.identifiant=table2.identifiant.*

- Lorsque deux tables ont des attributs identiques (du style *identifiant, nom…*), on différencie ceux-ci en utilisant les syntaxes *table1.attribut* et *table2.attribut*.

Exemple : **SELECT** VOLNUM,PLNUM,VOL.AVNUM,AVNOM,CAPACITE
**FROM** VOL **JOIN** AVION **ON** VOL.AVNUM=AVION.AVNUM

*retourne :*

| VOLNUM | PLNUM | VOL.AVNUM | AVNOM | CAPACITE |
|---|---|---|---|---|
| 100 | 1 | 1 | A300 | 300 |
| 101 | 1 | 2 | A310 | 300 |
| 102 | 2 | 1 | A300 | 300 |

En créant ainsi des n-uplets grâce à deux tables et une condition pertinente pour que ces n-uplets aient un sens, on a réalisé ce que l'on appelle **une jointure** entre les tables.

Exemple : Toujours en utilisant la base de données Vols, écrire une requête permettant d'obtenir le nom de tous les pilotes qui effectuent un vol au départ de Nice :

### 3) Jointure avec trois tables ou plus

La syntaxe est la suivante :

```
SELECT attribut(s)
FROM Table1 JOIN Table2 ON Table1.colonne1=Table2.colonne1
JOIN Table3 ON Table1.colonne2=Table3.colonne1
```

### 4) Auto-jointure

Une auto-jointure est la jointure d'une table avec elle-même.
Pour réaliser une auto-jointure, un renommage des tables est obligatoire, puisque tous les attributs sont ambigus. Pour donner un alias à une table, on note dans la clause FROM l'alias après le nom de la relation : FROM nom_table AS alias.

```
SELECT attribut(s) FROM table AS T1 JOIN table AS T2 ON T1.nom=T2.nom
```

Remarque : de manière générale, pour créer un alias d'une table ou d'un attribut, le mot-clef AS est optionnel (cependant je vous conseille de l'écrire pour plus de lisibilité), on pourrait donc écrire

```
SELECT attribut(s) FROM table T1 JOIN table T2 ON T1.nom=T2.nom
```

Exemple : Considérons une entreprise qui possède la table de ces employés (appelée `table_emp`) décrivant la hiérarchie entre ces employés : les employés peuvent être dirigé par un supérieur direct qui se trouve lui-même dans la table.

| id | prenom | nom | email | manager_id |
|---|---|---|---|---|
| 1 | Sebastien | Martin | s.martin@example.com | NULL |
| 2 | Gustave | Dubois | g.dubois@example.com | NULL |
| 3 | Georgette | Leroy | g.leroy@example.com | 1 |
| 4 | Gregory | Roux | g.roux@example.com | 2 |

Si l'on veut lister le nom de tous les employés, suivi du nom de leur supérieur direct (dans le cas où ils en ont un), on peut utiliser la requête suivante :

```
SELECT T1.nom AS employé, T2.nom AS supérieur
     FROM table_emp AS T1 JOIN table_emp AS T2    ON T1.manager_id=T2.id
```

Qu'affiche concrètement cette requête avec la table donnée en exemple ci-dessous ?

### 5) Produit cartésien en général

On peut avoir besoin d'effectuer un produit cartésien entre deux tables, qui soit moins restrictif qu'en faisant une équi-jointure comme vue précédemment. (une *équi-jointure* est une jointure utilisant un test d'égalité après le ON, ce qu'on utilisera dans la très grande majorité des cas).

Faire un produit cartésien de façon quelconque est possible mais **attention !** il faudra bien prendre garde à la cohérence des lignes qui seront créées, bien souvent il faudra utiliser une clause WHERE bien choisie pour ne garder que des lignes pertinentes.

La syntaxe pour effectuer un produit cartésien sans restriction est :

```
SELECT attribut(s)  FROM table1 , table2
```

ou bien

```
SELECT attribut(s)  FROM table1 CROSS JOIN table2
```

Exemple : Soient les deux tables suivantes contenant les groupes sanguins d'une population de 4 mâles et 4 femelles souris d'un laboratoire :

| MALES | |
|---|---|
| **Groupe** | **Rhesus** |
| A | - |
| AB | + |
| B | + |
| B | - |

| FEMELLES | |
|---|---|
| **Groupe** | **Rhesus** |
| B | - |
| B | - |
| AB | + |
| A | - |

On souhaite créer une table de tous les mixages possibles afin d'étudier les groupes possibles des bébés souris issus de tous les croisements possibles.

On peut alors écrire : `SELECT * FROM MALES CROISS JOIN FEMELLES`

Voici les premières lignes du résultat obtenu :

| | | | |
|---|---|---|---|
| A | - | B | - |
| A | - | B | - |
| A | - | AB | + |
| A | - | A | - |
| AB | + | B | - |
| AB | + | B | - |
| … | … | … | … |

Ecrire à présent la requête permettant d'obtenir les mixages possibles où les rhésus des deux parents sont différents :