# TP 22
# Langage SQL partie 1

Le **langage SQL** permet de faire des recherches dans une base de données. Une vraie base de données peut contenir des millions d'informations rangées dans plusieurs tableaux et il n'est alors pas question de faire défiler les lignes à la main pour trouver l'information souhaitée ; de plus, il s'agit de pouvoir automatiser un certain nombre de tâches demandant à lire ou à enregistrer des données et de communiquer avec la base de données à travers d'autres programmes, par exemple avec un site web qui demande à la base d'enregistrer un nouvel utilisateur. Le mot SQL signifie *Structured Query Language* c'est-à-dire « langage de requêtes structurées ».

Pour l'illustrer nous allons utiliser une base de données contenant des informations sur tous les aéroports du monde, ainsi que les régions et les pays dans lesquels ils se trouvent. Attachez vos ceintures, nous allons décoller !

## I Bases de données

Adoptons le vocabulaire approprié ; dans ce contexte on appelle **table** un tableau, dont les colonnes sont plutôt appelées **attributs** et les lignes sont des **entrées**. En générale, les colonnes sont fixées à l'avance (ce sont les attributs que l'on veut étudier), mais on veut pouvoir rajouter des lignes au fur et à mesure, sans nécessairement qu'il y ait d'ordre entre elles, chacune est donc une nouvelle entrée.

Nous en avons déjà parlé dans le TP sur les prénoms, où nous avions affaire à une table de tous les prénoms contenant quatre attributs (nom, sexe, année de naissance, nombre) et de très nombreuses entrées.

Une base de donnée intéressante peut être constituée de *plusieurs* tables faisant référence les unes aux autres de façon croisée, mais nous étudierons cette situation plus en détail dans le TP d'après. Dans celui-ci, la table des aéroports contient un code de la région où se trouve l'aéroport, et une autre table contient toutes les régions et des informations sur chacune ; on peut donc obtenir des informations simultanément sur l'aéroport et sa région en « croisant les tables ».

## II Le langage SQL

Le logiciel que nous utilisons **DB Browser for SQLite** (*explorateur de base de données pour le langage SQLite*) permet d'ouvrir une base de données, de visualiser le contenu (onglet « Parcourir les données ») et de travailler dessus en exécutant des requêtes SQL (onglet « Exécuter le SQL »). Le résultat de chaque requête sera une table, qui apparaitra en dessous.

Après avoir récupéré et décompressé le fichier `materiel.zip`, aller chercher depuis le logiciel avec « Ouvrir une base de données en lecture seule » (pour éviter de faire des bêtises en modifiant la base) le fichier `airports.db`.

Pour démarrer, testez votre première requête SQL :

```
SELECT name, iso_country FROM airports;
```

Elle signifie de sélectionner seulement les noms et les codes pays dans la table `airports`.

On peut écrire plusieurs requêtes SQL dans la zone de texte en les terminant par un point-virgule `;` et on peut alors demander à n'exécuter que la requête où se trouve le curseur avec l'icône play-pause ⏯, aussi sur le raccourci clavier `Maj` + `F5`

Contrairement au langage Python, les indentations et sauts de ligne n'ont pas d'importance et permettent seulement de rendre le code plus lisible. Les commentaires sont contenus entre les deux symboles : `/* commentaire */`.

La base de données contient trois tables : `airports` avec tous les aéroports, `countries` avec tous les pays, et `regions` avec toutes les régions de tous les pays. Le code ISO des pays est un code standard à deux lettres qui identifie uniquement chaque pays ; le code IATA d'un aéroport est un code à trois lettres qui identifie uniquement les aéroports importants, c'est notamment le code qui est écrit sur les billets d'avion et les bagages. Certaines régions apparaissent comme des pays à part entière, comme les régions d'outre-mer françaises, ce qui a du sens du point de vue d'un pilote…

Ces données sont ouvertes. Source : `https://ourairports.com/data/`.

# III Requêtes simples

Dans tout ce TP nous nous intéressons avant tout aux requêtes permettant d'aller chercher des informations **dans une seule table**. La syntaxe la plus simple, la plus courante, est :

```
SELECT attributs
FROM table
WHERE condition
```

où

- `attributs` désigne un attribut de la table (une colonne), ou plusieurs séparés par des virgules, que l'on veut sélectionner, ou le caractère `*` pour sélectionner tous les attributs d'un coup,
- `table` est le nom de la table dans laquelle on sélectionne,
- `condition` est une condition exprimée sur les attributs.

Les attributs ont un type de donnée : entier, flottant, chaine de caractère (entourées préférablement par des apostrophes simples `'texte'`). Il existe éventuellement d'autres types, permettant par exemple de stocker facilement une date et de comparer rapidement les données selon l'ordre chronologique… Il est important aussi d'avoir une valeur spéciale `NULL` indiquant l'absence de donnée.

La condition peut être formulée avec :

- Des comparaisons mathématiques comme d'habitude : comparaisons strictes `>`, `<`, comparaisons larges `>=`, `<=`, égalité `=`, différence `<>` (attention pour ces deux dernières, ce n'est pas du Python !),
- Les opérations arithmétiques habituelles `+`, `-`, `*`, `/`,
- On peut combiner les conditions avec les mots-clés `AND`, `OR`, `NOT`, et utiliser des parenthèses.

Testez par exemple :

```
SELECT * FROM airports WHERE iso_country='FR';
```

(tous les aéroports français) ou

```
SELECT name FROM airports WHERE type='large_airport' AND iso_region='FR-IDF';
```

(les gros aéroports d'Île-de-France, donner seulement les noms).

**Exercice 1**
Écrire des requêtes SQL pour :

1. Sélectionner le nom de tous les pays.
2. Sélectionner le nom de tous les pays d'Europe.
3. Sélectionner toutes les bases d'hydravion (`seaplane_base`) du monde.
4. Sélectionner toutes les bases d'hydravion en Europe.
5. Sélectionner le nom et les villes des aéroports en France qui sont gros ou moyens.
6. Sélectionner tous les aéroports qui sont en-dessous du niveau de la mer.

Quelques compléments :

- On peut toujours renommer un attribut avec le mot-clé `AS` :

  ```
  SELECT attributs AS nouveau_nom
  ```

Cela fonctionne un peu comme donner un nom à une variable, et aussi comme titre (propre) à la colonne résultat. Le titre peut être encadré entre guillemets doubles.

```
SELECT name AS "Nom", municipality AS "Ville"
FROM airports WHERE type='large_airport' AND iso_region='FR-IDF';
```

- Il se peut qu'après une requête, une entrée apparaisse plusieurs fois. Pour éliminer les doublons, on utilise le mot-clé `SELECT DISTINCT` à la place de `SELECT`. Testez :

```
SELECT DISTINCT continent FROM countries;
```

- Après `WHERE`, on a souvent envie de renvoyer le résultat classé selon un attribut, qui est soit un nombre (ordre usuel) soit une chaine de caractères (ordre lexicographique = ordre du dictionnaire) avec le mot-clé `ORDER BY`, qu'on peut aussi faire suivre de `ASC` (*ascending*, croissant) ou bien `DESC` (*descending*, décroissant). Testez par exemple :

```
SELECT code, name FROM countries ORDER BY code ASC;
```

(noms et codes des pays, classés par code dans l'ordre alphabétique).

**À retenir**

Les requêtes les plus générales vues jusque là ont la forme suivante :

```
SELECT attributs
FROM table
WHERE condition
ORDER BY critère ASC | DESC
```

**Exercice 2**
Écrire des requêtes SQL pour :

1. Afficher les aéroports français moyens ou gros, classés du nord au sud.
2. Afficher les villes des gros aéroports des États-Unis, classés d'ouest en est.
3. (i) Afficher les noms et villes des héliports français, situés à plus de 1000 m d'altitude, sachant que 1 ft = 0,3048 m (sans utiliser sa calculatrice avant),
   (ii) … classés par altitude,
   (iii) … pouvez-vous afficher directement l'altitude en mètres ?
4. Afficher les codes ISO des pays africains contenant un gros aéroport. Sans doublon.
5. (i) Afficher les régions allemandes et leur code ISO, classées par ordre alphabétique.
   (ii) Afficher tous les villes de la région Rhénanie-du-Nord-Westphalie contenant un aéroport.

*Remarque.* Le saviez-vous ? La région Rhénanie-du-Nord-Westphalie est l'une des plus importantes d'Allemagne. La plus peuplée (18,2 million d'habitants), la plus importante économiquement, l'une des plus riches (avec la Bavière). Elle comprend de nombreuses villes très peuplées formant une énorme agglomération Rhin-Ruhr : Bonn, Cologne, Düsseldorf, Duisburg le long du Rhin, et Essen, Bochum, Dortmund, Wuppertal dans la vallée de la Ruhr. La capitale est Düsseldorf, où se trouve aussi un aéroport international, le troisième aéroport allemand après Francfort et Munich.

# IV Les fonctions d'agrégation

On appelle **fonction d'agrégation** une fonction qui calcule un résultat à partir de **toute une colonne**. Ce sont notamment les fonctions suivantes :

- `MIN()` : minimum,
- `MAX()` : maximum,
- `SUM()` : somme,
- `COUNT()` : nombre d'entrées,
- `AVG()` : moyenne (en anglais *average*).

Ce qui peut être surprenant au premier abord, c'est qu'elles s'utilisent **dans** `SELECT`, souvent en leur donnant un nom avec `AS`. Par exemple, si on veut connaitre le nombre total de pays, alors la bonne commande est :

```
SELECT COUNT(id) AS "Nombre total" FROM countries;
```

(on peut remplacer ici `id` par un autre des attributs de la table des pays). Et si on veut l'altitude moyenne de tous les aéroports présents :

```
SELECT AVG(elevation_ft) AS "Altitude moyenne" FROM airports;
```

Le résultat est un nombre seul — considéré comme une table avec un seul élément !

*Remarque.* En effet le langage SQL ne manipule fondamentalement que des tables, et le résultat de toute requête est une table. On ne doit pas penser à une table comme à un tableau fixe gravé dans le marbre, mais comme à un fragment de base de donnée qui représente des **relations logiques** entre les données (« cet aéroport est lié à cette ville et à ce pays »). Les tables peuvent être sans cesse filtrées, séparées et recombinées. D'ailleurs, les tables sont enregistrées par le logiciel d'une façon très optimisée pour pouvoir rapidement chercher des données dedans et les recombiner, et qui n'a rien à voir avec l'ordre logique dans lequel un humain écrirait un tableau. S'il est recommandé d'utiliser des `ORDER BY`, c'est que l'ordre des données que le logiciel sort n'a aucune garanti d'être constant.

Pour `COUNT()`, si toutes les entrées ont tous leurs attributs bien remplis, on peut utiliser n'importe lequel des attributs. Mais s'il y a des valeurs `NULL` présentes dans une colonne, elles seront ignorées par `COUNT()`. De même, par exemple le calcul de moyenne avec `AVG()` est fait en ignorant les valeurs `NULL` (ce qui ne donne certainement pas le même résultat que si on complétait avec des zéros !). Éventuellement, la syntaxe `COUNT(*)` permet de compter *toutes* les entrées, qu'elles aient ou non des valeurs manquantes.

Bien sûr, on peut combiner avec un `WHERE`…

**Exercice 3**
Écrire des requêtes SQL pour :

1. Donner l'élévation moyenne de tous les aéroports aux Pays-Bas (*Netherlands*…)
2. Même question pour la Suisse (*Confédération Helvétique*…) et pour le Népal.
3. Donner le nombre de gros aéroports en Europe.
4. Donner le nombre de régions allemandes.

Il se peut que l'on souhaite calculer les fonctions d'agrégation non pas sur *toute* la colonne, mais en regroupant entre elles les entrées selon certains critères, par exemple compter le nombre de pays *par continent*. On utilise alors le mot-clé `GROUP BY attribut`, qui se place après `WHERE` mais avant un éventuel `ORDER BY` :

```
SELECT COUNT(id) AS "Nombre", continent
FROM countries
GROUP BY continent;
```

Après un `GROUP BY`, on peut vouloir ne garder que les entrées satisfaisant une certaine condition qui est elle-même le résultat d'une fonction d'agrégation. On utilise alors la commande `HAVING condition`, qui se place juste après le `GROUP BY`. Ce n'est pas tout à fait la même chose que `WHERE` car `WHERE` agit **avant** la fonction d'agrégation, pour sélectionner seulement ce qui nous intéresse et calculer la fonction dessus, alors que `HAVING` agit **après le calcul**. L'exemple ci-dessous sélectionne les continents avec au moins 20 pays :

```
SELECT COUNT(id) AS "Nombre", continent
FROM countries
GROUP BY continent
HAVING "Nombre" >= 20;
```

**Exercice 4**
Écrire des requêtes SQL pour :

1. Donner le nombre de régions par pays d'Europe.
2. Donner l'élévation moyenne par pays des aéroports gros ou moyens d'Amérique du Sud.
3. Donner les villes de France ayant au moins trois aéroports.
4. Donner les villes d'Europe ayant plusieurs aéroports gros ou moyens.

**À retenir**

En général, la commande `SELECT` est suivie par une liste contenant des attributs ou de fonctions d'agrégations sur les attributs. Les attributs sur lesquels portent une fonction d'agrégation sont souvent renommés avec `AS` et peuvent être utilisés dans un `HAVING`, alors que les attributs seuls doivent se retrouver dans un `GROUP BY`.

On peut donc grossièrement résumer les requêtes les plus générales vues jusque là sous forme :

```
SELECT fonction(attribut1) AS résultat, attribut2
FROM table
WHERE condition /* condition sur la recherche, avant de faire un calcul */
GROUP BY attribut2
HAVING condition /* condition sur le résultat du calcul */
ORDER BY critère ASC | DESC
```

# V Sélection dans plusieurs tables

Parfois, il faut recouper les informations qui sont contenues à travers plusieurs tables. C'est le concept de **jointure** que nous approfondirons au prochain TP.

Par exemple, chaque aéroport est attaché à une région, par un code appelé `iso_region` dans la table `airports`. Les régions se trouvent elles dans une autre table appelée `regions`, dont le code est simplement donné par l'attribut `code`. Pour obtenir des informations simultanément sur les aéroports et leur région, il faut utiliser une jointure. Observez le résultat de la commande suivante :

```
SELECT * FROM airports
JOIN regions ON airports.iso_region=regions.code
WHERE airports.iso_country='FR' AND airports.type='large_airport';
```

Elle se lit comme : sélectionner les aéroports en France, de type large, et y joindre les informations de la table des régions, quand coïncident les codes de région (celui de l'aéroport défini dans la table `airports`, avec celui des régions défini dans la table `regions`).

Comme on manipule les attributs de plusieurs tables en même temps, on les note `table.attribut` pour bien les distinguer, mais on peut aussi s'en passer en utilisant le renommage `AS`.

Le résultat de la commande précédente serait plus lisible si on sélectionnait seulement ce qui nous intéresse :

```
SELECT airports.name AS "Nom", airports.municipality AS "Ville", regions.name AS "Région"
FROM airports
JOIN regions ON airports.iso_region=regions.code
WHERE airports.iso_country='FR' AND airports.type='large_airport';
```

En général le mot-clé `JOIN` fonctionne ainsi :

```
SELECT attributs
FROM table
JOIN autre_table
ON critère
```

où en général `critère` est simplement l'égalité entre un attribut de la première table et un attribut de la seconde.

Bien entendu, on peut rajouter à tout cela les `WHERE`, `ORDER BY`, les agrégations et autres.

**Exercice 5**
Écrire des requêtes SQL pour :

1. Donner la liste des gros aéroports européens et leur région.
2. Donner les gros aéroports d'Asie et leur pays, classés d'ouest en est.
3. Un voyageur a un billet pour SCL. Dans quelle ville et quel pays se rend-il ?
4. Un voyageur a un billet pour DPS. Dans quel pays et quelle région se rend-il ?
5. (i) Donner le classement des pays avec le plus de codes IATA.

   (ii) … en ne gardant que ceux avec plus de 100 codes.
6. Donner le classement des pays du monde dont l'altitude moyenne des aéroports est la plus élevée.

# VI Annexe

**Manipulation des bases de données** Nous nous sommes intéressé au minimum vital pour effectuer une requête SQL. Le langage SQL permet également d'*insérer* des entrées dans une base de données (mot-clé `INSERT`), de créer des tables (`CREATE TABLE`), de gérer en entier une base de données à travers des programmes informatique.

En général le SQL ne s'utilise pas ainsi tout seul, car on ne va pas insérer les entrées de la table unes par unes avec des requêtes SQL… Il est utilisé conjointement avec d'autres langages ou d'autres programmes pour en récupérer les données, faire des calculs dessus ou les représenter graphiquement. Il y a par exemple des fonctions Python permettant d'ouvrir une base de données et de lancer des requêtes SQL dessus, puis récupérer les résultats dans des listes Python, et appliquer tout ce que l'on veut ensuite. Les sites internet utilisent aussi du SQL côté serveur, par exemple un utilisateur se connecte et il faut demander à la base d'aller chercher les informations de l'utilisateur. C'est donc un programme qui communique avec un autre *via* le langage SQL.

**Optimisation des bases de données** Les vraies bases de données peuvent contenir des millions d'entrées, ainsi chaque requête pourrait demander beaucoup de travail à l'ordinateur. Une grosse part du travail de conception de bases de données, et des logiciels qui vont avec, est d'optimiser les données pour pouvoir travailler le plus efficacement avec. Les données sont très loin d'être enregistrées les unes à la suite des autres comme on peut le visualiser (lire par exemple l'annexe du TP sur les dictionnaires et les tables de hachage).

De plus le but de jeu est toujours d'écrire le minimum possible de requêtes SQL et que chacune soit la plus *efficace* possible, c'est ainsi qu'on utilise au mieux les possibilités d'optimisation des bases de données. Par exemple les fonctions d'agrégations fournies par le langage SQL sont aussi bien plus efficaces, quand on les utilise correctement, que de récupérer les données avec un autre programme par exemple en Python *puis* de faire les calculs dessus.

**Quelques ressources** Pour s'entrainer tout seul :

- Des exercices interactifs dans diverses tables : `https://sql-exercices.github.io/`
- Cours à la fois bien complet et interactif (mais en anglais) : `https://www.w3schools.com/sql/`
- Exercices interactifs (en anglais) : `https://sqlzoo.net/`