Maintenant que nous avons vu la théorie, nous allons passer à la pratique et voir comment traduire ces opérations en SQL. Nous allons donc étudier la partie du SQL qui permet de manipuler des tables : y insérer des données, les sélectionner, etc. Cette portion du SQL est appelée le SQL DML : *Data Manipulation Language*. Grosso-modo, celui-ci définit plusieurs instructions relativement simples, et nous allons nous concentrer sur :

- SELECT : Sélection de données,
- JOIN : Jointure de plusieurs sources,
- INSERT : Insertion de données,
- DELETE : Suppression de données,
- UPDATE : Mise à jour de données existantes.

SELECT...FROM...WHERE

Si on devait se fier uniquement à son nom, l'instruction SELECT devrait effectuer une sélection. Dans la réalité du SQL, elle est plus puissante et permet de faire aussi une projection, un ou plusieurs produit cartésien, et quelques autres opérations annexes. La syntaxe de l'opération SELECT est relativement complexe : elle est composée de plusieurs clauses, qui s'écrivent ainsi :

```
SELECT colonne
FROM table
WHERE condition
ORDER BY attribut
GROUP BY attribut
HAVING condition
LIMIT nombre_de_lignes ;
```

SELECT

Dans le cas le plus simple, la syntaxe de SELECT est la suivante :

```
SELECT nom_colonnes /* les noms des colonnes sont séparés par des virgules */
FROM nom_table ;
```

Le `SELECT` effectue une projection : les noms des colonnes indiqués après le SELECT correspondent aux colonnes qui seront conservées, les autres étant éliminées par la projection. Le `FROM` indique dans quelle table il faut effectuer la projection et la sélection. On peut se limiter à faire une projection, ce qui fait que la syntaxe suivante est parfaitement possible :

```
SELECT nom, prénom
FROM personne ;
```

Doublons

On peut préciser qu'il est possible que certaines lignes donnent des doublons dans la table. Par exemple, rien n'empêche que plusieurs personnes différentes aient le même nom : une projection de la colonne nom dans une table de personne donnera fatalement quelques doublons. Mais il est possible d'éliminer ces doublons en utilisant un mot-clé. Ce mot-clé `DISTINCT` doit être placé entre le `SELECT` et les noms des colonnes.

```
SELECT DISTINCT nom_colonnes
FROM nom_table ;
```

Par exemple, prenons cette table :

```
CREATE TABLE Animal (
    id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
    espece VARCHAR(40) NOT NULL,
    sexe CHAR(1),
    date_naissance DATETIME NOT NULL,
    nom VARCHAR(30),

    PRIMARY KEY (id)
)
```

Voici la requête SQL pour obtenir une table dans laquelle on ne récupérerait que les noms des animaux (sans doublons) :

```
SELECT DISTINCT nom
FROM Animal
```

Voici la requête SQL pour obtenir une table dans laquelle on ne récupérerait que les noms et les dates de naissance des animaux (sans doublons) :

```
SELECT DISTINCT nom , date_naissance
FROM Animal
```

Omettre la projection

Si jamais vous souhaitez conserver toutes les colonnes lors d'une projection, vous devez utiliser la syntaxe suivante :

```
SELECT *
FROM Nom_table
WHERE condition
```

Par exemple, la requête `SELECT` suivante sélectionne toutes les lignes de la table personne où l'attribut `prenom` vaut "Jean", sans faire de projection.

```
SELECT *
FROM personne
WHERE (prenom = "Jean")
```

Agrégations

Dans une requête `SELECT`, il est possible d'utiliser certaines fonctions qui permettent respectivement de :

- faire la somme de tous les attributs d'une colonne (leur somme) ;
- déterminer quelle est la plus petite valeur présente dans la colonne (son minimum) ;
- déterminer quelle est la plus grande valeur présente dans la colonne (son maximum) ;
- déterminer quelle est la moyenne de la colonne ;
- déterminer quelle est le nombre de lignes de la colonne projetée.

Ces fonctions sont respectivement nommées :

- SUM (somme d'une colonne) ;
- AVG (moyenne d'une colonne) ;
- MIN (minimum d'une colonne) ;
- MAX (maximum d'une colonne) ;
- COUNT (nombre de lignes d'une colonne).

Par exemple, imaginons que vous ayez une table Personne contenant une liste de personnes, avec un attribut/colonne age et un autre attribut pour le nom. Supposons que vous souhaitiez calculer la moyenne d'âge des personnes de cette liste : vous pouvez le faire en utilisant cette requête :

```
SELECT AVG(age)
FROM Personne ;
```

Comme autre exemple, vous pouvez prendre la personne la plus âgée en utilisant la requête suivante :

```
SELECT MAX(age)
FROM Personne ;
```

De même, vous pouvez savoir combien de noms différents contient la table en utilisant cette requête :

```
SELECT DISTINCT COUNT(nom)
FROM Personne ;
```

WHERE

Si on veut effectuer une sélection, c'est cette syntaxe qu'il faut utiliser

```
SELECT nom_colonnes
FROM nom_table
WHERE condition_selection ;
```

Le terme `WHERE`, facultatif, permet de faire une sélection : il précise la condition que les lignes doivent respecter pour être conservées dans la table résultat.

Comparaisons de base

Les tests autorisés en SQL sont les conditions les plus basiques :

- a < b ;
- a > b ;
- a <= b ;
- a >= b ;
- a = b ;
- a <> b.

De plus, les opérateurs booléens `AND`, `OR` et `NOT` sont possibles (avec le `XOR` en plus).

Les comparaisons vues plus haut ne permettent pas de vérifier si le contenu d'une colonne est `NULL`. Ainsi, une condition du type `age = NULL` ne sera tout simplement pas accepté par le SGBD. Pour vérifier si une colonne contient ou non la valeur `NULL`, il faut utiliser des conditions spéciales, notées respectivement :

- IS NULL ;
- IS NOT NULL.

Pour donner un exemple, la requête suivante renvoie toutes les personnes dont l'âge est inconnu :

```
SELECT *
FROM Personne
WHERE age IS NULL ;
```

Comme autre exemple, la requête suivante renvoie toutes les personnes dont l'adresse est connue :

```
SELECT *
FROM Personne
WHERE adresse IS NOT NULL ;
```

Condition intervallaire

Il est aussi possible de vérifier si tel attribut est compris dans un intervalle bien précis. Pour cela, la condition s'écrit comme ceci :

```
SELECT ...
FROM ...
WHERE attribut BETWEEN minimum AND maximum ;
```

Par exemple, cette requête renvoie les lignes de la table personne qui comprend les personnes dont l'âge est compris entre 18 et 62 ans (les personnes en âge de travailler) :

```
SELECT *
FROM Personne
WHERE age BETWEEN 18 AND 62 ;
```

Cela fonctionne aussi avec les chaînes de caractère ou les dates. Par exemple, la requête suivante renvoie toutes les personnes qui sont nées entre le 2 juillet 2000 et le 3 octobre 2014 :

```
SELECT *
FROM Personne
WHERE date_naissance BETWEEN '2000-07-02' AND '2014-10-3' ;
```

La condition inverse, qui vérifie que l'attribut n'est pas dans l'intervalle existe aussi : c'est la condition `NOT BETWEEN`. Elle s'utilise comme la condition `BETWEEN`. Par exemple, cette requête renvoie les lignes de la table personne qui comprennent les personnes dont l'âge n'est pas compris entre 18 et 62 ans (les personnes en âge de travailler) :

```
SELECT *
FROM Personne
WHERE age NOT BETWEEN 18 AND 62 ;
```

Exemples

Prenons maintenant la table PERSONNE définie au-dessus.

Voici comment récupérer les personnes dont on connaît l'âge :

```sql
SELECT *
FROM personne
WHERE (age IS NOT NULL) ;
```

Voici comment récupérer les personnes dont on ne connaît pas la taille :

```sql
SELECT *
FROM personne
WHERE (taille IS NULL) ;
```

Maintenant, prenons la table suivante :

```sql
create table PERSONNE
(
    ID_PERSONNE int not null ,
    NOM varchar not null ,
    PRENOM varchar not null ,
    AGE int ,
    CATEGORIE_PROFESSIONNELLE varchar ,
    NOMBRE_ENFANTS int ,
    TAILLE float ,

    primary key (ID_PERSONNE) ,
) ;
```

Voici la requête pour récupérer toutes les personnes dont la taille est supérieure à 170 centimètres :

```sql
SELECT *
FROM personne
WHERE (taille > 170) ;
```

Et voici la requête pour récupérer les noms et prénoms des personnes qui ont la majorité (dont l'âge est égal à 18 ans) :

```sql
SELECT nom, prénom
FROM personne
WHERE (age = "18") ;
```

ORDER BY

On peut demander au SGBD de trier les données dans la table résultat, que ce soit dans l'ordre croissant ou dans l'ordre décroissant. Pour cela, il faut utiliser l'instruction `ORDER BY`, juste en dessous du `WHERE`. Cette instruction `ORDER BY` a besoin de plusieurs informations pour fonctionner :

- quelles sont les colonnes à prendre en compte lors du tri ;
- dans quel ordre les prendre en compte ;
- faut-il trier chaque colonne par ordre croissant ou décroissant.

Pour cela, `ORDER BY` est suivi d'un ou de plusieurs noms de colonne. De plus, chaque nom de colonne est suivi d'un mot-clé qui indique s'il faut trier dans l'ordre croissant ou décroissant : ces mot-clés sont respectivement les mot-clés `ASC` et `DESC` (pour Ascendant et Descendant).

Par exemple, la requête suivante sélectionne les personnes majeures de la table Personne, les personnes de la table résultat étant triée de l'âge le plus petit vers l'âge le plus grand.

```sql
SELECT *
FROM Personne
WHERE age > 18 AND age IS NOT NULL
ORDER BY age ASC ;
```

Par contre, la requête suivante trie les résultats par âge décroissant.

```sql
SELECT *
FROM Personne
WHERE age > 18 AND age IS NOT NULL
ORDER BY age DESC ;
```

FROM

Maintenant que nous avons étudié en détail la requête `SELECT` et la condition `WHERE`, nous allons nous pencher plus en détail sur la directive `FROM`. Dans les exemples précédents, nous avons vu que cette directive permet de préciser la table sur laquelle nous voulons effectuer la requête. Mais dans les faits, `FROM` est aussi plus puissant que prévu : il permet aussi d'effectuer des produits cartésiens entre plusieurs tables et des jointures. Il permet aussi d'effectuer ce qu'on appelle des sous-requêtes. Tout ce qu'il faut retenir, c'est que l'expression qui suit `FROM` doit avoir pour résultat une table, cette table étant celle sur laquelle on effectue la requête.

Produit cartésien

Effectuer un produit cartésien est relativement simple : il suffit d'indiquer plusieurs tables, séparées par des virgules à la suite de `FROM`. Par exemple, cette requête effectue un produit cartésien des tables Personne et Emploi :

```
SELECT *
FROM Personne , Emploi ;
```

Jointures

On peut parfaitement créer les jointures à la main, en décrivant le produit cartésien dans `FROM` et en mettant la condition de la jointure dans `WHERE`. Mais le langage SQL fournit une syntaxe spéciale pour les jointures. Mieux : il fournit plusieurs types de jointures, qui diffèrent sur quelques points de détail. La syntaxe la plus simple est la jointure normale, aussi appelé jointure interne, ou encore `INNER JOIN`.

```
SELECT *
FROM table_1 INNER JOIN table_2 ON condition
WHERE ... ;
```

La jointure naturelle est un cas particulier de jointure interne, qui correspond au cas où la condition vérifie l'égalité de deux colonnes et où les deux colonnes en question ont le même nom de colonne. Avec ces jointures, il n'y a pas besoin de préciser la condition que doivent respecter les deux tables, celle-ci étant implicite : c'est l'égalité des deux colonnes qui ont le même nom dans les deux tables. Ces jointures naturelles s'écrivent avec les mots-clés `NATURAL JOIN`, qui sépare les deux tables :

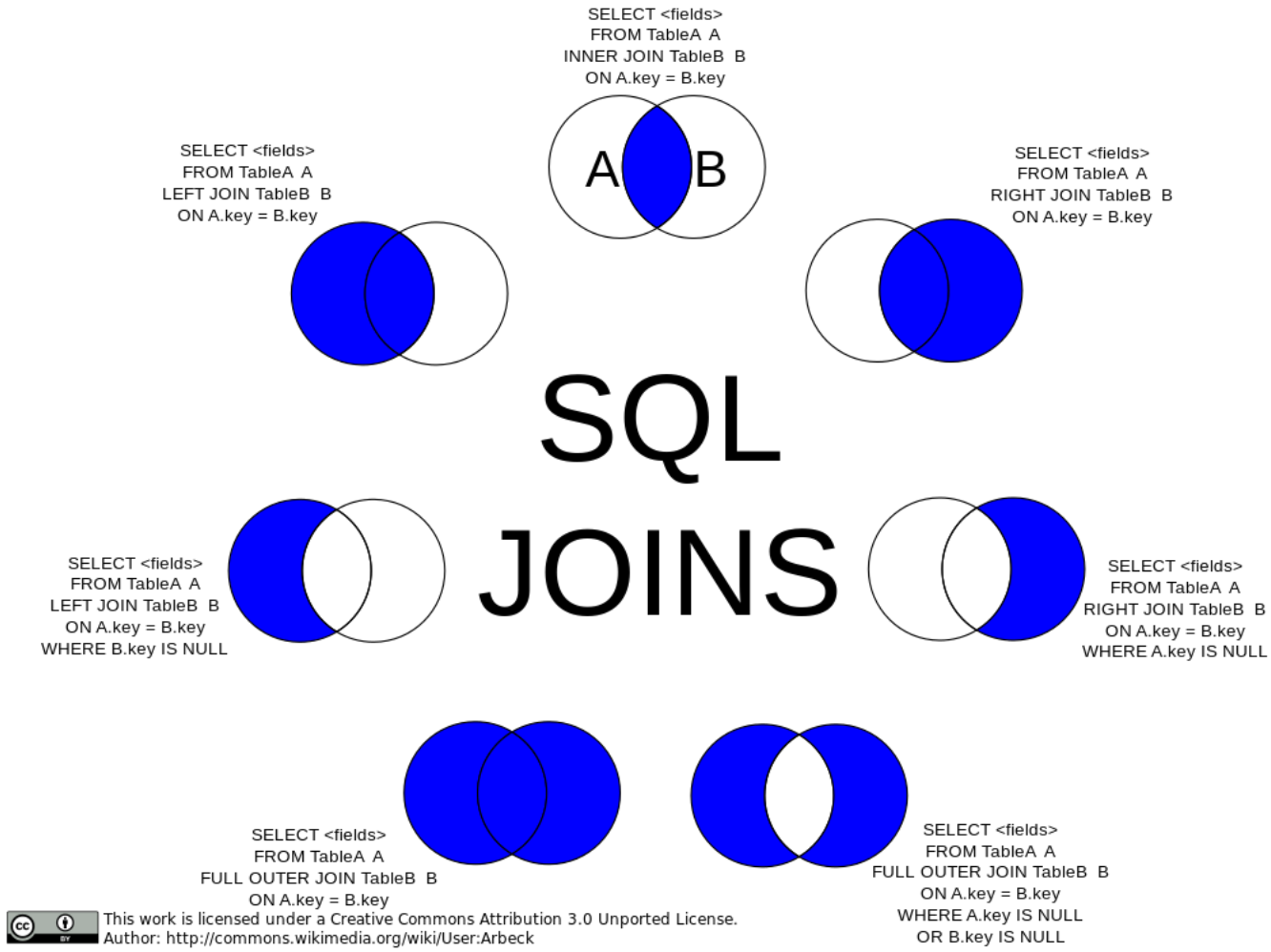
```
SELECT *
FROM table_1 NATURAL JOIN table_2
WHERE ... ;
```

À côté de cette jointure interne, il existe des jointures externes, qui ajoutent des lignes pour lesquelles la condition n'est pas respectée. Dans tous les cas, les lignes ajoutées voient leurs colonnes vides remplies avec la valeur `NULL` (la table résultat a plus de colonnes que les tables initiales). Il existe trois grands types de jointures externes :

- la plus commune est la jointure gauche, dans laquelle les lignes de la première table sont ajoutées dans la table résultat, même si elles ne respectent pas la condition ;
- la jointure droite est similaire, sauf que les lignes de la seconde table remplacent les lignes de la première table ;
- la jointure totale peut être vue comme un mélange d'une jointure gauche et droite : les lignes des deux tables sont copiées, même si elles ne respectent pas la condition et les colonnes en trop sont remplies avec `NULL`.

Les mot-clés pour ces jointures sont respectivement :

- LEFT JOIN ;
- RIGHT JOIN ;
- FULL JOIN.



GROUP BY

La clause `GROUP BY` permet de grouper plusieurs lignes en une seule, à la condition que ces lignes aient un attribut/colonne identique. Si on se contente de faire une projection sur la colonne identique dans les lignes, la clause `GROUP BY` élimine les doublons. On peut se demander à quoi cela peut servir, vu que le mot-clé distinct permet de faire exactement la même chose. Mais la différence apparaît quand on utilise des fonctions comme `MAX`, `MIN`, `AVG`, `SUM` ou `COUNT` : ces fonctions n'agissent plus sur une colonne complète, mais sur un groupe à la fois.

Prenons un exemple classique, avec une table ACHAT qui mémorise des informations sur des achats :

- quel client a effectué l'achat : clé étrangère client ;
- quel est le montant de l'achat : attribut montant, de type INT ;
- et d'autres dont on se moque pour cet exemple.

Il se trouve qu'un client peut faire plusieurs achats, à des jours différents, ou acheter des articles différents en une fois. Les clauses `GROUP BY` et une projection bien choisie nous permettent de calculer quel montant a dépensé le client dans le magasin. Pour cela, il suffit de regrouper toutes les lignes qui font référence à un même client avec un `GROUP BY client`. Une fois cela fait, on ajoute une fonction SUM dans la projection, afin de sommer les montants pour chaque groupe (et donc pour chaque client). Au final, on se retrouve avec une table résultat qui contient une ligne par client, chaque ligne contenant la somme totale que ce client a dépensé dans le magasin.

```
SELECT SUM(montant)
FROM Achats
GROUP BY client ;
```

HAVING

La clause `HAVING` est similaire à la clause `WHERE`, à un détail prêt : elle permet d'utiliser des conditions qui impliquent les fonctions `MAX`, `MIN`, `AVG`, `SUM` et `COUNT`, ainsi que quelques autres. Elle s'utilise le plus souvent avec un `GROUP BY`, même si ce n'est pas systématique.

Par exemple, on peut modifier l'exemple précédent de manière à ne conserver que les clients qui ont acheté plus de 500 euros dans le magasin, en utilisant cette requête :

```
SELECT SUM(montant)
FROM Achats
```

```
GROUP BY client
HAVING SUM(montant) > 500 ;
```

INSERT, DELETE, UPDATE

Dans cet extrait, nous allons voir comment ajouter, modifier ou supprimer des lignes dans une table. Ces opérations sont respectivement prises en charge par les instructions INSERT, UPDATE et DELETE.

INSERT

INSERT sert à ajouter des lignes dans une table. Sa syntaxe est la suivante :

INSERT INTO nom_table VALUES (liste des valeurs de chaque attribut, séparés par des virgules)

Insertion simple

Prenons la table définie comme suit :

```
create table PERSONNE
(
    ID_PERSONNE int not null ,
    NOM varchar not null ,
    PRENOM varchar not null ,
    AGE int ,
    CATEGORIE_PROFESSIONNELLE varchar ,
    NOMBRE_ENFANTS int ,
    TAILLE float ,

    primary key (ID_PERSONNE) ,
) ;
```

Supposons que nous souhaitons ajouter la personne numéro 50, nommée Pierre Dupont, qui a 25 ans, sans emploi, sans enfants et de taille 174 centimètres. Voici comment utiliser INSERT pour cela :

```
INSERT INTO personne VALUES (50, "Dupont", "Pierre", 25, "Sans emploi", 0, 174)
```

On remarquera que les informations sont données dans l'ordre des colonnes.

Maintenant, passons à l'exemple suivant. Prenons la table suivante :

```
create table ENFANT
(
    ID_ENFANT int not null ,
    NOM varchar not null ,
    PRENOM varchar not null ,
    ADRESSE varchar not null ,
    DATE_NAISSANCE date not null ,
) ;
```

Pour y ajouter l'enfant numéro 27, appelé "Jean Moreno", qui habite "22 rue des tuileries Paris", né le 17/01/2009, voici comment utiliser INSERT :

```
INSERT INTO enfant VALUES (27, "Moreno", "Jean", "22 rue des tuileries Paris", 17/01/2009)
```

Insertion multiple

Il est possible d'insérer plusieurs lignes à la fois dans une table en utilisant une seule instruction INSERT. Pour cela, les diverses lignes à ajouter sont simplement placées les unes après les autres à la suite du VALUES, entre parenthèses.

```
INSERT INTO nom_table VALUES
( première ligne )
( seconde ligne )
( troisième ligne )
( ... )
```

DELETE

L'instruction DELETE permet de supprimer les lignes d'une table. On peut l'utiliser sous deux formes :

- une qui supprime toutes les lignes de la table ;
- une autre qui supprime seulement les lignes qui respectent une certaine condition.

Suppression totale

Pour supprimer toutes les lignes d'une table, il faut préciser quelle est la table concernée. La syntaxe suivante permet de supprimer toutes les lignes de la table nommée "nom_table" :

```
DELETE FROM nom_table ;
```

En supprimant toutes les lignes, la table n'est pas supprimée : on obtient une table vide.

Suppression conditionnelle

Pour supprimer les lignes d'une table qui respectent une condition précise, il faut ajouter quelque chose à la syntaxe précédente, histoire de préciser quelle est la condition en question. Pour préciser la condition, on fait comme avec l'instruction SELECT : on utilise une clause WHERE. La syntaxe suivante permet de supprimer toutes les lignes de la table nommée "nom_table", qui respectent la condition nommée "condition" :

```
DELETE FROM nom_table
WHERE condition ;
```

Prenons l'exemple d'une table "Mineurs" qui mémorise une liste de personnes mineures pour une application judiciaire destinée à un tribunal pour enfants. Cette table mémorise, pour chaque enfant, son nom, prénom, age, date de naissance, et bien d'autres informations dans des attributs éponymes. Tous les ans, cette table est mise à jour pour que l'age mémorisé soit le bon. Cependant, suite à cette mise à jour, des lignes ont un age qui vaut 18, ce qui fait que la ligne correspond à des personnes majeures. Voici la requête qui permet de supprimer ces lignes :

```
DELETE FROM Mineurs WHERE age >= 18 ;
```

UPDATE

Pour mettre à jour certaines lignes d'une table, on doit utiliser l'instruction UPDATE. Celle-ci fonctionne comme pour la suppression des lignes : on doit préciser quelles sont les lignes à modifier avec une condition : toutes les lignes de la table qui respectent cette condition seront modifiées, alors que les autres ne seront pas touchées. Toutes les colonnes sont mises à jour avec la valeur indiquée dans le UPDATE. Pour faire la mise à jour, il faut ainsi préciser :

- quelle table modifier à la suite de l'instruction UPDATE ;
- quelles colonnes modifier et par quoi remplacer leur contenu : c'est le rôle de l'instruction SET ;
- et enfin quelle est la condition avec, encore une fois, un WHERE.

```
UPDATE table
SET colonne_1 = 'valeur 1', colonne_2 = 'valeur 2', colonne_3 = 'valeur 3'
WHERE condition ;
```