

# Cours SQL

ML Pro

Janvier 2025

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Qu'est ce que SQL ? . . . . .	3
1.2	Les types de bases de données . . . . .	3
1.3	La base des bases de données relationnelles . . . . .	3
1.3.1	Détails d'une table . . . . .	4
1.4	Les différents types de relations dans les bases de données relationnelles . . . . .	5
1.4.1	Relation 1: 1 . . . . .	5
1.4.2	Relation 1: $n$ . . . . .	5
1.4.3	Relation $n$ : $m$ . . . . .	6
1.4.4	Clés primaires et étrangères dans les relations . . . . .	7
1.4.5	Les différentes notations pour les relations . . . . .	7
1.5	Du modèle conceptuel au modèle relationnel . . . . .	9
1.5.1	Modèle conceptuel . . . . .	9
1.5.2	Modèle relationnel . . . . .	10
1.5.3	Génération automatique du modèle relationnel . . . . .	11
<b>2</b>	<b>Le langage SQL</b>	<b>12</b>
2.1	Le schéma du cours . . . . .	12
2.2	Créer des tables . . . . .	13
2.3	Supprimer des tables . . . . .	14
2.4	Modifier des tables . . . . .	14
2.5	Ajouter des lignes . . . . .	15
2.6	Supprimer des lignes . . . . .	16
2.7	Sélectionner des lignes . . . . .	17
2.8	Filtrer les résultats . . . . .	18
2.9	Trier les résultats . . . . .	19
2.10	Conditions logiques . . . . .	20
2.11	Calculs sur des colonnes (Avec AS) . . . . .	21
2.12	Les jointures . . . . .	22
2.12.1	INNER JOIN . . . . .	22
2.12.2	LEFT JOIN (ou LEFT OUTER JOIN) . . . . .	24
2.12.3	RIGHT JOIN (ou RIGHT OUTER JOIN) . . . . .	25
2.12.4	FULL JOIN (ou FULL OUTER JOIN) . . . . .	27
2.12.5	SELF JOIN . . . . .	28
2.12.6	Jointures multiples . . . . .	29
2.13	Utiliser les fonctions SUM, COUNT, AVG, ... avec GROUP BY . . . . .	30
2.14	Combiner WHERE et GROUP BY . . . . .	31
<b>3</b>	<b>Fiche résumé</b>	<b>33</b>

# 1 Introduction

## 1.1 Qu'est ce que SQL ?

Le **Structured Query Language** (SQL) est un langage conçu pour gérer et manipuler les bases de données relationnelles. Créé en 1974 par IBM, il est devenu un standard pour l'interaction avec les bases de données. SQL permet d'effectuer diverses opérations telles que l'insertion de données, la mise à jour, la suppression et la récupération d'informations dans des bases de données.

Voici un exemple simple d'une requête SQL qui récupère les prénoms, noms et âges des joueurs dont l'âge est supérieur ou égal à 18 ans :

```
SELECT Prenom, Nom, Age FROM Joueurs WHERE Age >= 18;
```

## 1.2 Les types de bases de données

Il existe plusieurs types de bases de données, chacun étant adapté à des besoins spécifiques et offrant des fonctionnalités différentes. Le tableau suivant présente les principaux types de bases de données ainsi que leurs caractéristiques et quelques exemples de systèmes de gestion de bases de données (SGBD) associés :

Type	Description	SGBD
<b>Relationnelles</b>	Organisées en tables avec des relations entre elles, utilisant SQL pour les requêtes.	MySQL, PostgreSQL, Oracle, SQLite
<b>Documentaires (NoSQL)</b>	Stockent des documents semi-structurés (JSON, XML), souvent utilisés dans les applications web modernes.	MongoDB
<b>En colonnes (NoSQL)</b>	Optimisées pour les requêtes analytiques et le stockage de grandes quantités de données, particulièrement adaptées au Big Data.	Cassandra
<b>Graphes (NoSQL)</b>	Conçues pour gérer des relations complexes entre les données, idéales pour les réseaux sociaux et les systèmes de recommandations.	Neo4j
<b>Clé-Valeur (NoSQL)</b>	Stockent des paires clé-valeur simples, idéales pour des caches ou des sessions utilisateurs.	Redis

TABLE 1 – Principaux types de bases de données et leurs SGBD

Dans le reste du cours nous utiliserons donc les bases de données relationnelles.

## 1.3 La base des bases de données relationnelles

Les bases de données relationnelles sont organisées autour de plusieurs concepts clés, notamment les **tables**, les **relations** et les **clés**. Une table est une collection d'enregistrements, chacun étant une ligne dans la table, et chaque colonne représente un attribut de ces enregistrements.

Les tables sont liées entre elles par des **relations**, qui sont définies à l'aide de clés. Il existe plusieurs types de clés, les deux principales étant la **clé primaire** et la **clé étrangère** :

- Une **clé primaire** est un attribut ou un ensemble d'attributs qui identifie de manière unique chaque ligne d'une table.
- Une **clé étrangère** est un attribut dans une table qui fait référence à la clé primaire d'une autre table, établissant ainsi une relation entre les deux tables.

Le schéma ci-dessous illustre cette organisation avec des relations entre trois tables, où des clés primaires et étrangères sont utilisées pour lier les données :

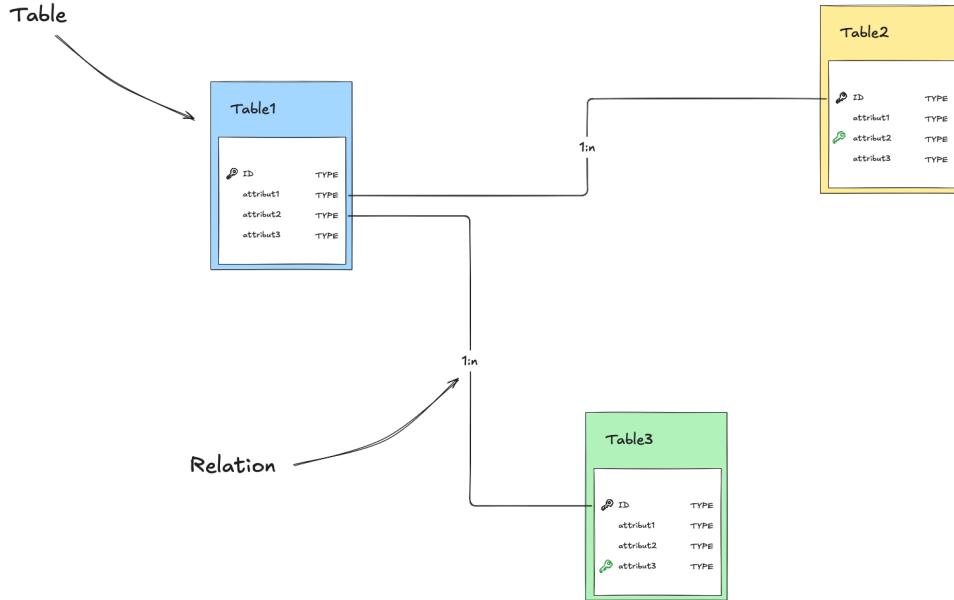


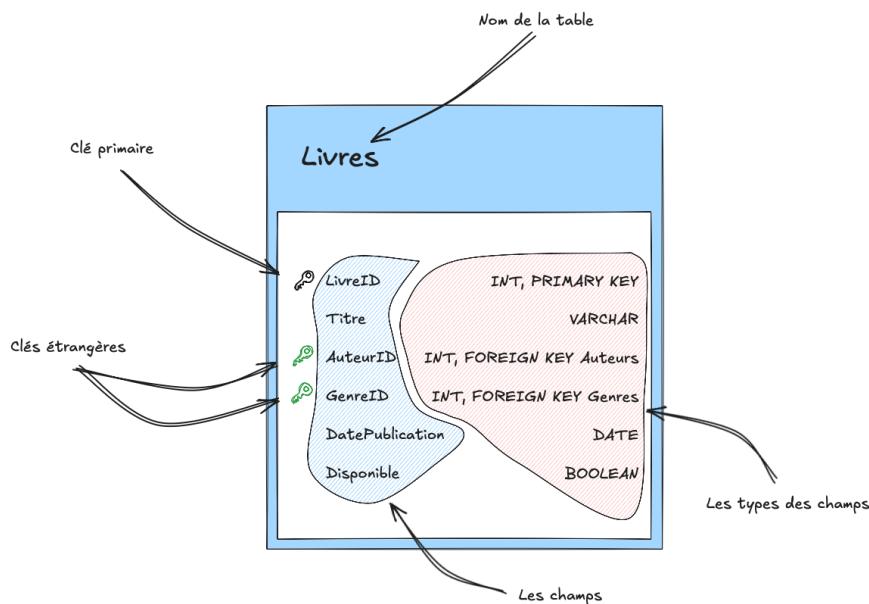
FIGURE 1 – Schéma des relations entre tables dans une base de données relationnelle

### 1.3.1 Détails d'une table

Chaque table dans une base de données relationnelle contient des **champs** qui décrivent les attributs de l'entité représentée par la table. Par exemple, une table *Livres* pourrait contenir des champs comme le *LivreID*, le *Titre*, l'*AuteurID*, le *GenreID*, et d'autres informations spécifiques aux livres.

Voici un aperçu des différents éléments que l'on retrouve dans une table :

- Le **nom de la table**, qui identifie l'entité représentée.
- Les **champs**, qui sont les colonnes de la table. Chaque champ a un type de données spécifique (par exemple, **INT** pour un entier, **VARCHAR** pour une chaîne de caractères).
- La **clé primaire**, qui identifie de manière unique chaque ligne de la table.
- Les **clés étrangères**, qui créent des liens avec d'autres tables.



Le contenu d'une table est présenté sous forme de lignes, chaque ligne représentant un enregistrement unique. Les valeurs dans chaque colonne correspondent aux attributs de cet enregistrement.

L'exemple suivant montre comment les données peuvent être stockées dans une table *Livres* :

LivreID	Titre	AuteurID	GenreID	DatePublication	Disponible
1	"Etang"	2	1	01/02/2009	0
2	"Bonheur"	4	3	12/11/2021	1

FIGURE 2 – Détails d'une table dans une base de données relationnelle

## 1.4 Les différents types de relations dans les bases de données relationnelles

Dans une base de données relationnelle, les relations entre les tables sont fondamentales pour structurer les données de manière cohérente et efficace. Une relation fait référence à la manière dont les tables sont connectées entre elles à travers des **clés**. Il existe trois principaux types de relations : 1: 1, 1: n, et n: m. Ces relations sont définies par la manière dont une table fait référence à une autre à travers des clés primaires et des clés étrangères.

### 1.4.1 Relation 1: 1

Une relation de type un-à-un (1: 1) signifie que chaque ligne d'une table est liée à une seule ligne d'une autre table, et vice versa. Ce type de relation est relativement rare et souvent utilisé lorsque des informations doivent être séparées en deux tables pour des raisons de conception ou de sécurité.

- La clé primaire de la première table est également la clé étrangère de la deuxième table, et cette clé ne peut apparaître qu'une seule fois dans chaque table.
- Cette relation est représentée par une ligne reliant les deux tables avec une indication "1: 1".

Exemple : Une table *Clients* est liée à une table *NuméroFiscales* par une relation 1: 1, où chaque client possède un numéro fiscal unique.

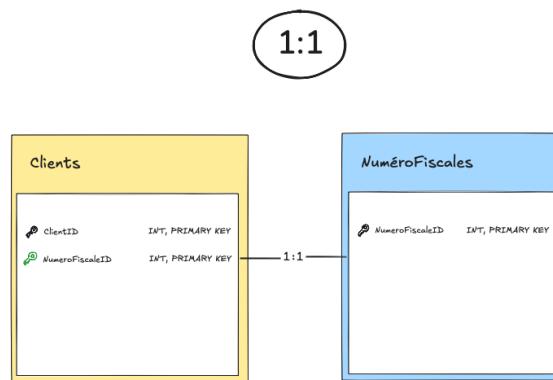


FIGURE 3 – Relation 1: 1 entre les tables *Clients* et *NuméroFiscales*

### 1.4.2 Relation 1: n

Une relation de type **un-à-plusieurs** (1: n) signifie qu'une ligne d'une table peut être associée à plusieurs lignes d'une autre table, mais chaque ligne de cette deuxième table ne peut être associée qu'à une seule ligne de la première table.

- La table qui contient l'attribut unique (souvent la clé primaire) est du côté "1", tandis que la table qui contient plusieurs références est du côté "n".

- La clé primaire de la table du côté "1" est utilisée comme clé étrangère dans la table du côté "n", permettant de relier plusieurs enregistrements.

Exemple : Une table *Clients* est liée à une table *Commandes*, où chaque client peut avoir plusieurs commandes, mais chaque commande est liée à un seul client.

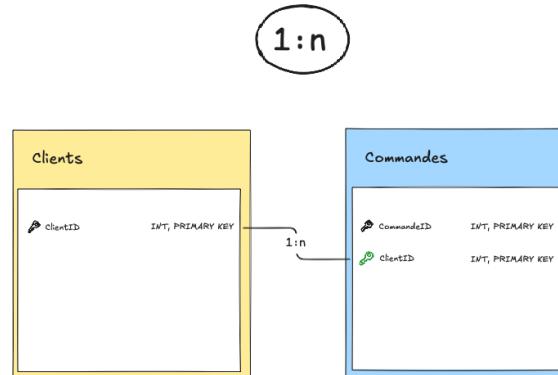


FIGURE 4 – Relation 1: n entre les tables *Clients* et *Commandes*

#### 1.4.3 Relation n: m

Une relation de type **plusieurs-à-plusieurs** (*n: m*) signifie qu'une ligne d'une table peut être liée à plusieurs lignes d'une autre table, et vice versa. Cependant, les bases de données relationnelles classiques ne permettent pas de modéliser directement les relations *n: m* entre deux tables. Pour cela, une table intermédiaire (ou table d'association) est nécessaire.

- Une table intermédiaire est créée pour gérer les relations *n: m*. Cette table contient au moins deux clés étrangères, qui font référence aux clés primaires des deux tables liées.
- Les deux relations 1:n sont ainsi modélisées à travers cette table intermédiaire, transformant la relation *n: m* en deux relations 1:n.

Exemple : Une table *Produits* est liée à une table *Commandes* par une relation *n: m*. Chaque produit peut apparaître dans plusieurs commandes, et chaque commande peut contenir plusieurs produits. La table intermédiaire *DétailsCommandes* gère cette relation en associant les IDs des produits et des commandes.

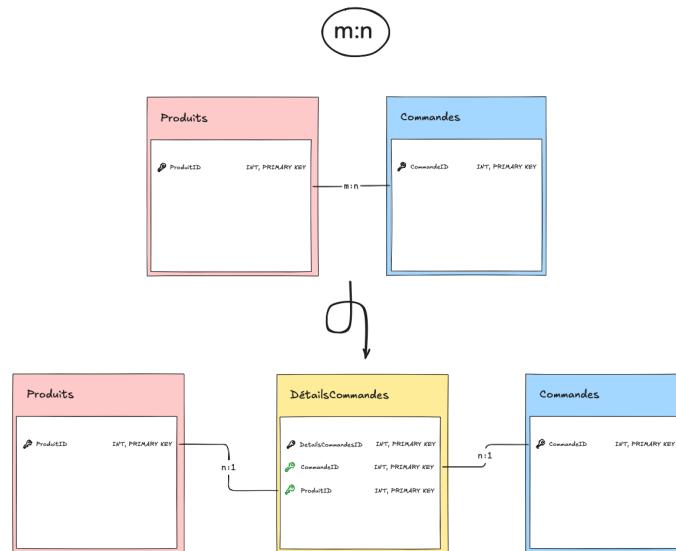


FIGURE 5 – Relation n :m entre les tables *Produits*, *Commandes* et la table intermédiaire *DétailsCommandes*

#### 1.4.4 Clés primaires et étrangères dans les relations

**Clé primaire (Primary Key)** : C'est un identifiant unique pour chaque enregistrement d'une table. Il garantit l'unicité des données dans une colonne. Dans les relations, la clé primaire d'une table est souvent utilisée pour créer une relation avec une autre table.

**Clé étrangère (Foreign Key)** : C'est une colonne ou un ensemble de colonnes dans une table qui référence la clé primaire d'une autre table. Elle permet de maintenir l'intégrité référentielle en s'assurant que les données d'une table correspondent à celles d'une autre.

Dans une relation  $1:n$ , la clé primaire du côté "1" devient la clé étrangère du côté "n". Dans une relation  $n:m$ , les deux clés primaires des tables en relation sont introduites dans la table intermédiaire comme clés étrangères, qui deviennent conjointement la clé primaire de cette table d'association.

#### 1.4.5 Les différentes notations pour les relations

Dans ce cours, nous utilisons la notation UML simplifiée.

##### Les relations en notation UML

En **UML**, les associations entre deux entités (par exemple, deux tables de base de données) se décrivent en indiquant, pour chaque extrémité, une valeur représentant le nombre maximal d'occurrences possibles. Mais cela peut correspondre à plusieurs situations, qui dépendent des spécifications du projet en question.

La relation  $1:1$  peut correspondre aux situations suivantes :

- '0..1 : 0..1'
- '0..1 : 1..1'
- '1..1 : 0..1'
- '1..1 : 1..1'

En pratique, cela signifie qu'une entité A est liée au plus à une entité B, et réciproquement.

**Exemple** : Un client peut éventuellement ('0..1') ou obligatoirement ('1..1') être associé à une carte de fidélité unique, et inversement.

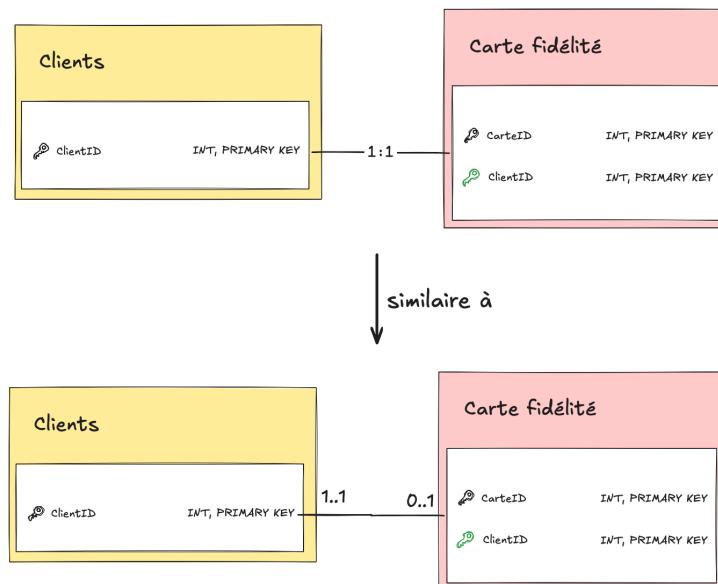
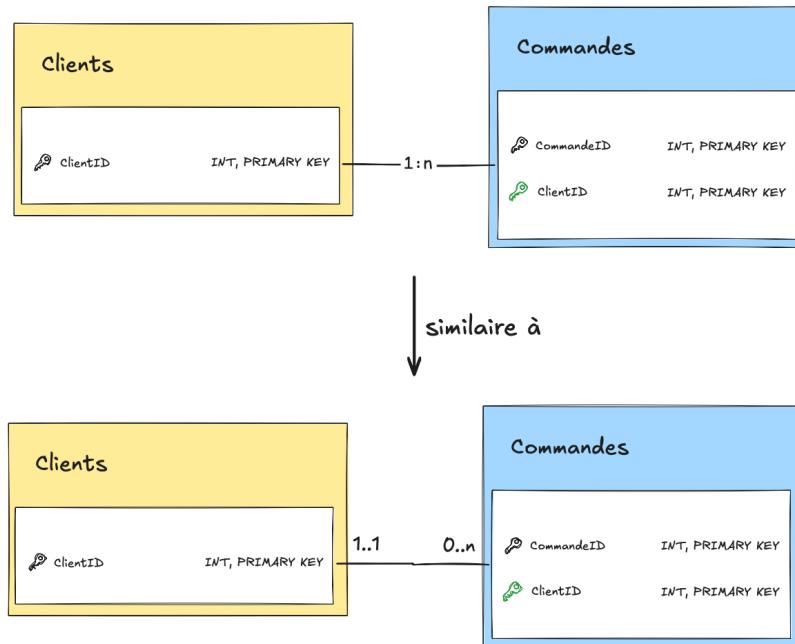


FIGURE 6 – Ici un client peut éventuellement avoir une carte de fidélité (0..1) mais une carte de fidélité est obligatoirement associée à un seul client (1..1). Ici une carte est forcément associée à un client.

La relation  $1 : N$  peut correspondre aux situations suivantes :

- ‘0..1 : 0..N‘
- ‘0..1 : 1..N‘
- ‘1..1 : 0..N‘
- ‘1..1 : 1..N‘

Cela indique qu’un élément du côté «1» peut éventuellement (‘0’) ou obligatoirement (‘1’) être associé à plusieurs éléments (‘N’) de l’autre entité, tandis que ceux-ci ne peuvent être reliés qu’à un seul élément du côté «1». **Exemple** : Un client (‘1..1’) passe plusieurs commandes (‘0..N’), mais chaque commande est liée à un unique client.



La relation  $N : M$  englobe les cas :

- ‘0..N : 0..N‘
- ‘0..N : 1..N‘
- ‘1..N : 0..N‘
- ‘1..N : 1..N‘

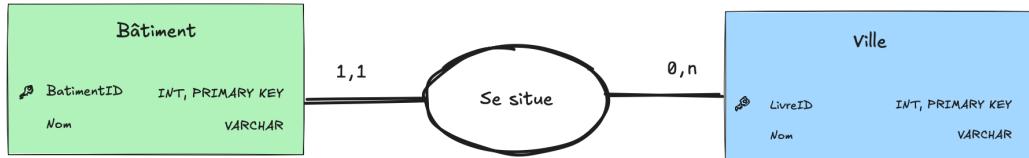
Chaque entité peut être associée à plusieurs occurrences de l’autre. Dans une base de données relationnelle, on modélise cette relation via une table d’association (ou table intermédiaire). **Exemple** : Un produit (‘1..N’) peut apparaître dans plusieurs commandes, et chaque commande (‘1..N’) peut contenir plusieurs produits.

#### Les relations avec la méthode Merise

La **méthode Merise** représente aussi les cardinalités en utilisant deux valeurs (minimum et maximum) pour chaque côté de la relation, mais sous la forme ‘(min, max)’. L’ordre dans lequel on écrit ces cardinalités correspond généralement à l’emplacement visuel : la partie gauche de la relation correspond à la première paire, et la partie droite, à la seconde.

-  $(1, 1) - (0, n)$  signifie, par exemple, qu'une entité A est liée à exactement 1 occurrence d'une entité B, tandis que B peut être lié à zéro, une ou plusieurs occurrences de A.

**Exemple :** Un bâtiment  $(1, 1)$  se situe exactement dans une seule ville, tandis qu'une ville  $(0, n)$  peut regrouper zéro ou plusieurs bâtiments.



-  $(0, 1) - (1, 1)$  ou toute autre combinaison suit la même logique : le premier couple  $(0, 1)$  ou  $(1, 1)$  concerne l'entité A à gauche, et le second couple  $(1, 1)$  à droite concerne l'entité B, toujours sous la forme (min, max).

## 1.5 Du modèle conceptuel au modèle relationnel

La conception d'une base de données suit plusieurs étapes importantes, permettant de passer d'une représentation abstraite des entités du système à un modèle détaillé prêt à être implémenté dans un SGBD relationnel. Ce processus comprend deux étapes majeures : la création du **modèle conceptuel** et sa transformation en **modèle relationnel**.

### 1.5.1 Modèle conceptuel

Le modèle conceptuel, souvent représenté par un **diagramme Entité-Relation (ER)**, est une représentation graphique des entités et des relations entre elles dans le système. À ce stade, le modèle n'entre pas dans les détails techniques, mais se concentre sur la compréhension des entités métiers et de leurs interactions.

#### Éléments du modèle conceptuel :

- **Entités** : Les entités sont les objets ou concepts essentiels à représenter dans la base de données. Chaque entité possède des attributs qui décrivent ses caractéristiques. Par exemple, dans une bibliothèque, des entités typiques seraient *Emprunteurs*, *Livres*, *Auteurs*, etc.
- **Relations** : Les relations indiquent comment les entités interagissent entre elles. Elles peuvent être de différents types ( $1: 1$ ,  $1: n$ ,  $n: m$ ). Par exemple, un *Emprunteur* peut emprunter plusieurs *Livres*, mais un *Livre* ne peut être emprunté que par un seul emprunteur à un moment donné, ce qui correspond à une relation  $1: n$ .

Voici un exemple de diagramme Entité-Relation pour une bibliothèque :

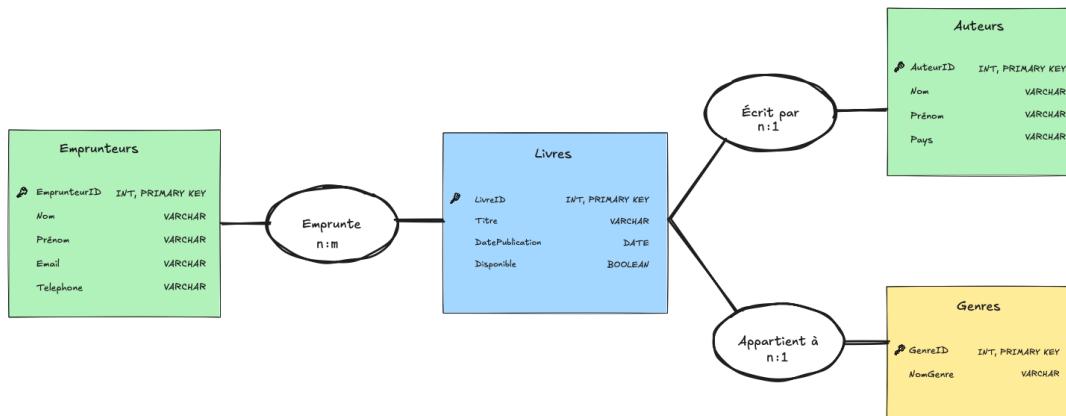


FIGURE 7 – Modèle conceptuel : Diagramme Entité-Relation

Dans cet exemple, on retrouve les entités *Emprunteurs*, *Livres*, *Auteurs* et *Genres*, avec les relations associées comme *Écrit par* (relation 1: n entre *Auteurs* et *Livres*) ou encore *Emprunte* (relation n: m entre *Emprunteurs* et *Livres*).

### 1.5.2 Modèle relationnel

Le modèle relationnel est la traduction directe du modèle conceptuel en un ensemble de tables prêtes à être implémentées dans une base de données relationnelle. Chaque entité devient une table, et les relations sont traduites en clés étrangères pour maintenir l'intégrité référentielle.

**Étapes pour passer du modèle conceptuel au modèle relationnel :**

- Transformation des entités en tables** : Chaque entité du modèle conceptuel devient une table dans le modèle relationnel. Les attributs de l'entité deviennent les colonnes de la table. Par exemple, l'entité *Livres* devient une table contenant les colonnes *LivreID*, *Titre*, *DatePublication*, etc.
- Transformation des relations en clés étrangères** : Les relations entre entités sont représentées par des clés étrangères. Par exemple, dans une relation 1: n, la clé primaire de l'entité du côté "1" est utilisée comme clé étrangère dans la table de l'entité du côté "n". Dans une relation n: m, une table intermédiaire est nécessaire pour représenter la relation.
- Gestion des cardinalités** : Les cardinalités des relations doivent être respectées dans le modèle relationnel, en ajoutant des contraintes sur les clés étrangères pour garantir l'intégrité des données.

Voici le modèle relationnel correspondant au modèle conceptuel de la bibliothèque :

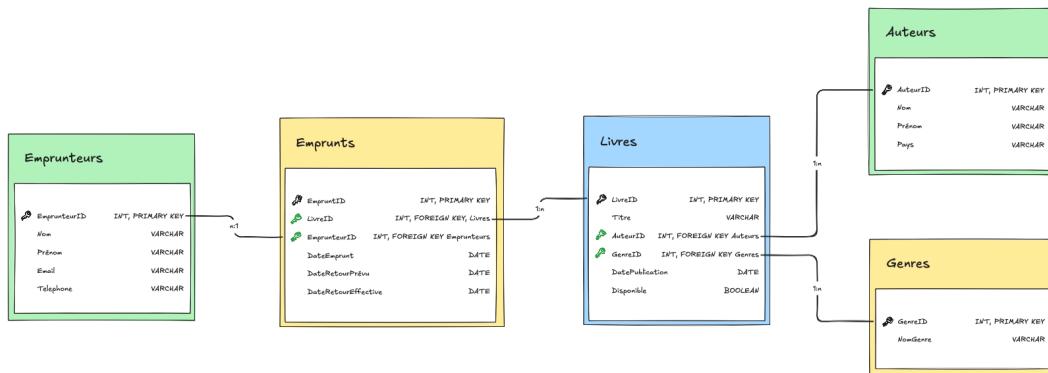


FIGURE 8 – Modèle relationnel : Transformation en tables et clés étrangères

Dans cet exemple, on observe la transformation des entités en tables (*Emprunteurs*, *Livres*, *Auteurs*, *Genres*) et des relations en clés étrangères. Par exemple, la table *Emprunts* gère la relation n: m entre les emprunteurs et les livres. Elle contient des clés étrangères (*EmprunteurID* et *LivreID*) qui font référence aux tables *Emprunteurs* et *Livres*, respectivement.

### 1.5.3 Génération automatique du modèle relationnel

#### Génération automatique du modèle relationnel

Il existe plusieurs logiciels qui permettent de générer automatiquement le modèle relationnel à partir d'un **Modèle Conceptuel de Données (MCD)**. Ces outils permettent non seulement de créer graphiquement le modèle conceptuel, mais aussi de le convertir en modèle relationnel et d'exporter les requêtes SQL nécessaires à la création des tables.

Parmi ces outils, on trouve :

- **JMerise** : Un logiciel populaire en France pour modéliser les bases de données avec la méthode Merise. Il permet de passer du MCD au modèle relationnel et d'exporter le tout en SQL.
- **MySQL Workbench** : Un outil de conception de bases de données qui permet de créer un diagramme EER (Entité-Relation étendu) et de générer des requêtes SQL pour la création des tables.
- **DbSchema** : Un autre outil visuel pour concevoir des bases de données, qui permet également de générer des scripts SQL à partir du modèle relationnel.
- **SQL Power Architect** : Un outil open source qui aide à modéliser les bases de données relationnelles et à générer les requêtes SQL associées.

Ces logiciels sont très utiles pour automatiser la phase de transformation du modèle conceptuel en modèle relationnel, ce qui simplifie le processus de développement d'une base de données.

## 2 Le langage SQL

### 2.1 Le schéma du cours

Le schéma ci-dessous présente l'organisation des tables dans une base de données relationnelle destinée à gérer les informations sur les employés, leurs départements, leurs adresses et leurs postes. Ce schéma met en évidence les relations entre les différentes entités et illustre comment les clés primaires et étrangères sont utilisées pour structurer les données.

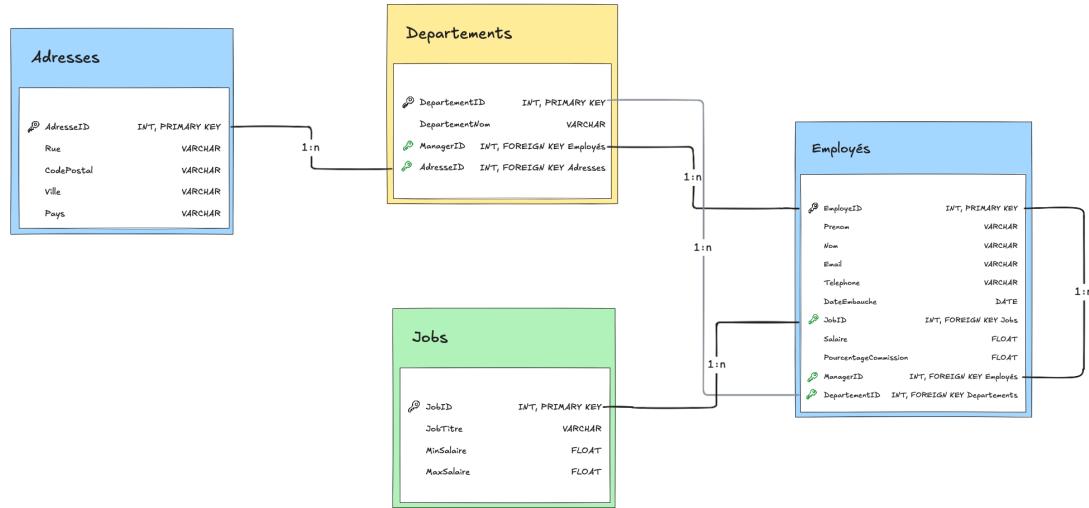


FIGURE 9 – Schéma de la base utilisée dans ce cours

#### Description des entités :

- **Adresses** : Cette table stocke les informations relatives aux adresses des départements, avec un identifiant unique **AdresseID** pour chaque adresse. Chaque adresse est caractérisée par des attributs tels que **Rue**, **CodePostal**, **Ville**, et **Pays**.
- **Départements** : Les départements de l'organisation sont répertoriés dans cette table. Chaque département est identifié par **DepartmentID**, et possède des informations telles que le nom du département (**DepartmentNom**), l'identifiant du manager (**ManagerID**), et l'adresse du département via une clé étrangère **AdresseID**.
- **Employés** : Cette table contient les informations des employés, tels que leur **EmployeeID**, **Prenom**, **Nom**, **Email**, et leur date d'embauche (**DateEmbauche**). Chaque employé est associé à un poste (**JobID**), un département (**DepartmentID**), et peut avoir un manager (**ManagerID**), qui est lui-même un employé.
- **Jobs** : La table *Jobs* répertorie les différents postes dans l'organisation. Chaque poste est identifié par **JobID**, et contient des informations telles que le titre du poste (**JobTitre**), le salaire minimum et maximum (**MinSalaire**, **MaxSalaire**).

#### Relations entre les tables :

- La relation entre la table *Adresses* et la table *Départements* est de type **1 :n**, chaque adresse pouvant être associée à plusieurs départements.
- La table *Employés* est liée à la table *Départements* par une relation **1 :n**, un département pouvant avoir plusieurs employés.
- Chaque employé est lié à un poste spécifique dans la table *Jobs* par une relation **1 :n**.
- Enfin, un employé peut être le manager d'autres employés, créant ainsi une relation récursive **1 :n** dans la table *Employés*.

Le schéma permet de visualiser les dépendances entre les différentes entités de la base de données, facilitant ainsi la compréhension de la structure relationnelle et de l'intégrité référentielle. Ce modèle relationnel est utilisé pour s'assurer que les données sont organisées de manière efficace et cohérente.

## 2.2 Créer des tables

La commande `CREATE TABLE` permet de créer une nouvelle table dans une base de données en spécifiant ses colonnes ainsi que les types de données associés à chaque colonne. Cette commande définit également les contraintes, telles que les clés primaires et étrangères, qui assurent l'intégrité référentielle dans le schéma relationnel.

### Syntaxe générale

```
CREATE TABLE NomTable (
    NomColonne TypeData Contraintes,
    ...
);
```

La commande suit une structure simple : chaque colonne est déclarée avec son type de données (INT, VARCHAR, DATE, etc.), et on peut également ajouter des contraintes comme `PRIMARY KEY` pour les colonnes qui doivent être uniques ou `FOREIGN KEY` pour les colonnes qui font référence à une autre table.

### Application : création des tables du schéma

Dans l'exemple du schéma précédent, nous allons créer les tables `Adresses`, `Departements`, `Jobs` et `Employés`, chacune avec les colonnes et relations spécifiées. Voici les requêtes SQL pour créer ces tables :

```
CREATE TABLE Adresses (
    AdresseID INT PRIMARY KEY,
    Rue VARCHAR(100),
    CodePostal VARCHAR(20),
    Ville VARCHAR(50),
    Pays VARCHAR(50)
);

CREATE TABLE Departements (
    DepartementID INT PRIMARY KEY,
    DepartementNom VARCHAR(100),
    ManagerID INT,
    AdresseID INT,
    FOREIGN KEY (ManagerID) REFERENCES Employés(EmployeID),
    FOREIGN KEY (AdresseID) REFERENCES Adresses(AdresseID)
);

CREATE TABLE Jobs (
    JobID INT PRIMARY KEY,
    JobTitre VARCHAR(100),
    MinSalaire FLOAT,
    MaxSalaire FLOAT
);

CREATE TABLE Employés (
    EmployeID INT PRIMARY KEY,
    Prenom VARCHAR(100),
    Nom VARCHAR(100),
    Email VARCHAR(150),
    Telephone VARCHAR(20),
    DateEmbauche DATE,
    JobID INT,
    Salaire FLOAT,
    PourcentageCommission FLOAT,
```

```

    ManagerID INT,
    DepartementID INT,
    FOREIGN KEY (JobID) REFERENCES Jobs(JobID),
    FOREIGN KEY (ManagerID) REFERENCES Employés(EmployeID),
    FOREIGN KEY (DepartementID) REFERENCES Departements(DepartementID)
);

```

#### Explication :

- **Adresses** : Cette table stocke les informations relatives aux adresses, avec **AdresseID** comme clé primaire.
- **Departements** : La table **Departements** est liée aux tables **Adresses** et **Employés** via des clés étrangères **AdresseID** et **ManagerID**.
- **Jobs** : Cette table stocke les différents postes avec **JobID** comme clé primaire.
- **Employés** : Cette table est liée aux tables **Jobs**, **Departements** et à elle-même pour gérer les managers, grâce aux clés étrangères **JobID**, **DepartementID**, et **ManagerID**.

Ces commandes SQL permettent de définir la structure de la base de données, de spécifier les relations entre les différentes tables, et de garantir l'intégrité des données via les contraintes des clés primaires et étrangères. Chaque table représente une entité du schéma et est reliée aux autres pour assurer une organisation logique et cohérente des informations.

### 2.3 Supprimer des tables

La commande **DROP TABLE** permet de supprimer une table existante dans une base de données. Cette opération est irréversible : une fois la table supprimée, toutes les données qu'elle contient sont également perdues. Il est donc important de s'assurer qu'on souhaite réellement supprimer la table avant d'exécuter cette commande.

#### Syntaxe générale

```
DROP TABLE NomTable;
```

La commande suit une structure simple : il suffit de spécifier le nom de la table que l'on souhaite supprimer.

#### Application : suppression d'une table

Dans l'exemple suivant, nous allons supprimer la table **Adresses** qui a été créée précédemment :

```
DROP TABLE Adresses;
```

Cette commande supprime définitivement la table **Adresses** ainsi que toutes les données qu'elle contenait. Il est important de noter que la suppression d'une table peut affecter les relations avec d'autres tables si elle contient des clés étrangères. Il est donc nécessaire de bien planifier la suppression pour ne pas compromettre l'intégrité de la base de données.

### 2.4 Modifier des tables

La commande **ALTER TABLE** permet d'ajouter, de supprimer ou de modifier des colonnes dans une table existante. Elle est utilisée lorsque la structure d'une table doit être modifiée après sa création, sans avoir à la supprimer et à la recréer.

### Syntaxe générale

Ajouter une colonne :

```
ALTER TABLE NomTable ADD NomColonne TypeData
```

Supprimer une colonne :

```
ALTER TABLE NomTable DROP NomColonne
```

Modifier le nom d'une colonne :

```
ALTER TABLE NomTable RENAME NomColonneAncien TO NomColonneNouveau
```

La commande suit une structure flexible qui permet de spécifier l'opération souhaitée. On peut ajouter une nouvelle colonne en utilisant **ADD**, supprimer une colonne existante avec **DROP**, ou encore renommer une colonne avec **RENAME**. Cette commande permet d'adapter la structure d'une table aux besoins changeants de la base de données.

### Application : modification d'une table

Dans l'exemple suivant, nous allons modifier la table **Adresses** en changeant le nom de la colonne **Ville** en **City** :

```
ALTER TABLE Adresses RENAME Ville TO City;
```

Cette commande renomme simplement la colonne **Ville** pour refléter un changement dans la structure ou les conventions de nommage de la base de données. De manière générale, la commande **ALTER TABLE** est très puissante et peut être utilisée pour divers ajustements sur la structure d'une table, tout en préservant les données existantes.

### 2.5 Ajouter des lignes

La commande **INSERT INTO** permet d'insérer une nouvelle ligne dans une table existante. Cette opération est utilisée pour ajouter des enregistrements dans une table en spécifiant les valeurs correspondant aux colonnes définies lors de la création de la table.

### Syntaxe générale

```
INSERT INTO NomTable (Colonne1, Colonne2, ...) VALUES (Valeur1, Valeur2, ...);
```

Il est possible de préciser les colonnes dans lesquelles les valeurs doivent être insérées. Si toutes les colonnes doivent recevoir des valeurs, il est également possible de ne pas mentionner les noms des colonnes, à condition que l'ordre des valeurs respecte l'ordre des colonnes de la table.

### Application : insertion de données

Dans l'exemple suivant, nous allons insérer des données dans les différentes tables du schéma :

```
INSERT INTO Adresses (AdresseID, Rue, CodePostal, Ville, Pays) VALUES
(1, 'Rue de Paris', '75001', 'Paris', 'France'),
(2, 'Rue de Lyon', '69001', 'Lyon', 'France'),
(3, 'Rue de Lille', '59000', 'Lille', 'France'),
(4, 'Rue de Marseille', '13001', 'Marseille', 'France');
```

```
INSERT INTO Departements (DepartementID, DepartementNom, ManagerID, AdresseID)
VALUES
(1, 'Ressources Humaines', NULL, 1),
```

```
(2, 'Informatique', 1, 2),
(3, 'Marketing', 2, 3),
(4, 'Ventes', NULL, 4);

INSERT INTO Jobs (JobID, JobTitre, MinSalaire, MaxSalaire) VALUES
(1, 'Développeur', 30000, 60000),
(2, 'Analyste', 35000, 70000),
(3, 'Manager', 45000, 90000),
(4, 'Commercial', 25000, 50000),
(5, 'Technicien', 20000, 40000);

INSERT INTO Employés (EmployeID, Prenom, Nom, Email, Telephone, DateEmbauche,
JobID, Salaire,
PourcentageCommission, ManagerID, DepartementID) VALUES
(1, 'John', 'Doe', 'john.doe@example.com', '0601020304', '2020-01-10', 1, 50000,
5.0, NULL, 1),
(2, 'Jane', 'Smith', 'jane.smith@example.com', '0602030405', '2019-05-15', 2,
60000, 7.0, 1, 2),
(3, 'Tom', 'Brown', 'tom.brown@example.com', NULL, '2021-07-20', 3, 80000, 10.0,
2, 3),
(4, 'Emma', 'Wilson', NULL, '0604050607', '2021-03-14', 4, 45000, 4.0, 3, 4),
(5, 'Noah', 'Taylor', 'noah.taylor@example.com', '0605060708', '2021-06-20', 1,
35000, 2.5, NULL, 2),
(6, 'Olivia', 'Johnson', 'olivia.johnson@example.com', '0607080910',
'2022-01-15', 2, 37000, 3.0, 1, 2),
(7, 'Liam', 'Lee', NULL, '0607080910', '2020-08-25', 1, 30000, 2.0, 2, 1);
```

### Explication :

- **Adresses** : Les lignes insérées dans cette table représentent les différentes adresses des départements.
- **Departements** : Chaque département est inséré avec un identifiant de manager (ou NULL si aucun manager n'est affecté) et une adresse.
- **Jobs** : Cette table est remplie avec différents intitulés de poste et les plages salariales associées.
- **Employés** : Les lignes insérées incluent des employés avec des informations comme le salaire, les commissions, le manager et le département auquel ils sont affectés. Certains employés peuvent ne pas avoir d'email ou de manager.

La commande `INSERT INTO` permet d'ajouter de nouvelles données dans les tables de la base, tout en respectant les contraintes de chaque colonne (comme les types de données, les valeurs NULL autorisées ou non, etc.).

## 2.6 Supprimer des lignes

La commande `DELETE` permet de supprimer une ou plusieurs lignes dans une table en fonction d'une condition spécifique. Cette commande est utilisée lorsque certaines données doivent être retirées de la table, que ce soit pour corriger des erreurs ou gérer des informations obsolètes.

### Syntaxe générale

```
DELETE FROM NomTable WHERE Condition;
```

La condition après `WHERE` permet de spécifier quelle(s) ligne(s) doit/doivent être supprimée(s). Si la condition est omise, toutes les lignes de la table seront supprimées. Il est donc essentiel d'utiliser `WHERE` avec précaution pour éviter de supprimer toutes les données de la table par accident.

### Application : suppression d'une ligne

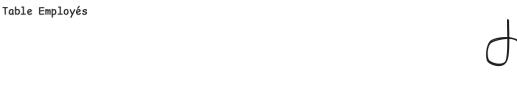
Dans l'exemple suivant, nous allons supprimer l'employé ayant l'`EmployeID` 2 dans la table `Employés` :

```
DELETE FROM Employés WHERE EmployeID = 2;
```

Cette commande va supprimer uniquement l'employé dont l'identifiant est égal à 2. Il est important de noter que la suppression d'une ligne est irréversible, les données seront définitivement perdues après l'exécution de cette commande.

#### Résultat :

La figure ci-dessous montre l'état de la table `Employés` avant et après l'exécution de la commande `DELETE`. L'employé "Jane Smith", ayant l'ID 2, a été supprimé de la table :



EmployeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2019-01-10	2	50000	5.0	NULL	1
2	Jane	Smith	jane.smith@example.com	0602030405	2018-07-20	3	60000	7.0	1	2
3	Tom	Brown	tom.brown@example.com	0603040506	2021-03-10	4	80000	10.0	2	3
4	Emma	Wilson	emma.wilson@example.com	0604050607	2021-06-20	1	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2022-01-15	2	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	1	37000	3.0	1	2
7	Liam	Lee	liam.lee@example.com	0607080910	2020-08-25	1	39000	0.0	2	1

EmployeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	emma.wilson@example.com	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	liam.lee@example.com	0607080910	2020-08-25	1	39000	0.0	2	1

FIGURE 10 – Table `Employés` avant et après la suppression de l'employé avec l'ID 2

Dans cet exemple, la suppression de l'employé "Jane Smith" affecte uniquement cette ligne, et les autres enregistrements de la table restent inchangés.

### 2.7 Sélectionner des lignes

La commande `SELECT` permet de récupérer des données spécifiques d'une ou plusieurs tables. Elle est utilisée pour extraire des informations précises à partir d'une table, en choisissant les colonnes à afficher.

#### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM NomTable;
```

La commande `SELECT` permet de spécifier les colonnes que l'on souhaite afficher. Il est aussi possible d'utiliser `SELECT *` pour sélectionner toutes les colonnes d'une table.

### Application : sélection de colonnes spécifiques

Dans l'exemple suivant, nous allons récupérer les colonnes `Prenom`, `Nom`, et `Email` de la table `Employés` :

```
SELECT Prenom, Nom, Email FROM Employés;
```

Cette requête permet d'extraire uniquement les informations de prénom, nom, et email pour chaque employé.

### Résultat :

L'image ci-dessous montre l'état de la table **Employés** et le résultat de la requête SELECT. La table complète contient plusieurs colonnes, mais seules les colonnes **Prenom**, **Nom**, et **Email** sont sélectionnées par la requête :



The diagram illustrates the selection process. At the top, there is a large rectangular table labeled "Table Employés" containing 10 rows of employee data. An arrow originates from the bottom right corner of this table and points downwards to a smaller, highlighted rectangular box below it. This smaller box contains only the columns "Prenom", "Nom", and "Email" for the same 10 employees, demonstrating how the query has filtered the original data.

EmployeeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	60000	10.0	2	3
4	Emma	Wilson	emma.wilson@example.com	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	liam.lee@example.com	0607080910	2020-08-25	1	39000	0.0	2	1

Prenom	Nom	Email
John	Doe	john.doe@example.com
Tom	Brown	tom.brown@example.com
Emma	Wilson	emma.wilson@example.com
Noah	Taylor	noah.taylor@example.com
Olivia	Johnson	olivia.johnson@example.com
Liam	Lee	null

FIGURE 11 – Table **Employés** et résultat de la requête SELECT **Prenom**, **Nom**, **Email**

Comme on peut le voir, les informations des employés sont filtrées pour ne montrer que les colonnes sélectionnées, simplifiant ainsi l'affichage des données spécifiques demandées.

## 2.8 Filtrer les résultats

La commande WHERE permet de filtrer les résultats d'une requête en fonction d'une condition spécifiée. Elle est utilisée pour ne récupérer que les lignes qui respectent certains critères, ce qui est très utile pour effectuer des recherches spécifiques dans une table.

### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM NomTable WHERE Condition;
```

La condition peut inclure des comparaisons avec des valeurs (=, >, <, etc.), ainsi que des opérateurs logiques comme AND ou OR pour combiner plusieurs critères.

### Application : filtrage des employés ayant un salaire supérieur à 40000

Dans l'exemple suivant, nous allons sélectionner les colonnes **Prenom**, **Nom**, et **Salaire** pour les employés qui ont un salaire supérieur à 40000 :

```
SELECT Prenom, Nom, Salaire FROM Employés WHERE Salaire > 40000;
```

Cette requête permet d'afficher uniquement les employés dont le salaire dépasse 40000, en filtrant les résultats de la table **Employés**.

### Résultat :

L'image ci-dessous montre la table **Employés** avant et après l'application de la condition WHERE. Seuls les employés qui gagnent plus de 40000 sont affichés :

Table Employés

```

graph TD
    A[Table Employés] --> B[SELECT Prenom, Nom, Salaire WHERE Salaire > 40000]
    B --> C[Resultant]

```

The diagram illustrates a selection operation. It starts with a large table labeled "Table Employés". An arrow points from this table to a smaller, highlighted result set below it. The result set contains three columns: "Prenom", "Nom", and "Salaire". The data shows four rows: John Doe (50000), Tom Brown (80000), and Emma Wilson (45000). The row for Noah Taylor (35000) is omitted because its salary is less than 40000.

EmployeeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	NULL	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	NULL	0607080910	2020-08-25	1	39000	0.0	2	1

Prenom	Nom	Salaire
John	Doe	50000
Tom	Brown	80000
Emma	Wilson	45000

FIGURE 12 – Table Employés et résultatat de la requête SELECT Prenom, Nom, Salaire WHERE Salaire > 40000

Comme on peut le voir, la requête a filtré les résultats pour ne conserver que les employés ayant un salaire supérieur à 40000. Les informations des autres employés ne sont pas affichées.

## 2.9 Trier les résultats

La commande ORDER BY permet de trier les résultats d'une requête en fonction d'une ou plusieurs colonnes. Par défaut, les résultats sont triés dans l'ordre croissant (**ASC**), mais il est également possible de les trier dans l'ordre décroissant (**DESC**).

### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM NomTable ORDER BY Colonne ASC|DESC;
```

La commande ORDER BY est souvent utilisée pour organiser les résultats de manière lisible, en triant par une colonne numérique, alphabétique ou chronologique. Le tri peut se faire dans l'ordre croissant (**ASC**) ou décroissant (**DESC**).

### Application : tri par salaire décroissant

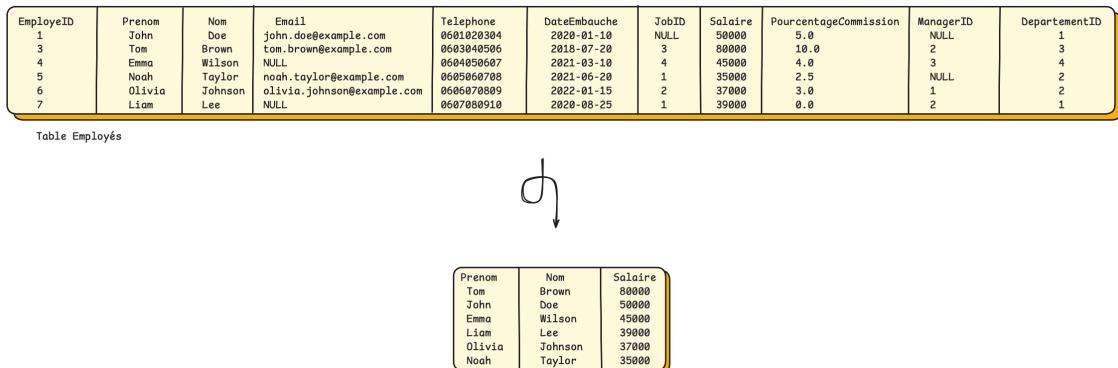
Dans l'exemple suivant, nous allons récupérer les colonnes **Prenom**, **Nom**, et **Salaire** pour les employés, et trier les résultats par salaire dans l'ordre décroissant :

```
SELECT Prenom, Nom, Salaire FROM Employés ORDER BY Salaire DESC;
```

Cette requête trie les employés par salaire du plus élevé au plus bas.

### Résultat :

L'image ci-dessous montre la table **Employés** ainsi que le résultat trié selon le salaire en ordre décroissant :



The diagram illustrates the relationship between a database table and its query results. At the top, a table named "Employés" is shown with columns: EmployéID, Prenom, Nom, Email, Telephone, DateEmbauche, JobID, Salaire, PourcentageCommission, ManagerID, and DépartementID. Below it, a curved arrow points down to a smaller table containing only the columns Prenom, Nom, and Salaire. This smaller table's data is ordered by Salaire in descending order, resulting in the highest salaries at the top.

EmployéID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DépartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	emma.wilson@example.com	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	liam.lee@example.com	0607080910	2020-08-25	1	39000	0.0	2	1

Prenom	Nom	Salaire
Tom	Brown	80000
John	Doe	50000
Emma	Wilson	45000
Liam	Lee	39000
Olivia	Johnson	37000
Noah	Taylor	35000

FIGURE 13 – Table Employés et résultat de la requête `SELECT Prenom, Nom, Salaire ORDER BY Salaire DESC`

Comme on peut le voir, la requête a trié les employés selon leur salaire, du plus élevé au plus bas, ce qui permet de visualiser rapidement les employés les mieux rémunérés.

## 2.10 Conditions logiques

Les conditions logiques comme AND, OR, et NOT permettent de combiner plusieurs critères dans une requête WHERE. Elles sont utilisées pour affiner la recherche et filtrer les résultats selon plusieurs conditions à la fois.

### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM NomTable WHERE Condition1 AND|OR Condition2;
```

- AND est utilisé pour récupérer les résultats qui satisfont à toutes les conditions.
- OR permet de récupérer les résultats qui satisfont à au moins une des conditions.
- NOT est utilisé pour exclure certains résultats qui répondent à une condition spécifique.

### Application : filtrer les employés avec plusieurs conditions

Dans l'exemple suivant, nous allons sélectionner les colonnes Prenom, Nom, Email et ManagerID des employés qui ont un email et qui ne travaillent pas sous le manager ayant l'ID 2 :

```
SELECT Prenom, Nom, Email, ManagerID FROM Employés
WHERE Email IS NOT NULL AND NOT ManagerID = 2;
```

Cette requête filtre les résultats pour ne garder que les employés ayant une adresse email (donc l'email n'est pas NULL) et qui ne sont pas sous la supervision du manager d'ID 2.

### Résultat :

L'image ci-dessous montre la table Employés et le résultat de la requête, qui filtre les employés selon les conditions combinées :

EmployeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	NULL	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	NULL	0607080910	2020-08-25	1	39000	0.0	2	1

Table Employés

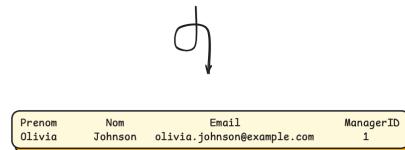


FIGURE 14 – Table Employés et résultat de la requête SELECT ... WHERE Email IS NOT NULL AND NOT ManagerID = 2

Comme on peut le voir, la requête a renvoyé l'employé "Olivia Johnson", qui a un email et dont le ManagerID n'est pas 2.

## 2.11 Calculs sur des colonnes (Avec AS)

Il est possible d'effectuer des calculs sur les colonnes d'une table, tels que des sommes, des multiplications, des soustractions, ou d'autres opérations arithmétiques. L'opérateur **AS** permet de renommer le résultat de ces calculs pour une lecture plus claire.

### Syntaxe générale

```
SELECT Colonne1, (Colonne2 * Colonne3) AS NouvelleColonne FROM NomTable;
```

L'opérateur **AS** est utile pour donner un nom explicite à une nouvelle colonne résultant d'un calcul entre plusieurs colonnes existantes.

### Application : calcul du salaire total avec le pourcentage de commission

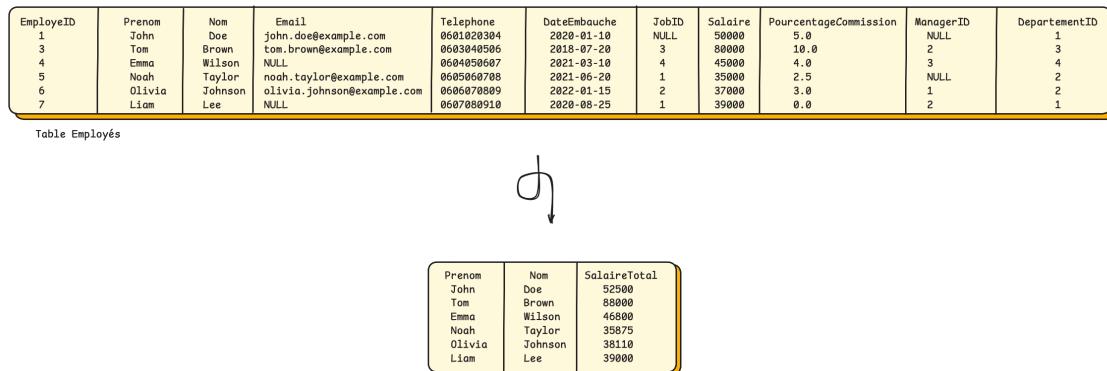
Dans l'exemple suivant, nous allons calculer le salaire total de chaque employé en ajoutant au salaire de base le montant de la commission, qui est calculé en fonction du PourcentageCommission :

```
SELECT Prenom, Nom, (Salaire + (Salaire * PourcentageCommission / 100))
AS SalaireTotal FROM Employés;
```

Cette requête calcule une nouvelle colonne nommée **SalaireTotal**, qui prend en compte le salaire de base et la commission basée sur un pourcentage.

### Résultat :

L'image ci-dessous montre la table **Employés** et le résultat de la requête qui calcule le salaire total de chaque employé :



The diagram illustrates a SELECT query operation. At the top, there is a table labeled "Table Employés" with the following data:

EmployeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	NULL	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	NULL	0607080910	2020-08-25	1	39000	0.0	2	1

An arrow points from the bottom right of the original table to a smaller, highlighted table below, which contains the results of the query:

Prenom	Nom	SalaireTotal
John	Doe	52500
Tom	Brown	88000
Emma	Wilson	46800
Noah	Taylor	35875
Olivia	Johnson	38110
Liam	Lee	39000

FIGURE 15 – Table Employés et résultatat de la requête SELECT Prenom, Nom, SalaireTotal

Comme on peut le voir, la colonne **SalairéTotal** a été ajoutée au résultatat, indiquant le salaire total pour chaque employé en incluant la commission.

## 2.12 Les jointures

Les jointures en SQL permettent de combiner des données provenant de plusieurs tables en fonction de critères communs, généralement une colonne partagée entre les tables. Elles sont indispensables lorsqu'on travaille avec des bases de données relationnelles, car elles permettent de relier des informations dispersées dans différentes tables.

Chacune des jointures est adaptée à des cas d'usage différents, selon les besoins de la requête et la manière dont les tables sont reliées entre elles. Nous allons explorer différents types de jointures à travers des exemples pratiques.

### 2.12.1 INNER JOIN

L'**INNER JOIN** est le type de jointure le plus couramment utilisé. Il ne renvoie que les lignes qui ont des correspondances dans les deux tables. Si une ligne d'une table n'a pas de correspondance dans l'autre table, elle n'apparaît pas dans le résultatat.

#### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM TableGauche
INNER JOIN TableDroite ON TableGauche.Collonne = TableDroite.Collonne;
```

Cette jointure est utilisée lorsqu'on souhaite récupérer uniquement les données qui ont des correspondances dans les deux tables.

#### Illustration :

L'image ci-dessous illustre le fonctionnement d'une jointure **INNER JOIN**, où seules les données présentes dans les deux tables sont renvoyées :

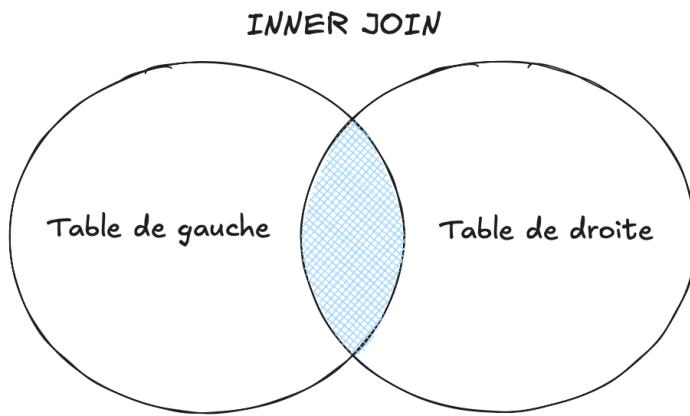


FIGURE 16 – Illustration de l'INNER JOIN

### Application : récupérer le titre de poste des employés

Dans l'exemple suivant, nous allons récupérer le prénom, le nom, et le titre de poste (`JobTitre`) des employés, mais seulement pour les employés qui ont un job associé dans la table `Jobs` :

```
SELECT Employés.Prenom, Employés.Nom, Jobs.JobTitre
FROM Employés
INNER JOIN Jobs ON Employés.JobID = Jobs.JobID;
```

Cette requête relie les tables `Employés` et `Jobs` en utilisant la colonne `JobID` qui est commune aux deux tables. Seuls les employés ayant un job défini dans la table `Jobs` apparaîtront dans les résultats.

### Résultat :

Voici le résultat de la requête, qui combine les informations de la table `Employés` et de la table `Jobs` pour afficher le titre de poste des employés :

EmployéID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0610028304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0633040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	NULL	0604056067	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605069708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070889	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	NULL	0607080910	2020-08-25	1	39000	0.0	2	1

Table Employés

JobID	JobTitre	MinSalaire	MaxSalaire
1	Développeur	30000	60000
2	Analyste	35000	70000
3	Manager	45000	90000
4	Commercial	25000	50000
5	Technicien	20000	40000

Table Jobs

Prenom	Nom	JobTitre
Tom	Brown	Manager
Emma	Wilson	Commercial
Noah	Taylor	Technicien
Olivia	Johnson	Analyste
Liam	Lee	Développeur

FIGURE 17 – Table `Employés`, Table `Jobs` et résultat de l'INNER JOIN

Comme on peut le voir, seuls les employés qui ont un `JobID` associé dans la table `Jobs` apparaissent dans le résultat, avec leur titre de poste correspondant.

### 2.12.2 LEFT JOIN (ou LEFT OUTER JOIN)

Le LEFT JOIN renvoie toutes les lignes de la table de gauche (la première table mentionnée) et les lignes correspondantes de la table de droite. Si une ligne de la table de gauche n'a pas de correspondance dans la table de droite, les colonnes de la table de droite seront complétées par NULL.

#### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM TableGauche
LEFT JOIN TableDroite ON TableGauche.Collonne = TableDroite.Collonne;
```

Cette jointure est utile lorsqu'on souhaite récupérer toutes les données de la première table, même si certaines d'entre elles n'ont pas de correspondance dans la seconde table.

#### Illustration :

L'image ci-dessous illustre le fonctionnement d'un LEFT JOIN, où toutes les lignes de la table de gauche sont renvoyées, avec les correspondances de la table de droite :

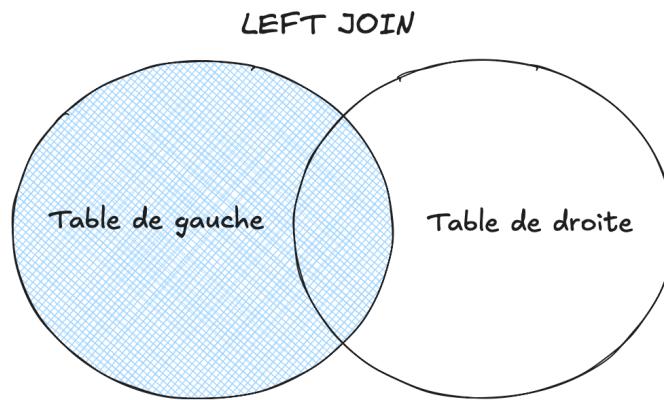


FIGURE 18 – Illustration du LEFT JOIN

#### Application : récupérer tous les employés, même ceux sans job associé

Dans l'exemple suivant, nous allons récupérer le prénom, le nom, et le titre de poste (`JobTitre`) des employés. Contrairement à un INNER JOIN, cette requête renverra tous les employés, même ceux qui n'ont pas de job associé dans la table `Jobs` :

```
SELECT Employés.Prenom, Employés.Nom, Jobs.JobTitre
FROM Employés
LEFT JOIN Jobs ON Employés.JobID = Jobs.JobID;
```

Avec cette requête, les employés sans job auront des valeurs NULL pour les colonnes venant de la table `Jobs`, mais ils apparaîtront quand même dans les résultats.

#### Résultat :

Voici le résultat de la requête, qui combine les informations des tables `Employés` et `Jobs`, en incluant les employés sans job associé :

The diagram shows three tables: **Table Employés**, **Table Jobs**, and the resulting **LEFT JOIN** result.

**Table Employés:**

EmployeeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020384	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040586	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	emma.wilson@example.com	0604050687	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060788	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070889	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	liam.lee@example.com	0607080910	2020-08-25	1	39000	0.0	2	1

**Table Jobs:**

JobID	JobTitre	MinSalaire	MaxSalaire
1	Développeur	30000	60000
2	Analyste	35000	70000
3	Manager	45000	90000
4	Commercial	25000	50000
5	Technicien	20000	40000

**Resultat du LEFT JOIN:**

Prenom	Nom	JobTitre
John	Doe	Null
Tom	Brown	Manager
Emma	Wilson	Commercial
Noah	Taylor	Technicien
Olivia	Johnson	Analyste
Liam	Lee	Développeur

FIGURE 19 – Table Employés, Table Jobs et résultat du LEFT JOIN

Comme on peut le voir, tous les employés sont inclus dans le résultat, même ceux qui n'ont pas de job associé, et leurs colonnes liées à la table Jobs sont remplies par des valeurs NULL.

### 2.12.3 RIGHT JOIN (ou RIGHT OUTER JOIN)

Le RIGHT JOIN est l'inverse du LEFT JOIN. Il renvoie toutes les lignes de la table de droite et les lignes correspondantes de la table de gauche. Si une ligne de la table de droite n'a pas de correspondance dans la table de gauche, les colonnes de la table de gauche seront complétées par NULL.

#### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM TableGauche
RIGHT JOIN TableDroite ON TableGauche.Collonne = TableDroite.Collonne;
```

Cette jointure est utile lorsqu'on souhaite récupérer toutes les données de la seconde table (la table de droite), même si certaines d'entre elles n'ont pas de correspondance dans la première table.

#### Illustration :

L'image ci-dessous illustre le fonctionnement d'un RIGHT JOIN, où toutes les lignes de la table de droite sont renvoyées, avec les correspondances de la table de gauche :

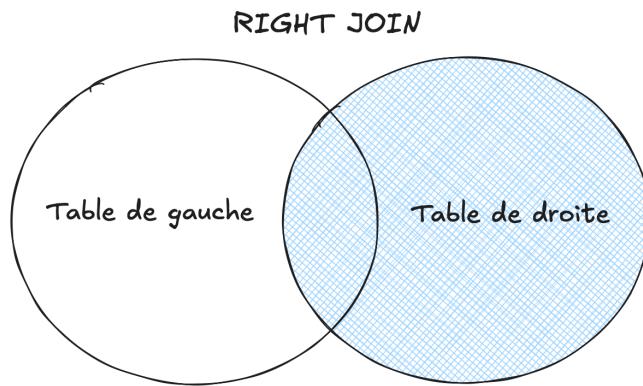


FIGURE 20 – Illustration du RIGHT JOIN

### Application : récupérer tous les jobs, même ceux sans employé associé

Dans l'exemple suivant, nous allons récupérer le prénom, le nom, et le titre de poste (`JobTitre`) des employés. Contrairement à un `LEFT JOIN`, cette requête renverra tous les jobs, même ceux qui ne sont associés à aucun employé dans la table `Employés` :

```
SELECT Employés.Prenom, Employés.Nom, Jobs.JobTitre
FROM Employés
RIGHT JOIN Jobs ON Employés.JobID = Jobs.JobID;
```

Avec cette requête, les jobs sans employés auront des valeurs `NULL` pour les colonnes venant de la table `Employés`, mais ils apparaîtront quand même dans les résultats.

### Résultat :

Voici le résultat de la requête, qui combine les informations des tables `Employés` et `Jobs`, en incluant les jobs sans employé associé :

EmployéID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	NULL	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	NULL	0607080910	2020-08-25	1	39000	0.0	2	1

Table Employés

JobID	JobTitre	MinSalaire	MaxSalaire
1	Développeur	30000	60000
2	Analyste	35000	70000
3	Manager	45000	90000
4	Commercial	25000	50000
5	Technicien	20000	40000

Table Jobs

↓

Prenom	Nom	JobTitre
Tom	Brown	Manager
Emma	Wilson	Commercial
Noah	Taylor	Développeur
Olivia	Johnson	Analyste
Liam	Lee	Développeur
null	null	Technicien

FIGURE 21 – Table `Employés`, Table `Jobs` et résultat du RIGHT JOIN

Comme on peut le voir, tous les jobs sont inclus dans le résultat, même ceux qui ne sont associés à aucun employé, et leurs colonnes liées à la table `Employés` sont remplies par des valeurs `NULL`.

#### 2.12.4 FULL JOIN (ou FULL OUTER JOIN)

Le FULL JOIN renvoie toutes les lignes des deux tables, que la correspondance existe ou non. Si une ligne de la table de gauche ou de la table de droite n'a pas de correspondance dans l'autre table, des valeurs NULL seront affichées pour les colonnes manquantes.

##### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM TableGauche
FULL JOIN TableDroite ON TableGauche.Collonne = TableDroite.Collonne;
```

Le FULL JOIN est utile lorsque l'on souhaite récupérer toutes les données de deux tables, même celles sans correspondance, afin d'avoir une vue d'ensemble complète des informations.

##### Illustration :

L'image ci-dessous illustre le fonctionnement d'un FULL JOIN, où toutes les lignes des deux tables sont renvoyées, qu'il y ait correspondance ou non :

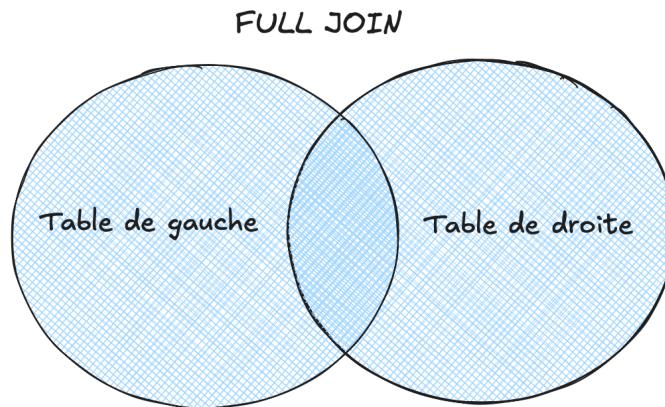


FIGURE 22 – Illustration du FULL JOIN

##### Application : récupérer tous les employés et tous les jobs, même sans correspondance

Dans l'exemple suivant, nous allons récupérer le prénom, le nom, et le titre de poste (`JobTitre`) des employés. Contrairement aux autres types de jointures, cette requête renverra tous les employés et tous les jobs, même s'ils ne sont pas associés entre eux :

```
SELECT Employés.Prenom, Employés.Nom, Jobs.JobTitre
FROM Employés
FULL OUTER JOIN Jobs ON Employés.JobID = Jobs.JobID;
```

Avec cette requête, tous les employés et tous les jobs apparaîtront dans le résultat, et les colonnes qui n'ont pas de correspondance seront remplies par des valeurs NULL.

##### Résultat :

Voici le résultat de la requête, qui combine les informations des tables `Employés` et `Jobs`, en incluant les employés sans job et les jobs sans employés :

The diagram illustrates the execution of a FULL JOIN between two tables: `Employés` and `Jobs`. It shows the initial tables, the resulting full join table, and the final result after applying a WHERE clause.

**Table Employés**

EmployeeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	NULL	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	NULL	0607080910	2020-08-25	1	39000	0.0	2	1

**Table Jobs**

JobID	JobTitre	MinSalaire	MaxSalaire
1	Développeur	30000	60000
2	Analyste	35000	70000
3	Manager	45000	90000
4	Commercial	25000	50000
5	Technicien	20000	40000

**Resultat du FULL JOIN**

Prenom	Nom	JobTitre
John	Doe	null
Tom	Brown	Manager
Emma	Wilson	Commercial
Noah	Taylor	Développeur
Olivia	Johnson	Analyste
Liam	Lee	Développeur
null	null	Technicien

FIGURE 23 – Table `Employés`, Table `Jobs` et résultat du FULL JOIN

Comme on peut le voir, tous les employés et tous les jobs sont inclus dans le résultat, même ceux qui n'ont pas de correspondance, avec des valeurs NULL pour les colonnes non appariées.

#### 2.12.5 SELF JOIN

Le SELF JOIN permet de joindre une table à elle-même. Ce n'est pas une commande spécifique, mais plutôt une utilisation d'un JOIN sur une seule et même table. Il est principalement utilisé pour établir des relations entre des lignes d'une même table, comme relier des employés à leur manager.

#### Application : récupérer les employés et leur manager

Dans l'exemple suivant, nous allons sélectionner les employés et les lier à leur manager en utilisant un LEFT JOIN sur la table `Employés`, en la joignant à elle-même :

```
SELECT E1.Prenom, E1.Nom, E2.Prenom as ManagerPrenom, E2.Nom as ManagerNom
FROM Employés E1
LEFT JOIN Employés E2 ON E1.ManagerID = E2.EmployeeID;
```

Cette requête permet de récupérer tous les employés ainsi que leurs managers, même pour les employés qui n'ont pas de manager (les managers sont représentés par des valeurs NULL dans ce cas).

#### Résultat :

Voici le résultat de la requête, qui combine les informations des employés et de leurs managers en utilisant un SELF JOIN sur la table `Employés` :

EmployeeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	NULL	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	NULL	0607080910	2020-08-25	1	39000	0.0	2	1

Table Employés E1

EmployeeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	NULL	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	NULL	0607080910	2020-08-25	1	39000	0.0	2	1

Table Employés E2



Prenom	Nom	ManagerPrenom	ManagerNom
John	Doe	null	null
Tom	Brown	null	Brown
Emma	Wilson	Tom	Tom
Noah	Taylor	null	null
Olivia	Johnson	John	John
Liam	Lee	Doe	Doe
		null	null

On rappelle qu'on a supprimé l'employé avec l'ID 2 au début, c'est pour cela que le manager est NULL ici

FIGURE 24 – Table Employés E1, Table Employés E2 et résultat du SELF JOIN

Dans cet exemple, nous récupérons tous les employés avec les informations de leur manager, s'ils en ont un. Pour les employés sans manager, les colonnes ManagerPrenom et ManagerNom contiennent des valeurs NULL. Il est important de rappeler qu'un employé a été supprimé précédemment avec l'ID 2, ce qui explique pourquoi le manager de certains employés est NULL.

### 2.12.6 Jointures multiples

Il est tout à fait possible d'utiliser plus de deux tables dans une même jointure. Chaque jointure peut être un INNER JOIN, LEFT JOIN, RIGHT JOIN ou FULL JOIN, selon les besoins, et elles s'enchaînent simplement dans la requête.

#### Syntaxe générale

```
SELECT Colonne1, Colonne2, Colonne3
FROM TableA
JOIN TableB ON TableA.Key = TableB.Key
JOIN TableC ON TableB.Key = TableC.Key;
```

Dans cette syntaxe, la première jointure est effectuée entre TableA et TableB, puis une deuxième jointure relie TableB à TableC. Ce processus peut être étendu à plusieurs tables en fonction des besoins de la requête.

### Application : récupérer les employés, leurs jobs et leurs départements

Dans l'exemple suivant, nous allons récupérer les informations des employés, ainsi que leur titre de poste et le nom de leur département. Nous utilisons des LEFT JOIN pour nous assurer que tous les employés apparaissent, même s'ils n'ont pas de job ou de département associé :

```
SELECT Employés.Prenom, Employés.Nom, Jobs.JobTitre, Départements.DepartementNom
FROM Employés
LEFT JOIN Jobs ON Employés.JobID = Jobs.JobID
LEFT JOIN Départements ON Employés.DepartementID = Départements.DepartementID;
```

Cette requête permet d'afficher tous les employés, y compris ceux qui n'ont pas de job ou de département assigné. Les colonnes sans correspondance seront remplacées par des valeurs NULL.

Résultat :

Voici le résultatat de la requête, qui combine les informations des tables Employés, Jobs et Departements :

The diagram illustrates the joining of three tables: Employes, Jobs, and Departements. It shows the individual tables at the top, followed by a join operation indicated by a curved arrow pointing downwards, which results in a single joined table at the bottom.

EmployeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	0601020304	2013-07-20	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2013-07-20	3	80000	10.0	2	3
4	Emma	Wilson	emma.wilson@example.com	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	liam.lee@example.com	0607080910	2020-08-25	1	39000	8.0	2	1

JobID	JobTitre	MinSalaire	MaxSalaire
1	Développeur	30000	60000
2	Analyste	35000	70000
3	Manager	45000	90000
4	Commercial	25000	50000
5	Technicien	20000	40000

DepartementID	DepartementNom	ManagerID	AdresseID
1	Ressources Humaines	null	1
2	Informatique	1	2
3	Marketing	2	3
4	Ventes	null	4

Prenom	Nom	JobTitre	DepartementNom
John	Doe	null	Ressources Humaines
Tom	Brown	Manager	Informatique
Emma	Wilson	Commercial	Marketing
Noah	Taylor	Développeur	Ventes
Olivia	Johnson	Analyste	Informatique
Liam	Lee	Développeur	Ressources Humaines

FIGURE 25 – Tables Employés, Jobs, Departements et résultatat des jointures multiples

Comme on peut le voir, tous les employés sont inclus dans le résultatat, même ceux sans job ou sans département associé. Les valeurs manquantes sont remplies par des NULL.

## 2.13 Utiliser les fonctions SUM, COUNT, AVG, ... avec GROUP BY

Les fonctions d'agrégation permettent de calculer des statistiques telles que la somme (SUM), la moyenne (AVG), le nombre (COUNT), le minimum (MIN), et le maximum (MAX) sur des colonnes spécifiques. Elles sont très utiles pour obtenir des résumés de données.

La commande GROUP BY permet de regrouper les résultats en fonction d'une ou plusieurs colonnes, et de calculer les agrégations pour chaque groupe.

### Syntaxe générale

```
SELECT SUM(Colonne), COUNT(Colonne), AVG(Colonne)
FROM NomTable
GROUP BY Colonne;
```

En utilisant la commande GROUP BY, on peut regrouper les lignes en fonction des valeurs d'une ou plusieurs colonnes et appliquer des fonctions d'agrégation sur ces groupes.

### Application : calculer le salaire moyen par département

Dans l'exemple suivant, nous allons calculer le salaire moyen des employés pour chaque département en utilisant la fonction AVG et en regroupant les résultats par DepartementID :

```
SELECT DepartementID, AVG(Salaire) AS SalaireMoyen
FROM Employés
GROUP BY DepartementID;
```

Cette requête calcule la moyenne des salaires pour chaque département, et les résultats sont affichés dans une colonne nommée **SalaireMoyen**.

### Résultat :

L'image ci-dessous montre la table **Employés** et le résultat de la requête qui calcule le salaire moyen par département :

The diagram illustrates a query flow. At the top, there is a table labeled "Table Employés" containing data for seven employees. An arrow points downwards from this table to a second table below it, which contains four rows representing the average salary for each department.

EmployeeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	061028304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603046506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	emma.wilson@example.com	0604056607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605066708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	liam.lee@example.com	0607080910	2020-08-25	1	39000	0.0	2	1

DepartementID	SalaireMoyen
1	44500
2	36000
3	80000
4	45000

FIGURE 26 – Table **Employés** et résultat du calcul du salaire moyen par département avec **GROUP BY**

Comme on peut le voir, chaque département est affiché avec le salaire moyen de ses employés.

## 2.14 Combiner WHERE et GROUP BY

Il est possible d'utiliser la clause **WHERE** avant **GROUP BY** pour filtrer les résultats avant de les regrouper. Cela permet d'appliquer une condition sur les données avant que celles-ci ne soient regroupées par la commande **GROUP BY**.

### Syntaxe générale

```
SELECT Colonne, SUM(Colonne)
FROM NomTable
WHERE Condition
GROUP BY Colonne;
```

Cette structure permet de filtrer les lignes à l'aide de la clause **WHERE** avant de les regrouper selon la colonne spécifiée dans **GROUP BY**.

### Application : calculer le nombre d'employés par département avec un salaire supérieur à 36 000

Dans l'exemple suivant, nous allons compter le nombre d'employés par département, mais uniquement pour ceux ayant un salaire supérieur à 36 000 :

```
SELECT DepartementID, COUNT(EmployeeID) AS NombreEmployes
FROM Employés
WHERE Salaire > 36000
GROUP BY DepartementID;
```

Cette requête calcule le nombre d'employés pour chaque département, en filtrant d'abord ceux dont le salaire est supérieur à 36 000.

### Résultat :

L'image ci-dessous montre la table **Employés** et le résultat de la requête qui calcule le nombre d'employés par département ayant un salaire supérieur à 36 000 :

EmployeeID	Prenom	Nom	Email	Telephone	DateEmbauche	JobID	Salaire	PourcentageCommission	ManagerID	DepartementID
1	John	Doe	john.doe@example.com	061020304	2020-01-10	NULL	50000	5.0	NULL	1
3	Tom	Brown	tom.brown@example.com	0603040506	2018-07-20	3	80000	10.0	2	3
4	Emma	Wilson	NULL	0604050607	2021-03-10	4	45000	4.0	3	4
5	Noah	Taylor	noah.taylor@example.com	0605060708	2021-06-20	1	35000	2.5	NULL	2
6	Olivia	Johnson	olivia.johnson@example.com	0606070809	2022-01-15	2	37000	3.0	1	2
7	Liam	Lee	NULL	0607080910	2020-08-25	1	39000	0.0	2	1

Table Employés



DepartementID	NombreEmployes
1	2
2	1
3	1
4	1

FIGURE 27 – Table **Employés** et résultat de la combinaison WHERE et GROUP BY

Comme on peut le voir, seuls les employés ayant un salaire supérieur à 36 000 sont comptabilisés dans les résultats par département.

### 3 Fiche résumé

#### Créer des tables

##### Syntaxe générale

```
CREATE TABLE NomTable (
    NomColonne TypeData Contraintes,
    ...
);
```

Exemple :

```
CREATE TABLE Employés (
    EmployeID INT PRIMARY KEY,
    Prenom VARCHAR(100),
    Nom VARCHAR(100)
);
```

#### Supprimer des tables

##### Syntaxe générale

```
DROP TABLE NomTable;
```

Exemple :

```
DROP TABLE Employés;
```

#### Modifier des tables

##### Ajouter supprimer renommer des colonnes

```
ALTER TABLE NomTable ADD NomColonne TypeData;
ALTER TABLE NomTable DROP NomColonne;
ALTER TABLE NomTable RENAME NomColonneAncien TO NomColonneNouveau;
```

Exemple :

```
ALTER TABLE Employés RENAME Nom TO LastName;
```

#### Ajouter des lignes

##### Syntaxe générale

```
INSERT INTO NomTable (Colonne1, Colonne2) VALUES (Valeur1, Valeur2);
```

Exemple :

```
INSERT INTO Employés (EmployeID, Prenom, Nom) VALUES (1, 'John', 'Doe');
```

#### Supprimer des lignes

##### Syntaxe générale

```
DELETE FROM NomTable WHERE Condition;
```

Exemple :

```
DELETE FROM Employés WHERE EmployeID = 1;
```

#### Sélectionner des lignes

##### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM NomTable;
```

Exemple :

```
SELECT Prenom, Nom FROM Employés;
```

## Filtrer les résultats

### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM NomTable WHERE Condition;
```

Exemple :

```
SELECT Prenom, Nom FROM Employés WHERE Salaire > 40000;
```

## Trier les résultats

### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM NomTable ORDER BY Colonne ASC|DESC;
```

Exemple :

```
SELECT Prenom, Nom FROM Employés ORDER BY Salaire DESC;
```

## Conditions logiques

### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM NomTable  
WHERE Condition1 AND/OR Condition2;
```

Exemple :

```
SELECT Prenom, Nom FROM Employés  
WHERE Salaire > 40000 AND ManagerID IS NOT NULL;
```

## Calculs sur les colonnes (Avec AS)

### Syntaxe générale

```
SELECT Colonne1, (Colonne2 * Colonne3) AS NouvelleColonne  
FROM NomTable;
```

Exemple :

```
SELECT Prenom, Nom, (Salaire + (Salaire * PourcentageCommission / 100))  
AS SalaireTotal FROM Employés;
```

## Les jointures

### INNER JOIN

#### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM TableA  
INNER JOIN TableB ON TableA.Collonne = TableB.Collonne;
```

### LEFT JOIN (ou LEFT OUTER JOIN)

#### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM TableA  
LEFT JOIN TableB ON TableA.Collonne = TableB.Collonne;
```

### RIGHT JOIN (ou RIGHT OUTER JOIN)

#### Syntaxe générale

```
SELECT Colonne1, Colonne2 FROM TableA  
RIGHT JOIN TableB ON TableA.Collonne = TableB.Collonne;
```

### FULL JOIN (ou FULL OUTER JOIN)

**Syntaxe générale**

```
SELECT Colonne1, Colonne2 FROM TableA  
FULL JOIN TableB ON TableA.Collonne = TableB.Collonne;
```

**SELF JOIN****Syntaxe générale**

```
SELECT E1.Collonne1, E2.Collonne1  
FROM TableA E1  
LEFT JOIN TableA E2 ON E1.Collonne = E2.Collonne;
```

**Jointures multiples****Syntaxe générale**

```
SELECT Colonne1, Colonne2, Colonne3 FROM TableA  
JOIN TableB ON TableA.Key = TableB.Key  
JOIN TableC ON TableB.Key = TableC.Key;
```

**Utiliser les fonctions SUM, COUNT, AVG, ... avec GROUP BY****Syntaxe générale**

```
SELECT SUM(Collonne), COUNT(Collonne), AVG(Collonne)  
FROM NomTable  
GROUP BY Collonne;
```

**Combiner WHERE et GROUP BY****Syntaxe générale**

```
SELECT Collonne, SUM(Collonne)  
FROM NomTable  
WHERE Condition  
GROUP BY Collonne;
```