



TRAVAIL PERSONNEL
COURS

Cours Complet SQL

By :
Rémy GASMI

14 mai 2025



Table des matières

1	Les bases de données relationnelles	6
1.1	Définition	6
1.2	Qu'est-ce que SQL ?	6
1.3	Pourquoi le SQL ?	6
1.4	Différents modèles	7
1.5	Les différents systèmes de gestion de bases de données relationnelles	7
1.6	Les différents types de données	8
1.7	Notions de clés primaires/secondaires	9
1.7.1	Clé primaire (PRIMARY KEY)	9
1.7.2	Clé secondaire (FOREIGN KEY)	9
1.7.3	Clé unique (UNIQUE KEY)	10
1.7.4	Clé Primaire VS Clé unique	11
2	Sélection de données	12
2.1	SELECT	13
2.1.1	SELECT DISTINCT	13
2.2	FROM	13
2.3	WHERE	14
2.3.1	Les opérateurs de comparaisons	14
2.3.2	AND & OR	14
2.4	GROUP BY	15
2.5	HAVING	15
2.6	ORDER BY	15
2.7	LIMIT	16
2.7.1	Limit et Offset	16
3	Fonctions utiles et classiques (Fonctions d'agrégation)	18
3.1	COUNT()	18
3.2	SUM()	18
3.3	AVG()	19
3.4	ROUND()	19
3.5	ISNULL()	19
3.6	MAX()	20
3.7	MIN()	20
4	Jointures	21
4.1	JOIN/INNER JOIN	21
4.2	LEFT JOIN	21
4.3	RIGHT JOIN	21
4.4	FULL OUTER JOIN	21
4.5	CROSS JOIN	21
4.6	Récapitulatif	22

5	CTE(Common Table Expression)	23
5.1	Présentation des CTE	23
5.2	Plusieurs CTE	24
5.3	UNION et UNION ALL	25
6	Les requêtes temporelles et les dates	26
6.1	DATE_FORMAT()	26
6.2	DATEDIFF()	28
6.3	DATE_ADD/DATE_SUB	28
6.4	NOW, DAY, MONTH, YEAR	29
6.4.1	NOW	29
6.4.2	CURDATE/CURRENT_DATE	29
6.4.3	CURRENT_TIME	29
6.4.4	DAY	29
6.4.5	MONTH	29
6.4.6	YEAR	29
6.5	FORMULAIRE DATE	30
7	Logique Conditionnelle	33
7.1	COALESCE	33
7.2	NULLIF	33
7.3	CASE WHEN	33
7.4	IF	34
7.5	EXISTS	34
7.6	GREATEST et LEAST	35
8	Opérations de pivotement	36
8.1	PIVOT	36
8.2	UNPIVOT	37
9	Les expressions régulières en SQL (REGEX)	38
9.1	Définitions et Bases	38
9.2	Exemples et astuces	39
9.3	Ressources utiles	39
10	Fenêtres analytiques/Fonctions de fenêtrages (Window Functions)	40
10.1	Qu'est-ce qu'une fenêtre?	40
10.2	OVER(PARTITION BY...)	40
10.3	ROW_NUMBER	41
10.4	RANK	42
10.5	LAG() et LEAD()	42
10.5.1	LAG()	42
10.5.2	LEAD()	42
10.5.3	Source	42

11 Les opérations CRUD et quelques autres commandes	43
11.1 CREATE TABLE	43
11.2 READ	43
11.3 UPDATE	44
11.4 DELETE	44
11.5 TRUNCATE	44
11.6 DROP TABLE	45
11.7 ALTER TABLE	45
11.7.1 Ajouter une colonne	45
11.7.2 Supprimer une colonne	45
11.7.3 Modifier une colonne	46
11.7.4 Renommer une colonne	46
12 Transactions et propriétés ACID	47
12.1 Définition d'une transaction	47
12.2 Les propriétés ACID	47
12.2.1 L'isolation : les niveaux	47
12.3 Commandes de gestion des transactions en SQL	48
12.3.1 START TRANSACTION ou BEGIN	48
12.3.2 COMMIT	48
12.3.3 ROLLBACK	49
12.3.4 SAVEPOINT et ROLLBACK TO SAVEPOINT	49
12.4 Détection et gestion des erreurs transactionnelles	49
12.5 Transactions imbriquées et limites selon les SGBD	50
12.6 Bonnes pratiques	50
13 Procédures stockées	51
13.1 Définition et utilité	51
13.2 Syntaxe générale (MySQL ou PostgreSQL)	51
13.2.1 Syntaxe	51
13.2.2 Exemple	52
13.3 Contrôle de flux dans les procédures	52
14 Triggers (déclencheurs)	53
14.1 Définition	53
14.2 Fonctionnement général	53
14.2.1 Moment de déclenchement	53
14.2.2 Type d'événement déclencheur	53
14.2.3 Portée de déclenchement	54
14.3 Syntaxe générale d'un Trigger	54
15 Index & Optimisation des requêtes	55
15.1 Types d'index	55
15.2 Création et gestion des index	56
15.2.1 Création d'un index simple	56
15.2.2 Création d'un index composé	56
15.2.3 Création d'un index unique	56
15.2.4 Suppression d'un index	56

15.2.5	Index implicites	57
15.2.6	Bonnes pratiques	57
15.3	Optimisation des requêtes avec les index	57
15.3.1	Index et tri (ORDER BY)	58
15.3.2	Index et jointures	58
15.3.3	Choix des colonnes à indexer	59
15.3.4	Impact sur les performances en écriture	59
15.4	Analyse des performances avec EXPLAIN	60
15.4.1	Syntaxe de base	60
15.4.2	Résultat typique (MySQL)	60
15.4.3	Avec EXPLAIN ANALYZE (PostgreSQL)	61
16	Boite à outils	62
16.1	STRCMP	62
16.2	IN/NOT IN	62
16.3	LIKE	62
16.3.1	Wildcards	62
16.3.2	Syntaxe et exemple	64
16.4	BETWEEN ... AND	64
16.5	TRIM	64
16.6	CONCAT	65
16.7	GROUP_CONCAT	66
16.8	REPLACE	67
16.9	CAST	67
16.10	Les chaînes de caractères	68
16.10.1	Fonctions MID, RIGHT, LEFT	68
16.10.2	Obtenir le nombre de caractères	68
16.10.3	Majuscule/Minsucule	68
16.10.4	Segmenter une chaîne de caractère	68
16.10.5	Obtenir la position d'une occurrence	68
16.10.6	Gérer les Noms Propres	69
17	SQL avec Python	70
17.1	sqlite3	70
17.2	sqlglot	71
18	Le SQL dans la Data Analyse	72
18.1	Identifier les doublons	72
18.2	Identifier les valeurs NULL	72
18.3	Vérifier la validité des valeurs	72
18.4	Vérifier les valeurs de date	72
18.5	Vérifier l'intégrité des données	72
18.6	Identifier les données orphelines	72
18.7	Vérifier si la quantité est un nombre	72
18.8	Détecter les valeurs aberrantes à l'aide de l'IQR (Interquartile Range)	73
18.9	Vérifier le format des adresses e-mail	73
18.10	Valider les valeurs de la colonne status	73

19 Bonnes pratiques	74
20 Sources	75

1 Les bases de données relationnelles

1.1 Définition

Une **base de données** est une collection organisée d'informations ou de données, structurée de manière à pouvoir être facilement accédée, gérée et mise à jour.

Les données sont présentes dans tous les domaines, que ce soit le football, le cinéma ou les diverses professions. Une base de données constitue un moyen efficace de structurer ces informations, facilitant ainsi leur accès et leur utilisation.

1.2 Qu'est-ce que SQL ?

Le SQL, Structured Query Language, est un langage Standard permettant à un client de communiquer des instructions à une base de données créée en 1974 et normalisée depuis 1986. Il se décline en quatre parties :

- le **DDL** (Data definition language) comporte les instructions qui permettent de définir la façon dont les données sont représentées.
- le **DML** (Data manipulation language) permet d'écrire dans la base et donc de modifier les données.
- le **DQL** (Data query language) est la partie la plus complexe du SQL, elle permet de lire les données dans la base à l'aide de requêtes.
- le **DCL** (Data control language) permet de gérer les droits d'accès aux données.

A cela s'ajoute des extensions procédurales du SQL (appelé PL/SQL en Oracle). Celui-ci permet d'écrire des scripts exécutés par le serveur de base de données.

Dans ce cours, nous allons nous pencher en grande partie sur le DQL et nous verrons une partie du DML.

1.3 Pourquoi le SQL ?

Souvent les données sont stockées dans des fichiers de type **.csv**.

Cela peut poser des problèmes si plusieurs personnes doivent travailler sur ce fichier, car chaque utilisateur doit le charger pour l'utiliser.

En plus, il devient compliqué de gérer les modifications apportées par chacun : comment s'assurer que toutes les modifications soient prises en compte ?

Enfin, dans des situations qui nécessitent beaucoup de données, stocker toutes les informations dans un seul tableau n'est pas optimale.

C'est pour ces raisons que le SQL existe et c'est pour cela qu'on parle de base de donnée relationnelle : pour faire communiquer les bases de données (qui sont donc des tableaux) entre elles avec un langage de requête simple.

1.4 Différents modèles

Il existe divers modèles de bases de données :

- Les **bases de données relationnelles** utilisent des tableaux de données organisés en lignes et colonnes auxquelles on accède grâce au langage **SQL**.
- Les **bases de données non-relationnelles** dites **noSQL** qui peuvent stocker des données de formes diverses comme des graphes, des documents, etc.

Dans ce cours, nous ne nous intéresserons qu'aux bases de données relationnelles et au langage SQL.

Dans une base de donnée relationnelle, plutôt que de stocker toutes les données dans une table, on les stocke dans plusieurs tables reliées entre elles par des clés.

Une base de données est définie par son schéma relationnel qui spécifie :

- La liste des tables avec pour chaque table :
 - Le nom de la table (appelée relation)
 - La liste des colonnes (appelées attributs) avec leur nom et le domaine de valeurs des données qui y sont stockées (entiers, chaînes de caractères, dates, etc.)
 - Les identifiants (qui sont uniques) de chaque ligne
- Les tables doivent être reliées entre elles grâce à des identifiants secondaires

1.5 Les différents systèmes de gestion de bases de données relationnelles

MySQL et SQL Server sont tous deux des systèmes de gestion de bases de données relationnelles à hautes performances.

SQL Server propose un optimiseur de requêtes et des index de type « columnstore » pour optimiser les performances. À mesure que vous augmentez la charge de travail des bases de données, les performances de SQL Server sont généralement légèrement supérieures à celles de MySQL.

MySQL utilise le regroupement de connexions et la mise en cache des requêtes pour optimiser les performances.

1.6 Les différents types de données

Les principaux types de données en SQL sont :

- CHARACTER (ou CHAR) : valeur alpha de longueur fixe.
- CHARACTER VARYING (ou VARCHAR) : valeur alpha de longueur maximale fixée.
- NATIONAL CHARACTER VARYING (ou NVARCHAR) : Contrairement au type VARCHAR, qui stocke les caractères selon un encodage dépendant de la page de code (non Unicode), NVARCHAR utilise un encodage Unicode (UTF-16 ou UCS-2 selon la version de SQL Server), permettant de gérer correctement un large éventail de langues et de symboles.
- TEXT : suite longue de caractères (sans limite de taille).
- FLOAT : 32 bits (1 bit de signe, 8 bits d'exposant, 23 bits de mantisse), environ 6 à 7 chiffres significatifs
- DOUBLE : 64 bits (1 bit de signe, 11 bits d'exposant, 52 bits de mantisse), environ 15 à 16 chiffres significatifs.
- INTEGER (ou INT) : entier long
- BIGINT : entier de généralement 64 bits (8 octets).
- REAL : réel à virgule flottante dont la représentation est binaire.
- BOOLEAN (ou LOGICAL) : vrai/faux
- BIT : (0 ou 1) dans certains SGBD.
- DATE : Date du calendrier grégorien.
- TIME : Stocke l'heure (heures, minutes, secondes) avec éventuellement une précision (microsecondes ou nanosecondes selon le SGBD).
- DATETIME : Combine la date et l'heure.
- TIMESTAMP : gère souvent un intervalle plus restreint que DATETIME mais est souvent associé au fuseau horaire ou capable d'enregistrer l'information de date/heure sous forme de nombre de secondes écoulées depuis une référence (Unix epoch).

Il y en a évidemment d'autres, mais c'est une bonne introduction. Voir le lien suivant :

https://webusers.i3s.unice.fr/~rueher/Cours/BD/DocPostgresSQL9-5_HTML/datatype.html

1.7 Notions de clés primaires/secondaires

1.7.1 Clé primaire (PRIMARY KEY)

Définition La clé primaire est un ou plusieurs champs (colonnes) d'une table dont les valeurs identifient de façon unique chaque enregistrement (ligne) de cette table.

Propriétés :

- (Unicité) Elle est unique : deux lignes ne peuvent pas avoir la même valeur de clé primaire.
- (Non nulle) Elle est non nulle : la clé primaire ne peut pas contenir de NULL.
- (Unicité structurelle) Une table ne peut avoir qu'une seule clé primaire.
- Elle peut être :
 - **simple** : construite à partir d'une seule colonne
 - **composite** : construite à partir de plusieurs colonnes

Exemple Voici un exemple de mise en place d'une clé primaire simple :

```
CREATE TABLE users (  
    user_id SERIAL PRIMARY KEY,  
    username VARCHAR(100) NOT NULL,  
    email VARCHAR(100) NOT NULL  
);
```

Ici, `user_id` est donc la clé primaire, identifiant unique pour chaque utilisateur.

Voyons maintenant un exemple de clé primaire composite :

```
CREATE TABLE enrollment (  
    student_id INT,  
    course_id INT,  
    PRIMARY KEY (student_id, course_id)  
);
```

Ici, un étudiant peut être inscrit à plusieurs cours, mais ne peut pas être inscrit deux fois au même cours.

1.7.2 Clé secondaire (FOREIGN KEY)

Définition Une clé secondaire (ou étrangère) est un champ (ou combinaison de champs) dans une table qui fait référence à la clé primaire d'une autre table. Elle permet de créer une relation entre deux tables. Son but est donc d'assurer l'intégrité référentielle, c'est-à-dire qu'une ligne ne peut référencer qu'une valeur existante dans la table cible.

Propriétés :

- Une clé étrangère peut contenir des doublons.
- Elle peut contenir des valeurs NULL (sauf si explicitement interdit).
- Elle garantit que les valeurs référencées existent dans la table cible.

Exemple

```
CREATE TABLE orders (  
    order_id SERIAL PRIMARY KEY,  
    user_id INT,  
    FOREIGN KEY (user_id) REFERENCES users(user_id)  
);
```

Ici `order_id` est la clé primaire de la table `orders` et `user_id` est une clé étrangère qui pointe vers `users(user_id)`.

1.7.3 Clé unique (UNIQUE KEY)

Définition Une clé unique est une contrainte qui garantit que les valeurs d'une ou plusieurs colonnes sont uniques dans toute la table. Elle sert à empêcher les doublons sur des colonnes qui ne sont pas la clé primaire, comme un email ou un numéro de téléphone.

Propriétés

- Unicité : Aucune ligne ne peut avoir la même valeur sur cette ou ces colonnes.
- NULL autorisé : Oui (mais attention, le comportement dépend du SGBD : PostgreSQL accepte plusieurs NULL, MySQL non).
- Multiplicité : On peut créer plusieurs contraintes UNIQUE dans une même table.
- Pas forcément identifiant : Contrairement à une clé primaire, une clé UNIQUE ne sert pas nécessairement à identifier une ligne.

Exemple

```
CREATE TABLE utilisateurs (  
    id INT PRIMARY KEY,  
    email VARCHAR(255) UNIQUE  
);
```

On peut également définir des clés uniques composites :

```
CREATE TABLE commandes (  
    id_commande INT,  
    id_client INT,  
    date_commande DATE,  
    PRIMARY KEY(id_commande),  
    UNIQUE(id_client, date_commande)  
);
```

1.7.4 Clé Primaire VS Clé unique

La clé primaire garantit l'unicité et est utilisée pour identifier de manière unique chaque ligne d'une table. Une table ne peut avoir qu'une seule clé primaire. Elle ne peut jamais contenir de valeur NULL.

La clé unique garantit aussi l'unicité mais une table peut avoir plusieurs contraintes de clé unique. Les colonnes qui utilisent une clé unique peuvent contenir des valeurs NULL.

Pourquoi utiliser des clés uniques ? Assurer l'intégrité des données : Une clé unique empêche les données dupliquées, ce qui est essentiel pour maintenir une base de données propre et fiable.

Optimiser les recherches : En indexant les colonnes avec des contraintes uniques, les performances des requêtes qui utilisent ces colonnes en filtre ou en jointure peuvent être améliorées.

Prévenir les erreurs : Lors de l'insertion de nouvelles lignes, si une violation de la contrainte unique se produit, cela permet de prévenir les erreurs avant qu'elles ne contaminent les données.

2 Sélection de données

Une requête SQL est une série d'instructions basiques utilisant des mots anglais tels que **SELECT**, **FROM**, etc.

Vous remarquerez qu'ils sont toujours écrits en majuscules.

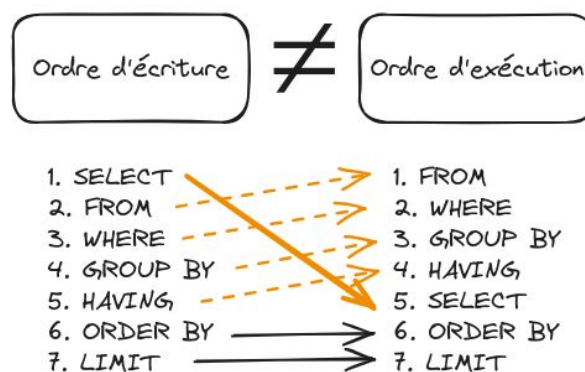
La structure générale de la majorité des commandes SQL est de la forme :

```
SELECT liste des attributs
FROM table_name;
WHERE liste des conditions
GROUP BY expression
HAVING condition
UNION | INTERSECT | EXCEPT
ORDER BY expression
LIMIT count
OFFSET start ;
```

Attention : Il est vraiment primordiale de respecter l'ordre et d'ajouter un point virgule ";" à la fin de la requête.

En SQL, le point-virgule (;) est utilisé pour marquer la fin d'une instruction. C'est un séparateur de commandes qui permet à l'interpréteur SQL de comprendre où se termine une commande et où commence la suivante. C'est donc une habitude à prendre pour ce langage.

L'ordre d'écriture de nos requêtes SQL n'est pas le même que leur ordre d'exécution, et c'est crucial de le comprendre pour optimiser nos performances.



Ainsi :

- Ordre d'écriture : **SELECT** → **FROM** → **WHERE** → **GROUP BY** → **HAVING** → **ORDER BY** → **LIMIT**
- Ordre d'exécution : **FROM** → **WHERE** → **GROUP BY** → **HAVING** → **SELECT** → **ORDER BY** → **LIMIT**

Pourquoi c'est important ? Prenons un exemple concret : Si vous filtrez vos données avec **WHERE** avant de faire un **GROUP BY**, vous économisez des ressources en n'agrégeant que les données pertinentes.

Gardez toujours cet ordre en tête lors de l'optimisation de vos requêtes. Cela peut faire une différence significative sur les performances, particulièrement avec de gros volumes de données.

2.1 SELECT

L'utilisation la plus courante de SQL consiste à lire des données issues de la base de données. Cela s'effectue avec la requête **SELECT** qui permet d'afficher les colonnes sélectionnées.

Avec cette dernière, on peut sélectionner une ou plusieurs colonnes d'une table.

Voici un exemple :

```
SELECT col1, col2  
FROM table_name;
```

Cette requête va sélectionner (SELECT) les colonnes "col1", "col2" provenant (FROM) de la table appelée "table_name".

Si on veut afficher toutes les colonnes d'une table, on utilisera le caractère "*" (étoile ou astérisque) :

```
SELECT *  
FROM table_name;
```

Cet astérisque est appelé "wildcard".

2.1.1 SELECT DISTINCT

La requête **SELECT DISTINCT** permet de sélectionner des valeurs uniques dans une colonne ou une combinaison de colonnes, en éliminant les doublons des résultats.

Pour le Système de Gestion de Bases de Données (SGBD) Oracle, cette requête est remplacée par la commande **UNIQUE**.

2.2 FROM

Pour chaque commande, il est nécessaire de spécifier dans quelle table rechercher les informations. Cela se fait avec l'instruction **FROM**. C'est pourquoi vous la voyez et verrez dans toutes les commandes qui suivent.

2.3 WHERE

La commande **WHERE** dans une requête SQL permet d'extraire les lignes d'une base de données qui respectent une condition. Cela permet d'obtenir uniquement les informations désirées.

On l'utilise de la manière suivante :

```
SELECT nom_colonnes  
FROM table_name  
WHERE condition ;
```

La condition est toujours de la forme :

```
Colonne Opérateur_de_comparaison la_condition
```

2.3.1 Les opérateurs de comparaisons

Il existe plusieurs opérateurs de comparaisons. La liste ci-jointe présente quelques uns des opérateurs les plus couramment utilisés.

- **=** : Permet de définir une condition d'égalité
- **!=** : Permet de vérifier une inégalité (On pourra utiliser le symbole **<>**)
- **>** : Permet de définir une condition d'ordre "Supérieur à"
- **<** : Permet de définir une condition d'ordre "Inférieur à"
- **>=** : Permet de définir une condition d'ordre "Supérieur ou égale à"
- **<=** : Permet de définir une condition d'ordre "Inférieur ou égal à"
- **IN** : Permet de vérifier si une donnée est présente dans une liste de valeurs possibles
- **BETWEEN... AND...** : Permet de vérifier si une valeur est comprise dans un intervalle donné (utile pour les nombres ou dates)
- **LIKE** : Permet de définir une condition sur des aspects précis (utile dans la manipulation de caractères)
- **IS NULL** : Permet de vérifier si une valeur est nulle
- **IS NOT NULL** : Permet de vérifier si une valeur est non nulle

2.3.2 AND & OR

Lorsque l'on a plusieurs conditions à tester, on utilisera les opérateurs **AND** et **OR**. Ces opérateurs peuvent être combinés à l'infini et mélangés.

Voici un exemple simple alliant les deux et quelques opérateurs :

```
SELECT * FROM produit  
WHERE ( categorie = 'informatique' AND stock < 20 )  
OR ( categorie = 'fourniture' AND stock < 200 )
```

2.4 GROUP BY

La commande **GROUP BY** est utilisée en SQL pour grouper plusieurs résultats et utiliser une fonction de totaux sur un groupe de résultat. Sur une table qui contient toutes les ventes d'un magasin, il est par exemple possible de regrouper les ventes par clients identiques et d'obtenir le coût total des achats pour chaque client.

La syntaxe de la commande est :

```
SELECT colonne1, fonction(colonne2)
FROM table_name
GROUP BY colonne1 ;
```

A noter : cette commande doit toujours s'utiliser après la commande **WHERE** et avant la commande **HAVING**.

2.5 HAVING

La condition **HAVING** en SQL est presque similaire à **WHERE** à la seule différence que **HAVING** permet de filtrer en utilisant des fonctions telles que **SUM()**, **COUNT()**, **AVG()**, **MIN()** ou **MAX()** (voir section Fonctions utiles et classiques).

L'utilisation de **HAVING** s'utilise de la manière suivante :

```
SELECT colonne1, SUM(colonne2)
FROM table_name
GROUP BY colonne1
HAVING fonction(colonne2) operateur valeur ;
```

Cela permet donc de SÉLECTIONNER les colonnes DE la table "table_name" en GROUPANT les lignes qui ont des valeurs identiques sur la colonne "colonne1" et que la condition de **HAVING** soit respectée.

Important : **HAVING** est très souvent utilisé en même temps que **GROUP BY** bien que ce ne soit pas obligatoire.

Dans les consignes, on pensera à faire un **HAVING** dans les phrases types "affichez les profs qui ont plus de 5 qqch".

2.6 ORDER BY

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

Une requête où l'on souhaite filtrer l'ordre des résultats utilise la commande **ORDER BY** de la sorte :

```
SELECT colonne1, colonne2
FROM table_name
ORDER BY colonne1 ;
```


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

```
SELECT colonne1, colonne2, colonne3
FROM table_name
ORDER BY colonne1 DESC, colonne2 ASC ;
```

A noter : il n'est pas obligé d'utiliser le suffixe "ASC" sachant que les résultats sont toujours classés par ordre ascendant par défaut. Toutefois, c'est plus pratique pour mieux s'y retrouver, surtout si on a oublié l'ordre par défaut.

2.7 LIMIT

La clause **LIMIT** est à utiliser dans une requête SQL pour spécifier le nombre maximum de résultats que l'on souhaite obtenir. Cette clause est souvent associée à un OFFSET, c'est-à-dire effectuer un décalage sur le jeu de résultat. Ces 2 clauses permettent par exemple d'effectuer des systèmes de pagination (exemple : récupérer les 10 articles de la page 4).

ATTENTION : selon le système de gestion de bases de données, la syntaxe ne sera pas pareille.

La syntaxe est :

```
SELECT *
FROM table_name ;
LIMIT 10 ;
```

Cette requête permet de récupérer seulement les 10 premiers résultats d'une table. Bien entendu, si la table contient moins de 10 résultats, alors la requête retournera toutes les lignes.

Bon à savoir : la bonne pratique lorsque l'on utilise LIMIT consiste à utiliser également la clause ORDER BY pour s'assurer que quoi qu'il en soit ce sont toujours les bonnes données qui sont présentées. En effet, si le système de tri est non spécifié, alors il est en principe inconnu et les résultats peuvent être imprévisibles.

2.7.1 Limit et Offset

L'offset est une méthode simple de décaler les lignes à obtenir. La syntaxe PostgreSQL pour utiliser une limite et un offset est la suivante :

```
SELECT *
FROM table_name ;
LIMIT 10 OFFSET 5 ;
```

Cette requête permet de récupérer les résultats 6 à 15 (car l'OFFSET commence toujours à 0). A titre d'exemple, pour récupérer les résultats 16 à 25 il faudrait donc utiliser : LIMIT 10 OFFSET 15.

A noter : Utiliser OFFSET 0 revient au même que d'omettre l'OFFSET.

En MySQL, la syntaxe est légèrement différente :

```
SELECT *  
FROM table_name;  
LIMIT 5, 10;
```

Cette requête retourne les enregistrements 6 à 15 d'une table. Le premier nombre est l'OFFSET tandis que le suivant est la limite.

Bon à savoir : pour une bonne compatibilité, MySQL accepte également la syntaxe LIMIT nombre OFFSET nombre. En conséquent, dans la conception d'une application utilisant MySQL il est préférable d'utiliser cette syntaxe car c'est potentiellement plus facile de migrer vers un autre système de gestion de base de données sans avoir à ré-écrire toutes les requêtes.

3 Fonctions utiles et classiques (Fonctions d'agrégation)

3.1 COUNT()

En SQL, la fonction d'agrégation `COUNT()` permet de compter le nombre d'enregistrement dans une table.

Connaître le nombre de lignes dans une table est très pratique dans de nombreux cas, par exemple pour savoir combien d'utilisateurs sont présents dans une table ou pour connaître le nombre de commentaires sur un article.

En voici un exemple :

```
SELECT COUNT(*)  
FROM table;
```

A savoir : en général, en termes de performance il est plutôt conseillé de filtrer les lignes avec `GROUP BY` si c'est possible, puis d'effectuer un `COUNT(*)`.

On pourra également associer le mot clé `DISTINCT` à `COUNT()` afin de compter distinctement les entités :

```
SELECT COUNT(DISTINCT expression)  
FROM table;
```

3.2 SUM()

La fonction d'agrégation `SUM()` permet de calculer la somme totale d'une colonne contenant des valeurs numériques. Cette fonction n'est utile que sur des colonnes de types numériques (`INT`, `FLOAT` ...) et n'additionne pas les valeurs `NULL`.

La syntaxe pour utiliser cette fonction SQL peut être similaire à celle-ci :

```
SELECT SUM(nom_colonne)  
FROM table;
```

Cette requête SQL permet de calculer la somme des valeurs contenues dans la colonne "nom_colonne".

A savoir : Il est possible de filtrer les enregistrements avec la commande `WHERE` pour ne calculer la somme que des éléments souhaités.

3.3 AVG()

La fonction d'agrégation `AVG()` dans le langage SQL permet de calculer une valeur moyenne sur un ensemble d'enregistrement de type numérique et non nul.

Syntaxe :

```
SELECT AVG(nom_colonne)
FROM nom_table;
```

Cette requête permet de calculer la note moyenne de la colonne “nom_colonne” sur tous les enregistrements de la table “nom_table”. Il est possible de filtrer les enregistrements concernés à l'aide de la commande `WHERE`. Il est aussi possible d'utiliser la commande `GROUP BY` pour regrouper les données appartenant à la même entité.

3.4 ROUND()

La fonction `ROUND()` permet d'arrondir un résultat numérique. Cette fonction permet soit d'arrondir sans utiliser de décimale pour retourner un nombre entier (c'est-à-dire : aucun chiffre après la virgule), ou de choisir le nombre de chiffres après la virgule.

Syntaxe :

```
SELECT ROUND(nom_colonne)
FROM table;
```

Dans cet exemple la colonne “nom_colonne” contient des données numériques (exemple : FLOAT, INT, NUMERIC ...) et retournera uniquement des entiers.

Pour obtenir le résultat avec 2 chiffres de décimal il convient de spécifier le nombre de décimal souhaité comme 2ème argument :

```
SELECT ROUND(nom_colonne, 2)
FROM table;
```

3.5 ISNULL()

Dans le langage SQL la fonction `ISNULL()` peut s'avérer utile pour traiter des résultats qui possèdent des données nulles. Attention toutefois, cette fonction s'utilise différemment selon le système de gestion de bases de données :

- MySQL : la fonction `ISNULL()` prend un seul paramètre et permet de vérifier si une données est nulle
- SQL Server : la fonction `ISNULL()` prend 2 paramètres et sert à afficher la valeur du 2ème paramètre si le premier paramètre est null. Cette même fonctionnalité peut être effectuée par d'autres systèmes de gestion de base de données d'une manière différente :
 - MySQL : il faut utiliser la fonction `IFNULL()`
 - PostgreSQL : il faut utiliser la fonction `COALESCE()`

— Oracle : il faut utiliser la fonction NVL()

Voici un lien pour creuser : <https://sql.sh/fonctions/isnull>

3.6 MAX()

La fonction d'agrégation `MAX()` permet de retourner la valeur maximale d'une colonne dans un set d'enregistrement. La fonction peut s'appliquer à des données numériques ou alphanumériques. Il est par exemple possible de rechercher le produit le plus cher dans une table d'une boutique en ligne.

Syntaxe :

```
SELECT MAX(nom_colonne)
FROM table;
```

Lorsque cette fonctionnalité est utilisée en association avec la commande `GROUP BY`, la requête peut ressembler à l'exemple ci-dessous :

```
SELECT colonne1, MAX(colonne2)
FROM table
GROUP BY colonne1;
```

3.7 MIN()

La fonction d'agrégation `MIN()` de SQL permet de retourner la plus petite valeur d'une colonne sélectionnée. Cette fonction s'applique aussi bien à des données numériques qu'à des données alphanumériques et s'utilise comme la fonction `MAX()`.

4 Jointures

Les jointures permettent de regrouper des jeux de données. Elles sont essentielles pour l'extraction d'information quand on a plusieurs jeux de données.

On va donc avec ces jointures pouvoir exploiter les relations entre les différentes bases de données.

De manière générale, on écrira :

```
SELECT *  
FROM table1  
(LEFT)(RIGHT)JOIN table2  
ON table1.colonne_i = table2.colonne_j;
```

4.1 JOIN/INNER JOIN

En SQL standard moderne, **JOIN** est un alias pour **INNER JOIN**, donc il n'y a aucune différence pratique.

Cette commande sélectionne uniquement les lignes où il existe une correspondance dans les deux tables impliquées dans la jointure.

On notera que certains SGBD anciens ou spécifiques (comme Oracle dans ses versions historiques) peuvent exiger que l'utilisateur précise explicitement **INNER JOIN** pour éviter toute ambiguïté.

4.2 LEFT JOIN

LEFT JOIN permet de récupérer toutes les lignes de la table de gauche (table1), ainsi que les lignes correspondantes de la table de droite (table2). Si aucune correspondance n'est trouvée dans la table de droite, les colonnes de celle-ci contiendront des valeurs **NULL**.

On l'utilisera donc lorsque l'on souhaite conserver toutes les données d'une table principale (table1), même si elles n'ont pas de correspondance dans une autre table.

4.3 RIGHT JOIN

RIGHT JOIN récupère toutes les lignes de la table de droite (table2), ainsi que les lignes correspondantes de la table de gauche (table1). Si aucune correspondance n'est trouvée dans la table de gauche, les colonnes de celle-ci contiendront des valeurs **NULL**.

4.4 FULL OUTER JOIN

Cette jonction combine les résultats d'un **LEFT JOIN** et d'un **RIGHT JOIN**.

On peut alors récupérer toutes les lignes des deux tables, en insérant des **NULL** pour les colonnes de la table opposée lorsqu'il n'y a pas de correspondance.

4.5 CROSS JOIN

Cette jonction réalise le produit cartésien des deux tables, c'est-à-dire qu'il associe chaque ligne de la première table avec chaque ligne de la seconde table.

4.6 Récapitulatif

Tables en entrée	Type de jointure	Requête SQL	Résultat																																																																										
<table><tr><th colspan="2">Table 1</th></tr><tr><th>id</th><th>nom</th></tr><tr><td>1</td><td>a</td></tr><tr><td>2</td><td>b</td></tr><tr><td>3</td><td>c</td></tr></table> <table><tr><th colspan="2">Table 2</th></tr><tr><th>id</th><th>valeur</th></tr><tr><td>1</td><td>xxx</td></tr><tr><td>3</td><td>yy</td></tr><tr><td>3</td><td>yyyy</td></tr><tr><td>5</td><td>zzz</td></tr></table>	Table 1		id	nom	1	a	2	b	3	c	Table 2		id	valeur	1	xxx	3	yy	3	yyyy	5	zzz	<p>Croisée CROSS JOIN</p> <p>Pas de critère de jointure</p>	<p>SELECT * FROM table1 AS t1 CROSS JOIN table2 AS t2 ;</p>	<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>1</td><td>a</td><td>3</td><td>yy</td></tr><tr><td>1</td><td>a</td><td>3</td><td>yyyy</td></tr><tr><td>1</td><td>a</td><td>5</td><td>zzz</td></tr><tr><td>2</td><td>b</td><td>1</td><td>xxx</td></tr><tr><td>2</td><td>b</td><td>3</td><td>yy</td></tr><tr><td>2</td><td>b</td><td>3</td><td>yyyy</td></tr><tr><td>2</td><td>b</td><td>5</td><td>zzz</td></tr><tr><td>3</td><td>c</td><td>1</td><td>xxx</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr><tr><td>3</td><td>c</td><td>5</td><td>zzz</td></tr></table>	t1.id	t1.nom	t2.id	t2.valeur	1	a	1	xxx	1	a	3	yy	1	a	3	yyyy	1	a	5	zzz	2	b	1	xxx	2	b	3	yy	2	b	3	yyyy	2	b	5	zzz	3	c	1	xxx	3	c	3	yy	3	c	3	yyyy	3	c	5	zzz
Table 1																																																																													
id	nom																																																																												
1	a																																																																												
2	b																																																																												
3	c																																																																												
Table 2																																																																													
id	valeur																																																																												
1	xxx																																																																												
3	yy																																																																												
3	yyyy																																																																												
5	zzz																																																																												
t1.id	t1.nom	t2.id	t2.valeur																																																																										
1	a	1	xxx																																																																										
1	a	3	yy																																																																										
1	a	3	yyyy																																																																										
1	a	5	zzz																																																																										
2	b	1	xxx																																																																										
2	b	3	yy																																																																										
2	b	3	yyyy																																																																										
2	b	5	zzz																																																																										
3	c	1	xxx																																																																										
3	c	3	yy																																																																										
3	c	3	yyyy																																																																										
3	c	5	zzz																																																																										
	<p>Complète FULL JOIN</p> <p>Nécessité d'avoir un critère de jointure</p>	<p>SELECT * FROM table1 AS t1 FULL JOIN table2 AS t2 ON t1.id = t2.id ;</p>	<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>2</td><td>b</td><td></td><td></td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr><tr><td></td><td></td><td>5</td><td>zzz</td></tr></table>	t1.id	t1.nom	t2.id	t2.valeur	1	a	1	xxx	2	b			3	c	3	yy	3	c	3	yyyy			5	zzz																																																		
t1.id	t1.nom	t2.id	t2.valeur																																																																										
1	a	1	xxx																																																																										
2	b																																																																												
3	c	3	yy																																																																										
3	c	3	yyyy																																																																										
		5	zzz																																																																										
	<p>À gauche LEFT JOIN</p> <p>Nécessité d'avoir un critère de jointure</p>	<p>SELECT * FROM table1 AS t1 LEFT JOIN table2 AS t2 ON t1.id = t2.id ;</p>	<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>2</td><td>b</td><td></td><td></td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr></table>	t1.id	t1.nom	t2.id	t2.valeur	1	a	1	xxx	2	b			3	c	3	yy	3	c	3	yyyy																																																						
t1.id	t1.nom	t2.id	t2.valeur																																																																										
1	a	1	xxx																																																																										
2	b																																																																												
3	c	3	yy																																																																										
3	c	3	yyyy																																																																										
	<p>À droite RIGHT JOIN</p> <p>Nécessité d'avoir un critère de jointure</p>	<p>SELECT * FROM table1 AS t1 RIGHT JOIN table2 AS t2 ON t1.id = t2.id ;</p>	<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr><tr><td></td><td></td><td>5</td><td>zzz</td></tr></table>	t1.id	t1.nom	t2.id	t2.valeur	1	a	1	xxx	3	c	3	yy	3	c	3	yyyy			5	zzz																																																						
t1.id	t1.nom	t2.id	t2.valeur																																																																										
1	a	1	xxx																																																																										
3	c	3	yy																																																																										
3	c	3	yyyy																																																																										
		5	zzz																																																																										
	<p>Naturelle JOIN</p> <p>Nécessité d'avoir un critère de jointure</p>	<p>SELECT * FROM table1 AS t1 JOIN table2 AS t2 ON t1.id = t2.id ;</p>	<table><tr><th>t1.id</th><th>t1.nom</th><th>t2.id</th><th>t2.valeur</th></tr><tr><td>1</td><td>a</td><td>1</td><td>xxx</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yy</td></tr><tr><td>3</td><td>c</td><td>3</td><td>yyyy</td></tr></table>	t1.id	t1.nom	t2.id	t2.valeur	1	a	1	xxx	3	c	3	yy	3	c	3	yyyy																																																										
t1.id	t1.nom	t2.id	t2.valeur																																																																										
1	a	1	xxx																																																																										
3	c	3	yy																																																																										
3	c	3	yyyy																																																																										

Les espaces vides sont en fait des NULL.

5 CTE(Common Table Expression)

5.1 Présentation des CTE

Une expression de table commune (CTE) est comme une sous-requête nommée. Elle fonctionne comme une table virtuelle à laquelle seule la requête principale peut accéder. Les CTE permettent de simplifier, de raccourcir et d'organiser votre code.

On les définit dans une requête SQL à l'aide du mot-clé : **WITH** .

Voici la syntaxe :

```
WITH my_cte AS(  
    SELECT a,b,c  
    FROM T1)  
  
SELECT a,c  
FROM my_cte  
WHERE ...
```

Le nom de cette CTE est my_cte et la requête CTE est SELECT a,b,c FROM T1.

L'ETC commence par le mot-clé WITH, après quoi vous indiquez le nom de votre ETC, puis le contenu de la requête entre parenthèses. La requête principale vient après la parenthèse fermante et fait référence à l'ETC. Ici, la requête principale (également appelée requête externe) est SELECT a,c FROM my_cte WHERE

Étudions un exemple :

branch	date	seller	item	quantity	unit_price
Paris-1	2021-12-07	Charles	Headphones A2	1	80
London-1	2021-12-06	John	Cell Phone X2	2	120
London-2	2021-12-07	Mary	Headphones A1	1	60
Paris-1	2021-12-07	Charles	Battery Charger	1	50
London-2	2021-12-07	Mary	Cell Phone B2	2	90
London-1	2021-12-07	John	Headphones A0	5	75
London-1	2021-12-07	Sean	Cell Phone X1	2	100

Pour obtenir le prix de l'article le plus cher, nous utilisons une expression de table commune comme celle-ci :


```
WITH highest AS (  
    SELECT  
        branch,  
        date,  
        MAX(unit_price) AS highest_price  
    FROM sales  
    GROUP BY branch, date)  
  
SELECT sales.*, h.highest_price  
FROM sales  
JOIN highest h  
ON sales.branch = h.branch  
   AND sales.date = h.date;
```

Important : La plupart des fonctions des CTE dans SQL Server peuvent être réalisées avec une sous-requête. Mais imaginez à quoi ressembleraient les codes... Ce ne serait pas très lisible !

L'une des utilisations typiques des CTE dans SQL Server permet donc de vous aider à organiser les longues requêtes. Les CTEs rendent les requêtes plus lisibles en nommant les parties de la requête. Ainsi, vous pouvez facilement décomposer chaque partie d'un calcul complexe et rendre le calcul logique.

Il n'y a pas de secrets, pour vraiment les comprendre, il faut s'exercer ! Voici quelques liens :

- <https://learnsql.fr/blog/11-exercices-d-expression-de-table-commune-sql/>
- Problème Leetcode : 550. Game Play Analysis IV
- Problème Leetcode : 1070. Product Sales Analysis III

5.2 Plusieurs CTE

IMPORTANT :

Si j'ai plusieurs cte à taper, je n'utiliserai qu'un seul **WITH**. Ensuite je sépare mes cte avec une simple virgule. Enfin je tape la commande principale. Cela donne :

```
WITH cte1 AS (  
    requête SQL ),  
cte2 AS(  
    requête SQL)  
  
SELECT ...  
FROM ...  
WHERE
```

5.3 UNION et UNION ALL

La commande **UNION** permet de mettre bout-à-bout les résultats de plusieurs requêtes utilisant elles-même la commande **SELECT**.

C'est donc une commande qui permet de concaténer les résultats de 2 requêtes ou plus. Pour l'utiliser il est nécessaire que chacune des requêtes à concaténer retournes le même nombre de colonnes, avec les mêmes types de données et dans le même ordre.

Sa syntaxe est :

```
SELECT * FROM table1
UNION
SELECT * FROM table2
```

La particularité de **UNION ALL** est que cette commande conserve tous les résultats, y compris les doublons. Elle s'utilise de la même manière que **UNION**.

6 Les requêtes temporelles et les dates

Il est important en SQL de pouvoir connaître la date et l'heure. Il existe une multitude de fonctions qui concernent les éléments temporels pour pouvoir lire ou écrire plus facilement des données à une date précise ou à un intervalle de date.

Ces requêtes sont cruciales pour l'analyse de séries temporelles, le calcul de KPIs basés sur le temps, ou l'identification de tendances saisonnières.

6.1 DATE_FORMAT()

La fonction `DATE_FORMAT()`, dans le langage SQL et plus particulièrement avec MySQL, permet de formater une donnée DATE dans le format indiqué. Il s'agit de la fonction idéal si l'on souhaite définir le formatage de la date directement à partir du langage SQL et pas uniquement dans la partie applicative. Voici sa syntaxe :

```
SELECT DATE_FORMAT(date, format)
```

Le premier paramètre correspond à une donnée de type DATE (ou DATETIME), tandis que le deuxième paramètre est une chaîne de caractère contenant le choix de formatage (**voir page suivante**).

Il est évidemment possible de combiner plusieurs valeurs de formatage afin d'obtenir le formatage d'une date dans le format de son choix.

Quelques exemples :

- `SELECT DATE_FORMAT("2018-09-24", "%D %b %Y")` donne "24th Sep 2018"
- `SELECT DATE_FORMAT("2018-09-24 22:21:20", "%H:%i:%s")` donne "22:21:20"
- `SELECT DATE_FORMAT("2018-09-24", "Message du : %d/%m/%Y")` donne "Message du : 24/09/2018"

Et donc vis à vis d'une colonne entière on écrira par exemple :

```
SELECT *, DATE_FORMAT(date_inscription, "%d/%m/%Y")  
FROM utilisateur;
```

La requête ci-dessus permet d'obtenir la liste des données d'une table contenant des utilisateurs, ainsi que la date d'inscription avec un formatage jour/mois/année prêt à être exploité affiché dans un format facile à lire pour les utilisateurs.

Voici tous les formats possibles :

- %a : nom du jour de la semaine abrégé (Sun, Mon, ... Sat)
- %b : nom du mois abrégé (Jan, Feb, ... Dec)
- %c : numéro du mois (0, 1, 2, ... 12) (numérique)
- %D : numéro du jour, avec le suffixe anglais (0th, 1st, 2nd, 3rd, ...)
- %d : numéro du jour du mois, sous 2 décimales (00..31) (numérique)
- %e : numéro du jour du mois (0..31) (numérique)
- %f : microsecondes (000000..999999)
- %H : heure (00..23)
- %h : heure (01..12)
- %I : heure (01..12)
- %i : minutes (00..59) (numérique)
- %j : jour de l'année (001..366)
- %k : heure (0..23)
- %l : heure (1..12)
- %M : nom du mois (January..December)
- %m : mois (00..12) (numérique)
- %p : AM ou PM
- %r : heure au format 12-heures (hh :mm :ss suivi par AM ou PM)
- %S : secondes (00..59)
- %s : secondes (00..59)
- %T : heure au format 24 heures (hh :mm :ss)
- %U : Semaine (00..53), pour lequel le dimanche est le premier jour de la semaine ; WEEK() mode 0
- %u : Semaine (00..53), pour lequel le lundi est le premier jour de la semaine ; WEEK() mode 1
- %V : Semaine (01..53), pour lequel le dimanche est le premier jour de la semaine ; WEEK() mode 2 ; utilisé avec %X
- %v : Semaine (01..53), pour lequel le lundi est le premier jour de la semaine ; WEEK() mode 3 ; utilisé avec %x
- %W : nom du jour de la semaine (Sunday, Monday, ... Saturday)
- %w : numéro du jour de la semaine (0=dimanche, 1=lundi, ... 6=samedi)
- %X : numéro de la semaine de l'année, pour lequel le dimanche est le premier jour de la semaine, numérique, 4 digits ; utilisé avec %V
- %x : numéro de la semaine de l'année, pour lequel le lundi est le premier jour de la semaine, numérique, 4 digits ; utilisé avec %v
- %Y : année, numérique (avec 4 digits)
- %y : année, numérique (avec 2 digits)
- %% : un caractère %
- %x x, pour n'importe lequel "x" non listé ci-dessus

6.2 DATEDIFF()

La fonction **DATEDIFF** permet de déterminer l'intervalle entre 2 dates spécifiées.

La fonction est utilisée avec les systèmes MySQL et SQL Server, mais s'utilise différemment :

- MySQL : la fonction prend 2 dates en paramètres et retourne le nombre de jours entre les 2 dates
- SQL Server : la fonction prend 2 dates ainsi qu'un paramètre déterminant sous quel intervalle de distance temporelle le résultat doit être retourné (nombre de jours entre 2 dates, nombre d'années, nombre de semaines, nombre d'heures ...)

En MySQL, on tapera :

```
SELECT DATEDIFF(date1, date2);
```

Sous SQL Server, on tapera :

```
SELECT DATEDIFF( type_limite, date1, date2 );
```

Le paramètre "type_limite" permet de spécifier si le résultat doit être exprimé en jour, semaine, mois, année ... Ce paramètre utilise l'une de ses valeurs : year, quarter, month, dayofyear, day, week, weekday, hour, minute, second, millisecond, microsecond ou nanosecond.

Donc pour reproduire le même fonctionnement que la fonction DATEDIFF() de MySQL, il convient d'utiliser la valeur "day" pour le paramètre "type _limite".

6.3 DATE_ADD/DATE_SUB

DATE_ADD est utilisé pour ajouter un intervalle spécifique à une date au format TIME.

DATE_SUB est utilisé pour soustraire un intervalle spécifique à une date donnée au format TIME.

On tapera :

```
DATE_ADD(date, INTERVAL valeur unité)
```

- date : La date de départ.
- INTERVAL : Spécifie combien d'unités de temps ajouter.
- valeur : Le nombre d'unités (par exemple, jours, mois, années).
- unité : Le type d'intervalle (DAY, MONTH, YEAR, etc).

Exemple :

```
SELECT DATE_ADD('2025-01-24', INTERVAL 10 DAY) AS new_date;
```

6.4 NOW, DAY, MONTH, YEAR

6.4.1 NOW

La fonction `NOW()` en SQL est utilisée pour obtenir la date et l'heure actuelles du système de base de données. C'est une fonction courante et très pratique pour travailler avec des données temporelles. Le format de sortie dépend du SGBD, mais généralement c'est `YYYY-MM-DD HH :MM :SS`.

6.4.2 CURDATE/CURRENT_DATE

Permet d'obtenir seulement la date actuelle `YYYY-MM-DD`.

- `CURDATE` pour MySQL
- `CURRENT_DATE` pour du SQL standard

6.4.3 CURRENT_TIME

Heure actuelle uniquement `HH :MM :SS`

6.4.4 DAY

La fonction `DAY` extrait le jour du mois d'une date donnée. `SELECT DAY('2025-01-24') AS jour` donnera 24.

6.4.5 MONTH

La fonction `MONTH` extrait le mois (sous forme de nombre) d'une date donnée. `SELECT MONTH('2025-01-24') AS mois` donnera 1.

6.4.6 YEAR

La fonction `YEAR` extrait l'année d'une date donnée. `SELECT YEAR('2025-01-24') AS annee` donnera 2025.

6.5 FORMULAIRE DATE

Sur cette page, je fais un gros formulaire récapitulatif de toutes les commandes SQL pour les dates.

- AGE() soustraire 2 dates [PostgreSQL]
- ADDDATE() ajouter une période sous forme d'heures à une date [MySQL]
- ADDTIME() ajouter une période sous forme d'une date à une autre date [MySQL]
- CONVERT_TZ() convertir d'une "timezone" à une autre [MySQL]
- CURDATE() récupérer la date courante [MySQL]
- CURRENT_DATE() synonyme de CURDATE() [MySQL, PostgreSQL]
- CURRENT_TIME() synonyme de CURTIME() [MySQL, PostgreSQL]
- CURRENT_TIMESTAMP() synonyme de NOW() [MySQL, PostgreSQL, SQL Server]
- CURTIME() Return the current time [MySQL]
- DATE() extraire une date à partir d'une chaîne contenant une valeur au format DATE ou DATETIME [MySQL]
- DATE_ADD() ajouter une valeur au format TIME à une date [MySQL]
- DATE_FORMAT() formater la date pour l'afficher selon le format choisi [MySQL]
- DATE_PART() extraire un élément d'un DATETIME (cf. heure, minute, jour, mois ...) [PostgreSQL]
- DATE_SUB() soustraire une valeur au format TIME à une date [MySQL]
- DATE_TRUNC() tronquer un DATETIME avec la précision souhaitée (cf. mois, jour, heure, minute ...) [PostgreSQL]
- DATEADD() ajoute un élément (minute, heure, jour, mois, année ...) à une date spécifiée [SQL Server]
- DATEDIFF() déterminer le nombre de jours entre 2 dates [MySQL, SQL Server]
- DATENAME() retourner une partie d'une date (cf. minute, heure, jour, semaine, mois ...) [SQL Server]
- DATEPART() retourne un entier qui représente la partie d'une date (cf. minute, heure, jour, mois, année ...) [SQL Server]
- DAY() synonyme de DAYOFMONTH() [MySQL, SQL Server]
- DAYNAME() retourne le nom du jour de la semaine [MySQL]
- DAYOFMONTH() retourner le jour dans le mois (de 1 à 31) [MySQL]
- DAYOFWEEK() retourner le jour dans la semaine (1=dimanche, 2=lundi, 3=mardi ...) [MySQL]
- DAYOFYEAR() retourner le jour dans l'année (de 1 à 366) [MySQL]
- EXTRACT() extraire une partie d'une date [MySQL, PostgreSQL]
- FROM_DAYS() convertir un nombre de jour en une date [MySQL]
- FROM_UNIXTIME() convertir un timestamp UNIX en une date au format DATETIME [MySQL]
- GET_FORMAT() retourne le format d'une date dans une chaîne de caractère [MySQL]
- GETDATE() obtenir la date courante du système, sans le décalage horaire [SQL Server]
- GETUTCDATE() obtenir la date courante UTC [SQL Server]
- HOUR() extraire le nombre d'heure pour une heure au format HH :MM :SS [MySQL]
- ISDATE() retourne 1 si la valeur en paramètre est dans l'un des formats suivants :

- TIME, DATE ou DATETIME [SQL Server]
- ISFINITE() tester pour savoir si une date ou une période de temps est finie [PostgreSQL]
 - JUSTIFY_HOURS() ajuster un intervalle de 24 heures en tant que “1 jour” ou un intervalle de 30 jours en tant que “1 mois” [PostgreSQL]
 - LAST_DAY() retourner le dernier jour du mois d’une date [MySQL]
 - LOCALTIME() synonyme de NOW() [MySQL, PostgreSQL]
 - LOCALTIMESTAMP() synonyme de NOW() [MySQL, PostgreSQL]
 - MAKEDATE() retourne une date à partir d’une année et du numéro du jour dans cette année [MySQL]
 - MAKETIME() créer une heure au format TIME à partir d’une heure, des minutes et du nombre de secondes [MySQL]
 - MICROSECOND() retourne le nombre de microsecondes à partir d’une heure ou d’un DATETIME [MySQL]
 - MINUTE() extraire le nombre de minutes d’une heure au format HH :MM :SS [MySQL]
 - MONTH() extraire le numéro du mois à partir d’une date [MySQL, SQL Server]
 - MONTHNAME() retourne le nom du mois [MySQL]
 - NOW() obtenir la date courante [MySQL, PostgreSQL]
 - PERIOD_ADD() ajoute un nombre définie de mois à une période [MySQL]
 - PERIOD_DIFF() retourne le nombre de mois entre 2 périodes définies [MySQL]
 - QUARTER() Return the quarter from a date argument [MySQL]
 - SEC_TO_TIME() convertir un nombre de secondes en une heure et minutes au format HH :MM :SS [MySQL]
 - SECOND() extraire le nombre de secondes d’une heure au format HH :MM :SS [MySQL]
 - STR_TO_DATE() convertir une chaîne de caractère en date [MySQL]
 - SUBDATE() synonyme de DATE_SUB() lorsque la fonction est invoquée avec 3 arguments [MySQL]
 - SUBTIME() soustraire une heure [MySQL]
 - SWITCHOFFSET() retourne une valeur DATETIMEOFFSET en changeant le fuseau horaire [SQL Server]
 - SYSDATE() retourne la date et l’heure courante [MySQL]
 - SYSDATETIME() retourne le DATETIME (date et heure) de l’ordinateur sur lequel est installé la base de données [SQL Server]
 - SYSDATETIMEOFFSET() retourne le DATETIMEOFFSET du système avec le décalage du fuseau horaire inclus [SQL Server]
 - SYSUTCDATETIME() retourne le DATETIME en heure UTC de l’ordinateur sur lequel la base est installées [SQL Server]
 - TIME_FORMAT() formater une date dans un autre format [MySQL]
 - TIME_TO_SEC() convertir une heure de format HH :MM :SS en nombre de secondes [MySQL]
 - TIME() extraire l’heure/minutes/secondes au format HH :MM :SS à partir d’une date [MySQL]
 - TIMEDIFF() retourne la durée entre 2 heures [MySQL]
 - TIMESTAMP() permet de convertir une DATE au format DATETIME [MySQL]
 - TIMESTAMPADD() ajoute un intervalle à une expression au format DATETIME

- [MySQL]
- `TIMESTAMPDIFF()` soustraire un intervalle à partir d'une expression au format `DATETIME` [MySQL]
- `TIMEOFDAY()` retourne la date et heure courante [PostgreSQL]
- `TO_DAYS()` retourne le nombre de jour à partir d'une date [MySQL]
- `TO_SECONDS()` Return the date or datetime argument converted to seconds since Year 0 [MySQL]
- `TODATETIMEOFFSET()` retourne un `DATETIMEOFFSET` à partir d'un `DATETIME` [SQL Server]
- `UNIX_TIMESTAMP()` retourner le timestamp UNIX (nombre de secondes depuis le 1er janvier 1970) [MySQL]
- `UTC_DATE()` retourne la date GMT courante [MySQL]
- `UTC_TIME()` retourne l'heure GMT courante [MySQL]
- `UTC_TIMESTAMP()` retourne la date et heure GMT courante [MySQL]
- `WEEK()` déterminer le numéro de la semaine dans une année, à partir d'une date [MySQL]
- `WEEKDAY()` déterminer le jour de la semaine à partir d'une date (0=lundi, 1 :mardi, 2=mercredi ...) [MySQL]
- `WEEKOFYEAR()` déterminer le numéro de la semaine dans une année, à partir d'une date [MySQL]
- `YEAR()` extraire l'année d'une date [MySQL, SQL Server]
- `YEARWEEK()` retourne l'année et la semaine à partir d'une date [MySQL]

7 Logique Conditionnelle

7.1 COALESCE

La fonction `COALESCE` est utilisée pour remplacer les valeurs NULL par une valeur par défaut que l'on spécifie. On écrit alors :

```
COALESCE(expression, valeur_par_defaut)
```

De ce fait si l'expression n'est pas NULL, elle retourne sa valeur et si l'expression est NULL, elle retourne `valeur_par_defaut`.

7.2 NULLIF

La fonction `NULLIF` compare deux expressions et retourne `NULL` si elles sont égales. Sinon, elle retourne la première expression.

```
NULLIF(expression1, expression2)
```

Utilisé pour éviter une division par zéro :

```
SELECT total / NULLIF(count, 0) AS moyenne  
FROM Table;
```

Si `count = 0`, alors `NULLIF(count, 0)` retourne NULL, évitant ainsi une erreur de division par zéro.

7.3 CASE WHEN

La structure `CASE WHEN` est une instruction conditionnelle utilisée pour évaluer des expressions et retourner une valeur en fonction d'une condition.

```
CASE WHEN condition THEN resultat ELSE resultat_defaut END
```

Dans un cas multi-conditions :

```
CASE  
  WHEN condition1 THEN resultat1  
  WHEN condition2 THEN resultat2  
  ELSE resultat_defaut  
END
```

Exemple d'utilisation : calculer des agrégats conditionnels, comme compter les confirmations réussies :

```
SUM(CASE WHEN action = 'confirmed' THEN 1 ELSE 0 END) AS confirmed_count
```

Ici, on compte seulement les lignes où `action = 'confirmed'`.

7.4 IF

La fonction `IF()` renvoie une valeur si une condition est VRAIE, ou une autre valeur si une condition est FAUSSE.

Voici la syntaxe :

```
IF(condition, value_if_true,  
value_if_false)
```

Voici des exemples :

```
SELECT IF(500 < 1000, 5, 10);  
SELECT IF(500<1000, "YES", "NO");
```

Un autre exemple :

```
SELECT  
    OrderID,  
    Quantity,  
    IF(Quantity>10, "MORE", "LESS")  
FROM OrderDetails;
```

7.5 EXISTS

Dans le langage SQL, la commande `EXISTS` s'utilise dans une clause conditionnelle pour savoir s'il y a une présence ou non de lignes lors de l'utilisation d'une sous-requête.

Voici la syntaxe :

```
SELECT nom_colonne1  
FROM table1  
WHERE EXISTS (  
    SELECT nom_colonne2  
    FROM table2  
    WHERE nom_colonne3 = 10);
```

Important : cette commande n'est pas à confondre avec la clause `IN`. La commande `EXISTS` vérifie si la sous-requête retourne un résultat ou non, tandis que `IN` vérifie la concordance d'une à plusieurs données.

7.6 GREATEST et LEAST

En SQL, les fonctions `GREATEST()` et `LEAST()` permettent de comparer plusieurs expressions (colonnes, valeurs littérales, variables, etc.) et de récupérer respectivement la plus grande ou la plus petite valeur parmi ces arguments.

- `GREATEST(val1, val2, ...)` : retourne la valeur maximale parmi les arguments transmis.
- `LEAST(val1, val2, ...)` : retourne la valeur minimale parmi les arguments transmis.

Ces fonctions sont disponibles dans plusieurs SGBD, notamment MySQL, PostgreSQL et Oracle. Elles ne font pas partie du core standard SQL ISO, mais sont très largement supportées.

Voici la syntaxe :

```
SELECT
    employe_id,
    GREATEST(salaire, prime, bonus) AS montant_max,
    LEAST(salaire, prime, bonus) AS montant_min
FROM contrats ;
```

Points importants :

- Renvoie NULL : si un des arguments est NULL, la plupart des implémentations (par exemple MySQL, PostgreSQL, Oracle) retournent NULL (sauf configuration particulière).
- Nombre d'arguments : on peut passer 2, 3, 4 ou plus de valeurs à comparer, sous forme de champs ou de valeurs constantes.
- Type de retour : les SGBD essaient souvent de promouvoir ou de convertir les valeurs pour qu'elles soient comparables. Il est préférable que les arguments soient du même type ou de types compatibles.

8 Opérations de pivotement

L'opérateur **PIVOT** dans SQL Server et Oracle est une technique extrêmement utile qui transforme les tableaux en colonnes.

Transformez vos données de lignes en colonnes (ou vice-versa) pour une meilleure visualisation et analyse.

Utilité : Essentiel pour créer des tableaux croisés dynamiques (TCD) ou pour préparer les données pour une visualisation.

8.1 PIVOT

L'opérateur SQL Server **PIVOT** est utile pour résumer des données car il permet de transformer des lignes en colonnes.

Examinons le tableau `city_sales` ci-dessous, qui indique les ventes générales d'un produit dans cinq grandes villes des États-Unis.

	city	year	sales
1	New York	2019	1000
2	New York	2019	1100
3	New York	2020	1500
4	New York	2020	1600
5	New York	2021	2000
6	Los Angeles	2019	1100
7	Los Angeles	2019	1150
8	Los Angeles	2020	1600
9	Los Angeles	2020	1650
10	Los Angeles	2021	2100
11	Chicago	2019	1200
12	Chicago	2019	1250
13	Chicago	2020	1700
14	Chicago	2020	1750
15	Chicago	2021	2200

Nous allons utiliser la requête suivante, qui utilise l'opérateur **PIVOT**, pour faire pivoter plusieurs colonnes dans le tableau ci-dessus :

```
-- Select the columns for the output: city and sales data for 2019, 2020, and 2021
SELECT
    city,
    [2019] AS Sales_2019,
    [2020] AS Sales_2020,
    [2021] AS Sales_2021
FROM
(
    -- Subquery to select city, year, and sales from city_sales table
    SELECT city, year, sales
    FROM city_sales
) AS src
PIVOT
(
    -- Pivot the sales data to have years as columns and sum the sales for each y
    SUM(sales)
    FOR year IN ([2019], [2020], [2021])
) AS pvt;
```

On obtient alors :

	city	Sales_2019	Sales_2020	Sales_2021
1	Chicago	2450	3450	2200
2	Houston	2650	3650	2300
3	Los Angeles	2250	3250	2100
4	New York	2100	3100	2000

8.2 UNPIVOT

UNPIVOT est un opérateur relationnel qui nous permet de convertir des colonnes en lignes dans une table (l'inverse de PIVOT).

Pour plus de détails : <https://www.geeksforgeeks.org/unpivot-in-sql-server/>

9 Les expressions régulières en SQL (REGEX)

9.1 Définitions et Bases

Une expression régulière, plus communément appelée REGEX (REGular EXpression) permet de faire des recherches et de la reconnaissance sur des chaînes de caractères.

Par exemple on peut extraire des numéros de téléphone d'une chaîne de caractère, ou vérifier que l'email ressemble bien à un email, exemple :

- **Adresses email** : `^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`
- **Numéros de téléphone** : `^[+]?[1-9]\d{1,14}$`

Il ne s'agit pas d'un langage de programmation à proprement parler : la quasi-intégralité des langages dispose d'une bibliothèque REGEX permettant l'utilisation des expressions régulières.

De plus, la syntaxe ne varie que très peu d'un langage ou d'un outil à l'autre, ce qui facilite grandement la compatibilité entre les différentes plateformes.

En SQL, on s'en servira entre autres pour la validation des données, l'extraction de motifs complexes ou le nettoyage de texte à grande échelle.

Voici un tableau récapitulatif :

Caractère	Signification
<code>^</code>	Début d'une ligne
<code>\$</code>	Fin d'une ligne
<code>.</code>	Correspond à n'importe quel caractère
<code>\s</code>	Correspond à un espace
<code>\S</code>	Correspond à n'importe quel caractère sauf l'espace
<code>*</code>	Répète un caractère zéro ou plusieurs fois
<code>+</code>	Répète un caractère une ou plusieurs fois
<code>[^XYZ]</code>	Correspond à un caractère unique qui n'est pas dans la liste XYZ
<code>[a-z0-9]</code>	Correspond aux lettres minuscules de A à Z ou chiffres de 0 à 9
<code>[:alpha:]</code>	Correspond aux lettres minuscules ou majuscules
<code>[:punct:]</code>	Correspond aux ponctuations
<code>\w</code>	Correspond à n'importe quel caractère alphanumérique (<code>[a-zA-Z0-9]</code>)
<code>\W</code>	Correspond à n'importe quel caractère non alphanumérique (<code>[^a-zA-Z0-9]</code>)
<code>\d</code>	Correspond à n'importe quel caractère numérique
<code>\D</code>	Correspond à n'importe quel caractère non numérique
<code>(</code>	Indique le début de l'extraction de la chaîne
<code>)</code>	Indique la fin de l'extraction de la chaîne
<code>R S</code>	L'expression régulière R ou S
Quantificateurs	
<code>?</code>	Répète un caractère zéro ou une fois
<code>*</code>	Répète un caractère zéro ou plusieurs fois
<code>+</code>	Répète un caractère une ou plusieurs fois
<code>{n}</code>	Répétition n fois consécutives
<code>{n,m}</code>	Répétition n à m fois consécutives
<code>{,n}</code>	Répétition zéro à n fois consécutives
<code>{n,}</code>	Répétition au moins n fois consécutives
Les caractères spéciaux	
<code>[] { } () \ ^ \$. ? * +</code>	Ces caractères sont spéciaux . Pour les afficher, il faut au préalable les échapper avec \

9.2 Exemples et astuces

En SQL (avec LIKE), le % est un joker qui signifie n'importe quelle suite de caractères. C'est vraiment le symbole principal que l'on utilisera

On pourra le placer avant ou après la cible de la requête.

Supposons qu'on cherche tous les patients atteints du diabète. Une colonne indiquera si c'est le cas avec un code DIAB1.

Voici la syntaxe :

```
SELECT *  
FROM Patients  
WHERE conditions LIKE "%DIAB1%" OR conditions LIKE "DIAB1%
```

On pourra également directement faire du REGEX dans un WHERE. Par exemple pour trouver toutes les adresses mails valides de @leetcode.com :

```
SELECT *  
FROM Users  
WHERE mail REGEXP '^[a-zA-Z][a-zA-Z0-9_.-]*@leetcode[.]com$';
```

9.3 Ressources utiles

Pour faire rapidement du REGEX ou vérifier son expression :

- N'importe quel LLM (Chat GPT ou autre)
- <https://regex101.com/>
- <https://regexlearn.com/fr>

10 Fenêtres analytiques/Fonctions de fenêtrages (Window Functions)

Les fonctions de fenêtrage vous permettent d'effectuer des calculs sur un ensemble de lignes liées à la ligne actuelle.

Idéal pour les calculs de classement, les moyennes mobiles, ou les totaux cumulatifs sans avoir à utiliser de sous-requêtes complexes.

Vous pouvez également entendre parler de fonctions de fenêtrage en SQL, de fonctions analytiques ou de fonctions `OVER()`.

10.1 Qu'est-ce qu'une fenêtre ?

Un ensemble de lignes liées à la ligne courante est appelé une fenêtre ou un cadre de fenêtre. D'où le nom de ces fonctions : leur résultat est basé sur un cadre de fenêtre coulissante.

10.2 `OVER(PARTITION BY...)`

L'expression SQL `PARTITION BY` est une sous-clause de la `OVER` clause, qui est utilisée dans presque toutes les invocations de fonctions de fenêtre telles que `AVG()`, `MAX()` et `RANK()`.

La `PARTITION BY` sous-clause SQL est utilisée pour définir les enregistrements à inclure dans le cadre de fenêtre associée à chaque enregistrement du résultat.

Voici comment utiliser la `PARTITION BY` clause SQL à travers un exemple :

marque_de_voiture	modèle_de_voiture	type_de_voiture	prix_de_la_voiture
Gué	Mondeo	prime	18200
Renault	Feu	sport	16500
Citroën	Cactus	prime	19000
Gué	Faucon	faible coût	8990
Gué	Galaxie	standard	12400
Renault	Mégane	standard	14300
Citroën	Picasso	prime	23400

Pour chaque voiture, nous souhaitons obtenir la marque, le modèle, le prix, le prix moyen de toutes les voitures et le prix moyen du même type de voiture (pour avoir une meilleure idée de la façon dont le prix d'une voiture donnée se compare à celui des autres voitures). Voici la requête :

```
SELECT
    car_make,
    car_model,
    car_price,
    AVG(car_price) OVER() AS "overall average price",
    AVG(car_price) OVER (PARTITION BY car_type) AS "car type average price"
FROM car_list_prices;
```

Le résultat de cette requête est :

marque_de_voiture	modèle_de_voiture	prix_de_la_voiture	prix moyen global	type de voiture prix moyen
Gué	Mondeo	18200	16112.85	20200,00
Renault	Feu	16500	16112.85	16500,00
Citroën	Cactus	19000	16112.85	20200,00
Gué	Faucon	8990	16112.85	8990,00
Gué	Galaxie	12400	16112.85	13350,00
Renault	Mégane	14300	16112.85	13350,00
Citroën	Picasso	23400	16112.85	20200,00

La requête ci-dessus utilise deux fonctions de fenêtre. La première est utilisée pour calculer le prix moyen de toutes les voitures de la liste de prix. Elle utilise la fonction de fenêtre `AVG()` avec une `OVER` clause vide. La deuxième fonction de fenêtre est utilisée pour calculer le prix moyen d'un produit spécifique `car_type` comme standard, premium, sport, etc. C'est là que nous utilisons une `OVER` clause avec une `PARTITION BY` sous-clause.

Autre exemple : `SUM(ventes) OVER (PARTITION BY produit ORDER BY date)`

10.3 ROW_NUMBER

`ROW_NUMBER` est une valeur temporaire calculée lorsque la requête est exécutée. Elle numérote toutes les lignes dans l'ordre (par exemple 1, 2, 3, 4, 5).

Exemple :

```
SELECT
    ROW_NUMBER() OVER(ORDER BY name ASC) AS Row#,
    name,
    recovery_model_desc
FROM sysdatabases
WHERE database_id < 5;
```

Remarque : Vous devez déplacer la clause `ORDER BY` vers le haut jusqu'à la clause `OVER`.

10.4 RANK

Retourne le rang de chaque ligne au sein de la partition d'un jeu de résultats. Le rang d'une ligne est un, plus le nombre de rangs précédant la ligne en question.

RANK fournit la même valeur numérique pour les liens (par exemple 1, 2, 2, 4, 5).

10.5 LAG() et LEAD()

LAG() et **LEAD()** sont des fonctions positionnelles.

10.5.1 LAG()

La fonction **LAG()** est une fonction de fenêtre (ou analytic function) utilisée pour accéder à une ligne précédant la ligne courante dans un ensemble de résultats. Elle est souvent utilisée pour comparer une valeur avec celle d'une ligne précédente sans avoir à effectuer de jointure.

Voici sa syntaxe :

```
LAG(expression [, offset [, default]])  
OVER (PARTITION BY partition_expression ORDER BY order_expression)
```

10.5.2 LEAD()

La fonction **LEAD()** permet d'accéder à la valeur de la ligne suivante dans un ensemble de résultats sans avoir besoin de joindre la table à elle-même. C'est particulièrement utile pour effectuer des calculs sur des lignes successives, comme obtenir la différence entre des valeurs dans des lignes adjacentes. Voici un exemple :

```
SELECT  
    mois,  
    chiffre_affaires,  
    LEAD(chiffre_affaires) OVER (ORDER BY mois) AS chiffre_affaires_suivant  
FROM ventes ;
```

Ici on voulait savoir quel est le chiffre d'affaires du mois suivant pour chaque ligne. On renverra NULL pour le dernier mois de la base de données (vu qu'on aura atteint la limite).

10.5.3 Source

Cette section est inspirée de ce site : <https://learnsql.fr/blog/la-fonction-lag-et-la-fonction-lead-en-sql/>

11 Les opérations CRUD et quelques autres commandes

Les opérations CRUD sont les fondements de toutes interactions avec une base de données. Ces opérations permettent de manipuler les données de manière structurée et efficace, que ce soit pour créer de nouveaux enregistrements, lire des informations existantes, mettre à jour des données ou supprimer des entrées obsolètes.

CRUD est un acronyme des noms des quatre opérations de base de la gestion de la persistance des données et applications :

- CREATE (créer)
- READ ou RETRIEVE (lire)
- UPDATE (mettre à jour)
- DELETE ou DESTROY (supprimer)

ATTENTION : Ce sont des commandes à manipuler avec précaution.

Avant d'essayer de modifier des lignes, il est recommandé d'effectuer une sauvegarde de la base de données, ou tout du moins de la table concernée par la modification. Ainsi, s'il y a une mauvaise manipulation il est toujours possible de restaurer les données.

11.1 CREATE TABLE

La commande `CREATE TABLE` permet de créer une table en SQL.

Un tableau est une entité qui est contenu dans une base de données pour stocker des données ordonnées dans des colonnes. La création d'une table sert à définir les colonnes et le type de données qui seront contenus dans chacun des colonne (entier, chaîne de caractères, date, valeur binaire ...).

La syntaxe générale pour créer une table est la suivante :

```
CREATE TABLE nom_de_la_table (  
    colonne1 type_donnees,  
    colonne2 type_donnees,  
    colonne3 type_donnees,  
    colonne4 type_donnees  
)
```

Dans cette requête, 4 colonnes ont été définies. Le mot-clé "type_donnees" sera à remplacer par un mot-clé pour définir le type de données (INT, DATE, TEXT ...). Pour chaque colonne, il est également possible de définir des options telles que (liste non-exhaustive) :

- (IS) NOT NULL : empêche d'enregistrer une valeur nulle pour une colonne.
- DEFAULT : attribuer une valeur par défaut si aucune données n'est indiquée pour cette colonne lors de l'ajout d'une ligne dans la table.
- PRIMARY KEY : indiquer si cette colonne est considérée comme clé primaire pour un index.
- Revoir la section "Notions de clés"
- etc...

11.2 READ

Il s'agit ni plus ni moins que du `SELECT` introduit au départ.

11.3 UPDATE

La commande `UPDATE` permet d'effectuer des modifications sur des lignes existantes. Très souvent cette commande est utilisée avec `WHERE` pour spécifier sur quelles lignes doivent porter la ou les modifications.

Syntaxe :

```
UPDATE table  
SET nom_colonne_1 = nouvelle valeur  
WHERE condition
```

Pour spécifier en une seule fois plusieurs modification, il faut séparer les attributions de valeur par des virgules. Ainsi la syntaxe deviendrait la suivante :

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

Exemple : Update the patients table for the allergies column. If the patient's allergies is null then replace it with 'NKA' :

```
UPDATE patients  
SET allergies = 'NKA'  
WHERE allergies IS NULL;
```

11.4 DELETE

La commande `DELETE` permet de supprimer des lignes dans une table. En utilisant cette commande associé à `WHERE` il est possible de sélectionner les lignes concernées qui seront supprimées.

Voici la syntaxe :

```
DELETE FROM table  
WHERE condition;
```

Attention : s'il n'y a pas de condition `WHERE` alors toutes les lignes seront supprimées et la table sera alors vide.

11.5 TRUNCATE

Pour supprimer toutes les lignes d'une table, il est aussi possible d'utiliser la commande `TRUNCATE`, de la façon suivante :

```
TRUNCATE TABLE utilisateur;
```

La différence majeure étant que la commande `TRUNCATE` va ré-initialiser l'auto-incrémente s'il y en a un. Tandis que la commande `DELETE` ne ré-initialise pas l'auto-incrément.

11.6 DROP TABLE

La commande `DROP TABLE` permet de supprimer définitivement une table d'une base de données. Cela supprime en même temps les éventuels index, trigger, contraintes et permissions associées à cette table.

Une fois supprimée, les données sont perdues : Avant de l'utiliser sur une base importante il peut être judicieux d'effectuer un backup (une sauvegarde) pour éviter les mauvaises surprises.

On l'utilise ainsi :

```
DROP TABLE nom_table
```

S'il y a une dépendance avec une autre table, il est recommandé de les supprimer avant de supprimer la table. C'est le cas par exemple s'il y a des clés étrangères.

11.7 ALTER TABLE

La commande `ALTER TABLE` permet de modifier une table existante. Idéal pour ajouter une colonne, supprimer une colonne ou modifier une colonne existante, par exemple pour changer le type.

D'une manière générale, la commande s'utilise de la manière suivante :

```
ALTER TABLE nom_table  
instruction
```

Voyons les différentes instructions de cette commande.

11.7.1 Ajouter une colonne

L'ajout d'une colonne dans une table devra suivre la syntaxe suivante :

```
ALTER TABLE nom_table  
ADD nom_colonne type_donnees
```

Ici, le mot-clé "type_donnees" est à remplacer par un type de données tel que INT, VARCHAR, TEXT, DATE ...

11.7.2 Supprimer une colonne

Il y a 2 manières totalement équivalente pour supprimer une colonne :

```
ALTER TABLE nom_table  
DROP nom_colonne  
OU  
ALTER TABLE nom_table  
DROP COLUMN nom_colonne
```

11.7.3 Modifier une colonne

Pour modifier une colonne, comme par exemple changer le type d'une colonne :

```
ALTER TABLE nom_table  
MODIFY nom_colonne type_donnees
```

11.7.4 Renommer une colonne

Pour renommer une colonne, il convient d'indiquer l'ancien nom de la colonne et le nouveau nom de celle-ci ainsi que son type :

```
ALTER TABLE nom_table  
CHANGE colonne_ancien_nom colonne_nouveau_nom type_donnees
```

12 Transactions et propriétés ACID

12.1 Définition d'une transaction

Une transaction est une séquence d'opérations SQL qui est exécutée comme une unité indivisible. Elle est utilisée pour garantir la cohérence des données dans un système de gestion de base de données relationnelle (SGBDR).

Exemple simple : Transférer de l'argent d'un compte A à un compte B

```
START TRANSACTION ;  
UPDATE comptes SET solde = solde - 100 WHERE id = 1 ;  
UPDATE comptes SET solde = solde + 100 WHERE id = 2 ;  
COMMIT ;
```

Les transactions permettent :

- D'assurer l'intégrité des données
- D'éviter les incohérences en cas de panne
- De gérer les accès concurrents

12.2 Les propriétés ACID

L'acronyme ACID décrit les 4 garanties fondamentales qu'une transaction doit respecter :

Lettre	Nom	Description
A	Atomicité	La transaction est une unité indivisible. Soit toutes les opérations sont exécutées, soit aucune ne l'est. En cas d'échec, toutes les modifications sont annulées.
C	Cohérence	Une transaction fait passer la base de données d'un état valide à un autre, en respectant toutes les règles d'intégrité.
I	Isolation	Chaque transaction s'exécute comme si elle était seule. Les modifications d'une transaction ne sont pas visibles par les autres tant qu'elle n'est pas validée.
D	Durabilité	Une fois validée (commit), une transaction est permanente, même en cas de panne système.

TABLE 1 – Propriétés ACID des transactions

12.2.1 L'isolation : les niveaux

L'isolation est une propriété ACID qui garantit que les transactions concurrentes n'interfèrent pas entre elles de façon indésirable. En SQL, cela se traduit par différents niveaux d'isolation, chacun offrant un compromis entre concurrence et sécurité des données.

Mais avant de voir les niveaux, il faut comprendre les problèmes qu'ils permettent d'éviter :

- Lecture sale (Dirty Read) : lire des données non encore validées par une autre transaction.
- Lecture non répétable (Non-Repeatable Read) : lire deux fois la même donnée et obtenir des résultats différents.
- Phantom Read (Lecture fantôme) : une requête voit un ensemble de lignes différent à chaque exécution, car d'autres transactions insèrent ou suppriment des lignes.

Voici donc les niveaux d'isolation standard :

- **READ UNCOMMITTED** : Le niveau le plus permissif. Les transactions peuvent lire des données modifiées mais non validées par d'autres transactions (lecture sale). Rarement utilisé en pratique.
- **READ COMMITTED** (par défaut dans SQL Server, Oracle) : Empêche les lectures sales, mais permet que la même donnée soit modifiée entre deux lectures dans la même transaction.
- **REPEATABLE READ** (par défaut dans MySQL InnoDB) : Empêche les lectures sales et non répétables. En revanche, les lectures fantômes sont encore possibles.
- **SERIALIZABLE** : Le niveau le plus strict. Toutes les transactions sont exécutées comme si elles étaient en série (l'une après l'autre). Aucune lecture sale, ni non répétable, ni fantôme.

Pour préciser le niveau, on utilisera la syntaxe suivante :

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;  
START TRANSACTION;  
– opérations SQL ici  
COMMIT;
```

12.3 Commandes de gestion des transactions en SQL

A travers l'exemple de la section définition, on a pu commencer à voir avec quelles commandes on réalise une transaction. Pour contrôler explicitement le comportement des transactions, SQL propose plusieurs commandes. Voici les principales :

12.3.1 START TRANSACTION ou BEGIN

Cette commande démarre une nouvelle transaction. À partir de ce moment, les modifications apportées à la base ne seront visibles qu'à l'intérieur de cette transaction jusqu'à validation.

```
START TRANSACTION ; – ou simplement BEGIN ;
```

12.3.2 COMMIT

Valide (enregistre) définitivement les modifications effectuées depuis le début de la transaction. Après un **COMMIT**, les changements deviennent visibles pour les autres utilisateurs et sont persistés dans la base.

12.3.3 ROLLBACK

Annule toutes les modifications effectuées depuis le début de la transaction. La base de données revient à son état initial, comme si aucune instruction n'avait été exécutée.

12.3.4 SAVEPOINT et ROLLBACK TO SAVEPOINT

Ces commandes permettent de définir des points intermédiaires dans une transaction, utiles pour annuler partiellement certaines opérations sans revenir totalement en arrière.

12.4 Détection et gestion des erreurs transactionnelles

Dans un système de gestion de base de données, il est fréquent qu'une transaction rencontre des erreurs : violation de contraintes d'intégrité, conflits d'accès entre plusieurs utilisateurs, ou encore pannes systèmes ou réseau. Pour garantir la fiabilité de l'application et la cohérence des données, il est essentiel de savoir gérer ces erreurs correctement.

Lorsqu'une erreur survient en cours de transaction, il est indispensable de pouvoir restaurer la base de données dans un état cohérent. C'est précisément le rôle de la commande **ROLLBACK**, qui annule toutes les modifications réalisées depuis le début de la transaction. Ainsi, la base revient à son état initial, comme si aucune instruction n'avait été exécutée. C'est ce qu'on appelle le principe d'atomicité, une des propriétés fondamentales des transactions.

Dans les cas plus complexes, on peut vouloir annuler partiellement une transaction sans tout recommencer. SQL permet cela grâce aux points de sauvegarde, ou **SAVEPOINT**. En plaçant des **SAVEPOINT** à des étapes critiques de la transaction, on peut revenir en arrière jusqu'à ce point précis grâce à la commande **ROLLBACK TO SAVEPOINT**. Cela permet de mieux contrôler le flux d'exécution, notamment lorsqu'on exécute plusieurs opérations indépendantes dans la même transaction.

Voici un exemple : imaginons que l'on retire une somme d'un compte, puis que l'on tente de créditer un second compte. Si la seconde opération échoue (par exemple parce que le compte est bloqué), on peut revenir au point intermédiaire après le retrait, sans annuler toute la transaction :

```
START TRANSACTION ;
UPDATE comptes SET solde = solde - 100 WHERE id = 1 ;
SAVEPOINT retrait_effectue ;
UPDATE comptes SET solde = solde + 100 WHERE id = 2 ;
– Si une erreur survient ici, on peut annuler uniquement cette étape :
ROLLBACK TO retrait_effectue ;
COMMIT ;
```

12.5 Transactions imbriquées et limites selon les SGBD

Une transaction imbriquée (ou nested transaction) est une transaction définie à l'intérieur d'une autre. Cela permet de structurer des blocs de traitement indépendants, chacun pouvant être validé ou annulé sans affecter toute la transaction principale.

Exemple logique :

- La transaction principale gère une commande client.
- À l'intérieur, on lance une transaction secondaire pour réserver le stock.
- Si la réservation échoue, on veut annuler juste cette partie, pas l'ensemble de la commande.

La plupart des SGBD ne supportent pas directement les transactions imbriquées complètes (avec commit partiel), mais ils permettent de simuler ce comportement grâce à la combinaison suivante :

- SAVEPOINT : pour marquer le début d'une sous-transaction
- ROLLBACK TO SAVEPOINT : pour annuler la sous-partie sans affecter le reste
- COMMIT : reste global à la transaction principale

Voyons un exemple détaillé :

```
START TRANSACTION ;
- Traitement principal -
UPDATE commandes SET statut = 'traitement' WHERE id = 100 ;
- Début de transaction imbriquée simulée -
SAVEPOINT reservation_stock ;
- Sous-opération : réserver du stock -
UPDATE stock SET disponible = disponible - 10 WHERE produit_id = 42 ;
- Imaginons une erreur ici : stock insuffisant -
- => on annule uniquement la réservation -
ROLLBACK TO reservation_stock ;
- La transaction principale peut continuer -
UPDATE logs SET message = 'Stock insuffisant' WHERE commande_id = 100 ;
COMMIT ;
```

12.6 Bonnes pratiques

Voici quelques bonnes pratiques dans ce contexte de transactions :

- Toujours utiliser des transactions pour les opérations critiques (finance, commandes...).
- Préférer des transactions courtes pour éviter les blocages : Plus une transaction dure longtemps, plus elle a de chances de générer des conflits avec d'autres, ou de bloquer des ressources. Il faut donc éviter de faire traîner des transactions en mémoire, surtout dans des systèmes multi-utilisateurs à forte charge.
- Gérer les erreurs : toujours prévoir un ROLLBACK en cas d'exception.
- En dehors du SQL pur, la gestion des erreurs doit également être prise en compte dans l'application cliente. Dans un langage comme Python, Java ou PHP, on encapsule généralement les transactions dans des blocs try/except, afin de détecter les erreurs, exécuter un ROLLBACK si nécessaire, puis informer l'utilisateur ou journaliser l'incident.

13 Procédures stockées

13.1 Définition et utilité

Une procédure stockée (ou stored procedure) est un ensemble d'instructions SQL enregistrées directement dans le Système de Gestion de Base de Données (SGBD). Contrairement aux requêtes classiques qui sont exécutées une seule fois, une procédure peut être appelée à tout moment par son nom, ce qui permet de réutiliser du code facilement et efficacement.

Une procédure stockée fonctionne un peu comme une fonction dans un langage de programmation : elle regroupe un ensemble d'instructions, reçoit éventuellement des paramètres, et peut effectuer des actions conditionnelles, des calculs, des mises à jour, etc.

Avantages principaux :

- Réutilisabilité : on écrit une fois, on exécute autant de fois qu'on veut.
- Performance : comme la procédure est stockée côté serveur, cela réduit les allers-retours entre le client et la base.
- Sécurité : on peut autoriser l'accès à certaines procédures sans donner un accès direct aux tables.
- Centralisation de la logique métier : les traitements complexes sont gérés directement dans la base.

Attention : le code des procédures peut varier selon le SGBD (MySQL, PostgreSQL, Oracle, etc.), ce qui peut rendre les migrations plus délicates.

13.2 Syntaxe générale (MySQL ou PostgreSQL)

13.2.1 Syntaxe

Voici la syntaxe générale pour définir une procédure :

```
DELIMITER //  
CREATE PROCEDURE nom_procedure (IN param1 TYPE, OUT param2 TYPE, ...)  
BEGIN  
    – Instructions SQL  
END //  
DELIMITER ;
```

Et voici la syntaxe pour appeler cette procédure :

```
CALL nom_procedure(param1, param2, ...)
```

13.2.2 Exemple

Voici un exemple dans lequel on va lister les pays d'un continent donné :

```
DELIMITER //
CREATE PROCEDURE country_hos (IN con CHAR(20))
BEGIN
    SELECT Name, HeadOfState
    FROM Country
    WHERE Continent = con ;
END //
DELIMITER ;
```

Pour ensuite utiliser cette procédure :

```
CALL country_hos('Europe');
```

13.3 Contrôle de flux dans les procédures

Les procédures stockées peuvent intégrer une logique conditionnelle comme :

- IF...THEN...ELSE...END IF ;
- WHILE...DO...END WHILE ;
- LOOP...LEAVE...END LOOP ;
- REPEAT...UNTIL...END REPEAT ;

Cela permet d'automatiser des traitements complexes ou conditionnels au sein de la base.

14 Triggers (déclencheurs)

14.1 Définition

Un trigger (ou déclencheur) est un objet de base de données qui permet d'exécuter automatiquement un bloc d'instructions SQL lorsqu'un événement précis se produit sur une table (comme une insertion, une mise à jour ou une suppression). Contrairement à une requête manuelle ou une procédure stockée qui est appelée volontairement, un trigger s'exécute de manière automatique et transparente.

Les triggers sont utilisés pour ajouter de la logique métier au niveau de la base de données, afin de :

- garantir l'intégrité des données,
- enregistrer automatiquement des actions (comme un historique),
- faire appliquer des règles spécifiques sans dépendre du code applicatif.

14.2 Fonctionnement général

Un trigger s'exécute automatiquement lorsqu'un événement spécifique se produit sur **une seule** table. Donc un trigger est attaché à une table spécifique. Il ne peut pas modifier directement la table sur laquelle il est défini (cela causerait une boucle récursive, sauf configuration explicite). L'ordre d'exécution des triggers multiples peut être contrôlé dans certains SGBD (ex : PostgreSQL avec **FOLLOWS**).

Il peut intervenir avant ou après une opération de modification de données.

14.2.1 Moment de déclenchement

Il existe principalement trois types de déclenchements (selon les SGBD) :

- **BEFORE** : le trigger s'exécute avant que l'opération ne soit appliquée sur la table. Utilisé pour valider ou modifier les données avant insertion ou mise à jour.
- **AFTER** : le trigger s'exécute après l'exécution de l'opération. Souvent utilisé pour créer un historique ou déclencher des actions secondaires.
- **INSTEAD OF** (dans certains SGBD comme SQL Server ou Oracle) : remplace l'opération (utile surtout sur des vues).

14.2.2 Type d'événement déclencheur

Ensuite, un trigger peut réagir à trois événements fondamentaux :

- **INSERT** : lorsqu'une nouvelle ligne est ajoutée
- **UPDATE** : lorsqu'une ligne existante est modifiée
- **DELETE** : lorsqu'une ligne est supprimée

Il est également possible de combiner plusieurs événements dans un même trigger selon le SGBD.

14.2.3 Portée de déclenchement

Les triggers peuvent être définis :

- **FOR EACH ROW** : s'exécute une fois pour chaque ligne affectée (le plus courant)
- **FOR EACH STATEMENT** : s'exécute une seule fois par requête, quel que soit le nombre de lignes concernées (moins courant, et dépend du SGBD).

14.3 Syntaxe générale d'un Trigger

La création d'un trigger varie légèrement selon le SGBD utilisé (MySQL, PostgreSQL, Oracle, etc.), mais les éléments de base restent similaires.

Voici la structure générale d'un trigger dans **MySQL** :

```
DELIMITER //  
CREATE TRIGGER nom_du_trigger  
BEFORE | AFTER INSERT | UPDATE | DELETE  
ON nom_de_la_table  
FOR EACH ROW  
BEGIN  
    - Instructions SQL END //  
DELIMITER;
```

Voici la signification de chaque élément utilisé :

DELIMITER // (spécifique à MySQL) : par défaut, le délimiteur des instructions SQL est le point-virgule ;. Lorsqu'un trigger contient plusieurs instructions entre **BEGIN** et **END**, on redéfinit temporairement le délimiteur afin d'éviter toute confusion.

CREATE TRIGGER nom_du_trigger Permet de créer un nouveau trigger. Le nom doit être unique dans le schéma. Il est conseillé d'adopter une convention de nommage explicite (ex. : `trg_audit_insert_user`).

BEFORE ou AFTER Indique le moment d'exécution du trigger :

- **BEFORE** : le trigger s'exécute avant l'action (utile pour valider ou modifier des données),
- **AFTER** : le trigger s'exécute après l'action (souvent utilisé pour journaliser ou lancer des actions secondaires).

INSERT, UPDATE, DELETE Spécifie le type d'événement sur la table cible qui déclenchera le trigger.

ON nom_de_la_table Indique la table sur laquelle le trigger est attaché. Toute modification sur cette table liée à l'événement spécifié activera le trigger.

FOR EACH ROW Signifie que le trigger est exécuté pour chaque ligne affectée. Si une requête modifie plusieurs lignes, le trigger est exécuté autant de fois que de lignes concernées.

BEGIN ... END Délimite le bloc d'instructions SQL qui seront exécutées lorsque le trigger est déclenché.

OLD et NEW Ces pseudo-enregistrements permettent d'accéder aux anciennes et nouvelles valeurs des colonnes :

- **NEW.colonne** : valeur après l'opération (**INSERT** ou **UPDATE**),
- **OLD.colonne** : valeur avant l'opération (**UPDATE** ou **DELETE**).

15 Index & Optimisation des requêtes

En SQL, un **index** est une structure de données associée à une ou plusieurs colonnes d'une table, destinée à améliorer la rapidité des opérations de recherche. Il fonctionne de manière similaire à l'index d'un livre : au lieu de parcourir toutes les pages (lignes d'une table), on utilise un repère direct pour accéder rapidement à l'information souhaitée.

Sans index, le SGBD doit effectuer une *recherche séquentielle*, c'est-à-dire lire chaque ligne une par une pour vérifier si elle satisfait la condition d'une requête. Cela devient coûteux en temps dès que la taille de la table augmente.

L'utilisation d'index permet donc d'**optimiser fortement les performances des requêtes**, en particulier celles contenant des clauses `WHERE`, `JOIN`, `ORDER BY` ou `GROUP BY`.

Cependant, il est important de noter que les index ne sont pas gratuits :

- Ils occupent de l'espace disque supplémentaire.
- Ils peuvent ralentir les opérations d'insertion, de mise à jour ou de suppression, car l'index doit être maintenu à jour.

Ainsi, l'indexation doit être réfléchie : il ne s'agit pas de créer un index sur chaque colonne, mais sur celles qui sont fréquemment utilisées dans des filtres ou des regroupements.

15.1 Types d'index

Il existe plusieurs types d'index, chacun ayant des cas d'usage spécifiques selon le type de données ou les besoins en performance. Voici les principaux :

Index simple Un index portant sur une seule colonne. Il est le plus courant et est utilisé pour accélérer les recherches ou tris sur cette colonne.

Index composé (ou multicolonne) Index portant sur plusieurs colonnes. Il est utile pour optimiser les requêtes utilisant des conditions sur plusieurs champs, notamment dans les clauses `WHERE` ou `ORDER BY`.

- L'ordre des colonnes dans l'index est important : un index sur `(col1, col2)` peut optimiser une requête filtrant sur `col1` seul ou sur `col1` et `col2`, mais pas sur `col2` seul.

Index unique Il impose l'unicité des valeurs dans la ou les colonnes indexées. Il est souvent utilisé pour les identifiants ou les colonnes avec contrainte d'unicité.

Index FULLTEXT (texte intégral) Utilisé pour rechercher des mots ou expressions dans de longues chaînes de texte. Très utile dans des champs de type `TEXT`, par exemple pour l'indexation de contenus d'articles ou de commentaires.

Index spatial Spécifique aux bases de données prenant en charge des types géographiques (comme PostGIS, MySQL Spatial). Ces index optimisent les requêtes sur des données géospatiales (ex : recherche de points dans un rayon).

Index partiel (ou filtré) Index portant uniquement sur une portion des lignes de la table, selon une condition spécifiée. Par exemple, indexer uniquement les utilisateurs actifs :

```
CREATE INDEX idx_actifs ON utilisateurs(email)
WHERE actif = true ;
```

Cela réduit la taille de l'index et peut améliorer l'efficacité pour certains cas ciblés.

Index fonctionnel (ou exprimé) L'index est créé non pas sur une colonne brute, mais sur le résultat d'une expression ou d'une fonction. Par exemple :

```
CREATE INDEX idx_lower_email
ON utilisateurs(LOWER(email)) ;
```

Cela permet d'optimiser les recherches insensibles à la casse.

15.2 Création et gestion des index

La création d'un index se fait à l'aide de l'instruction `CREATE INDEX`. Selon le type d'index, la syntaxe peut légèrement varier.

15.2.1 Création d'un index simple

```
CREATE INDEX nom_index
ON nom_table(nom_colonne) ;
```

15.2.2 Création d'un index composé

```
CREATE INDEX nom_index
ON nom_table(colonne1, colonne2) ;
```

15.2.3 Création d'un index unique

```
CREATE UNIQUE INDEX nom_index
ON nom_table(nom_colonne) ;
```

15.2.4 Suppression d'un index

```
DROP INDEX nom_index ;
```

Selon le SGBD, la syntaxe peut varier. Par exemple, en MySQL :

```
DROP INDEX idx_nom
ON clients ;
```

15.2.5 Index implicites

Certains index sont créés automatiquement par le SGBD :

- Lorsqu'une colonne est définie comme **PRIMARY KEY**, un index unique est créé.
- Les colonnes déclarées avec une contrainte **UNIQUE** ou **FOREIGN KEY** génèrent également des index sous-jacents.

15.2.6 Bonnes pratiques

- Donner des noms explicites aux index : `idx_nom_colonne` ou `idx_table_champ1_champ2`.
- Éviter de multiplier les index sur des colonnes peu utilisées dans les clauses **WHERE** ou **JOIN**.
- Supprimer les index inutilisés ou redondants pour optimiser les performances en écriture.

15.3 Optimisation des requêtes avec les index

Les index permettent au moteur SQL d'exécuter plus rapidement certaines requêtes en réduisant le volume de données à parcourir. Ils sont particulièrement efficaces dans les cas suivants :

- Recherche par filtre dans une clause **WHERE**
- Tri des résultats avec **ORDER BY**
- Regroupement de données avec **GROUP BY**
- Requêtes utilisant des jointures (**JOIN**)

Utilisation dans les clauses WHERE

La clause **WHERE** permet de filtrer les lignes selon une condition. Sans index, le moteur de base de données est obligé de parcourir chaque ligne de la table (ce qu'on appelle un *full table scan*).

Sans index :

```
SELECT *  
FROM clients  
WHERE email = 'a@exemple.com';
```

Recherche séquentielle sur toute la table (scan complet).

Dans ce cas, chaque ligne de la table `clients` doit être lue pour vérifier si la colonne `email` correspond à la valeur recherchée.

Si un index existe sur la colonne `nom`, la recherche sera beaucoup plus rapide, surtout sur de grandes tables.

Avec index sur email :

```
CREATE INDEX idx_email  
ON clients(email);
```

Grâce à cet index, la base peut effectuer une recherche plus rapide, en accédant directement à la valeur via une structure comme un arbre B+. Cela réduit considérablement le temps de réponse. Accès rapide via l'index (recherche en $\log(n)$).

15.3.1 Index et tri (ORDER BY)

Un index peut aussi optimiser le tri si l'ordre demandé correspond à celui défini dans l'index. Les requêtes utilisant `ORDER BY` demandent un tri explicite. Par défaut, le moteur trie manuellement les résultats après lecture complète des lignes.

Sans index :

```
SELECT *  
FROM commandes  
ORDER BY date_commande ;
```

Ici, le tri est réalisé en mémoire après avoir chargé toutes les lignes, ce qui peut devenir très coûteux.

Avec index sur `date_commande` :

```
CREATE INDEX idx_date  
ON commandes(date_commande) ;
```

L'index garde déjà les données triées. Le SGBD peut ainsi parcourir directement les valeurs dans l'ordre voulu, sans avoir à les trier lui-même.

15.3.2 Index et jointures

Les jointures entre tables reposent généralement sur des clés. Si aucune clé n'est indexée, la base de données doit comparer chaque ligne de la première table à toutes celles de la seconde.

Sans index :

```
SELECT *  
FROM commandes  
JOIN clients  
ON commandes.id_client = clients.id ;
```

Cette approche est inefficace sur de grosses tables (complexité quadratique).

Avec index sur `clients.id` :

```
CREATE INDEX idx_client_id  
ON clients(id) ;
```

La recherche du client associé à chaque commande est alors optimisée. Le SGBD utilise l'index pour trouver rapidement la ligne correspondante dans `clients`.

Index et regroupement (GROUP BY)

Le regroupement de données implique souvent un tri préalable pour regrouper les lignes similaires. Cela peut être coûteux si non optimisé.

Sans index :

```
SELECT categorie, COUNT(*)  
FROM produits  
GROUP BY categorie;
```

Sans tri préalable, le moteur doit traiter ligne par ligne, puis regrouper les catégories en mémoire.

Avec index sur categorie :

```
CREATE INDEX idx_categorie  
ON produits(categorie);
```

L'index rend les données déjà triées, ce qui rend le regroupement beaucoup plus rapide.

15.3.3 Choix des colonnes à indexer

Il est judicieux d'indexer :

- les colonnes souvent utilisées dans des filtres (**WHERE**)
- les colonnes de jointure (**ON**)
- les colonnes utilisées dans des tris ou des regroupements

En revanche, il est déconseillé d'indexer :

- les colonnes peu discriminantes (ex. : booléens)
- les colonnes très fréquemment modifiées
- trop de colonnes d'une même table (surcoût en écriture)

15.3.4 Impact sur les performances en écriture

Les index doivent être mis à jour à chaque insertion, modification ou suppression. Trop d'index ralentissent donc les opérations en écriture (**INSERT**, **UPDATE**, **DELETE**).

Il est donc essentiel de trouver un équilibre entre performance en lecture (requêtes rapides) et performance en écriture.

15.4 Analyse des performances avec EXPLAIN

EXPLAIN est une commande utilisée dans les principaux SGBD (MySQL, PostgreSQL, SQL Server, etc.) pour analyser le plan d'exécution d'une requête. Elle montre comment le moteur de base de données va accéder aux données (type d'accès, index utilisés, coût estimé, etc.), sans exécuter la requête elle-même.

15.4.1 Syntaxe de base

```
EXPLAIN
SELECT *
FROM commandes
WHERE client_id = 1001 ;
```

Selon le SGBD utilisé, la commande peut varier légèrement :

- **MySQL** : `EXPLAIN [FORMAT=JSON] SELECT ...`
- **PostgreSQL** : `EXPLAIN ANALYZE SELECT ...`
- **SQL Server** : `SET SHOWPLAN_ALL ON;`

15.4.2 Résultat typique (MySQL)

Voici un exemple de sortie :

id	select_type	table	type	key
1	SIMPLE	commandes	ref	idx_client_id

Interprétation des colonnes importantes :

- **type** : indique le mode d'accès. Les valeurs vont de **ALL** (scan complet de table, mauvais signe) à **const**, **ref**, **range** (bonnes performances).
- **key** : index utilisé (ici `idx_client_id`).
- **rows** : nombre estimé de lignes à lire.

Amélioration avec un index

Si aucune clé n'est indiquée et que le type est **ALL**, cela signifie qu'aucun index n'est utilisé :

```
type : ALL
key : NULL
rows : 10523
```

Ce diagnostic invite à créer un index ciblé :

```
CREATE INDEX idx_client_id
ON commandes(client_id);
```

Puis relancer **EXPLAIN** pour constater que l'index est bien pris en compte, avec une réduction significative des lignes analysées.

15.4.3 Avec EXPLAIN ANALYZE (PostgreSQL)

Dans PostgreSQL, la commande **EXPLAIN ANALYZE** exécute réellement la requête et affiche son plan d'exécution avec les temps réels d'exécution pour chaque étape :

```
EXPLAIN ANALYZE
SELECT *
FROM commandes
WHERE client_id = 1001 ;
```

Cela permet de diagnostiquer très précisément les goulets d'étranglement d'une requête.

16 Boîte à outils

16.1 STRCMP

Fonction pour comparer deux strings.

La syntaxe est la suivante : `STRCMP(string1, string2)`

Renvoie :

- If `string1 = string2`, this function returns 0
- If `string1 < string2`, this function returns -1
- If `string1 > string2`, this function returns 1

16.2 IN/NOT IN

Très efficace pour définir une liste. Par exemple, si je veux afficher toutes les lignes concernant les patients dont l'id est 1, 45, 534, 879, 1000 alors j'écrirai :

```
SELECT *  
FROM patients  
WHERE patient_id IN (1, 45, 534, 879, 1000);
```

On le voit tout de suite, cela évitera d'écrire :

```
SELECT *  
FROM patients  
WHERE patient_id = 1  
OR patient_id = 45  
OR patient_id = 534  
OR patient_id = 879  
OR patient_id = 1000;
```

On peut également réaliser sa négation avec `NOT IN`.

Attention : avec `NOT IN` si la table contient des valeurs `NULL` !

En effet, sans travail préventif sur ces valeurs nulles, le comportement de la requête peut devenir inattendu : Elle ne retournera aucun résultat.

En SQL, toute comparaison avec `NULL` donne `UNKNOWN` et donc `FALSE` pour toutes les lignes. On utilisera alors un filtre avec par exemple la commande `IS NOT NULL` avant d'utiliser `NOT IN`.

16.3 LIKE

16.3.1 Wildcards

Avant de parler de la commande, nous allons introduire les Wildcards.

Un caractère "wildcard" est utilisé pour se substituer à n'importe quel autre caractère, au sein d'une chaîne de caractère. Ces caractères wildcards sont utilisés dans l'opérateur `LIKE`. Voici les 2 exemples de caractères wildcards :

- % : le symbole pourcentage représente zéro, un ou plusieurs caractères joker.
- _ : le symbole underscore représente un seul caractère joker.

Toutefois, il ne faut pas confondre cette section avec le terme "wildcard" qui désigne l'astérisque utilisé dans une requête de lecture.

16.3.2 Syntaxe et exemple

Ainsi, pour sélectionner tous les prénoms commençant par la lettre C, on pourra écrire :

```
SELECT first_name  
FROM customers  
WHERE first_name LIKE "C%";
```

Ainsi, pour sélectionner tous les prénoms commençant par la lettre C et qui finisse par la lettre e, on écrira donc `LIKE "C%e"`.

Enfin, si je veux afficher tous les patients dont le diagnostic contient le mot "malade", je ferai `LIKE "%malade%"`.

16.4 BETWEEN ... AND ...

L'opérateur `BETWEEN` est utilisé dans une requête SQL pour sélectionner un intervalle de données dans une requête utilisant `WHERE`. L'intervalle peut être constitué de chaînes de caractères, de nombres ou de dates. L'exemple le plus concret consiste par exemple à récupérer uniquement les enregistrements entre 2 dates définies :

```
SELECT *  
FROM table  
WHERE nom_colonne BETWEEN 'valeur1' AND 'valeur2';
```

16.5 TRIM

La fonction `TRIM()` permet de supprimer des caractères au début et en fin d'une chaîne de caractère.

Le plus souvent la fonction `TRIM()` permet de supprimer les caractères invisibles, c'est-à-dire les caractères tels que l'espace, la tabulation, le retour à la ligne ou bien même le retour chariot.

Basiquement :

```
SELECT TRIM(' Exemple ');  
Résultat : "Exemple" sans espaces.
```

Pour supprimer un caractère spécifique en début et en fin de chaîne :

```
SELECT TRIM(BOTH 'x' FROM 'xxxExemplexxx');  
Résultat : "Exemple"
```

Supprimer un caractère spécifique uniquement en début de chaîne :

```
SELECT TRIM(LEADING 'x' FROM 'xxxExemplexxx');  
Résultat : "Exemplexxx"
```

Supprimer un caractère spécifique uniquement en fin de chaîne :

```
SELECT TRIM(TRAILING 'x' FROM 'xxxExemplexxx');  
Résultat : "xxxExemple"
```

16.6 CONCAT

Dans le langage SQL la fonction `CONCAT()` permet de concaténer les valeurs de plusieurs colonnes pour ne former qu'une seule chaîne de caractère. Cette fonction SQL peut se révéler pratique pour mettre bout-à-bout les valeurs de plusieurs colonnes pour n'en afficher qu'une. C'est donc pratique pour afficher un résultat facile à lire tout en maintenant les données dans plusieurs colonnes pour une bonne maintenance des données.

Plusieurs façons de l'utiliser :

```
SELECT
    CONCAT(first_name, ' ', last_name) AS full_name
FROM patients;
```

OU

```
SELECT
    first_name || ' ' || last_name
FROM patients;
```

On peut également l'utiliser dans un `WHERE` :

```
SELECT id
FROM table
WHERE colonne1 = CONCAT(colonne2, colonne3);
```

16.7 GROUP_CONCAT

La fonction `GROUP_CONCAT()` permet de regrouper les valeurs non nulles d'un groupe en une chaîne de caractère. Cela est utile pour regrouper des résultats en une seule ligne autant d'avoir autant de ligne qu'il y a de résultat dans ce groupe.

Attention : la fonction `GROUP_CONCAT()` est actuellement disponible que pour MySQL. Les autres Systèmes de Gestion de Base de Données (SGBD) tel que PostgreSQL, SQLite ou SQL Server n'intègrent pas cette fonctionnalité.

Voici la syntaxe :

```
SELECT id, GROUP_CONCAT(nom_colonne)
FROM table
GROUP BY id;
```

Il est possible de choisir le caractère qui sert de séparateur en respectant la syntaxe suivante :

```
SELECT id, GROUP_CONCAT(nom_colonne SEPARATOR ',')
FROM table
GROUP BY id;
```

Voici un exemple issu d'un des exercices de leet code :

Example 1:

Input:

Activities table:

sell_date	product
2020-05-30	Headphone
2020-06-01	Pencil
2020-06-02	Mask
2020-05-30	Basketball
2020-06-01	Bible
2020-06-02	Mask
2020-05-30	T-Shirt

Output:

sell_date	num_sold	products
2020-05-30	3	Basketball,Headphone,T-shirt
2020-06-01	2	Bible,Pencil
2020-06-02	1	Mask

La requête pour obtenir l'output est :

```
SELECT
    sell_date,
    COUNT(DISTINCT product) as num_sold,
    GROUP_CONCAT(DISTINCT product ORDER BY product ASC SEPARATOR ',')
    AS products
FROM Activities
GROUP BY sell_date;
```

16.8 REPLACE

La fonction `REPLACE` dans le langage SQL permet de remplacer des caractères alphanumérique dans une chaîne de caractère. Cela sert particulièrement à mettre à jour des données dans une base de données ou à afficher des résultats personnalisés.

```
SELECT REPLACE('Hello tout le monde', 'Hello', 'Bonjour');
```

16.9 CAST

La fonction `CAST()` est une fonction de transtypage qui permet de convertir une données d'un type en un autre. Il est par exemple possible de transformer une date au format DATETIME en DATE, ou l'inverse.

```
SELECT CAST( expression AS type );
```

16.10 Les chaînes de caractères

16.10.1 Fonctions MID, RIGHT, LEFT

La fonction **RIGHT** retourne un morceau d'une chaîne de caractère, d'une longueur renseignée en paramètre et en commençant pas les caractères de droite. **RIGHT(chaine_a_couper, 5)** prendra les 5 caractères en partant de la droite.

La fonction **LEFT** retourne un morceau d'une chaîne de caractère, d'une longueur renseignée en paramètre et en commençant pas les caractères de gauche.

La fonction **MID** retourne un morceau d'une chaîne de caractère, d'une longueur renseignée en paramètre et en commençant pas le caractère dont le numéro est aussi renseigné en paramètre.

16.10.2 Obtenir le nombre de caractères

On utilisera la fonction **LENGTH(...)**.

16.10.3 Majuscule/Minsucule

- **LCASE()** transformer une chaîne de caractère en minuscule (ou LOWER)
- **UCASE()** transformer une chaîne de caractère en majuscule (Ou UPPER)

16.10.4 Segmenter une chaîne de caractère

On utilisera **SUBSTRING()** (Ou **SUBSTR**) pour segmenter une chaîne de caractère.

La fonction **SUBSTRING()** peut s'utiliser de 4 façons différentes, que voici :

- **SUBSTRING(chaine, debut)** : retourne la chaîne de caractère de "chaine" à partir de la position définie par "debut" (position en nombre de caractères)
- **SUBSTRING(chaine FROM debut)** : idem que précédent
- **SUBSTRING(chaine, debut, longueur)** : retourne la chaîne de caractère "chaine" en partant de la position définie par "debut" et sur la longueur définie par "longueur"
- **SUBSTRING(chaine FROM debut FOR longueur)** : idem que précédent

16.10.5 Obtenir la position d'une occurrence

La fonction **INSTR()** est utilisé par MySQL et Oracle pour obtenir la position d'une occurrence dans une chaîne de caractères.

Cette fonction permet donc de savoir si une chaîne est présente dans une autre chaîne et d'obtenir par la même occasion sa position.

Exemple :

```
SELECT SUBSTR('abcdefghij', INSTR('123321234', '2', 3, 2), 2)
FROM STUDENT;
```

16.10.6 Gérer les Noms Propres

Savante combinaison des fonctions décrites précédemment :

```
SELECT user_id,  
       CONCAT(UPPER(LEFT(name,1)),LOWER(RIGHT(name,LENGTH(name)-1))) as name  
FROM Users ;
```

17 SQL avec Python

17.1 sqlite3

Le module `sqlite3` est la bibliothèque standard de Python permettant d'interagir avec une base de données SQLite.



Intégré par défaut dans la bibliothèque standard, il permet de manipuler facilement des bases relationnelles sans configuration serveur, en stockant les données dans un simple fichier ou même directement en mémoire.

Il est possible d'exécuter des requêtes SQL classiques — création de tables, insertion, sélection, mise à jour, suppression — tout en restant dans un environnement Python pur.

L'interaction se fait via des connexions à une base et des curseurs (cursor), comme dans la plupart des drivers SQL. Voyons comment l'utiliser :

```
import sqlite3

# Connexion à un fichier "ma_base.db" :
conn = sqlite3.connect("ma_base.db")
cur = conn.cursor()

# Création de table
cur.execute("CREATE TABLE users (id INTEGER, name TEXT);")

# Insertion de données
cur.execute("INSERT INTO users VALUES (1, 'Alice');")
cur.execute("INSERT INTO users VALUES (2, 'Bob');")

# Requête SELECT
cur.execute("SELECT * FROM users;")
rows = cur.fetchall()
print(rows) # [(1, 'Alice'), (2, 'Bob')]

conn.close()
```

17.2 sqlglot

Le module `sqlglot`, quant à lui, est une bibliothèque externe dédiée à l'analyse et à la transformation des requêtes SQL.



Il permet de parser une requête en une structure arborescente (AST), facilitant ainsi son inspection, sa modification ou sa validation.

L'un des atouts majeurs de `sqlglot` est sa capacité à convertir automatiquement une requête d'un dialecte SQL à un autre : par exemple, il peut transformer une requête écrite pour PostgreSQL en une version compatible avec SQLite, BigQuery ou Snowflake.

Cela en fait un outil de choix pour le développement d'outils multi-SGBD, la migration de bases de données, ou encore l'optimisation et la réécriture dynamique de requêtes.

Contrairement à `sqlite3`, `sqlglot` n'exécute pas de requêtes : il les comprend, les analyse et les reformule. Il doit être installé via `pip`.

Très important : `sqlglot` est un transpileur syntaxique, pas sémantique :

- Il corrige la structure des requêtes d'un dialecte à un autre.
- Mais il n'invente pas ou n'interprète pas automatiquement des types comme par exemple "TEXT → VARCHAR(255)". Pour lui, TEXT est valide aussi en MySQL (et en effet, MySQL accepte TEXT aussi, même si c'est différent de VARCHAR). Donc `sqlglot` laisse TEXT tel quel car c'est syntaxiquement correct en MySQL.

Voici un exemple d'utilisation de ce module python :

```
import sqlglot
# Requête SQL d'origine
query = "SELECT id, name FROM users WHERE active = TRUE"

# Parsing en AST
expression = sqlglot.parse_one(query)

# Traduction en dialecte SQLite
sqlite_query = expression.to_sql(dialect="sqlite")
print(sqlite_query)
# Résultat : SELECT id, name FROM users WHERE active = 1
```


18 Le SQL dans la Data Analyse

18.1 Identifier les doublons

```
SELECT customer_id, COUNT(*)  
FROM customers  
GROUP BY customer_id  
HAVING COUNT(*) > 1;
```

18.2 Identifier les valeurs NULL

```
SELECT *  
FROM customers  
WHERE status IS NULL
```

18.3 Vérifier la validité des valeurs

```
SELECT *  
FROM products  
WHERE price <= 0;
```

18.4 Vérifier les valeurs de date

```
SELECT *  
FROM orders  
WHERE order_date < "2022-01-01" OR order_date > "2022-12-31";
```

18.5 Vérifier l'intégrité des données

```
SELECT *  
FROM orders  
WHERE customer_id NOT IN (SELECT customer_id FROM customers);
```

18.6 Identifier les données orphelines

```
SELECT *  
FROM orders  
WHERE order_id NOT IN (SELECT DISTINCT order_id FROM order_details);
```

18.7 Vérifier si la quantité est un nombre

```
SELECT *  
FROM order_details  
WHERE TRY_CAST(quantity AS FLOAT) IS NULL;
```

18.8 Détecter les valeurs aberrantes à l'aide de l'IQR (Interquartile Range)

```
WITH Quartiles AS (  
  SELECT  
    PERCENTILE_CONT(0.25) WITHIN GROUP (ORDER BY valeur) AS Q1,  
    PERCENTILE_CONT(0.75) WITHIN GROUP (ORDER BY valeur) AS Q3  
  FROM ma_table),  
  
  IQR_Calcul AS (  
    SELECT Q1, Q3,  
      (Q3 - Q1) AS IQR,  
      (Q1 - 1.5 * (Q3 - Q1)) AS lower_bound,  
      (Q3 + 1.5 * (Q3 - Q1)) AS upper_bound  
    FROM Quartiles)  
  
  SELECT t.*  
  FROM ma_table t  
  JOIN IQR_Calcul iqr  
  ON t.valeur < iqr.lower_bound OR t.valeur > iqr.upper_bound;
```

18.9 Vérifier le format des adresses e-mail

```
SELECT *  
FROM customers  
WHERE email NOT LIKE "%_@__%.__%";
```

18.10 Valider les valeurs de la colonne status

```
SELECT DISTINCT status  
FROM orders  
WHERE status NOT IN ("Completed", "Pending", "Cancelled");
```

19 Bonnes pratiques

Voyons dans cette section des bonnes pratiques à garder en tête :

1. **Commencez chaque nom de colonne sur une nouvelle ligne** : Cela assure la clarté et la lisibilité, permettant de commenter facilement une colonne.
2. **Utilisez des alias descriptifs et compréhensibles** : Cette pratique améliore la lisibilité des requêtes et aide les autres à comprendre votre code plus facilement.
3. **Évitez d'utiliser SELECT *** dans le cas de grosse BD : Spécifiez explicitement les colonnes dont vous avez besoin plutôt que d'utiliser *. L'étoile peut ralentir la performance des requêtes.
4. **Évitez les sous-requêtes si possible** : Utilisez des JOINS pour de meilleures performances.
5. **Gérez les valeurs NULL** : Les valeurs NULL peuvent créer des comportements inattendus dans vos requêtes SQL. Utilisez des fonctions comme COALESCE ou NULLIF pour les gérer.
6. **Utilisez des commentaires** : C'est une pratique de code à adopter le plus tôt possible. Pour commenter son code en SQL, on utilisera : "--" avant la phrase de commentaire (deux tirets du six collés).
7. **Commencez chaque colonne par une virgule** : Au cas où vous commenteriez amount, votre code fonctionnera sans erreur.
8. **Utilisez WHERE TRUE** : Au cas où vous commenteriez status = 'completed', votre code fonctionnera sans erreur.
9. **Organisez votre code avec des CTEs** : La clause WITH rend le code lisible, car la requête principale n'est pas interrompue par les sous-requêtes.
10. **Supprimez les clauses ORDER BY inutiles** : La clause ORDER BY ajoute de la complexité et ralentit l'exécution, surtout quand vous travaillez avec de gros datasets. Utilisez-la si nécessaire !

20 Sources

Ce document s'appuie sur la documentation du langage, un grand nombre de cours présent sur Internet, sur mes lectures et tous les posts LinkedIn que j'ai pu trouver dans ce domaine.

Les sites :

- La doc du langage : <https://sql.sh/>
- Site de cours lié à la documentation : <https://sql.sh/2143-cours-sql-pdf>
- Super cours de SQL : <https://enseignement.alexandre-mesle.com/sql/sql.pdf>
- SQL Tutorial - Full Database Course for Beginners : <https://www.youtube.com/watch?v=HXV3zeQKqGY>
- <https://mode.com/sql-tutorial>
- <https://generation-prepa.com/wpcontent/uploads/2021/12/SQL.pdf>
- <https://learnsql.com/cookbook/how-to-get-the-year-and-the-month-from-a-date-in-mysql/>
- Cours de leetcode : <https://leetcode.com/explore/featured/card/sql-language/>
- https://webusers.i3s.unice.fr/~rueher/Cours/BD/DocPostgresSQL9-5_HTML/datatype.html
- TOP-70 Most Important SQL Queries in 2023 : <https://bytescout.com/blog/20-important-sql-queries.html>
- <https://learnsql.fr/blog/qu-est-ce-qu-une-expression-de-table-commune-cte-en-sql/>
- <https://www.datacamp.com/fr/tutorial/sql-pivot>
- <https://aws.amazon.com/fr/compare/the-difference-between-sql-and-mysql/#:~:text=MySQL%20et%20SQL%20Server%20sont,columnstore%20%C2%BB%20pour%20optimiser%20les%20performances.>
- [https://data.sigea.educagri.fr/download/sigea/supports/PostGIS/distance/perfectionnement/M04_complement_SQL/co/30_types_jointure.html#:~:text=Une%20jointure%20crois%C3%A9e%20\(cross%20join,toutes%20les%20colonnes%20de%20t2.](https://data.sigea.educagri.fr/download/sigea/supports/PostGIS/distance/perfectionnement/M04_complement_SQL/co/30_types_jointure.html#:~:text=Une%20jointure%20crois%C3%A9e%20(cross%20join,toutes%20les%20colonnes%20de%20t2.)
- Comprendre le fenêtrage : <https://datajourney.io/les-fonctions-de-fenetrage-en-sql.html>

Super site pour le Regex :

- <https://regexlearn.com/fr>
- <https://datascientest.com/regex-tout-savoir>
- <https://regex101.com/>
- <https://community.alteryx.com/t5/Blog/Pense-bete-RegEx/ba-p/1239471>

Super sites pour apprendre :

- W3School : <https://www.w3schools.com/sql/>
- Geeksforgeeks : <https://www.geeksforgeeks.org/sql-tutorial/>
- SQL Bolt : <https://sqlbolt.com/>
- SQL Zoo : https://sqlzoo.net/wiki/SQL_Tutorial

- Select Star : <https://selectstarsql.com/>
- SQLTutorial : <https://www.sqltutorial.org/>
- Data Lemur : <https://datalemur.com/>

Les plateformes d'entraînement, à faire dans l'ordre :

- <https://www.sql-practice.com/>
- <https://leetcode.com/studyplan/top-sql-50/>
- <https://www.sqlnoir.com/>
- <https://www.kaggle.com/code/dillonmyrick/sql-beginner-to-advanced-with-practical-examples>

Faire du SQL avec Python :

- <https://docs.python.org/3/library/sqlite3.html>

Mes sources LinkedIn et Newsletters :

- <https://www.linkedin.com/in/gael-penessot/>
- <https://www.linkedin.com/in/fphuongnguyen/>
- <https://www.linkedin.com/in/fphuongnguyen/>

La section Bonnes pratiques provient de ce lien : https://www.linkedin.com/posts/fphuongnguyen_10-tips-de-pro-pour-un-code-sql-impeccablepdf-ugcPost-7304754235194159105-xP0u?utm_source=share&utm_medium=member_desktop&rcm=ACoAADIZc5QB1CXXmYmgtD6_ZpOS4ATZsuRulXk