Bases De Données Avancées

Gebril Amor ;)

SQL (Structured Query Language)

SQL est le langage standard pour gérer et manipuler les bases de données.

Opérations de Base

a. Créer une Table

```
sql
CREATE TABLE nom_table (
    colonne1 type_donnee,
    colonne2 type_donnee,
    ...
);
```

b. Insérer des Données

```
sql
INSERT INTO nom_table (colonne1, colonne2)
VALUES (valeur1, valeur2);
```

c. Sélectionner des Données

```
sql
SELECT colonne1, colonne2
FROM nom_table
WHERE condition;
```

d. Mettre à Jour des Données

```
sql
UPDATE nom_table
SET colonne1 = valeur1
WHERE condition;
```

e. Supprimer des Données

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

3. Clauses Courantes

- **WHERE** : Filtre les enregistrements selon des conditions.
- **ORDER BY** : Trie le jeu de résultats.
- **GROUP BY** : Regroupe les lignes ayant les mêmes valeurs.
- **HAVING** : Filtre les groupes selon des conditions.
- **LIMIT** : Restreint le nombre de lignes retournées.

4. Jointures

- **INNER JOIN** : Combine les lignes de deux tables où il y a des correspondances.
- **LEFT JOIN** : Inclut toutes les lignes de la table de gauche et les lignes correspondantes de la table de droite.
- **RIGHT JOIN** : Inclut toutes les lignes de la table de droite et les lignes correspondantes de la table de gauche.
- **FULL JOIN** : Combine les résultats des jointures LEFT et RIGHT.

5. Fonctions Agrégées

- **COUNT()** : Compte le nombre de lignes.
- **SUM()** : Additionne des valeurs.
- **AVG()** : Calcule la valeur moyenne.
- **MAX()/MIN()** : Trouve la valeur maximale ou minimale.

Voici des exemples concrets pour illustrer les clauses **GROUP BY**, **COUNT**, **HAVING**, et **ORDER BY** en SQL, avec les résultats attendus.

Supposons que nous ayons une table `Ventes` :

Produit	Vendeur	Montant
A	Jean	100
A	Marie	150
B	Jean	200
B	Marie	250
A	Jean	300
C	Pierre	400

1. GROUP BY avec SUM

Nous voulons obtenir le montant total des ventes pour chaque produit.

```
sql
SELECT Produit, SUM(Montant) AS TotalVentes
FROM Ventes
GROUP BY Produit;
```

Résultat :

Produit	TotalVentes
A	550
B	450
C	400

Explication: La clause `GROUP BY Produit` regroupe les lignes par produit, et `SUM(Montant)` calcule le total des montants de vente pour chaque produit.

2. COUNT avec GROUP BY

Nous voulons connaître le nombre de ventes réalisées pour chaque produit.

```sql
SELECT Produit, COUNT(*) AS NombreVentes
FROM Ventes
GROUP BY Produit;
```

Résultat :

Produit	NombreVentes
A	3
B	2
C	1

Explication: `COUNT(*)` compte le nombre de lignes pour chaque produit, montrant combien de ventes ont été enregistrées pour chaque produit.

3. HAVING pour Filtrer les Groupes

Nous voulons afficher uniquement les produits qui ont un nombre de ventes supérieur à 1.

```sql
SELECT Produit, COUNT(*) AS NombreVentes
FROM Ventes
GROUP BY Produit
HAVING COUNT(*) > 1;
```

Résultat :

Produit	NombreVentes
A	3
B	2

Explication: `HAVING COUNT(*) > 1` filtre les groupes pour n'afficher que les produits ayant plus d'une vente.

4. ORDER BY pour Trier les Résultats

Nous voulons afficher les produits avec leur nombre de ventes, triés par nombre de ventes en ordre décroissant.

```sql
SELECT Produit, COUNT(*) AS NombreVentes
FROM Ventes
GROUP BY Produit
ORDER BY NombreVentes DESC;
```

Résultat :

Produit NombreVentes

A 3

B 2

C 1

Explication: `ORDER BY NombreVentes DESC` trie les résultats en fonction de `NombreVentes` du plus élevé au plus bas.

Exemple Complet avec GROUP BY, COUNT, HAVING et ORDER BY

Nous voulons obtenir les produits ayant plus de 1 vente, avec le nombre de ventes trié en ordre décroissant.

```
sql
SELECT Produit, COUNT(*) AS NombreVentes
FROM Ventes
GROUP BY Produit
HAVING COUNT(*) > 1
ORDER BY NombreVentes DESC;
```

Résultat :

Produit NombreVentes

A 3

B 2

Explication: Ce résultat montre uniquement les produits ayant plus d'une vente, triés par nombre de ventes décroissant.

Sous-requête

Une **sous-requête** (ou requête imbriquée) est une requête SQL qui est intégrée dans une autre requête SQL. Les sous-requêtes permettent d'effectuer des opérations qui nécessitent plusieurs étapes ou qui dépendent des résultats d'une autre requête. Elles peuvent être utilisées dans plusieurs clauses, telles que `SELECT`, `FROM`, `WHERE`, et `HAVING`.

Types de Sous-requêtes

1. **Sous-requête à une seule ligne** : Renvoie une seule valeur (une ligne et une colonne). Elle peut être utilisée avec des opérateurs de comparaison comme `=`, `>`, `<`, etc.
2. **Sous-requête à plusieurs lignes** : Renvoie plusieurs valeurs (plusieurs lignes). Elle peut être utilisée avec des opérateurs comme `IN`, `ANY`, ou `ALL`.
3. **Sous-requête corrélée** : Une sous-requête qui fait référence à des colonnes de la requête externe. Elle est exécutée une fois pour chaque ligne traitée par la requête externe.

Exemples

Exemple 1 : Sous-requête à une seule ligne

Supposons que nous ayons une table `Produits` et que nous voulons trouver les produits dont le prix est supérieur au prix moyen.

Table Produits :

ProduitID	NomProduit	Prix
1	Produit A	100
2	Produit B	150
3	Produit C	200
4	Produit D	250

```
sql
SELECT NomProduit
FROM Produits
WHERE Prix > (SELECT AVG(Prix) FROM Produits);
```

Résultat :

NomProduit
Produit C
Produit D

Explication : La sous-requête `(SELECT AVG(Prix) FROM Produits)` calcule le prix moyen de tous les produits. La requête externe sélectionne les `NomProduit` dont le prix est supérieur à cette moyenne.

Exemple 2 : Sous-requête à plusieurs lignes

Si nous voulons trouver les clients ayant passé des commandes dont le montant total dépasse 1000, nous pouvons utiliser :

Table Clients :

ClientID NomClient

1 Client A

2 Client B

3 Client C

Table Commandes :

CommandeID ClientID MontantTotal

1 1 1500

2 1 800

3 2 1200

4 3 300

```
sql
SELECT NomClient
FROM Clients
WHERE ClientID IN (
    SELECT ClientID
    FROM Commandes
    WHERE MontantTotal > 1000
);
```

Résultat :

NomClient

Client A

Client B

Explication : La sous-requête sélectionne les `ClientID` dans la table `Commandes` où le `MontantTotal` est supérieur à 1000. La requête externe récupère les noms des clients correspondants.

Exemple 3 : Sous-requête corrélée

Pour trouver les employés qui gagnent plus que le salaire moyen dans leur département, nous pouvons écrire :

Table Employés :

EmployéID	NomEmployé	Salaire	DépartementID
1	Alice	3000	1
2	Bob	4000	1
3	Carol	2500	2
4	David	3500	2

```
sql
SELECT NomEmployé, Salaire
FROM Employés e1
WHERE Salaire > (
    SELECT AVG(Salaire)
    FROM Employés e2
    WHERE e1.DépartementID = e2.DépartementID
);
```

Résultat :

NomEmployé	Salaire
Bob	4000
David	3500

Explication : La sous-requête calcule le salaire moyen pour le département de chaque employé évalué dans la requête externe (`e1.DépartementID = e2.DépartementID`). La requête externe sélectionne les employés dont le salaire est supérieur à celui de leur département.

Jointure

Une **jointure** est une clause utilisée pour combiner des enregistrements de deux ou plusieurs tables basées sur des colonnes liées entre elles. Les jointures sont essentielles pour interroger des données réparties sur plusieurs tables dans une base de données relationnelle. Voici un aperçu des principaux types de jointures :

Types de Jointures

1. **INNER JOIN** :
   - Récupère les enregistrements qui ont des valeurs correspondantes dans les deux tables.
   - Seules les lignes qui satisfont la condition de jointure sont incluses dans le résultat.

   Exemple : Supposons que nous ayons deux tables : `Clients` et `Commandes`.

   Table Clients :

   ClientID NomClient

   1 Alice

   2 Bob

   3 Carol

   Table Commandes :

   CommandeID ClientID MontantTotal

   1 1 100

   2 1 150

   3 2 200

   4 4 250

```
sql
SELECT Clients.NomClient, Commandes.MontantTotal
FROM Clients
INNER JOIN Commandes ON Clients.ClientID = Commandes.ClientID;
```

   Résultat :

   NomClient MontantTotal

   Alice 100

NomClient MontantTotal

Alice 150

Bob 200

Explication : Cette requête récupère les noms et les montants des commandes des clients qui ont passé des commandes.

2. **LEFT JOIN (ou LEFT OUTER JOIN)** :
   - Récupère tous les enregistrements de la table de gauche et les enregistrements correspondants de la table de droite.
   - S'il n'y a pas de correspondance, des valeurs NULL sont retournées pour les colonnes de la table de droite.

```
sql
SELECT Clients.NomClient, Commandes.MontantTotal
FROM Clients
LEFT JOIN Commandes ON Clients.ClientID = Commandes.ClientID;
```

Résultat :

NomClient MontantTotal

Alice 100

Alice 150

Bob 200

Carol NULL

Explication : Tous les clients sont listés, mais Carol n'a pas de commandes, donc son `MontantTotal` est NULL.

3. **RIGHT JOIN (ou RIGHT OUTER JOIN)** :
   - Récupère tous les enregistrements de la table de droite et les enregistrements correspondants de la table de gauche.
   - S'il n'y a pas de correspondance, des valeurs NULL sont retournées pour les colonnes de la table de gauche.

```
sql
SELECT Clients.NomClient, Commandes.MontantTotal
FROM Clients
RIGHT JOIN Commandes ON Clients.ClientID = Commandes.ClientID;
```

Résultat :

NomClient MontantTotal

NomClient MontantTotal

Alice 100

Alice 150

Bob 200

NULL 250

Explication : Toutes les commandes sont listées, et puisque la CommandeID 4 n'a pas de client correspondant, `NomClient` est NULL.

4. **FULL OUTER JOIN** :
   - Combine les résultats des jointures LEFT et RIGHT.
   - Récupère tous les enregistrements des deux tables, en remplissant avec des NULL là où il n'y a pas de correspondance.

```
sql
SELECT Clients.NomClient, Commandes.MontantTotal
FROM Clients
FULL OUTER JOIN Commandes ON Clients.ClientID = Commandes.ClientID;
```

Résultat :

NomClient MontantTotal

Alice 100

Alice 150

Bob 200

Carol NULL

NULL 250

Explication : Tous les clients et toutes les commandes sont listés, avec des NULL là où il n'y a pas de correspondances.

5. **CROSS JOIN** :
   - Produit un produit cartésien des deux tables.
   - Chaque ligne de la première table est combinée avec chaque ligne de la seconde table.

```
sql
SELECT Clients.NomClient, Commandes.MontantTotal
FROM Clients
CROSS JOIN Commandes;
```

Résultat (en supposant 3 clients et 4 commandes) :

NomClient	MontantTotal
Alice	100
Alice	150
Alice	200
Alice	250
Bob	100
Bob	150
Bob	200
Bob	250
Carol	100
Carol	150
Carol	200
Carol	250

Explication : Chaque client est associé à chaque commande.

Explication des Objets SQL : Vue, Séquence et Synonyme

Examinons les objets SQL **Vue**, **Séquence**, et **Synonyme**, chacun avec un exemple pour une meilleure compréhension.

1. Vue

Une **vue** est une table virtuelle basée sur une requête. Elle simplifie l'accès à des requêtes complexes et peut restreindre l'accès aux données en exposant seulement certaines colonnes ou lignes.

Exemple de Table

Supposons que nous ayons une table appelée `Employes`:

```sql
CREATE TABLE Employes (
    EmployeID INT PRIMARY KEY,
    Prenom VARCHAR(50),
    Nom VARCHAR(50),
    Departement VARCHAR(50),
    Salaire DECIMAL(10, 2),
    DateEmbauche DATE
);
```

Création d'une Vue

Créons une vue qui montre seulement les employés du département IT:

```sql
CREATE VIEW IT_Employes AS
SELECT Prenom, Nom, Salaire
FROM Employes
WHERE Departement = 'IT';
```

Utilisation de la Vue

Vous pouvez maintenant interroger `IT_Employes` comme une table :

```sql
SELECT * FROM IT_Employes;
```

Cela renverra uniquement le `Prenom`, `Nom`, et `Salaire` des employés du département IT, sans exposer toutes les données de la table `Employes`.

2. Séquence

Une **séquence** est un objet qui génère des nombres uniques et séquentiels, souvent utilisé pour les valeurs de clés primaires en auto-incrémentation.

Création d'une Séquence

Créons une séquence pour générer des identifiants uniques pour les employés :

```sql
CREATE SEQUENCE EmployeID_Seq
START WITH 1
INCREMENT BY 1;
```

Utilisation de la Séquence

Lors de l'insertion d'un nouvel employé, utilisez `NEXTVAL` pour obtenir le prochain nombre dans la séquence :

```sql
INSERT INTO Employes (EmployeID, Prenom, Nom, Departement, Salaire, DateEmbauche)
VALUES (NEXTVAL('EmployeID_Seq'), 'Alice', 'Vert', 'Finance', 55000.00, '2023-02-10');
```

Chaque appel à `NEXTVAL('EmployeID_Seq')` génère le prochain nombre unique, garantissant des valeurs `EmployeID` uniques.

3. Synonyme

Un **synonyme** est un alias pour un autre objet de base de données. Il simplifie l'accès aux objets ayant des noms longs ou résidant dans différents schémas.

Exemple de Synonyme

Supposons que nous ayons une table appelée `RH.Employes` dans un autre schéma, et que nous souhaitons y accéder sans toujours spécifier `RH.` devant.

```sql
CREATE SYNONYM Emp FOR RH.Employes;
```

Utilisation du Synonyme

Maintenant, au lieu de requêter `RH.Employes`, nous pouvons simplement utiliser `Emp` :

```sql
SELECT * FROM Emp;
```

Cette requête récupérera les données de la table `RH.Employes` via le synonyme `Emp`, facilitant son utilisation dans les requêtes.

Ces objets SQL—vues, séquences et synonymes—améliorent la gestion des données en simplifiant l'accès, en automatisant la génération de clés uniques, et en créant des alias pour une meilleure lisibilité.

4. Index

En SQL, un **index** est un objet de base de données qui améliore la vitesse des opérations de recherche de données sur une table en permettant une consultation rapide des enregistrements. Il fonctionne de manière similaire à un index dans un livre, où l'on peut localiser des informations sans parcourir chaque page. En créant un index sur une colonne ou un ensemble de colonnes, les bases de données SQL peuvent retrouver rapidement des lignes sans parcourir toute la table.

Points Clés
- **But** : Accélère la récupération des données.
- **Types d'index** :
  - **Index de Clé Primaire** : Créé automatiquement sur les colonnes de clé primaire pour garantir l'unicité.
  - **Index Unique** : Garantit que toutes les valeurs d'une colonne sont uniques.
  - **Index Composite** : Un index sur plusieurs colonnes, utile pour les requêtes filtrant par plusieurs colonnes.
  - **Index en Texte Intégral** : Utilisé pour la recherche de texte dans de grands champs textuels.
- **Compromis** : Les index améliorent la vitesse de lecture, mais peuvent ralentir les opérations d'écriture (`INSERT`, `UPDATE`, `DELETE`) car ils nécessitent une mise à jour.

Syntaxe
Pour créer un index, utilisez :
```sql
CREATE INDEX nom_index ON nom_table (nom_colonne);
```

Exemple
Supposons que nous ayons une table `Clients` :
```sql
CREATE TABLE Clients (
    ClientID INT PRIMARY KEY,
    Nom VARCHAR(100),
    Email VARCHAR(100),
    Ville VARCHAR(50)
);
```
Si nous interrogeons souvent les clients par `Ville`, nous pouvons créer un index sur la colonne `Ville` pour accélérer ces recherches :
```sql
CREATE INDEX idx_ville ON Clients (Ville);
```

Utilisation de l'Index
Maintenant, lorsque vous exécutez une requête comme celle-ci :
```sql
SELECT * FROM Clients WHERE Ville = 'Paris';
```
La base de données utilisera `idx_ville` pour localiser rapidement les lignes ayant `Ville = 'Paris'`, ce qui améliore considérablement les performances, surtout dans les grandes tables.

Note sur la Maintenance des Index
Bien que les index optimisent les performances de lecture, ils nécessitent une maintenance lors des opérations d'écriture. Chaque fois que des données sont insérées, mises à jour ou supprimées dans la table, les index doivent être mis à jour, ce qui peut légèrement impacter les performances de ces opérations. Il est donc recommandé de créer des index uniquement sur les colonnes fréquemment utilisées dans les conditions de recherche ou les jointures.