SQL (légèrement) avancé ...

Jointures Variées (INNER JOIN, LEFT JOIN, RIGHT JOIN, CROSS JOIN)

```sql
-- INNER JOIN récupère les enregistrements correspondants dans les deux tables.
SELECT Client.nom, Commande.date_commande
FROM Client
INNER JOIN Commande ON Client.id_client = Commande.id_client;

-- LEFT JOIN inclut tous les enregistrements de la table de gauche et les correspondants de la table de droite.
SELECT Client.nom, Commande.date_commande
FROM Client
LEFT JOIN Commande ON Client.id_client = Commande.id_client;

-- RIGHT JOIN n'est pas supporté dans toutes les bases de données, donc vérifiez la compatibilité.
-- Elle inclut tous les enregistrements de la table de droite et les correspondants de la table de gauche.
-- Si votre système de gestion de base de données ne supporte pas RIGHT JOIN, utilisez LEFT JOIN avec l'ordre des tables inversé.
SELECT Commande.date_commande, Client.nom
FROM Commande
LEFT JOIN Client ON Commande.id_client = Client.id_client;

-- CROSS JOIN produit le produit cartésien des deux tables.
SELECT Client.nom, Plat.nom_plat
FROM Client
CROSS JOIN Plat;
```

Fonctions SQL Avancées (GROUP BY, HAVING, DISTINCT)

```sql
-- Utilisation de GROUP BY avec HAVING pour filtrer les résultats.
SELECT Categorie.nom_categorie, COUNT(*) AS Nombre_Plats
FROM Plat
JOIN Categorie ON Plat.id_categorie = Categorie.id_categorie
GROUP BY Categorie.nom_categorie
HAVING COUNT(*) > 5;

-- Utilisation de DISTINCT pour éviter les doublons.
SELECT DISTINCT Client.prenom
FROM Client;
```

Fonctions de Date et d'Heure

```sql
-- Extrait l'année de la date de commande.
SELECT id_commande, YEAR(date_commande) AS Annee_Commande
FROM Commande;

-- Calcule la différence en jours entre deux dates.
SELECT DATEDIFF(CURDATE(), date_commande) AS Jours_Depuis_Commande
FROM Commande;
```

Utilisation de Fonctions d'Aggrégation (SUM, AVG, MIN, MAX)

```sql
-- Calcule le total des ventes, la commande moyenne, le montant maximum et
minimum des commandes.
SELECT SUM(total) AS Total_Ventes, AVG(total) AS Moyenne_Commande,
MIN(total) AS Commande_Min, MAX(total) AS Commande_Max
FROM Commande;
```

Sous-requêtes et Opérations sur les Sets (IN, NOT IN, EXISTS)

```sql
-- Utilise IN pour sélectionner les clients ayant passé une commande de
plus de 100 euros.
SELECT nom, prenom FROM Client
WHERE id_client IN (SELECT id_client FROM Commande WHERE total > 100);

-- Utilise EXISTS pour vérifier l'existence de certaines commandes.
SELECT nom FROM Client c
WHERE EXISTS (SELECT 1 FROM Commande co WHERE co.id_client = c.id_client
AND total > 150);
```

Insertions, Mises à jour, et Suppressions de données

```sql
-- Insertion d'un nouveau client.
INSERT INTO Client (id_client, nom, prenom, email) VALUES (101, 'Dupont',
'Jean', 'jean.dupont@example.com');

-- Mise à jour de l'email d'un client.
UPDATE Client SET email = 'nouveau.email@example.com' WHERE id_client =
101;

-- Suppression d'un client.
DELETE FROM Client WHERE id_client = 101;
```

Transactions (START TRANSACTION, COMMIT, ROLLBACK)

Les transactions en SQL sont des séquences d'opérations de gestion de base de données qui sont traitées de manière logique et indivisible. Elles sont essentielles pour maintenir l'intégrité des données, en particulier dans les environnements où plusieurs utilisateurs ou applications accèdent et modifient simultanément la base de données. Une transaction en SQL commence par une commande de démarrage et se termine par un commit ou un rollback.

Principe des Transactions

1. **Atomicité** : Chaque transaction est atomique, ce qui signifie qu'elle est indivisible. Toutes les opérations au sein de la transaction sont effectuées avec succès, ou aucune n'est appliquée. Si une opération échoue, toutes les modifications précédentes dans la transaction sont annulées (rollback).
2. **Cohérence** : Une transaction transforme la base de données d'un état valide à un autre état valide, en préservant l'intégrité des données. Toutes les règles d'intégrité doivent être respectées.
3. **Isolation** : Chaque transaction doit être isolée des autres transactions. Les modifications effectuées dans une transaction ne doivent pas être visibles par les autres transactions avant que la transaction ne soit terminée (commit).
4. **Durabilité** : Une fois qu'une transaction a été validée (commit), les modifications qu'elle a introduites dans la base de données doivent être permanentes, même en cas de panne du système.

Exemple de Transaction en SQL

Pour illustrer comment les transactions fonctionnent en pratique, imaginons une base de données de gestion de comptes bancaires où les transactions sont cruciales pour assurer que tous les transferts de fonds sont effectués de manière sécurisée et cohérente.

```
-- Démarrage de la transaction
START TRANSACTION;

-- Tentative de transfert de 100 euros du compte 001 vers le compte 002
-- Débit du compte 001
UPDATE Comptes SET solde = solde - 100 WHERE numero_compte = '001';

-- Crédit du compte 002
UPDATE Comptes SET solde = solde + 100 WHERE numero_compte = '002';

-- Vérification que le compte débiteur a suffisamment de fonds (ne doit pas
être négatif)
SELECT solde INTO @solde FROM Comptes WHERE numero_compte = '001';
IF @solde < 0 THEN
    -- Si le solde est insuffisant, annulation de la transaction
    ROLLBACK;
ELSE
    -- Si tout est en règle, validation de la transaction
    COMMIT;
END IF;
```

Commentaires sur l'Exemple

- **START TRANSACTION** démarre la transaction.
- Les commandes **UPDATE** sont utilisées pour transférer les fonds entre les comptes.
- La commande **SELECT INTO** récupère le solde après le débit pour vérifier si le compte ne passe pas en négatif.
- **IF** permet de tester si le solde est inférieur à zéro. Si c'est le cas, la transaction est annulée avec **ROLLBACK**; sinon, elle est validée avec **COMMIT**.
- **ROLLBACK** annule toutes les modifications effectuées dans la transaction.
- **COMMIT** applique toutes les modifications de manière permanente dans la base de données.
- **@solde** cré une variable

    1. Portée et Durée de vie : Les variables utilisateur comme @solde ont une portée limitée à la session dans laquelle elles sont définies. Elles perdent leur valeur une fois la session terminée.
    2. Sécurité des transactions : L'utilisation de cette variable dans le contexte d'une transaction bancaire est cruciale pour assurer l'intégrité des opérations financières et prévenir les erreurs telles que les découverts non autorisés.

Cet exemple montre l'utilisation essentielle des transactions pour gérer des opérations financières complexes de manière sécurisée et fiable, garantissant ainsi l'intégrité et la cohérence des données dans des systèmes transactionnels critiques.