HIS

Administration des Bases de Données Avancées

Cours de soutien pour le module BDA

BAHRI Somia
23/06/2024

Table des matières

Introduction

L'optimisation des requêtes SQL est un aspect crucial des systèmes de gestion de bases de données (SGBD), visant à améliorer l'efficacité et la performance des opérations de traitement des données. Ce cours explore les différentes méthodes d'optimisation, notamment celles basées sur les arbres algébriques, les triggers et les contraintes d'intégrité. En nous concentrant particulièrement sur les arbres algébriques, nous verrons comment la restructuration et la réécriture des expressions algébriques peuvent réduire le coût d'exécution des requêtes. Les propriétés des opérateurs algébriques, telles que la commutativité et l'associativité, jouent un rôle clé dans cette optimisation, permettant de minimiser les ressources nécessaires pour manipuler de grandes quantités de données. Ce cours fournira également des exemples concrets et des exercices pour illustrer les concepts théoriques et leur application pratique.

SQL et PL/SQL

SQL (Structured Query Language) et PL/SQL (Procedural Language/SQL) sont deux langages utilisés dans le contexte des bases de données relationnelles, mais ils ont des objectifs et des fonctionnalités différents.

SQL

SQL est un langage de requête utilisé pour gérer et manipuler des bases de données relationnelles. Il est utilisé par diverses bases de données comme MySQL, PostgreSQL, Microsoft SQL Server, Oracle Database, etc.

C’est un langage de requête ensembliste utilisé pour manipuler et interroger les bases de données relationnelles. Voici ses aspects clés :

- Ensemble de requêtes distinctes :Chaque instruction SQL est une requête distincte qui spécifie les opérations à effectuer sur les données (par exemple SELECT, INSERT, UPDATE, DELETE).

- Langage de quatrième génération : SQL est souvent considéré comme un langage de 4ème génération car il permet de décrire le résultat souhaité sans spécifier comment accéder aux données ni les manipuler de manière détaillée.

- Encapsulation dans un Langage Hôte :SQL est intégré dans des langages hôtes de 3ème génération comme Java, C, Python, etc., pour obtenir des résultats spécifiques en interagissant avec une base de données.

Fonctionnalités :

- Requêtes: Permet de récupérer des données avec des commandes comme SELECT.
- Manipulation de données: Permet d'insérer (INSERT), mettre à jour (UPDATE), et supprimer (DELETE) des données.
- Définition de données: Permet de définir la structure de la base de données avec des commandes comme CREATE, ALTER, et DROP.
- Contrôle d'accès: Permet de gérer les permissions avec des commandes comme GRANT et REVOKE.

PL/SQL

PL/SQL est un langage procédural étendu au SQL, spécifiquement pour Oracle Database. Il est utilisé spécifiquement dans Oracle Database, bien que d'autres SGBD (comme PostgreSQL avec PL/pgSQL) aient des langages procéduraux similaires.

PL/SQL ajoute des fonctionnalités procédurales au SQL, permettant une programmation plus avancée et robuste pour la gestion de bases de données :

- Bloc PL/SQL :Tout programme PL/SQL est structuré entre les mots-clés `BEGIN` et `END`, formant ainsi un bloc de code exécutable.

- Gestion des Transactions : Adapté pour la gestion de transactions complexes où plusieurs opérations doivent être exécutées de manière atomique (tout ou rien).

- Architecture Client-Serveur :Utilisé dans une architecture client-serveur typique des systèmes de gestion de bases de données, où les clients envoient des requêtes SQL au serveur qui exécute le code PL/SQL et renvoie les résultats.

- Il permet également la gestion des exceptions, facilitant le contrôle des erreurs et la prise de décisions en fonction des conditions rencontrées lors de l'exécution des blocs de code.

En résumé, PL/SQL enrichit SQL en permettant une programmation procédurale plus avancée, adaptée aux besoins de manipulation de données complexes et de gestion des transactions dans les environnements de base de données Oracle.

Fonctionnalités:

- Procédures stockées: Permet d'écrire des blocs de code qui peuvent être stockés et exécutés dans la base de données.
- Fonctions: Permet de créer des fonctions définies par l'utilisateur pour effectuer des opérations complexes.
- Boucles et conditions: Supporte les structures de contrôle comme les boucles (FOR, WHILE) et les conditions (IF-THEN-ELSE).
- Gestion des exceptions: Permet de gérer les erreurs et exceptions de manière structurée.
- Paquets: Permet de grouper des procédures, des fonctions, des variables, et d'autres éléments en unités modulaires.

Différences principales entre SQL et PL/SQL

1. Nature du langage:

- SQL: Déclaratif, utilisé pour les requêtes et les opérations de manipulation des données.
- PL/SQL: Impératif, utilisé pour écrire des programmes complets avec des blocs de code, y compris les procédures et les fonctions.

2. Utilisation:

- SQL: Utilisé pour les opérations simples de récupération et de manipulation des données.
- PL/SQL: Utilisé pour les opérations complexes nécessitant une logique procédurale, comme les boucles, les conditions, et la gestion des exceptions.

3. Environnement d'exécution:

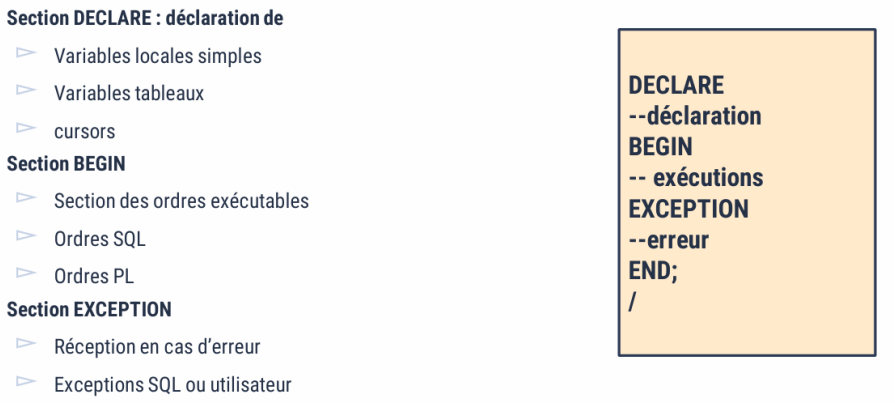
- SQL: Peut être exécuté sur n'importe quel SGBD relationnel.
- PL/SQL: Spécifique à Oracle Database, bien que d'autres bases de données aient leurs propres langages procéduraux similaires.

En résumé, SQL est utilisé pour les opérations de requête et de manipulation des données, tandis que PL/SQL est utilisé pour écrire des programmes complets avec une logique procédurale, spécifiquement dans le contexte des bases de données Oracle.

Le langage PL/SQL (Procedural Language/Structured Query Language) est une extension procédurale du SQL utilisé principalement dans les bases de données Oracle. Voici une explication détaillée de ses caractéristiques principales :

Prise de main PL/SQL

Declaration d'un bloc PL/SQL



Variables en PL/SQL

Les variables en PL/SQL permettent de stocker des valeurs temporaires ou des résultats intermédiaires lors de l'exécution des blocs de code. Voici une explication détaillée des différents types de variables et leurs utilisations :

Variables de Type SQL

1. Nombre : NUMBER(2);
Déclare une variable `nbr` de type `NUMBER` pouvant stocker des nombres entiers de 2 chiffres.

2. Chaîne de Caractères :VARCHAR2(30);
Déclare une variable `nom` de type 'VARCHAR2` pouvant stocker des chaînes de caractères jusqu`à 30 caractères de longueur.

3. Constante : minimum CONSTANT INTEGER := 5;
Déclare une constante `minimum` de type `INTEGER` avec une valeur constante initiale de 5. Cette valeur ne peut pas être modifiée pendant l'exécution du programme.

4. Nombre à Virgule Flottante :salaire NUMBER(8,2);

Déclare une variable `salaire` de type `NUMBER` qui peut stocker des nombres à virgule avec une précision totale de 8 chiffres et 2 chiffres après la virgule.

5. Nombre Non Nul : debut NUMBER NOT NULL;
Déclare une variable `debut` de type `NUMBER` qui ne peut pas être nulle (`NOT NULL`).

Variables de Type Booléen

1. Booléen : fin BOOLEAN;
Déclare une variable `fin` de type `BOOLEAN`, qui peut avoir les valeurs `TRUE`, `FALSE` ou `NULL`.

2. Booléen avec Valeur par Défaut : Reponse BOOLEAN DEFAULT TRUE;
Déclare une variable `Reponse` de type `BOOLEAN` avec une valeur par défaut de `TRUE`.

3. Initialisation Explicite du Booléen : ok BOOLEAN := TRUE;
Déclare une variable `ok` de type `BOOLEAN` initialisée explicitement à `TRUE`.

Variables Faisant Référence au Dictionnaire de Données

1. Référence à une Colonne : vsalaire employe.salaire%TYPE;
Déclare une variable `vsalaire` dont le type est déduit de la colonne `salaire` de la table `employe`.

2. Référence à une Ligne : vemploye employe%ROWTYPE;
Déclare une variable `vemploye` de type `employe%ROWTYPE`, qui est une structure contenant toutes les colonnes de la table `employe`.

3. Accès à une Colonne dans une Variable de Type `ROWTYPE` : vemploye.adresse;
Accède à la colonne `adresse` de la variable `vemploye`.

Exemples d'utilisation de ces variables en PL/SQL

```
DECLARE
    nbr NUMBER(2);
    nom VARCHAR2(30);
    minimum CONSTANT INTEGER := 5;
    salaire NUMBER(8,2);
    debut NUMBER NOT NULL;
    fin BOOLEAN;
    Reponse BOOLEAN DEFAULT TRUE;
    ok BOOLEAN := TRUE;
    vsalaire employe.salaire%TYPE;
    vemploye employe%ROWTYPE;
```

```
BEGIN
    -- Assignation de valeurs aux variables
    nbr := 10;
    nom := 'John Doe';
    salaire := 5000.50;
    debut := 1;
    fin := TRUE;
    vsalaire := 6000.75;

    -- Utilisation de la variable ROWTYPE
    vemploye.nom := 'Jane Smith';
    vemploye.salaire := 7000.00;
    vemploye.adresse := '123 Main St';

    -- Affichage des valeurs
    DBMS_OUTPUT.PUT_LINE('Nom: ' || nom);
    DBMS_OUTPUT.PUT_LINE('Salaire: ' || salaire);
    DBMS_OUTPUT.PUT_LINE('Employé: ' || vemploye.nom || ', Salaire: ' || vemploye.salaire)
END;
/
```

Ce bloc PL/SQL déclare différentes variables pour stocker des nombres, des chaînes de caractères, des constantes, des booléens et des structures de lignes. Il montre également comment assigner des valeurs à ces variables et comment accéder aux données structurées à l'intérieur de la variable `vemploye`.

Instructions PL/SQL

En PL/SQL, les instructions conditionnelles et les affectations sont essentielles pour contrôler le flux d'exécution du programme et affecter des valeurs aux variables. Voici une explication détaillée avec des exemples :

Affectation (`:=`)

L'opérateur d'affectation `:=` est utilisé pour assigner une valeur à une variable en PL/SQL.

```
DECLARE
    A NUMBER;
    B NUMBER := 10;
BEGIN
    A := B; -- A prend la valeur de B
    DBMS_OUTPUT.PUT_LINE('A: ' || A); -- Affiche : A: 10
END;
/
```

Dans cet exemple, la variable `A` prend la valeur de la variable `B` qui est égale à 10.

Structure Alternative ou Conditionnelle (`IF ... THEN ... ELSE ... END IF`)

Les structures conditionnelles permettent d'exécuter certaines instructions en fonction d'une condition.

```
DECLARE
    x NUMBER := 15;
BEGIN
    IF x > 10 THEN
        DBMS_OUTPUT.PUT_LINE('x est supérieur à 10');
    ELSE
        DBMS_OUTPUT.PUT_LINE('x est inférieur ou égal à 10');
    END IF;
END;
/
```

Dans cet exemple, si `x` est supérieur à 10, le programme affichera "x est supérieur à 10", sinon il affichera "x est inférieur ou égal à 10".

Structure `IF ... THEN ... ELSIF ... ELSE ... END IF`

Cette structure permet de tester plusieurs conditions de manière séquentielle.

```
DECLARE
    x NUMBER := 7;
BEGIN
    IF x > 10 THEN
        DBMS_OUTPUT.PUT_LINE('x est supérieur à 10');
    ELSIF x > 5 THEN
        DBMS_OUTPUT.PUT_LINE('x est supérieur à 5 mais inférieur ou égal à 10');
    ELSE
        DBMS_OUTPUT.PUT_LINE('x est inférieur ou égal à 5');
    END IF;
END;
/
```

Dans cet exemple, selon la valeur de `x`, le programme affichera le message correspondant à la première condition vérifiée.

Imbrication de Conditions

Il est possible d'imbriquer plusieurs structures `IF ... THEN ... ELSIF ... ELSE ... END IF` pour tester des conditions complexes.

```
DECLARE
    score NUMBER := 85;
BEGIN
    IF score >= 90 THEN
        DBMS_OUTPUT.PUT_LINE('Excellent');
    ELSIF score >= 80 THEN
        DBMS_OUTPUT.PUT_LINE('Très bien');
    ELSIF score >= 70 THEN
        DBMS_OUTPUT.PUT_LINE('Bien');
    ELSIF score >= 60 THEN
        DBMS_OUTPUT.PUT_LINE('Assez bien');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Insuffisant');
    END IF;
END;
/
```

Dans cet exemple, en fonction de la valeur de `score`, le programme affiche la mention correspondante selon les critères définis.

Les structures conditionnelles et les affectations en PL/SQL permettent de contrôler efficacement le flux d'exécution des programmes en fonction de diverses conditions. Elles sont essentielles pour automatiser des décisions dans les scripts SQL et PL/SQL, en fonction des données manipulées et des besoins spécifiques des applications.

Structure Alternative : CASE

En PL/SQL, la structure de contrôle CASE permet de simplifier le code lorsqu'on veut effectuer des actions différentes en fonction de la valeur d'une variable spécifique. Elle permet de choisir une action parmi plusieurs alternatives en fonction de la valeur d'une variable.

Supposons que nous voulons attribuer une mention en fonction de la note obtenue par un étudiant. Voici comment utiliser la structure CASE pour cela :

```
DECLARE
    note NUMBER := 75;
    mention VARCHAR2(20);
BEGIN
    CASE
        WHEN note >= 90 THEN mention := 'Excellent';
        WHEN note >= 80 THEN mention := 'Très bien';
        WHEN note >= 70 THEN mention := 'Bien';
        WHEN note >= 60 THEN mention := 'Assez bien';
        ELSE mention := 'Passable';
    END CASE;

    DBMS_OUTPUT.PUT_LINE('Note: ' || note || ', Mention: ' || mention);
END;
/
```

Dans cet exemple :
- Si `note` est supérieure ou égale à 90, la variable `mention` prend la valeur `'Excellent'`.
- Si `note` est entre 80 et 89, `mention` prend la valeur `'Très bien'`.
- Si `note` est entre 70 et 79, `mention` prend la valeur `'Bien'`.
- Si `note` est entre 60 et 69, `mention` prend la valeur `'Assez bien'`.
- Sinon (pour toutes les autres valeurs de `note`), `mention` prend la valeur `'Passable'`.

La sortie affichera la note et la mention correspondante en fonction de la valeur de `note`.

Boucle POUR (FOR loop)

La boucle FOR en PL/SQL est utilisée lorsque le nombre d'itérations est connu à l'avance, souvent en utilisant un indice numérique.

```
FOR indice IN borne1..borne2 LOOP
    instructions;
END LOOP;
```

- `indice` : C'est la variable d'itération numérique.
- `borne1` et `borne2` : Ce sont les bornes (valeurs numériques) entre lesquelles l'indice va itérer.
- `instructions` : Ce sont les instructions exécutées à chaque itération de la boucle.

```
DECLARE
    total NUMBER := 0;
BEGIN
    FOR i IN 1..5 LOOP
        total := total + i;
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('Total: ' || total);
END;
/
```

Dans cet exemple, la boucle FOR itère de 1 à 5. À chaque itération, elle ajoute la valeur de `i` à la variable `total`. La sortie affiche ensuite la somme totale.

Boucle TANT QUE (WHILE loop)

La boucle TANT QUE en PL/SQL est utilisée lorsque le nombre d'itérations n'est pas connu à l'avance, mais dépend d'une condition.

```
WHILE condition LOOP
    instructions;
END LOOP;
```

- `condition` : C'est une condition booléenne qui détermine si la boucle continue à s'exécuter.
- `instructions` : Ce sont les instructions exécutées à chaque itération tant que la condition est vraie.

Exemple :

```
DECLARE
    total NUMBER := 0;
    i NUMBER := 1;
BEGIN
    WHILE i <= 5 LOOP
        total := total + i;
        i := i + 1;
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('Total: ' || total);
END;
/
```

Dans cet exemple, la boucle WHILE continue à s'exécuter tant que `i` est inférieur ou égal à 5. À chaque itération, elle ajoute la valeur de `i` à `total` et incrémente `i` de 1.

Boucle RÉPÉTER (LOOP loop)

La boucle RÉPÉTER (ou boucle LOOP) en PL/SQL est utilisée lorsque vous devez exécuter des instructions au moins une fois, puis vérifier une condition pour déterminer si la boucle doit continuer.

```
LOOP
    instructions;
    EXIT WHEN condition;
END LOOP;
```

- `instructions` : Ce sont les instructions exécutées à chaque itération de la boucle.
- `EXIT WHEN condition` : C'est une condition qui, si elle est vraie à un moment donné, provoque la sortie immédiate de la boucle.

Exemple :

```
DECLARE
    total NUMBER := 0;
    i NUMBER := 1;
BEGIN
    LOOP
        total := total + i;
        i := i + 1;
        EXIT WHEN i > 5;
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('Total: ' || total);
END;
/
```

Dans cet exemple, la boucle LOOP ajoute la valeur de `i` à `total` à chaque itération jusqu'à ce que `i` dépasse 5. Une fois que `i` est supérieur à 5, la boucle se termine.

DBMS_OUTPUT.PUT_LINE

La commande `DBMS_OUTPUT.PUT_LINE` est utilisée pour afficher du texte ou des valeurs dans la fenêtre de sortie (console) lors de l'exécution d'un programme PL/SQL.

```
DBMS_OUTPUT.PUT_LINE(texte);
```

- `texte` : C'est le texte ou la valeur que vous souhaitez afficher.

Exemple :

```
BEGIN
    DBMS_OUTPUT.PUT_LINE('Hello, World!');
    DBMS_OUTPUT.PUT_LINE('2 + 2 = ' || (2 + 2));
END;
/
```

Dans cet exemple, deux lignes sont affichées :
- "Hello, World!" est affiché en tant que texte.
- "2 + 2 = 4" calcule l'expression arithmétique et affiche le résultat.

L'affichage de résultats intermédiaires en PL/SQL est souvent réalisé à l'aide du package `DBMS_OUTPUT`. Ce package permet d'enregistrer des messages dans une mémoire tampon côté serveur Oracle, qui sont ensuite affichés sur le poste client à la fin de l'exécution du bloc PL/SQL.

DBMS_OUTPUT

1. Activation de DBMS_OUTPUT :
Avant de pouvoir utiliser les fonctions de `DBMS_OUTPUT`, il est nécessaire d'activer la sortie avec la commande `DBMS_OUTPUT.ENABLE`.

```
SET SERVEROUTPUT ON; -- Activation de la sortie du serveur SQL*Plus
```

2. Affichage de messages :
Pour afficher un message dans la mémoire tampon, on utilise la fonction `DBMS_OUTPUT.PUT_LINE`.

```
DBMS_OUTPUT.PUT_LINE('Message à afficher');
```

Cette fonction enregistre le message dans la mémoire tampon du serveur Oracle.

3. Affichage des résultats :
À la fin de l'exécution du bloc PL/SQL, les messages enregistrés dans la mémoire tampon sont affichés sur le poste client. Il est important de noter que l'affichage des résultats dépend de la configuration du client SQL*Plus ou de l'outil utilisé pour l'exécution de PL/SQL.

Exemple d'utilisation :
Considérons un exemple simple où nous calculons la somme des nombres de 1 à 10 et affichons chaque étape du calcul à l'aide de `DBMS_OUTPUT`.

```
SET SERVEROUTPUT ON;

DECLARE
    total NUMBER := 0;
BEGIN
    FOR i IN 1..10 LOOP
        total := total + i;
        DBMS_OUTPUT.PUT_LINE('Total partiel après ' || i || ' itérations : ' || total);
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('Total final : ' || total);
END;
/
```

Dans cet exemple :
- La boucle FOR calcule la somme des nombres de 1 à 10.
- À chaque itération de la boucle, `DBMS_OUTPUT.PUT_LINE` est utilisé pour afficher le total partiel après chaque ajout.
- À la fin de la boucle, le total final est également affiché.

Lorsque ce bloc PL/SQL est exécuté, les messages enregistrés avec `DBMS_OUTPUT.PUT_LINE` sont stockés dans la mémoire tampon côté serveur. Après l'exécution du bloc, les messages sont envoyés au poste client et affichés dans la fenêtre de sortie SQL*Plus.

DBMS_OUTPUT.PUT

Cette fonction écrit une chaîne de caractères dans le tampon de sortie sans ajouter de saut de ligne.

```
DBMS_OUTPUT.PUT('Message sans saut de ligne ');
```

Exemple d'utilisation :

```
DECLARE
    nom VARCHAR2(50) := 'Jean';
    prenom VARCHAR2(50) := 'Dupont';
BEGIN
    DBMS_OUTPUT.PUT('Nom : ');
    DBMS_OUTPUT.PUT(nom);
    DBMS_OUTPUT.PUT(', Prénom : ');
    DBMS_OUTPUT.PUT(prenom);
END;
/
```

Ce code affichera :

```
Nom : Jean, Prénom : Dupont
```

DBMS_OUTPUT.NEW_LINE

Cette procédure insère explicitement un saut de ligne dans le tampon de sortie.

```
DBMS_OUTPUT.NEW_LINE;
```

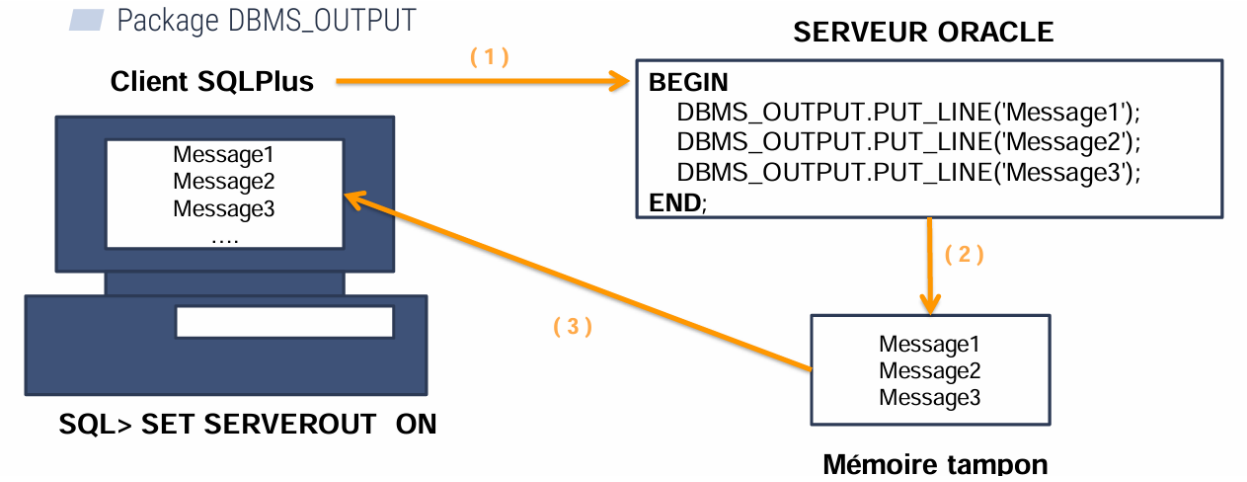
Exemple d'utilisation :

```
DECLARE
    tab_cars DBMS_OUTPUT.chararr;
    n INTEGER := 5;
BEGIN
    DBMS_OUTPUT.PUT_LINE('Affichage des premiers ' || n || ' caractères en ligne :');
    FOR i IN 1..n LOOP
        DBMS_OUTPUT.PUT(tab_cars(i));
    END LOOP;
    DBMS_OUTPUT.NEW_LINE;
END;
/
```

Ce code affichera chaque caractère stocké dans le tableau `tab_cars` sur une même ligne, suivis d'un saut de ligne à la fin de l'affichage.

Conclusion :

Le package `DBMS_OUTPUT` est essentiel pour le débogage et l'affichage des résultats intermédiaires lors de l'exécution de programmes PL/SQL. En utilisant `PUT_LINE`, `PUT` et `NEW_LINE`, les développeurs peuvent contrôler précisément le formatage et le contenu des messages écrits dans le tampon de sortie, ce qui est particulièrement utile pour le suivi du flux d'exécution et le dépannage des erreurs.



Contraintes d’intégrité

Définition

Les contraintes d'intégrité dans les bases de données sont des règles essentielles qui garantissent la précision et la cohérence des données stockées. Elles jouent un rôle crucial en assurant que les données représentent correctement la réalité et en préservant leur intégrité à travers les opérations de manipulation.

1. Définition générale des contraintes d'intégrité :
   Les contraintes d'intégrité sont définies comme des règles qui garantissent la précision et la cohérence des données au sein d'une base de données. Elles permettent d’assurer que les données d’une BD sont conformes à la réalité qu'elles représentent." Cette définition met en évidence leur rôle fondamental dans la maintenance de la qualité des données (Adamski et Plewczynski, 2019).

2. Nature des contraintes :
   Les contraintes d'intégrité peuvent être simples ou complexes et peuvent prendre la forme de valeurs booléennes (vrai ou faux). Elles "sont des règles qui peuvent être simples ou complexes pouvant prendre la valeur vraie ou faux." Cette caractéristique montre leur flexibilité en fonction des besoins spécifiques de la base de données et des exigences métier (Connolly et Begg, 2015).

3. Champ d'application :
   Ces règles s'appliquent directement aux données stockées dans une base de données. Elles définissent des conditions et des restrictions sur les données pour s'assurer de leur validité et de leur cohérence. Elles s’appliquent sur les données d’une base de données, ce qui souligne leur impact direct sur la qualité et la fiabilité des informations contenues (Elmasri et Navathe, 2016).

4. Maintien de la cohérence des données :
   En maintenant la cohérence des données, les contraintes d'intégrité empêchent les valeurs incorrectes ou incohérentes d'être introduites dans la base de données. Elles "permettent de maintenir la cohérence de données", assurant ainsi que les informations disponibles sont exactes et fiables pour les utilisateurs et les applications (Ramakrishnan et Gehrke, 2003).

En résumé, les contraintes d'intégrité sont cruciales pour assurer la fiabilité et l'exactitude des données dans les bases de données. Elles garantissent que les informations stockées reflètent de manière précise la réalité qu'elles représentent, en imposant des règles qui doivent être respectées lors de la manipulation des données. Bien sûr, intégrons les exemples dans le contexte des points détaillés précédemment sur les contraintes d'intégrité dans les bases de données :

Quand les déclarer ?

1. À la création de la base de données :

Au moment de créer la base de données, nous définissons les contraintes d'intégrité pour garantir la cohérence des données dès le départ. Par exemple, dans la création de la table `Utilisateurs`, nous déclarons une contrainte de clé primaire sur la colonne `id` :

```
CREATE TABLE Utilisateurs (
    id INT PRIMARY KEY,
    nom VARCHAR(50),
    age INT
);
```

Ici, `PRIMARY KEY` définit `id` comme la clé primaire, assurant l'unicité et la non-nullité de cette colonne dès la création de la table.

2. À tout instant de l'existence de la base de données :

Au fur et à mesure que la base de données évolue, de nouvelles contraintes peuvent être ajoutées ou modifiées pour répondre aux nouveaux besoins. Par exemple, si nous ajoutons une nouvelle table `Employes`, nous pourrions définir une contrainte d'unicité sur le numéro de sécurité sociale (`num_secu`) pour s'assurer qu'il est unique pour chaque employé :

```
CREATE TABLE Employes (
    id INT PRIMARY KEY,
    nom VARCHAR(50),
    num_secu VARCHAR(15) UNIQUE,
    salaire DECIMAL(10, 2)
);
```

Ici, `UNIQUE` garantit l'unicité du numéro de sécurité sociale pour chaque employé.

Comment les déclarer et vérifier ?

1. À l'aide de SQL et ses extensions :

Les contraintes d'intégrité sont définies à l'aide de SQL lors de la création de la table ou par des instructions `ALTER TABLE` pour modifier la structure existante. Par exemple, pour ajouter une nouvelle contrainte de clé étrangère reliant la table `Employes` à la table `Departements` :

```
ALTER TABLE Employes
ADD CONSTRAINT fk_departement_id
FOREIGN KEY (departement_id) REFERENCES Departements(id);
```

Ici, `fk_departement_id` est le nom de la contrainte de clé étrangère qui assure l'intégrité référentielle entre les tables.

2. Vérification automatique par le SGBD :

Une fois définies, ces contraintes sont automatiquement appliquées par le SGBD lors des opérations de manipulation de données. Par exemple, si nous tentons d'insérer un nouvel employé avec un `num_secu` déjà existant dans la table `Employes`, cette opération sera rejetée en raison de la contrainte d'unicité définie.

3. Exemples de vérification des contraintes :

- Insertion de données :

```
INSERT INTO Employes (id, nom, num_secu, salaire)
VALUES (1, 'Jean Dupont', '123-45-6789', 50000.00);
```

Si un autre employé avec le même `num_secu` ('123-45-6789') existe déjà, cette insertion échouera en raison de la contrainte d'unicité définie sur `num_secu`.

- Mise à jour des données :

```
UPDATE Utilisateurs
SET id = 5
WHERE nom = 'Alice';
```

Si un autre utilisateur avec `id = 5` existait déjà, cette mise à jour échouerait en raison de la violation de la contrainte de clé primaire définie sur `id`.

- Suppression de données :

```
DELETE FROM Employes
WHERE id = 3;
```

Si `id = 3` n'existe pas dans la table `Employes`, cette opération de suppression échouerait en raison de la contrainte de clé primaire définie sur `id`.

En résumé, ces exemples illustrent comment les contraintes d'intégrité sont déclarées lors de la création et modifiées dynamiquement tout au long de la vie d'une base de données, ainsi que comment elles sont vérifiées automatiquement par le SGBD pour assurer la cohérence et la validité des données stockées.

Type de contraintes d'intégrités

Les contraintes d'intégrité dans les bases de données peuvent être classées en deux types principaux : statiques et dynamiques. Voici une explication détaillée de chacun de ces types avec des exemples concrets :

Contraintes d'intégrité statiques

Les contraintes d'intégrité statiques sont des règles qui s'appliquent à un état spécifique ou à des données existantes dans la base de données. Elles garantissent que les données sont cohérentes au moment où elles sont introduites ou modifiées. Voici quelques exemples :

1. Contrainte de clé primaire :
   Une contrainte de clé primaire garantit qu'une colonne ou un groupe de colonnes dans une table identifie de manière unique chaque ligne de cette table. Par exemple, dans une table `Utilisateurs`, la contrainte de clé primaire sur la colonne `id` garantit que chaque utilisateur a un identifiant unique.

```
CREATE TABLE Utilisateurs (
    id INT PRIMARY KEY,
    nom VARCHAR(50),
    email VARCHAR(100) UNIQUE
);
```

Ici, `id` est défini comme la clé primaire, assurant que chaque valeur dans cette colonne est unique et non nulle.

2. Contrainte d'unicité :
   Une contrainte d'unicité garantit que les valeurs dans une colonne ou un groupe de colonnes sont uniques à travers toutes les lignes d'une table. Par exemple, dans la même table `Utilisateurs`, nous pourrions avoir

une contrainte d'unicité sur la colonne `email` pour nous assurer qu'aucun utilisateur ne partage le même email.

```
CREATE TABLE Utilisateurs (
    id INT PRIMARY KEY,
    nom VARCHAR(50),
    email VARCHAR(100) UNIQUE
);
```

Ici, `UNIQUE` sur `email` assure que chaque adresse email dans la table est unique.

Contraintes d'intégrité dynamiques

Les contraintes d'intégrité dynamiques sont des règles qui dépendent du passage d'un état à un autre dans la base de données. Elles s'assurent que les données respectent les règles même lorsque des modifications sont apportées. Voici des exemples :

1. Contrainte de clé étrangère :
   Une contrainte de clé étrangère garantit l'intégrité référentielle entre deux tables en s'assurant qu'une valeur dans une colonne (clé étrangère) correspond à une valeur existante dans une autre table (clé primaire ou une colonne unique). Par exemple, dans une base de données de gestion d'école, la table `Etudiants` pourrait avoir une clé étrangère faisant référence à la table `Cours` pour garantir qu'un étudiant ne peut être inscrit qu'à des cours existants.

```
CREATE TABLE Etudiants (
    id INT PRIMARY KEY,
    nom VARCHAR(50),
    cours_id INT,
    FOREIGN KEY (cours_id) REFERENCES Cours(id)
);
```

Ici, `FOREIGN KEY (cours_id) REFERENCES Cours(id)` définit une contrainte de clé étrangère qui assure que chaque `cours_id` dans la table `Etudiants` correspond à un `id` existant dans la table `Cours`.

2. Contrainte CHECK :
   Une contrainte CHECK définit une condition permettant de vérifier les valeurs avant leur insertion ou leur modification. Par exemple, dans une base de données de vente en ligne, une contrainte CHECK pourrait être utilisée pour s'assurer que le prix d'un produit est supérieur à zéro.

```sql
CREATE TABLE Produits (
    id INT PRIMARY KEY,
    nom VARCHAR(100),
    prix DECIMAL(10, 2) CHECK (prix > 0)
);
```

Ici, `CHECK (prix > 0)` est une contrainte CHECK qui garantit que le prix d'un produit est toujours supérieur à zéro.

Voici un tableau comparatif de ces contraintes d'intégrité statiques :

Contrainte	Syntaxe SQL	Explication
Clé Primaire (PRIMARY KEY)	PRIMARY KEY (colonne)	Assure que chaque valeur de la colonne ou combinaison de colonnes est unique et non nulle.
Non-Nul (NOT NULL)	colonne TYPE NOT NULL	Assure que la colonne ne peut pas contenir de valeur nulle.
Unique (UNIQUE)	UNIQUE (colonne)	Assure que chaque valeur de la colonne est unique.
Vérification (CHECK)	CHECK (condition)	Imposer une condition spécifique pour les valeurs d'une colonne.
Plage de Valeurs	CHECK (colonne BETWEEN valeur1 AND valeur2)	Assure que les valeurs d'une colonne sont comprises dans une plage définie.
Liste de Valeurs	CHECK (colonne IN (valeur1, valeur2, ...))	Assure que les valeurs d'une colonne appartiennent à une liste prédéfinie.
Contrainte de Format	colonne CHAR(n)	Spécifie un format fixe pour les valeurs de la colonne.
Contrainte Multi-Colonnes	CHECK (colonne1 < colonne2)	Imposer une relation entre les valeurs de deux colonnes ou plus.

Les contraintes d'intégrité statiques et dynamiques jouent des rôles essentiels dans la gestion des données pour assurer la cohérence et la validité des informations stockées dans une base de données. Les contraintes statiques s'appliquent à des états spécifiques des données, tandis que les contraintes dynamiques surveillent les transitions d'état pour maintenir l'intégrité des données tout au long de leur cycle de vie.

Contraintes d'intégrité statique

Les contraintes statiques intra-relation

Les contraintes statiques intra-relation sont des règles imposées au sein d'une seule table pour garantir l'intégrité des données. Elles se divisent en deux catégories principales : les contraintes individuelles intra-relation mono-attribut et les contraintes individuelles intra-relation multi-attribut.

CI Individuelles Intra-Relation Mono-Attribut

Ces contraintes concernent une seule colonne (ou attribut) d'une table.

Contrainte de Vérification (CHECK)

Une contrainte de vérification impose une condition spécifique sur les valeurs d'une colonne. Par exemple, pour s'assurer que le salaire d'un employé est compris entre 4 000 et 20 000 :

```
CREATE TABLE Employes (
    Id_Employe INT PRIMARY KEY,
    Nom VARCHAR(50) NOT NULL,
    Salaire DECIMAL(10, 2) CHECK (Salaire BETWEEN 4000 AND 20000)
);
```

Ici, la contrainte `CHECK` sur la colonne `Salaire` impose que la valeur du salaire doit être comprise entre 4 000 et 20 000.

Plage de Valeurs

Cette contrainte définit une plage de valeurs acceptables pour une colonne. Par exemple, si le salaire doit être compris entre 10 000 et 20 000 :

```
CREATE TABLE Employes (
    Id_Employe INT PRIMARY KEY,
    Nom VARCHAR(50) NOT NULL,
    Salaire DECIMAL(10, 2) CHECK (Salaire BETWEEN 10000 AND 20000)
);
```

Ici, la contrainte `CHECK` sur la colonne `Salaire` impose une plage de valeurs entre 10 000 et 20 000.

Liste de Valeurs

Cette contrainte impose que les valeurs d'une colonne doivent appartenir à une liste de valeurs prédéfinie. Par exemple, pour la couleur :

```
CREATE TABLE Produits (
    Id_Produit INT PRIMARY KEY,
    Nom VARCHAR(50) NOT NULL,
    Couleur VARCHAR(10) CHECK (Couleur IN ('bleu', 'rouge', 'vert'))
);
```

Ici, la contrainte `CHECK` sur la colonne `Couleur` impose que la valeur doit être soit 'bleu', 'rouge', soit 'vert'.

Contraintes de Format

Cette contrainte impose un format spécifique pour les valeurs d'une colonne. Par exemple, pour s'assurer que le nom est toujours de 20 caractères :

```
CREATE TABLE Clients (
    Id_Client INT PRIMARY KEY,
    Nom CHAR(20) NOT NULL
);
```

Ici, la colonne `Nom` est définie comme `CHAR(20)`, ce qui signifie que chaque nom doit être exactement de 20 caractères.

CI Individuelles Intra-Relation Multi-Attribut

Ces contraintes concernent plusieurs colonnes d'une même table.

Contraintes entre Constituants

Cette contrainte impose une relation entre les valeurs de différentes colonnes. Par exemple, pour s'assurer que les dépenses sont toujours inférieures aux recettes :

```
CREATE TABLE Budget (
    Id_Budget INT PRIMARY KEY,
    Recettes DECIMAL(10, 2) NOT NULL,
    Depenses DECIMAL(10, 2) NOT NULL,
    CHECK (Depenses < Recettes)
);
```

Ici, la contrainte `CHECK` impose que la valeur de la colonne `Depenses` doit être inférieure à la valeur de la colonne `Recettes`.

Exemple Détaillé
Supposons que nous ayons une table `Employes` pour stocker des informations sur les employés d'une entreprise.

```
CREATE TABLE Employes (
    Id_Employe INT PRIMARY KEY,
    Nom VARCHAR(50) NOT NULL,
    Salaire DECIMAL(10, 2) CHECK (Salaire BETWEEN 4000 AND 20000),
    Departement VARCHAR(20) CHECK (Departement IN ('RH', 'Finance', 'IT', 'Marketing')),
    Date_Embauche DATE,
    CHECK (EXTRACT(YEAR FROM Date_Embauche) >= 2000)
);
```

Dans cet exemple :
- PRIMARY KEY: La colonne `Id_Employe` est définie comme clé primaire, assurant que chaque employé a un identifiant unique.
- NOT NULL: La colonne `Nom` ne peut pas contenir de valeurs nulles, chaque employé doit avoir un nom.
- CHECK(Plage de valeurs) : La contrainte `CHECK` sur la colonne `Salaire` garantit que le salaire est compris entre 4 000 et 20 000.
- CHECK(Liste de valeurs) : La contrainte `CHECK` sur la colonne `Departement` impose que la valeur doit être soit 'RH', 'Finance', 'IT', soit 'Marketing'.
- CHECK(Multi-attribut) : La contrainte `CHECK` sur la colonne `Date_Embauche` impose que l'année d'embauche doit être postérieure ou égale à l'an 2000.

Ces contraintes assurent que les données dans la table `Employes` sont valides et cohérentes, respectant les règles métier définies.

Les contraintes d'intégrité statique ensembliste

Les contraintes d'intégrité statique ensembliste sont des règles imposées pour maintenir l'intégrité des données au niveau d'une table entière. Elles peuvent être appliquées au niveau des colonnes ou des tables pour garantir que les données respectent certaines conditions et relations spécifiques.

Niveau Colonne

Contrainte NOT NULL

La contrainte `NOT NULL` interdit la présence de valeurs nulles dans la colonne. Cela signifie que chaque enregistrement dans cette colonne doit avoir une valeur.

```
CREATE TABLE Employés (
    NSS NUMBER(4),
    Nomemp VARCHAR(10) NOT NULL,
    Numproj NUMBER(9),
    Numdept NUMBER(2) NOT NULL
);
```

Dans cet exemple, les colonnes `Nomemp` et `Numdept` de la table `Employés` sont définies avec la contrainte `NOT NULL`. Cela signifie qu'aucun enregistrement ne peut avoir une valeur nulle dans ces colonnes.

Vous pouvez également spécifier une contrainte `NOT NULL` avec un nom personnalisé :

```
Numdept NUMBER(2) CONSTRAINT C1 NOT NULL;
```

Niveau Colonne et Niveau Table

Contrainte de Clé Unique (UNIQUE)

La contrainte `UNIQUE` assure que toutes les valeurs d'une colonne ou d'un groupe de colonnes sont uniques. Cette contrainte peut être définie au niveau colonne ou au niveau table.

```
CREATE TABLE Departement (
    Numdept NUMBER(2),
    Nomdept VARCHAR(12) CONSTRAINT departement_nomdept_uk UNIQUE
);
```

Dans cet exemple, la colonne `Nomdept` est définie avec une contrainte `UNIQUE` nommée `departement_nomdept_uk`. Cela garantit que chaque valeur dans la colonne `Nomdept` est unique.

Vous pouvez également définir une contrainte `UNIQUE` au niveau table :

```
CREATE TABLE Departement (
    Numdept NUMBER(2),
    Nomdept VARCHAR(12),
    CONSTRAINT departement_nomdept_uk UNIQUE (Nomdept)
);
```

Contrainte de Clé Primaire (PRIMARY KEY)

La contrainte `PRIMARY KEY` combine les propriétés de `NOT NULL` et `UNIQUE`, garantissant que les valeurs de la colonne ou des colonnes spécifiées sont à la fois uniques et non nulles. Un index unique est automatiquement créé pour la clé primaire.

```
CREATE TABLE Departement (
    Numdept NUMBER(2),
    Nomdept VARCHAR(14),
    CONSTRAINT departement_nomdept_uk UNIQUE (Nomdept),
    CONSTRAINT departement_numdept_pk PRIMARY KEY (Numdept)
);
```

Dans cet exemple :
- La contrainte `UNIQUE` nommée `departement_nomdept_uk` garantit que les valeurs dans la colonne `Nomdept` sont uniques.
- La contrainte `PRIMARY KEY` nommée `departement_numdept_pk` sur la colonne `Numdept` assure que chaque valeur est unique et non nulle. Un index unique est automatiquement créé pour cette colonne.

Les Contraintes Intra-Relation Multi-Attributs

Les contraintes intra-relation multi-attributs concernent plusieurs colonnes (ou attributs) au sein d'une même table. Elles peuvent être classées en deux types principaux : verticales et horizontales.

Verticale

Les contraintes verticales contrôlent la valeur d'un attribut (colonne) d'un enregistrement (n-uplet) en fonction des valeurs de cet attribut pour d'autres enregistrements de la même table.

Exemple :
Supposons une table `Employé` avec les colonnes `Nom`, `Salaire` et `Nom_Rayon`.
- Contrainte :Un employé ne peut gagner plus du double de la moyenne des salaires de son rayon.

```
ASSERT CI2 ON Employé E :
Salaire ≤ 2 * (
    SELECT AVG(Salaire)
    FROM Employé
    WHERE Nom_Rayon = E.Nom_Rayon
);
```

Explication :
- La contrainte assure que le salaire de chaque employé est au maximum le double de la moyenne des salaires des employés travaillant dans le même rayon.
- La clause `ASSERT` vérifie cette condition pour chaque enregistrement en comparant le salaire de l'employé avec la moyenne des salaires dans le même rayon (`Nom_Rayon`).

Horizontale

Les contraintes horizontales contrôlent la valeur d'un attribut en fonction des valeurs apparaissant dans d'autres attributs du même enregistrement.
Exemple :
Supposons une table `Rayon` avec les colonnes `Nom_Rayon`, `Etage`, `Recettes`, `Dépenses` et `Nb_Employés`.
- Contrainte :Les dépenses doivent être inférieures ou égales aux recettes.

```
ASSERT CI3 ON Rayon :
Dépenses ≤ Recettes;
```

Explication :
- La contrainte assure que pour chaque enregistrement dans la table `Rayon`, les dépenses ne dépassent pas les recettes.
- La clause `ASSERT` vérifie cette condition pour chaque enregistrement de la table `Rayon`.

Contrôler les Relations entre Tables

Une autre forme de contrainte intra-relation consiste à vérifier que les valeurs d'un attribut dans une table apparaissent également dans une autre table. Cela peut être exprimé en utilisant des sous-requêtes ou des clés étrangères (`FOREIGN KEY`).

Exemple :
Supposons deux tables : `Employé` et `Rayon`.
- Contrainte :Tout rayon apparaissant dans la table `Employé` doit être décrit dans la table `Rayon`.

```
ASSERT CI5 (
    SELECT Nom_Rayon
    FROM Employé
) IS IN (
    SELECT Nom_Rayon
    FROM Rayon
);
```

Cette contrainte peut être remplacée par une contrainte de clé étrangère :
Clé Étrangère (FOREIGN KEY)

La contrainte `FOREIGN KEY` définit une colonne d'une table par référence à une colonne d'une autre table. Elle assure que les valeurs de la colonne référencée existent dans la table de référence.

Exemple :
Supposons une table `Employé` qui référence une table `Departement`.

```
CREATE TABLE Departement (
    Numdept NUMBER(2) PRIMARY KEY,
    Nomdept VARCHAR(14)
);

CREATE TABLE Employé (
    Numemp NUMBER(4),
    Nomemp VARCHAR(10) NOT NULL,
    Numdept NUMBER(2) NOT NULL,
    CONSTRAINT employe_numdept_fk FOREIGN KEY (Numdept)
    REFERENCES Departement(Numdept)
);
```

Explication :
- La table `Departement` contient des départements avec `Numdept` comme clé primaire.
- La table `Employé` contient une colonne `Numdept` qui référence `Numdept` dans la table `Departement`.
- La contrainte `FOREIGN KEY` `employe_numdept_fk` assure que chaque valeur de `Numdept` dans la table `Employé` correspond à une valeur existante de `Numdept` dans la table `Departement`.

Ces contraintes garantissent l'intégrité référentielle entre les tables et évitent les incohérences dans les relations entre les données.

Exercice 1 : Contrainte de Vérification (CHECK)

Créer une table `Produit` qui contient les informations suivantes :
- `Id_Produit` (clé primaire, entier)
- `Nom_Produit` (chaîne de caractères, non nul)
- `Prix` (décimal)
- `Quantite` (entier)

Implémentez les contraintes suivantes :
- Le prix doit être compris entre 1 et 1000.
- La quantité doit être au moins 1.

Exercice 2 : Contrainte de Clé Unique (UNIQUE)

Créer une table `Client` qui contient les informations suivantes :
- `Id_Client` (clé primaire, entier)
- `Nom_Client` (chaîne de caractères, non nul)
- `Email` (chaîne de caractères)

Implémentez les contraintes suivantes :
- L'email doit être unique parmi tous les clients..

Exercice 3 : Contrainte de Clé Étrangère (FOREIGN KEY)

Créer deux tables `Departement` et `Employé` :
- La table `Departement` contient les informations suivantes :
  - `Numdept` (clé primaire, entier)
  - `Nomdept` (chaîne de caractères)

- La table `Employé` contient les informations suivantes :
  - `Numemp` (clé primaire, entier)
  - `Nomemp` (chaîne de caractères, non nul)
  - `Numdept` (entier)

Implémentez les contraintes suivantes :
- Chaque employé doit être associé à un département existant dans la table `Departement`.

Solution Exercice 1 : Contrainte de Vérification (CHECK)

```sql
CREATE TABLE Produit (
    Id_Produit INT PRIMARY KEY,
    Nom_Produit VARCHAR(50) NOT NULL,
    Prix DECIMAL(10, 2) CHECK (Prix BETWEEN 1 AND 1000),
    Quantite INT CHECK (Quantite >= 1)
);
```

Explication :
1. `Id_Produit INT PRIMARY KEY`:
   - La colonne `Id_Produit` est définie comme clé primaire, ce qui signifie que chaque produit doit avoir un identifiant unique et non nul.
2. `Nom_Produit VARCHAR(50) NOT NULL`:
   - La colonne `Nom_Produit` est une chaîne de caractères d'une longueur maximale de 50 caractères et ne peut pas être nulle (`NOT NULL`), garantissant que chaque produit a un nom.
3. `Prix DECIMAL(10, 2) CHECK (Prix BETWEEN 1 AND 1000)`:
   - La colonne `Prix` est de type décimal avec une précision totale de 10 chiffres, dont 2 après la virgule.
   - La contrainte `CHECK (Prix BETWEEN 1 AND 1000)` impose que le prix des produits doit être compris entre 1 et 1000 inclus.
4. `Quantite INT CHECK (Quantite >= 1)`:
   - La colonne `Quantite` est un entier.
   - La contrainte `CHECK (Quantite >= 1)` garantit que la quantité des produits doit être au moins 1, évitant des valeurs nulles ou négatives.

Solution Exercice 2 : Contrainte de Clé Unique (UNIQUE)

```sql
CREATE TABLE Client (
    Id_Client INT PRIMARY KEY,
    Nom_Client VARCHAR(50) NOT NULL,
    Email VARCHAR(100) UNIQUE
);
```

Explication :
1. `Id_Client INT PRIMARY KEY`:
   - La colonne `Id_Client` est définie comme clé primaire, garantissant que chaque client a un identifiant unique et non nul.
2. `Nom_Client VARCHAR(50) NOT NULL`:
   - La colonne `Nom_Client` est une chaîne de caractères d'une longueur maximale de 50 caractères et ne peut pas être nulle (`NOT NULL`), assurant que chaque client a un nom.
3. `Email VARCHAR(100) UNIQUE`:
   - La colonne `Email` est une chaîne de caractères d'une longueur maximale de 100 caractères.
   - La contrainte `UNIQUE` sur l'email garantit que chaque adresse email dans la table est unique, évitant les doublons.

Solution Exercice 3 : Contrainte de Clé Étrangère (FOREIGN KEY)

```sql
CREATE TABLE Departement (
    Numdept INT PRIMARY KEY,
    Nomdept VARCHAR(50)
);

CREATE TABLE Employé (
    Numemp INT PRIMARY KEY,
    Nomemp VARCHAR(50) NOT NULL,
    Numdept INT,
    CONSTRAINT employe_numdept_fk FOREIGN KEY (Numdept)
    REFERENCES Departement(Numdept)
);
```

Explication :

1. Table `Departement` :
   - `Numdept INT PRIMARY KEY`:
     - La colonne `Numdept` est définie comme clé primaire, garantissant que chaque département a un identifiant unique et non nul.
   - `Nomdept VARCHAR(50)`:
     - La colonne `Nomdept` est une chaîne de caractères d'une longueur maximale de 50 caractères pour stocker le nom du département.
2. Table `Employé` :
   - `Numemp INT PRIMARY KEY`:
     - La colonne `Numemp` est définie comme clé primaire, garantissant que chaque employé a un identifiant unique et non nul.
   - `Nomemp VARCHAR(50) NOT NULL`:
     - La colonne `Nomemp` est une chaîne de caractères d'une longueur maximale de 50 caractères et ne peut pas être nulle (`NOT NULL`), assurant que chaque employé a un nom.
   - `Numdept INT`:
     - La colonne `Numdept` est un entier, représentant l'identifiant du département auquel l'employé est associé.
   - `CONSTRAINT employe_numdept_fk FOREIGN KEY (Numdept) REFERENCES Departement(Numdept)`:
     - Cette contrainte de clé étrangère (`FOREIGN KEY`) `employe_numdept_fk` assure que chaque valeur de `Numdept` dans la table `Employé` doit correspondre à une valeur existante de `Numdept` dans la table `Departement`.
     - Cela garantit l'intégrité référentielle, assurant que chaque employé est associé à un département valide.

Contraintes d’intégrité dynamiques

Les contraintes d'intégrité dynamiques sont des règles qui s'appliquent lors des changements d'état des données dans une base de données. Elles permettent de contrôler les transitions entre les états anciens et nouveaux des enregistrements, garantissant ainsi la cohérence et l'intégrité des données au cours des opérations de mise à jour ou d'insertion.

Les contraintes d'intégrité dynamiques sont définies au cours des changements d'états des enregistrements dans une base de données.
- Contrôle des transitions :Il est possible de contrôler le passage d’un état à un autre en utilisant des assertions ou des déclencheurs (triggers).
- Dépendance des états :Les nouvelles valeurs des enregistrements dépendent des anciennes valeurs.

Types de Contraintes Dynamiques

1. Directement à l’aide d’assertions :
   - Les assertions sont des conditions qui doivent être vraies à tout moment. Elles peuvent être utilisées pour vérifier les contraintes lors de la mise à jour des enregistrements.

2. Triggers ou déclencheurs :
   - Les triggers sont des mécanismes qui exécutent une action spécifique en réponse à un événement particulier sur une table, comme une insertion, une mise à jour ou une suppression.

Contrainte d'Intégrité Dynamique Intra-Relation

- Exemple :Le salaire d’un employé ne peut diminuer lors d’une mise à jour.
Assertion :

```
ASSERT CI6 ON UPDATE OF Employé(Salaire)
NEW.Salaire ≥ OLD.Salaire;
```

Explication :
- Cette assertion garantit que lors d’une mise à jour de la colonne `Salaire` dans la table `Employé`, la nouvelle valeur du salaire (`NEW.Salaire`) doit être supérieure ou égale à l'ancienne valeur (`OLD.Salaire`).

Contrainte d'Intégrité Dynamique Inter-Relation

- Exemple :Tout employé du département 'Recherche' doit avoir un salaire supérieur à 150000 DA.

Assertion :

```
CREATE ASSERTION CHECK NOT EXISTS (
    SELECT *
    FROM EMPLOYE
    WHERE salaire <= 150000 AND IDD = (
        SELECT IDD
        FROM DEPARTEMENT
        WHERE nom_dept = 'Recherche'
    )
);
```

Explication :

- Cette assertion vérifie qu'il n'existe aucun employé dans le département 'Recherche' ayant un salaire inférieur ou égal à 150000 DA. Si une telle condition est détectée, l'opération qui a provoqué cette situation est annulée.

Modification des Contraintes d'Intégrité

Ajouter une Contrainte d'Intégrité

```
ALTER TABLE <nom_table>
ADD CONSTRAINT <nom_contrainte> <definition_contrainte>;
```

Exemple :

```
ALTER TABLE Employé
ADD CONSTRAINT chk_salaire CHECK (Salaire >= 0);
```

Explication :

- Cette commande ajoute une contrainte de vérification (`CHECK`) à la table `Employé`, garantissant que le salaire est toujours supérieur ou égal à 0.

Supprimer une Contrainte d'Intégrité

```
ALTER TABLE <nom_table>
DROP CONSTRAINT <nom_contrainte>;
```

Exemple :

```
ALTER TABLE Employé
DROP CONSTRAINT chk_salaire;
```

Explication :
- Cette commande supprime la contrainte de vérification `chk_salaire` de la table `Employé`.

Triggers ou Déclencheurs

Un trigger est une règle qui suit la structure suivante :
- Structure :« Si événement et/ou condition alors action »
- Fonctionnement :L'action est déclenchée en réponse à l'événement, si la condition spécifiée est vérifiée.

Exemple de Trigger
- Lorsqu'un nouvel employé est inséré dans la base, le nombre d'employés du rayon concerné doit augmenter de un.

```
CREATE TRIGGER EMPINS
ON INSERTION OF Employé
FOR EACH ROW
BEGIN
    UPDATE RAYON
    SET Nb_Employés = Nb_Employés + 1
    WHERE Nom_Rayon = :NEW.Nom_Rayon;
END;
```

Explication :
- Déclencheur (`CREATE TRIGGER EMPINS`) :Le nom du trigger est `EMPINS`.
- Événement (`ON INSERTION OF Employé`) :Le trigger se déclenche lors de l'insertion d'un nouvel enregistrement dans la table `Employé`.
- Action :
  - L'action consiste à mettre à jour la table `RAYON`.
  - La colonne `Nb_Employés` est incrémentée de 1 pour le rayon (`Nom_Rayon`) correspondant au nouvel employé inséré (`:NEW.Nom_Rayon`)

Les contraintes d'intégrité dynamiques sont essentielles pour maintenir l'intégrité et la cohérence des données dans une base de données, surtout lorsque les données subissent des modifications fréquentes. Les assertions et les triggers offrent des moyens puissants pour imposer et vérifier ces contraintes dynamiques.

Triggers

Un trigger en PL/SQL est un bloc de code PL/SQL qui s'exécute automatiquement en réponse à certains événements sur une table ou une vue. Il est utilisé pour appliquer des règles d'intégrité dynamique, auditer les changements, ou effectuer des actions automatiques en réponse à des modifications de données.

Caractéristiques d'un trigger

1. Événements déclencheurs: Les triggers peuvent être déclenchés par des événements tels que `INSERT`, `UPDATE`, ou `DELETE` sur une table ou une vue.
2. Moment d'exécution: Ils peuvent être configurés pour s'exécuter avant ou après l'événement déclencheur.
3. Niveau d'application: Ils peuvent s'appliquer à chaque ligne affectée (`FOR EACH ROW`) ou une seule fois par opération sur la table (`FOR EACH STATEMENT`).

Utilisations courantes des triggers

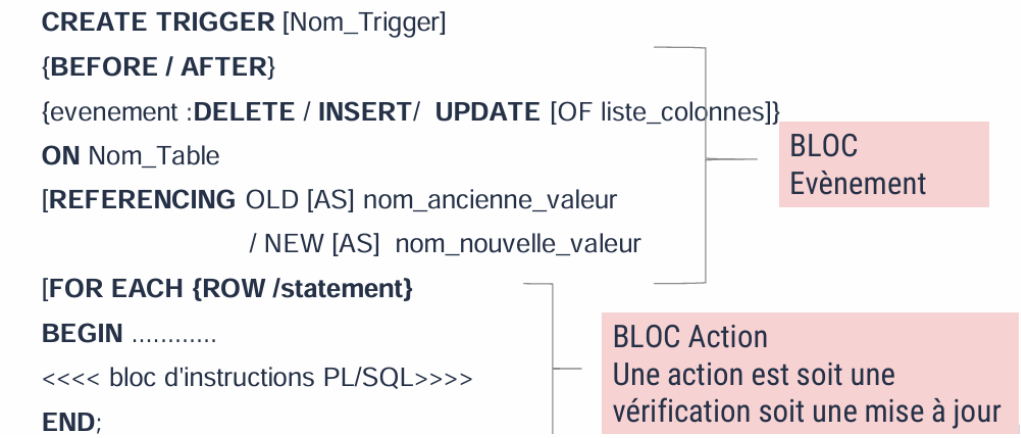
- Maintenir l'intégrité des données: Assurer que certaines conditions sont respectées lors de la modification des données.
- Auditer les modifications: Enregistrer les changements effectués sur les données pour des fins de traçabilité.
- Synchroniser des tables: Propager les modifications d'une table à une autre.
- Appliquer des règles de gestion: Automatiser des actions spécifiques en réponse à des changements de données.

Exemple simple de trigger en PL/SQL

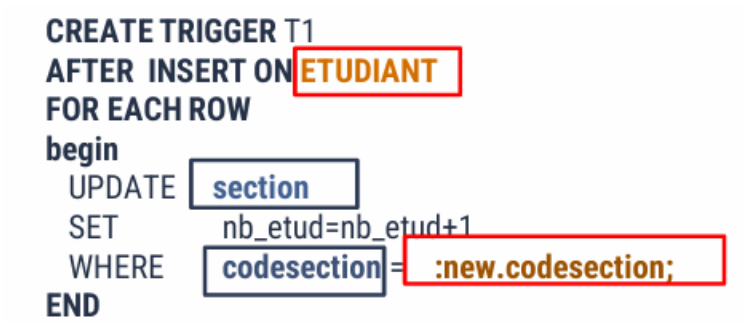
```
CREATE OR REPLACE TRIGGER enforce_salary_limit
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    IF :NEW.salary < 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Le salaire doit être positif.');
    END IF;
END;
```

Dans cet exemple, le trigger `enforce_salary_limit` s'assure qu'aucun salaire négatif ne peut être inséré ou mis à jour dans la table `employees`.

Syntaxe de creation de triggers



Exemple des triggers



Dans cet exemple, nous avons trois tables : `Etudiant`, `Evaluation` et `Section`. Le déclencheur (trigger) `T1` est utilisé pour mettre à jour le nombre d'étudiants (`NB_etud`) dans la table `Section` chaque fois qu'un nouvel étudiant est inséré dans la table `Etudiant`.
Schéma des Tables
- Etudiant (mat, nom, code_section): Cette table contient des informations sur les étudiants, y compris leur numéro matricule (`mat`), nom (`nom`) et le code de la section (`code_section`) à laquelle ils appartiennent.

- Evaluation (mat, codemodule, note): Cette table enregistre les évaluations des étudiants, associées à leur numéro matricule, au code du module et à la note obtenue.

- Section (codesection, nom, NB_etud): La table des sections contient des détails sur les différentes sections, y compris le code de section (`codesection`), le nom (`nom`) et le nombre d'étudiants (`NB_etud`) actuellement inscrits.

Déclencheur (Trigger) T1

```
CREATE TRIGGER T1
AFTER INSERT ON Etudiant
FOR EACH ROW
BEGIN
    UPDATE Section
    SET NB_etud = NB_etud + 1
    WHERE codesection = :new.code_section;
END;
```

Explication du Déclencheur (Trigger)
- `CREATE TRIGGER T1`: Cette instruction SQL crée un nouveau déclencheur nommé `T1`.
- `AFTER INSERT ON Etudiant`: Le déclencheur se déclenche après chaque insertion dans la table `Etudiant`.
- `FOR EACH ROW`: Indique que le déclencheur s'exécutera pour chaque ligne insérée dans la table `Etudiant`.
- `BEGIN ... END`: Contient le corps du déclencheur, où les actions spécifiées seront exécutées.
- `UPDATE Section ...`: C'est l'action principale du déclencheur. Cette instruction met à jour la table `Section`.
- `SET NB_etud = NB_etud + 1`: Incrémente le nombre d'étudiants (`NB_etud`) dans la table `Section` de 1.
- `WHERE codesection = :new.code_section`: Filtre les lignes de la table `Section` où le `codesection` correspond à la valeur `code_section` du nouvel étudiant inséré (`:new.code_section`).

Fonctionnement du Déclencheur

Chaque fois qu'une nouvelle ligne est insérée dans la table `Etudiant`, le déclencheur `T1` est déclenché. Il exécute une mise à jour sur la table `Section`, augmentant le nombre d'étudiants (`NB_etud`) pour la section spécifique à laquelle le nouvel étudiant appartient (`code_section`). Cela assure que la table `Section` reste à jour avec le nombre actuel d'étudiants inscrits dans chaque section, garantissant ainsi l'intégrité et l'exactitude des données.

Exercice

1. Créer une table `employees` avec les colonnes suivantes :
   - `employee_id` (NUMBER) - clé primaire
   - `first_name` (VARCHAR2(50))
   - `last_name` (VARCHAR2(50))
   - `salary` (NUMBER)

2. Créer un trigger `check_salary_before_insert` qui :
   - Se déclenche avant une opération d'insertion sur la table `employees`.
   - Vérifie que la valeur de la colonne `salary` est supérieure à zéro.
   - Lève une erreur si la condition n'est pas respectée.

Solution

1. Création de la table `employees`:

```
CREATE TABLE employees (
    employee_id NUMBER PRIMARY KEY,
    first_name VARCHAR2(50),
    last_name VARCHAR2(50),
    salary NUMBER
);
```

2. Création du trigger `check_salary_before_insert`:

```
CREATE OR REPLACE TRIGGER check_salary_before_insert
BEFORE INSERT ON employees
FOR EACH ROW
BEGIN
    IF :NEW.salary <= 0 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Le salaire doit être supérieur à zéro.');
    END IF;
END;
```

Explication du trigger

- `CREATE OR REPLACE TRIGGER check_salary_before_insert`: Cette commande crée un nouveau trigger nommé `check_salary_before_insert` ou le remplace s'il existe déjà.
- `BEFORE INSERT ON employees`: Indique que le trigger se déclenche avant chaque opération d'insertion sur la table `employees`.
- `FOR EACH ROW`: Spécifie que le trigger s'applique à chaque ligne affectée par l'opération d'insertion.
- `BEGIN ... END`: Bloc PL/SQL contenant le code du trigger.
- `IF :NEW.salary <= 0 THEN`: Vérifie si la valeur du nouveau salaire est inférieure ou égale à zéro.
- `RAISE_APPLICATION_ERROR(-20001, 'Le salaire doit être supérieur à zéro.');`: Lève une erreur avec le code d'erreur -20001 et le message spécifié si la condition est vraie.

Test du trigger

Essayons d'insérer des données dans la table `employees` pour voir le trigger en action.

Insertion valide :

```
INSERT INTO employees (employee_id, first_name, last_name, salary)
VALUES (1, 'John', 'Doe', 5000);

-- Résultat attendu : L'insertion réussit car le salaire est supérieur à zéro.
```

Insertion invalide :

```
INSERT INTO employees (employee_id, first_name, last_name, salary)
VALUES (2, 'Jane', 'Smith', -100);

-- Résultat attendu : L'insertion échoue avec le message d'erreur 'Le salaire doit être su
```

Ce trigger assure que toutes les insertions dans la table `employees` respectent la règle métier selon laquelle les salaires doivent être positifs.

Les triggers en SQL sont des objets de base de données qui sont déclenchés automatiquement en réponse à certains événements sur une table, comme une insertion, une mise à jour ou une suppression de données. Voici une explication détaillée des différents aspects des triggers PL/SQL avec des exemples.

Création de Triggers

Un trigger est créé pour surveiller un événement spécifique sur une table et exécuter des actions définies lors de cet événement. Voici les principales caractéristiques à considérer lors de la création d'un trigger :
- Nom du Trigger :Le nom du trigger doit être unique dans le schéma de la base de données.
- Événements :Les événements qui peuvent déclencher un trigger sont INSERT, UPDATE et DELETE.
- Définition de l'Événement :Pour l'événement UPDATE, on peut spécifier une liste de colonnes. Le trigger se déclenchera uniquement si l'instruction UPDATE modifie l'une au moins des colonnes spécifiées.
- Option BEFORE/AFTER :Spécifie si le déclencheur doit être exécuté avant ou après l'exécution de l'événement.

Exemple de Création de Trigger

```
CREATE OR REPLACE TRIGGER nom_trigger
BEFORE INSERT ON table
FOR EACH ROW
BEGIN
    -- Actions à exécuter
    -- Exemple : incrémenter une valeur dans une autre table
    UPDATE autre_table
    SET compteur = compteur + 1
    WHERE condition;
END;
```

Types de Triggers

Les triggers peuvent être de deux types principaux :

1. Triggers Ligne (`FOR EACH ROW`):

Ce type de trigger est déclenché pour chaque ligne affectée par l'événement. On utilise souvent la clause `WHEN` pour spécifier une condition logique sur les lignes à traiter.

- Exemple avec WHEN :

```
CREATE OR REPLACE TRIGGER trigger_ligne
BEFORE INSERT OR UPDATE ON table
FOR EACH ROW
WHEN (new.colonne > 0)
BEGIN
    -- Actions à exécuter si la nouvelle valeur de colonne est positive
END;
```

2. Triggers Global (Statement):

Ce type de trigger est déclenché une seule fois pour chaque instruction SQL exécutée, indépendamment du nombre de lignes affectées.

- Exemple de Trigger Global :

```
CREATE OR REPLACE TRIGGER trigger_global
BEFORE DELETE ON table
BEGIN
    -- Actions globales à exécuter avant la suppression de toutes les lignes
END;
```

Utilisation des Anciennes et Nouvelles Valeurs

Dans les triggers ligne, il est possible d'accéder aux anciennes (`:old.colonne`) et nouvelles (`:new.colonne`) valeurs des colonnes affectées par l'événement. Cela permet de comparer les valeurs avant et après l'événement pour prendre des décisions appropriées.

- Exemple d'utilisation des Anciennes et Nouvelles Valeurs :

```
CREATE OR REPLACE TRIGGER exemple_trigger
BEFORE UPDATE ON table
FOR EACH ROW
BEGIN
    IF :new.salaire < :old.salaire THEN
        -- Actions à exécuter si le nouveau salaire est inférieur à l'ancien salaire
    END IF;
END;
```

Option `REFERENCING`

L'option `REFERENCING` est utilisée pour spécifier un alias (`AS`) pour les anciennes et nouvelles valeurs (`OLD` et `NEW`) lorsque le nom de la table peut être ambigu dans le contexte du trigger.

- Exemple avec `REFERENCING` :

```
CREATE OR REPLACE TRIGGER trigger_referencing
BEFORE UPDATE ON table
REFERENCING OLD AS old_table NEW AS new_table
FOR EACH ROW
BEGIN
    :new_table.colonne1 := TO_CHAR(:new_table.colonne2);
END;
```

Les prédicats conditionnels (`INSERTING`, `DELETING` et `UPDATING`)

Les prédicats conditionnels (`INSERTING`, `DELETING` et `UPDATING`) sont utilisés dans les triggers PL/SQL pour exécuter des blocs de code spécifiques en fonction du type d'instruction SQL (`INSERT`, `DELETE`, `UPDATE`) qui a déclenché le trigger. Voici un exemple concret :

Explication des Prédicats Conditionnels

Lorsqu'un trigger est déclenché par plusieurs types d'instructions (par exemple `INSERT`, `DELETE` ou `UPDATE`), les prédicats conditionnels permettent de déterminer quelle action doit être effectuée en fonction de l'instruction SQL.
- `INSERTING`: Ce prédicat est vrai lorsque le trigger est déclenché par une instruction `INSERT`.
- `DELETING`: Ce prédicat est vrai lorsque le trigger est déclenché par une instruction `DELETE`.
- `UPDATING`: Ce prédicat est vrai lorsque le trigger est déclenché par une instruction `UPDATE`.

Exemple d'Utilisation

Considérons deux tables : `Employes` et `Service`.

- Employes: Contient les informations sur les employés.
  - Colonnes : `NSS`, `Nom`, `Prenom`, `NumService`

- Service: Contient les informations sur les services.
  - Colonnes : `CodeServ`, `Nom_service`, `NbEmp`

L'objectif est de maintenir à jour le nombre d'employés (`NbEmp`) dans la table `Service` en fonction des opérations effectuées sur la table `Employes`.

Exemple de Trigger

```
CREATE OR REPLACE TRIGGER trg_update_nbemp
BEFORE INSERT OR UPDATE OR DELETE ON Employes
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        UPDATE Service
        SET NbEmp = NbEmp + 1
        WHERE CodeServ = :new.NumService;
    END IF;

    IF UPDATING THEN
        UPDATE Service
        SET NbEmp = NbEmp - 1
        WHERE CodeServ = :old.NumService;

        UPDATE Service
        SET NbEmp = NbEmp + 1
        WHERE CodeServ = :new.NumService;
    END IF;

    IF DELETING THEN
        UPDATE Service
        SET NbEmp = NbEmp - 1
        WHERE CodeServ = :old.NumService;
    END IF;
END;
```

Explication de l'Exemple

Ce trigger `trg_update_nbemp` est déclenché avant toute opération d'`INSERT` ou d'`UPDATE` sur la table `Employes`. Voici ce que chaque partie du trigger fait :

1. IF INSERTING: Cette partie du code est exécutée si le trigger est déclenché par une instruction `INSERT`. Lorsqu'un nouvel employé est inséré dans la table `Employes`, le nombre d'employés (`NbEmp`) du service correspondant dans la table `Service` est augmenté de 1.

- `:new.NumService` fait référence à la nouvelle valeur de `NumService` pour l'employé inséré.

2. IF UPDATING: Cette partie du code est exécutée si le trigger est déclenché par une instruction `UPDATE`. Lorsqu'un employé existant est mis à jour dans la table `Employes`, deux opérations sont effectuées :

- La première mise à jour diminue de 1 le nombre d'employés (`NbEmp`) du service correspondant à l'ancienne valeur de `NumService` (`:old.NumService`).
- La deuxième mise à jour augmente de 1 le nombre d'employés (`NbEmp`) du service correspondant à la nouvelle valeur de `NumService` (`:new.NumService`).

3. IF DELETING : Cette partie du code est exécutée si le trigger est déclenché par une instruction DELETE. Lorsqu'un employé est supprimé de la table Employes, le nombre d'employés (NbEmp) du service correspondant dans la table Service est diminué de 1.
:old.NumService fait référence à l'ancienne valeur de NumService pour l'employé qui est en train d'être supprimé.
Le prédicat DELETING permet de s'assurer que le bloc de code à l'intérieur du IF DELETING est exécuté uniquement lorsque des lignes sont supprimées de la table Employes. Cela permet de maintenir la cohérence des données dans la table Service, en ajustant automatiquement le nombre d'employés (NbEmp) lorsque des employés sont retirés.

Les prédicats conditionnels (`INSERTING`, `DELETING`, `UPDATING`) sont des outils essentiels pour structurer les triggers PL/SQL en fonction du type d'opération SQL qui les déclenche. Ils offrent une flexibilité accrue pour automatiser les actions en réponse aux modifications des données, tout en garantissant l'intégrité et la cohérence des informations dans la base de données.

Les prédicats conditionnels `INSERTING`, `DELETING` et `UPDATING` permettent de structurer logiquement les triggers afin qu'ils réagissent de manière appropriée aux différentes opérations sur les données. Cela permet de maintenir l'intégrité des données et d'automatiser la gestion des relations entre les tables dans une base de données relationnelle.

En résumé, les prédicats conditionnels sont une fonctionnalité puissante de PL/SQL qui offre une flexibilité accrue dans la gestion des triggers en fonction des types d'instructions SQL qui les déclenchent.

Ordre de Traitement des Lignes dans les Triggers PL/SQL

Lorsqu'on travaille avec des triggers PL/SQL dans une base de données, il est essentiel de comprendre certains aspects concernant l'ordre de traitement des lignes et les triggers en cascade. Voici une explication détaillée avec des exemples :

Ordre de Traitement des Lignes

1. Non Contrôlable :Dans SQL et PL/SQL, il n'est pas possible de contrôler l'ordre dans lequel les lignes sont traitées par une instruction SQL. Par conséquent, il est impossible de créer un trigger qui dépend spécifiquement de l'ordre des lignes traitées par une opération comme `INSERT`, `UPDATE` ou `DELETE`.

- Exemple :Supposons que vous ayez un trigger qui doit mettre à jour une autre table en fonction des valeurs insérées dans une table principale. Même si les lignes dans la table principale sont insérées dans un certain ordre, SQL ne garantit pas cet ordre lors de l'exécution de l'instruction `INSERT`.

2. Triggers en Cascade :Un trigger peut déclencher l'exécution d'un autre trigger, créant ainsi une cascade de déclenchements. Cela signifie qu'un trigger peut provoquer l'exécution d'un autre trigger sur la même table ou sur une autre table.

Exemple :Supposons que vous ayez deux triggers sur la table `Employes` :

```
CREATE OR REPLACE TRIGGER trg1
AFTER INSERT ON Employes
FOR EACH ROW
BEGIN
    -- Actions à exécuter après l'insertion
END;
/

CREATE OR REPLACE TRIGGER trg2
AFTER INSERT ON Employes
FOR EACH ROW
BEGIN
    -- Actions à exécuter après l'insertion
END;
/
```

Si une ligne est insérée dans la table `Employes`, les deux triggers `trg1` et `trg2` seront déclenchés après l'insertion.

3. Limitation des Triggers en Cascade : Dans Oracle, il est possible d'avoir jusqu'à 32 niveaux de triggers en cascade pour une même opération. Cela signifie qu'un trigger peut en déclencher un autre, qui peut à son tour en déclencher un autre, et ainsi de suite, jusqu'à ce que la limite de 32 niveaux soit atteinte.

Utilisation et Impact

- Utilisation : Les triggers en cascade sont utiles pour automatiser des actions complexes qui nécessitent des mises à jour ou des vérifications sur plusieurs tables après une opération sur une table principale.

- Impact : Il est crucial de bien concevoir et tester les triggers en cascade pour éviter les boucles infinies ou les comportements imprévus dans la base de données. Une gestion appropriée des transactions et des validations est nécessaire pour maintenir la cohérence des données.

Comprendre l'ordre de traitement des lignes et les triggers en cascade est essentiel pour concevoir efficacement des systèmes de base de données robustes et fiables. Bien que l'ordre des lignes ne puisse pas être contrôlé, les triggers en cascade offrent une manière puissante d'automatiser les actions et les vérifications au niveau de la base de données, en assurant la cohérence des données et en améliorant la gestion des transactions.

Activation et Désactivation des Triggers dans Oracle

Les triggers dans Oracle peuvent être activés ou désactivés selon les besoins de gestion des données et des opérations sur la base de données. Voici une explication détaillée avec des exemples :

1. Activation par défaut : Lorsqu'un trigger est créé, il est activé par défaut. Cela signifie qu'il est prêt à être déclenché en réponse aux événements SQL correspondants (INSERT, UPDATE, DELETE).

2. Désactivation d'un Trigger : Un trigger peut être désactivé pour empêcher son exécution temporairement. Les raisons courantes pour désactiver un trigger incluent :
   - Référence à un objet non disponible.
   - Chargement rapide d'un grand volume de données sans déclencher les actions associées aux triggers.
   - Rechargement de données déjà contrôlées.

   Pour désactiver un trigger, utilisez l'instruction `ALTER TRIGGER` avec l'option `DISABLE` :

```
ALTER TRIGGER nomtrigger DISABLE;
```

   Par exemple, si vous avez un trigger nommé `trg_update_nbemp`, vous pouvez le désactiver :

```
ALTER TRIGGER trg_update_nbemp DISABLE;
```

3. Désactivation de tous les Triggers d'une Table : Vous pouvez désactiver tous les triggers associés à une table spécifique en utilisant la commande suivante :

```
ALTER TABLE nomtable DISABLE ALL TRIGGERS;
```

   Par exemple, pour désactiver tous les triggers sur la table `Employes` :

```
ALTER TABLE Employes DISABLE ALL TRIGGERS;
```

4. Réactivation d'un Trigger : Pour réactiver un trigger précédemment désactivé, utilisez l'instruction `ENABLE` :

```
ALTER TRIGGER nomtrigger ENABLE;
```

Par exemple, pour réactiver le trigger `trg_update_nbemp` :

```
ALTER TRIGGER trg_update_nbemp ENABLE;
```

5. Réactivation de tous les Triggers d'une Table : Pour réactiver tous les triggers associés à une table spécifique, utilisez la commande suivante :

```
ALTER TABLE nomtable ENABLE ALL TRIGGERS;
```

Par exemple, pour réactiver tous les triggers sur la table `Employes` :

```
ALTER TABLE Employes ENABLE ALL TRIGGERS;
```

Utilisation et Impact

- Utilisation : La capacité à activer ou désactiver des triggers est particulièrement utile lors du chargement initial de données dans une base de données ou lors de la maintenance des schémas de base de données.

- Impact : Désactiver un trigger peut avoir un impact significatif sur le comportement attendu des opérations SQL. Il est important de planifier soigneusement l'activation et la désactivation des triggers pour éviter des incohérences ou des perturbations dans la base de données.

La gestion des triggers dans Oracle, y compris leur activation et leur désactivation, offre une flexibilité précieuse pour gérer efficacement les opérations sur la base de données. En comprenant comment activer et désactiver les triggers, les administrateurs de base de données peuvent optimiser les performances et assurer l'intégrité des données tout en facilitant la maintenance du système.

Recherche d'Information sur les Triggers dans Oracle

Dans Oracle, les informations sur les triggers définis dans la base de données sont stockées dans des vues systèmes spécifiques. Ces vues permettent aux utilisateurs d'interroger et de comprendre les détails des triggers existants dans le schéma de la base de données.

Voici un aperçu des principales vues système utilisées pour rechercher des informations sur les triggers :

Vues de la Métabase des Triggers

1. USER_TRIGGERS :
   - Cette vue contient des informations sur les triggers définis par l'utilisateur connecté.
   - Elle inclut les détails tels que le nom du trigger, le type d'événement déclencheur (INSERT, UPDATE, DELETE), le moment de l'exécution (BEFORE, AFTER), le type de trigger (ROW ou STATEMENT), et d'autres propriétés.

Exemple d'Utilisation :

```
SELECT trigger_name, trigger_type, triggering_event, table_name
FROM user_triggers;
```

2. ALL_TRIGGERS :
   - Cette vue contient des informations sur tous les triggers accessibles à l'utilisateur connecté, y compris ceux définis par d'autres utilisateurs.
   - Les objets inclus dans cette vue sont limités aux objets auxquels l'utilisateur connecté a accès.

Exemple d'Utilisation :

```
SELECT trigger_name, trigger_type, triggering_event, table_name
FROM all_triggers;
```

3. DBA_TRIGGERS :
   - Cette vue contient des informations sur tous les triggers de la base de données, indépendamment de l'utilisateur qui les a définis.
   - Elle est accessible uniquement par les utilisateurs disposant du privilège DBA (administrateur de la base de données).

Exemple d'Utilisation :

```
SELECT trigger_name, trigger_type, triggering_event, table_name
FROM dba_triggers;
```

Utilisation des Vues

- Interrogation des Informations : Ces vues permettent aux administrateurs de base de données et aux développeurs d'interroger les détails des triggers pour comprendre leur fonctionnement, leur impact sur les opérations SQL, et pour la gestion générale des bases de données.

- Gestion et Maintenance : En utilisant ces vues, les administrateurs peuvent identifier les triggers existants, vérifier leur état (activé ou désactivé), et effectuer des actions de gestion telles que l'activation, la désactivation ou la suppression des triggers selon les besoins.

Les vues `USER_TRIGGERS`, `ALL_TRIGGERS` et `DBA_TRIGGERS` sont des outils essentiels pour rechercher des informations détaillées sur les triggers dans Oracle. Elles offrent une visibilité complète sur les déclencheurs définis dans le système, permettant ainsi une gestion efficace et une maintenance proactive des bases de données Oracle.

Gestion des Exceptions dans les Triggers Oracle

Lorsque vous travaillez avec des triggers dans Oracle, il est crucial de gérer les exceptions de manière appropriée pour assurer l'intégrité des données et éviter les situations imprévues. Voici une explication détaillée et des exemples sur la gestion des exceptions dans les triggers :

Principe de Gestion des Exceptions

1. Rollback en Cas d'Erreur : Lorsqu'une erreur se produit pendant l'exécution d'un trigger, Oracle effectue automatiquement un rollback de toutes les modifications apportées par le trigger ainsi que par l'instruction qui l'a déclenché. Cela permet de maintenir la cohérence des données dans la base de données.

2. Introduction des Exceptions :
   - Une exception se produit lorsqu'une erreur survient pendant l'exécution d'un bloc PL/SQL, y compris dans un trigger.
   - Les exceptions peuvent être prédéfinies (comme les erreurs standard Oracle) ou définies par l'utilisateur à l'aide de blocs `EXCEPTION`.

3. Bloc EXCEPTION :
   - Un bloc PL/SQL peut inclure un bloc `EXCEPTION` qui gère les erreurs potentielles avec des clauses `WHEN`.
   - La clause `WHEN OTHERS THEN ROLLBACK;` est souvent utilisée pour capturer toutes les erreurs non prévues et déclencher un rollback pour annuler les modifications potentiellement invalides.

Exemple 1 : Gestion d'une Exception Prédéfinie

```
CREATE OR REPLACE TRIGGER trg_example
BEFORE INSERT ON ma_table
FOR EACH ROW
DECLARE
    -- Déclaration d'une exception personnalisée
    custom_exception EXCEPTION;
    PRAGMA EXCEPTION_INIT(custom_exception, -20001); -- Associer un code d'erreur
BEGIN
    -- Votre logique métier
    IF :NEW.colonne <= 0 THEN
        -- Lever une exception personnalisée
        RAISE custom_exception;
    END IF;
EXCEPTION
    WHEN custom_exception THEN
        -- Gérer l'exception personnalisée
        DBMS_OUTPUT.PUT_LINE('Erreur : Valeur invalide pour colonne.');
        :NEW.colonne := DEFAULT; -- Rétablir la valeur par défaut ou faire autre action
    WHEN OTHERS THEN
        -- Gérer toutes les autres exceptions
        ROLLBACK; -- Annuler toutes les modifications
        DBMS_OUTPUT.PUT_LINE('Erreur non gérée : ' || SQLERRM);
END;
/
```

Dans cet exemple :
- Le trigger `trg_example` est déclenché avant chaque insertion dans `ma_table`.
- Il vérifie si la valeur de la colonne `colonne` est positive. Si ce n'est pas le cas, il lève une exception personnalisée `custom_exception`.
- Si une exception est levée, le bloc `EXCEPTION` capture cette exception et effectue des actions appropriées comme l'affichage d'un message et la gestion du rollback.

Exemple 2 : Gestion Générale des Exceptions

```
CREATE OR REPLACE TRIGGER trg_general_exception
BEFORE INSERT ON ma_table
FOR EACH ROW
BEGIN
    -- Votre logique métier
    IF :NEW.date_naissance > SYSDATE THEN
        RAISE_APPLICATION_ERROR(-20002, 'Date de naissance future non autorisée.');
    END IF;
EXCEPTION
    WHEN VALUE_ERROR THEN
        -- Gérer une exception spécifique
        DBMS_OUTPUT.PUT_LINE('Erreur de valeur : ' || SQLERRM);
    WHEN OTHERS THEN
        -- Gérer toutes les autres exceptions
        ROLLBACK; -- Annuler toutes les modifications
        DBMS_OUTPUT.PUT_LINE('Erreur non gérée : ' || SQLERRM);
END;
/
```

Dans cet exemple :
- Le trigger `trg_general_exception` est déclenché avant chaque insertion dans `ma_table`.
- Il vérifie si la date de naissance (`date_naissance`) est postérieure à la date actuelle (`SYSDATE`). Si oui, il lève une exception prédéfinie (`RAISE_APPLICATION_ERROR`) avec un code d'erreur spécifique.
- Le bloc `EXCEPTION` capture l'exception spécifique `VALUE_ERROR` ainsi que toutes les autres exceptions non prévues, déclenchant un rollback pour annuler les modifications en cours et affiche un message d'erreur correspondant.

Exceptions prédéfinies

Les exceptions prédéfinies sont des erreurs standard qui peuvent être rencontrées lors de l'exécution de blocs PL/SQL, y compris dans des triggers Oracle. Voici quelques-unes des exceptions prédéfinies les plus couramment utilisées et leurs significations :

1. NO_DATA_FOUND :
   - Cette exception est générée lorsque l'instruction `SELECT INTO` ne retourne aucune ligne.
   - Par exemple, si une requête est censée récupérer une valeur d'une table dans une variable à l'aide de `SELECT INTO`, mais qu'aucune ligne ne correspond aux critères de la requête, cette exception est levée.

Exemple :

```
DECLARE
    v_salaire employees.salary%TYPE;
BEGIN
    SELECT salary INTO v_salaire FROM employees WHERE employee_id = 1000;
    -- Si aucun employé avec ID 1000 n'est trouvé, NO_DATA_FOUND est levée
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Aucun employé trouvé avec ID 1000.');
END;
```

2. DUP_VAL_ON_INDEX :

- Cette exception survient lorsqu'une tentative est faite d'insérer une nouvelle ligne dans une table avec une valeur déjà existante dans une colonne définie comme index unique (ou clé primaire).
- Oracle garantit l'unicité des valeurs dans les colonnes avec une contrainte d'index unique. Si une tentative d'insertion viole cette contrainte, DUP_VAL_ON_INDEX est levée.

Exemple :

```
CREATE TABLE students (
    student_id NUMBER PRIMARY KEY,
    student_name VARCHAR2(50)
);

INSERT INTO students VALUES (1, 'Alice');
INSERT INTO students VALUES (2, 'Bob');
INSERT INTO students VALUES (1, 'Carol'); -- Provoque une exception DUP_VAL_ON_INDEX
```

3. ZERO_DIVIDE :

- Cette exception est générée lorsqu'une tentative est faite de diviser un nombre par zéro.
- Dans les calculs arithmétiques, diviser par zéro est une opération invalide et provoque cette exception.

Exemple :

```
DECLARE
    v_nombre NUMBER := 10;
    v_diviseur NUMBER := 0;
    v_resultat NUMBER;
BEGIN
    v_resultat := v_nombre / v_diviseur; -- Provoque une exception ZERO_DIVIDE
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Erreur : Division par zéro.');
END;
```

Exceptions Non SQL (Définies par l'Utilisateur)

Les exceptions non SQL sont des erreurs personnalisées définies par l'utilisateur pour gérer des scénarios spécifiques qui ne sont pas couverts par les exceptions SQL prédéfinies. Pour définir une exception non SQL, vous devez la déclarer explicitement avec un nom et un numéro d'erreur associé à l'aide de PRAGMA EXCEPTION_INIT.

Exemple :

```
DECLARE
    -- Déclaration de l'exception personnalisée
    my_custom_exception EXCEPTION;
    PRAGMA EXCEPTION_INIT(my_custom_exception, -20001); -- Numéro d'erreur personnalisé

    v_employee_name employees.employee_name%TYPE;
BEGIN
    SELECT employee_name INTO v_employee_name FROM employees WHERE employee_id = 1000;
EXCEPTION
    WHEN my_custom_exception THEN
        DBMS_OUTPUT.PUT_LINE('Erreur personnalisée : Valeur spécifique non trouvée.');
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Aucun employé trouvé avec ID 1000.');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Erreur non gérée : ' || SQLERRM);
END;
```

Dans cet exemple :

- my_custom_exception est définie comme une exception personnalisée avec le numéro d'erreur -20001.
- Lorsque le SELECT INTO ne trouve pas de correspondance pour employee_id = 1000, l'exception NO_DATA_FOUND est levée.
- Si une valeur spécifique attendue n'est pas trouvée (définie par my_custom_exception), l'exception personnalisée est gérée spécifiquement.

Syntaxe SQL des exceptions

Exceptions SQL

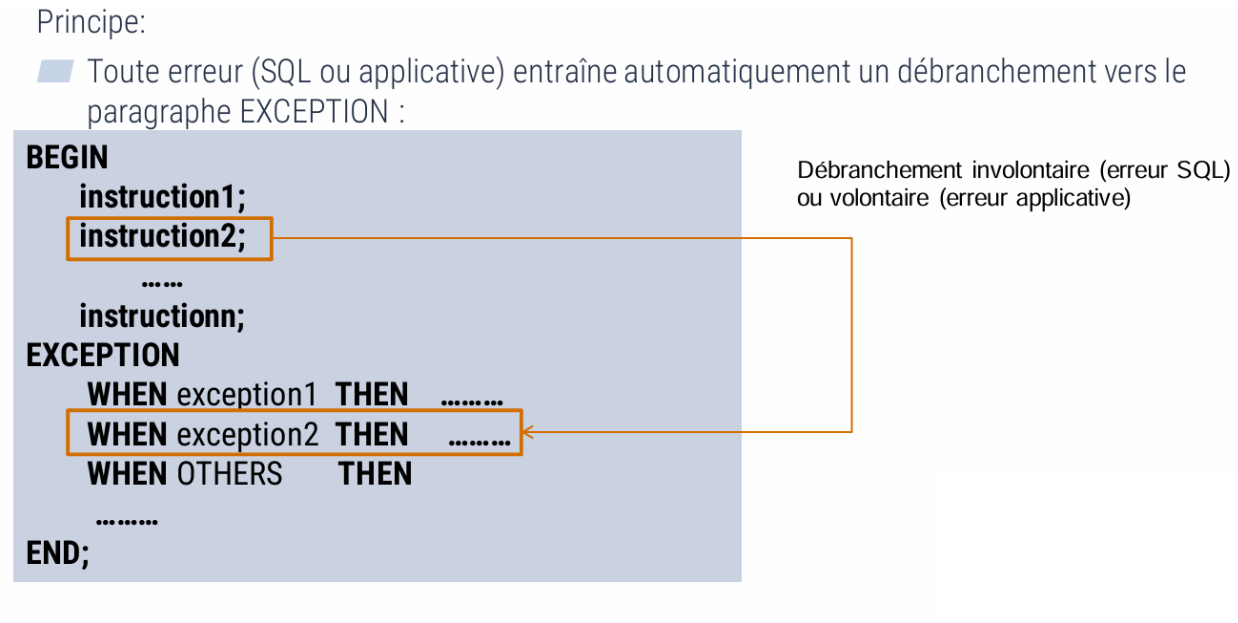
- Exceptions applicatives
  - Déclaration sans n° erreur

```
nomerreur    EXCEPTION;
```

Exemple

```
DECLARE
enfant_sans_parent  EXCEPTION;
PRAGMA EXCEPTION_INIT(enfant_sans_parent,-2291);
BEGIN
    INSERT INTO fils VALUES ( ……. );
    EXCEPTION
        WHEN enfant_sans_parent THEN
          ……..
        WHEN OTHERS THEN ……..
END;
```

```
DECLARE
  NomErreur EXCEPTION;
  PRAGMA EXCEPTION_INIT(NomErreur, -20001);
BEGIN
   ….
   RAISE_APPLICATION_ERROR(-20001, 'Une erreur s est produit lors du traitement');
   EXCEPTION
       WHEN NomErreur THEN
         DBMS_OUTPUT.PUT_LINE(' Erreur : ' || SQLCODE || ' - ' || sqlerrm );
       WHEN OTHERS THEN
         DBMS_OUTPUT.PUT_LINE('Erreur : ' || SQLCODE || ' - ' || sqlerrm);
END;
```



La gestion des exceptions dans les triggers Oracle est essentielle pour garantir la fiabilité des opérations effectuées sur la base de données. En utilisant des blocs `EXCEPTION` appropriés avec des clauses `WHEN` et en levant des exceptions personnalisées quand nécessaire, vous pouvez assurer que les transactions sont sécurisées et que les données restent cohérentes même en cas d'erreurs imprévues.

Curseurs

En PL/SQL, la sélection mono-ligne avec `SELECT ... INTO` est utilisée pour assigner directement les résultats d'une requête SQL à des variables spécifiées. Cela permet de récupérer une seule ligne de résultats dans des variables scalaires ou dans une variable de type `ROWTYPE`.

Utilisation de SELECT ... INTO :

1. Syntaxe de base avec INTO :

```
SELECT colonne1, colonne2, ...
INTO variable1, variable2, ...
FROM table
WHERE condition;
```

Dans cette structure :

- `SELECT colonne1, colonne2, ...` spécifie les colonnes à sélectionner.
- `INTO variable1, variable2, ...` assigne les valeurs de ces colonnes aux variables spécifiées.
- `FROM table` indique la table à partir de laquelle les données sont sélectionnées.
- `WHERE condition` est une condition facultative pour filtrer les lignes.

2. Exemples d'utilisation :

Exemple 1 :

```
DECLARE
    vnom VARCHAR2(50);
    vadresse VARCHAR2(100);
    vtel VARCHAR2(15);
    matricule NUMBER := 1001;
BEGIN
    SELECT nom, adresse, tel
    INTO vnom, vadresse, vtel
    FROM Employés
    WHERE NSS = matricule;

    DBMS_OUTPUT.PUT_LINE('Nom : ' || vnom);
    DBMS_OUTPUT.PUT_LINE('Adresse : ' || vadresse);
    DBMS_OUTPUT.PUT_LINE('Téléphone : ' || vtel);
END;
/
```

Explication :

- Dans cet exemple, les variables `vnom`, `vadresse` et `vtel` sont utilisées pour stocker respectivement le nom, l'adresse et le numéro de téléphone d'un employé dont le NSS (Numéro de Sécurité Sociale) est égal à `matricule` (dans ce cas, 1001).
- La requête `SELECT ... INTO` récupère exactement une seule ligne de résultats correspondant à la condition spécifiée (`WHERE NSS = matricule`) et assigne les valeurs des colonnes `nom`, `adresse` et `tel` aux variables correspondantes.

Exemple 2 : Utilisation de `ROWTYPE` :

```
DECLARE
    vempl Employés%ROWTYPE;
    matricule NUMBER := 1002;
BEGIN
    SELECT *
    INTO vempl
    FROM Employés
    WHERE NSS = matricule;

    DBMS_OUTPUT.PUT_LINE('Nom Employé : ' || vempl.nom);
    DBMS_OUTPUT.PUT_LINE('Adresse : ' || vempl.adresse);
    DBMS_OUTPUT.PUT_LINE('Service : ' || vempl.nom_service);
END;
/
```

Explication :

- Ici, `vempl` est une variable de type `Employés%ROWTYPE`, ce qui signifie qu'elle contiendra toutes les colonnes de la table `Employés`.
- La requête `SELECT * INTO vempl` sélectionne toutes les colonnes de la table `Employés` pour l'employé dont le NSS est égal à `matricule` (dans ce cas, 1002).
- Les valeurs de chaque colonne sélectionnée sont assignées aux champs correspondants de la structure `vempl`.
- Ensuite, les informations sont affichées à l'aide de `DBMS_OUTPUT.PUT_LINE`.

L'utilisation de `SELECT ... INTO` est une technique courante en PL/SQL pour récupérer des données spécifiques d'une requête SQL dans des variables. Cela permet d'interagir efficacement avec la base de données Oracle en assignant directement les résultats de la requête à des variables déclarées localement dans un bloc PL/SQL.

Sélection de plusieurs lignes

En PL/SQL, les curseurs sont des structures utilisées pour traiter et manipuler des résultats de requêtes SQL ligne par ligne. Ils sont essentiels lorsque vous avez besoin de traiter plusieurs lignes de résultats dans une procédure ou une fonction PL/SQL.

Principe des curseurs

1. Obligatoire pour sélectionner plusieurs lignes :
Les curseurs sont nécessaires lorsque vous devez traiter plusieurs lignes de résultats d'une requête SQL.

2. Zone mémoire partagée (SGA) :
Les résultats des requêtes SQL sont stockés temporairement dans la mémoire partagée (SGA) du système de base de données Oracle.

3. Curseur implicite :
Lorsque vous exécutez une requête SQL simple dans PL/SQL sans utiliser explicitement un curseur, un curseur implicite est automatiquement créé pour gérer les résultats. Par exemple :

```
SELECT * FROM table WHERE condition;
```

Ici, un curseur implicite est utilisé pour gérer les résultats de la requête.

4. Curseur explicite :
Pour un contrôle plus précis sur la gestion des résultats, vous pouvez déclarer et utiliser un curseur explicite.

Curseur explicite

Un curseur explicite est défini de manière explicite dans la section de déclaration d'un bloc PL/SQL. Voici les étapes pour utiliser un curseur explicite :

- Déclaration du curseur :
Avant d'utiliser un curseur, vous devez le déclarer. Cela inclut spécifier la requête SQL que le curseur va exécuter. Par exemple :

```
DECLARE
    CURSOR nom_curseur IS
        SELECT colonne1, colonne2, ...
        FROM table
        WHERE condition;
```

- Ouverture du curseur :

Une fois déclaré, le curseur est ouvert pour exécuter la requête SQL et charger les résultats dans la mémoire partagée (SGA). Cela se fait avec l'instruction `OPEN`.

```
OPEN nom_curseur;
```

- Sélection d'une ligne :

Pour accéder aux données du curseur, vous utilisez l'instruction `FETCH`. Chaque `FETCH` récupère une ligne de résultats dans le programme PL/SQL.

```
FETCH nom_curseur INTO variable1, variable2, ...;
```

- Traitement des lignes :

Vous pouvez itérer à travers les résultats à l'aide d'une boucle `LOOP` et de l'instruction `FETCH`. Cela vous permet de manipuler chaque ligne de résultats individuellement.

```
LOOP
    FETCH nom_curseur INTO variable1, variable2, ...;
    EXIT WHEN nom_curseur%NOTFOUND;  -- Condition de sortie de la boucle
    -- Traitements sur les variables récupérées
END LOOP;
```

- Fermeture du curseur :

Une fois que vous avez fini de traiter les résultats du curseur, vous devez le fermer pour libérer l'espace mémoire SGA qu'il occupe.

```
CLOSE nom_curseur;
```

Exemple :

```
DECLARE
    CURSOR cur_employes IS
        SELECT nom, prenom, salaire
        FROM Employes
        WHERE service = 'Informatique';

    v_nom Employes.nom%TYPE;
    v_prenom Employes.prenom%TYPE;
    v_salaire Employes.salaire%TYPE;
BEGIN
    OPEN cur_employes;

    LOOP
        FETCH cur_employes INTO v_nom, v_prenom, v_salaire;
        EXIT WHEN cur_employes%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('Nom : ' || v_nom || ', Prénom : ' || v_prenom || ', Salaire

        -- Autres traitements sur les données récupérées

    END LOOP;

    CLOSE cur_employes;
END;
/
```

Explication de l'exemple :
- Un curseur `cur_employes` est déclaré pour sélectionner les noms, prénoms et salaires des employés travaillant dans le service "Informatique".
- Le curseur est ouvert avec `OPEN cur_employes;`, ce qui exécute la requête SQL et initialise le curseur pour la récupération des résultats.
- À l'intérieur d'une boucle `LOOP`, chaque ligne de résultats est récupérée avec `FETCH cur_employes INTO ...` et affichée à l'écran avec `DBMS_OUTPUT.PUT_LINE`.
- La boucle continue à itérer jusqu'à ce que toutes les lignes soient récupérées (`EXIT WHEN cur_employes%NOTFOUND;`).
- Enfin, le curseur est fermé avec `CLOSE cur_employes;` pour libérer les ressources occupées dans la mémoire SGA.

En résumé, les curseurs explicites offrent un contrôle plus fin sur la manipulation des résultats de requêtes SQL en PL/SQL, en permettant de traiter ligne par ligne les résultats récupérés dans des variables locales.

En PL/SQL, lorsqu'on travaille avec des curseurs explicites pour manipuler les résultats de requêtes SQL, il est courant d'utiliser les variables systèmes associés aux curseurs pour contrôler leur comportement et effectuer des traitements conditionnels. Voici les variables système les plus utilisées avec les curseurs :

Variables système des curseurs

1. Curseur%FOUND
   - Description : Cette variable booléenne indique si le curseur a récupéré au moins une ligne de résultats.
   - Utilisation : Elle est souvent utilisée pour vérifier si le curseur a encore des lignes à traiter.

```
OPEN cur_employes;
LOOP
    FETCH cur_employes INTO v_nom, v_prenom, v_salaire;
    EXIT WHEN cur_employes%NOTFOUND;
    -- Traitement sur les données récupérées
END LOOP;
CLOSE cur_employes;
```

2. Curseur%NOTFOUND
   - Description : Cette variable booléenne est l'opposé de `%FOUND`. Elle indique si le curseur n'a pas récupéré de ligne supplémentaire lors du dernier `FETCH`.
   - Utilisation : Utile pour sortir d'une boucle lorsque toutes les lignes ont été récupérées.
   - Exemple : Utilisé avec `%FOUND` dans l'exemple précédent.

3. Curseur%COUNT
   - Description : Cette variable numérique retourne le nombre de lignes déjà retournées par le curseur depuis son ouverture.
   - Utilisation : Permet de connaître le nombre de lignes déjà traitées.

```
OPEN cur_employes;
FETCH cur_employes INTO v_nom, v_prenom, v_salaire;
DBMS_OUTPUT.PUT_LINE('Nombre de lignes traitées : ' || cur_employes%COUNT);
CLOSE cur_employes;
```

4. Curseur%ISOPEN
   - Description : Cette variable booléenne indique si le curseur est actuellement ouvert (`TRUE`) ou fermé (`FALSE`).
   - Utilisation : Permet de vérifier si le curseur est encore actif avant de l'utiliser.

```
IF cur_employes%ISOPEN THEN
    CLOSE cur_employes;
END IF;
```

Exemple complet

Voici un exemple complet utilisant ces variables système avec un curseur explicite :

```
DECLARE
    CURSOR cur_employes IS
        SELECT nom, prenom, salaire
        FROM Employes
        WHERE service = 'Informatique';

    v_nom Employes.nom%TYPE;
    v_prenom Employes.prenom%TYPE;
    v_salaire Employes.salaire%TYPE;
BEGIN
    OPEN cur_employes;

    LOOP
        FETCH cur_employes INTO v_nom, v_prenom, v_salaire;
        EXIT WHEN cur_employes%NOTFOUND;

        -- Traitement sur les données récupérées
        DBMS_OUTPUT.PUT_LINE('Nom : ' || v_nom || ', Prénom : ' || v_prenom || ', Salaire

        -- Exemple d'utilisation de %COUNT
        DBMS_OUTPUT.PUT_LINE('Nombre de lignes traitées : ' || cur_employes%COUNT);
    END LOOP;

    CLOSE cur_employes;
END;
/
```

s` est déclaré pour sélectionner les noms, prénoms et salaires des employés travaillant dans le service "Informatique".
- Le curseur est ouvert avec `OPEN cur_employes;`, ce qui exécute la requête SQL et initialise le curseur pour la récupération des résultats.
- À l'intérieur d'une boucle `LOOP`, chaque ligne de résultats est récupérée avec `FETCH cur_employes INTO ...`.
- `%FOUND` est utilisé dans `EXIT WHEN cur_employes%NOTFOUND;` pour sortir de la boucle lorsque toutes les lignes ont été récupérées.
- `%COUNT` est utilisé pour afficher le nombre de lignes déjà traitées.
- Le curseur est fermé avec `CLOSE cur_employes;` pour libérer les ressources occupées dans la mémoire SGA.

Procédures et les fonctions stockées en PL/SQL

Les procédures et les fonctions stockées en PL/SQL sont des éléments essentiels pour encapsuler la logique métier dans la base de données Oracle. Voici une explication détaillée avec des exemples pour chaque type.

Procédures Stockées

Les procédures stockées sont des blocs de code PL/SQL qui peuvent contenir des instructions SQL et PL/SQL. Elles sont utilisées pour effectuer des tâches spécifiques et peuvent être appelées depuis d'autres programmes ou directement depuis SQL*Plus.

Déclaration d'une procédure stockée :

```
CREATE OR REPLACE PROCEDURE nom_procedure (
    parametre1 IN type1,
    parametre2 OUT type2,
    parametre3 IN OUT type3
) AS
    -- Déclarations des variables locales et curseurs
BEGIN
    -- Instructions SQL et PL/SQL
    INSERT INTO table_exemple (colonne1, colonne2) VALUES (parametre1, parametre2);

    -- Gestion des exceptions
    EXCEPTION
        WHEN others THEN
            DBMS_OUTPUT.PUT_LINE('Erreur : ' || SQLERRM);
END;
/
```

Exemple d'appel d'une procédure depuis SQL*Plus :

```
EXECUTE nom_procedure('valeur1', 'valeur2', 'valeur3');
```

Fonctions Stockées

Les fonctions stockées sont similaires aux procédures, mais elles retournent une seule valeur. Elles peuvent être utilisées dans des requêtes SQL, des expressions PL/SQL ou dans d'autres fonctions ou procédures.
Déclaration d'une fonction stockée :

```
CREATE OR REPLACE FUNCTION nom_fonction (
    parametre1 IN type1,
    parametre2 OUT type2
) RETURN type_resultat IS
    -- Déclarations de variables locales et curseurs
    variable type_resultat;
BEGIN
    -- Instructions SQL et PL/SQL
    SELECT colonne INTO variable FROM table_exemple WHERE condition;

    -- Retourner le résultat
    RETURN variable;

    -- Gestion des exceptions
    EXCEPTION
        WHEN no_data_found THEN
            variable := null;
            RETURN variable;
END;
/
```

Exemple d'appel d'une fonction depuis SQL :

```
SELECT nom_fonction('valeur1', 'valeur2') FROM dual;

-- Utilisation dans une requête
SELECT nom, prenom FROM employes WHERE salaire > nom_fonction('parametre1', 'parametre2');
```

Les procédures stockées sont utilisées pour effectuer des actions qui ne nécessitent pas de retourner une valeur spécifique, tandis que les fonctions stockées sont conçues pour calculer et retourner des résultats spécifiques. Elles offrent une encapsulation efficace de la logique métier directement dans la base de données Oracle, facilitant ainsi la réutilisation et la maintenance du code.
Voici un exercice qui combine plusieurs concepts de PL/SQL, y compris les triggers, les boucles, les structures conditionnelles, les curseurs, les fonctions et procédures stockées, l'utilisation des fonctions du DBMS_OUTPUT, la gestion des INSERT, UPDATE, DELETE, ainsi que la gestion des exceptions.

Exercice : Gestion d'une base de données d'employés

Considérons une base de données simple pour gérer les informations des employés et des départements. L'objectif est de créer des procédures et des fonctions pour gérer l'insertion, la mise à jour, et la suppression des employés, en utilisant des triggers pour mettre à jour automatiquement les informations sur les départements.

```
CREATE TABLE Employes (
    matricule NUMBER PRIMARY KEY,
    nom VARCHAR2(100),
    salaire NUMBER(8,2),
    code_departement VARCHAR2(10)
);

CREATE TABLE Departements (
    code_departement VARCHAR2(10) PRIMARY KEY,
    nom VARCHAR2(100),
    nb_employes NUMBER
);
```

1. Création des tables :
   - Table `Employes` : contient les informations sur les employés (matricule, nom, salaire, département).
   - Table `Departements` : contient les informations sur les départements (code, nom, nombre d'employés).

2. Procédures et Fonctions :
   - Procédure pour l'insertion d'un employé :Ajoute un nouvel employé à la table `Employes` et met à jour le nombre d'employés dans le département correspondant dans la table `Departements`.

```
CREATE OR REPLACE PROCEDURE ajouter_employe (
    p_matricule IN NUMBER,
    p_nom IN VARCHAR2,
    p_salaire IN NUMBER,
    p_code_dep IN VARCHAR2
) AS
BEGIN
    INSERT INTO Employes(matricule, nom, salaire, code_departement)
    VALUES(p_matricule, p_nom, p_salaire, p_code_dep);

    UPDATE Departements
    SET nb_employes = nb_employes + 1
    WHERE code_departement = p_code_dep;

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Employé ajouté avec succès.');
EXCEPTION
    WHEN others THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Erreur lors de l\'ajout de l\'employé : ' || SQLERRM);
END;
/
```

Fonction pour la mise à jour du salaire d'un employé :

```
CREATE OR REPLACE FUNCTION mettre_a_jour_salaire (
    p_matricule IN NUMBER,
    p_nouveau_salaire IN NUMBER
) RETURN BOOLEAN IS
BEGIN
    UPDATE Employes
    SET salaire = p_nouveau_salaire
    WHERE matricule = p_matricule;

    -- Obtenir le code département de l'employé
    DECLARE
        v_code_dep Employes.code_departement%TYPE;
    BEGIN
        SELECT code_departement INTO v_code_dep
        FROM Employes
        WHERE matricule = p_matricule;

        -- Mettre à jour le nombre d'employés dans le département
        UPDATE Departements
        SET nb_employes = nb_employes - 1
        WHERE code_departement = v_code_dep;
```

```
        UPDATE Departements
        SET nb_employes = nb_employes + 1
        WHERE code_departement = v_code_dep;

        COMMIT;
        DBMS_OUTPUT.PUT_LINE('Salaire de l\'employé mis à jour avec succès.');
        RETURN TRUE;
    EXCEPTION
        WHEN no_data_found THEN
            DBMS_OUTPUT.PUT_LINE('Employé non trouvé.');
            RETURN FALSE;
        WHEN others THEN
            ROLLBACK;
            DBMS_OUTPUT.PUT_LINE('Erreur lors de la mise à jour du salaire : ' || SQLERRM)
            RETURN FALSE;
    END;
END;
/
```

3. Trigger pour mettre à jour le nombre d'employés lors de l'insertion d'un nouvel employé :

```
CREATE OR REPLACE PROCEDURE supprimer_employe (
    p_matricule IN NUMBER
) AS
    v_code_dep Employes.code_departement%TYPE;
BEGIN
    -- Obtenir le code département de l'employé
    SELECT code_departement INTO v_code_dep
    FROM Employes
    WHERE matricule = p_matricule;

    DELETE FROM Employes
    WHERE matricule = p_matricule;

    UPDATE Departements
    SET nb_employes = nb_employes - 1
    WHERE code_departement = v_code_dep;

    COMMIT;
    DBMS_OUTPUT.PUT_LINE('Employé supprimé avec succès.');
EXCEPTION
    WHEN no_data_found THEN
        DBMS_OUTPUT.PUT_LINE('Employé non trouvé.');
    WHEN others THEN
        ROLLBACK;
        DBMS_OUTPUT.PUT_LINE('Erreur lors de la suppression de l\'employé : ' || SQLERRM);
END;
/
```

- Trigger pour mettre à jour le nombre d'employés lors de la suppression d'un employé :

```
CREATE OR REPLACE TRIGGER trg_delete_employe
BEFORE DELETE ON Employes
FOR EACH ROW
BEGIN
    UPDATE Departements
    SET nb_employes = nb_employes - 1
    WHERE code_departement = :old.code_departement;
    DBMS_OUTPUT.PUT_LINE('Trigger : Nombre d\'employés dans le département mis à jour.');
EXCEPTION
    WHEN others THEN
        DBMS_OUTPUT.PUT_LINE('Erreur dans le trigger : ' || SQLERRM);
END;
/
```

4. Utilisation des procédures et fonctions :

```
-- Appel de la procédure pour ajouter un nouvel employé
EXECUTE ajouter_employe(101, 'John Doe', 5000, 'IT');

-- Appel de la fonction pour mettre à jour le salaire d'un employé
DECLARE
    v_success BOOLEAN;
BEGIN
    v_success := mettre_a_jour_salaire(101, 6000);
    IF v_success THEN
        DBMS_OUTPUT.PUT_LINE('Salaire mis à jour avec succès.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Échec de la mise à jour du salaire.');
    END IF;
END;
/

-- Appel de la procédure pour supprimer un employé
EXECUTE supprimer_employe(101);
```

Optimisation des requêtes SQL

Pour optimiser les requêtes SQL et réduire leur temps d'exécution, il est essentiel de comprendre les étapes de traitement d'une requête, les plans d'exécution, l'algèbre relationnelle sous-jacente, ainsi que les techniques spécifiques d'optimisation. Voici une explication détaillée avec des exemples pour chaque aspect pertinent :

Étapes de traitement d'une requête SQL

1. Analyse Syntaxique :Vérifie la syntaxe de la requête pour s'assurer qu'elle est correcte et valide.
2. Transformation en Plan d'Évaluation :La requête SQL est transformée en un plan d'évaluation, qui est représenté sous forme d'arbre d'opérations de l'algèbre relationnelle. Chaque opération (sélection, jointure, projection, etc.) dans le plan est associée à un algorithme spécifique.
3. Optimisation du Plan :Le SGBD évalue plusieurs plans alternatifs pour la requête et choisit celui qui est estimé le moins coûteux en termes de ressources (temps d'exécution, utilisation de l'index, utilisation du cache, etc.).
4. Exécution du Plan :Une fois le plan choisi, le SGBD exécute la requête et récupère les résultats.

Algèbre Relationnelle et Arbre Algébrique

L'algèbre relationnelle est un modèle formel permettant d'exprimer et de calculer des requêtes sur des bases de données relationnelles. Voici les points clés concernant l'arbre algébrique :

- Arbre Algébrique :Représente graphiquement le plan d'évaluation d'une requête SQL. Les relations de base sont les nœuds feuilles, les opérations (sélection, jointure, etc.) sont les nœuds internes, et le nœud racine est le résultat final de la requête.

Catalogues et Statistiques

Les catalogues contiennent des informations essentielles sur les relations et les indexes, nécessaires à l'optimisation des requêtes :
- Informations dans les Catalogues :Nombre de tuples, nombre de pages, histogrammes (pour la distribution des valeurs), hauteur des index, etc.

Les chemins d'accès déterminent comment le SGBD va accéder aux données pour exécuter une requête :
- Index B-Arbre :Correspond à des sélections sur des préfixes de la clé d'index.
  - Exemple : Index sur `(a, b, c)` correspond à `a = 5 AND b = 3`, mais pas à `b = 3`.
- Index à Hachage :Correspond à des sélections où chaque attribut de la clé d'index est spécifié.
  - Exemple : Index sur `(a, b, c)` correspond à `a = 5 AND b = 3 AND c = 5`.

Techniques d'Optimisation

Pour optimiser les requêtes SQL, plusieurs techniques sont utilisées :
- Expressions Algébriques :Transformation des requêtes pour obtenir la meilleure séquence d'opérations.
- Optimisation Basée sur les Règles (RBO) :Utilisation de règles prédéfinies pour choisir le plan d'exécution.
- Optimisation Basée sur les Coûts (CBO) :Estimation des coûts des différentes stratégies en fonction des statistiques et choix du plan le moins coûteux.

Exemple d'Optimisation

Considérons une requête simple pour trouver tous les employés d'un département particulier avec un salaire supérieur à une valeur donnée :

```sql
SELECT nom, salaire
FROM Employes
WHERE code_departement = 'IT' AND salaire > 5000;
```

Pour optimiser cette requête, le SGBD pourrait utiliser un index sur `(code_departement, salaire)` pour filtrer rapidement les employés du département 'IT' avec un salaire élevé, réduisant ainsi le temps d'accès aux données.

L'optimisation des requêtes SQL est cruciale pour améliorer les performances des bases de données. En comprenant les principes d'algèbre relationnelle, les chemins d'accès aux données et en utilisant des techniques comme l'optimisation basée sur les coûts, les développeurs et administrateurs de bases de données peuvent réduire efficacement le temps d'exécution des requêtes et améliorer la réactivité des systèmes.

Méthodes d’Optimisation Basées sur les Arbres Algébriques

Les méthodes d’optimisation basées sur les arbres algébriques visent à réorganiser les expressions de requêtes SQL pour minimiser le coût d'exécution. Voici un examen approfondi :

Problème de l'Ordre des Opérateurs

L'ordre dans lequel les opérateurs algébriques sont appliqués dans un arbre de requête peut avoir un impact significatif sur le temps d'exécution. Par exemple, l'application d'une sélection (σ) avant une jointure peut réduire le nombre de tuples sur lesquels la jointure est effectuée, ce qui peut économiser des ressources.

Exemple :Considérons une requête avec une sélection et une jointure :

```
SELECT *
FROM Etudiant
WHERE age > 20
JOIN Cours ON Etudiant.cours_id = Cours.id;
```

Optimiser cette requête impliquerait de placer la sélection `age > 20` avant la jointure pour réduire le volume de données jointes.

Coût des Opérateurs

Le coût des opérateurs varie en fonction du volume des données traitées. Par exemple, les jointures sur de grandes tables peuvent être coûteuses en termes de ressources CPU et d'E/S par rapport aux jointures sur des tables plus petites.

Opérateurs qui Diminuent le Volume des Données

Certains opérateurs, comme la restriction (σ) et la projection (π), réduisent le volume des données en limitant les lignes ou les colonnes traitées.

Exemple :Une requête avec projection et sélection :

```
SELECT nom, prénom
FROM Etudiant
WHERE age > 20;
```

La projection

La projection limitant les colonnes sélectionnées et la sélection limitant les lignes basées sur la condition `age > 20` réduisent le volume de données manipulées.

Restructuration Algébrique

La restructuration algébrique consiste à transformer une requête SQL en une forme équivalente mais plus efficace en utilisant des règles de réécriture prédéfinies.

Règles de Réécriture

Les règles de réécriture permettent de transformer une expression algébrique en une autre équivalente mais optimisée en exploitant les propriétés algébriques des opérateurs.

Exemple :Réécrire une expression avec une sélection et une jointure :

```
SELECT *
FROM Etudiant
JOIN Cours ON Etudiant.cours_id = Cours.id
WHERE Cours.nom = 'Mathématiques';
```

La restructuration pourrait déplacer la sélection `Cours.nom = 'Mathématiques'` avant la jointure pour limiter le nombre de lignes jointes.

Propriétés des Opérateurs Algébriques

Les opérateurs algébriques (jointure, sélection, projection) possèdent des propriétés algébriques qui facilitent leur manipulation et leur réorganisation dans une requête.

1. Commutativité et Associativité
   - La commutativité (par exemple, \(R \join S = S \join R \)) et l'associativité (par exemple, \((R \join S) \join T = R \join (S \join T) \)) permettent de réorganiser les opérations sans changer le résultat final.

2. Groupage des Sélections et Projections
   - Les sélections (σ) et les projections (π) peuvent être regroupées pour minimiser le volume de données manipulées.

3. Inversion des Opérateurs
   - Inverser les sélections et les projections peut souvent optimiser la requête en réduisant le volume de données traitées à chaque étape.

Principe d’Optimisation des Expressions Algébriques

Les principes suivants guident l'optimisation des expressions algébriques pour minimiser le coût d'exécution et maximiser l'efficacité :

- Exécuter les Sélections Aussitôt que Possible
  - Pousser les opérations de sélection le plus bas possible dans l'arbre de requête pour limiter le volume de données avant d'effectuer d'autres opérations.

- Réduire la Taille des Relations à Manipuler
  - Combiner les sélections avec les produits cartésiens pour réduire le nombre de lignes jointes lors de l'exécution de la jointure.

- Mémoriser les Sous-Expressions Communes
  - Si une sous-expression est utilisée plusieurs fois, la calculer une seule fois peut améliorer les performances globales de la requête.

Algorithme d’Optimisation

Les systèmes de gestion de bases de données utilisent des algorithmes pour optimiser automatiquement les requêtes en suivant un ensemble d'étapes :

1. Séparer et Descendre les Sélections
   - Diviser les sélections complexes en sélections simples et les appliquer le plus bas possible dans l'arbre de requête.

2. Descendre les Projections
   - Réorganiser les projections pour minimiser le nombre de colonnes manipulées dès que possible.

3. Combinaison des Opérations Unaires
   - Combinaison efficace des opérations de sélection et de projection pour éviter le surtraitement des données.

4. Regrouper les Nœuds Intérieurs
   - Regrouper les nœuds autour des opérateurs binaires (par exemple *, -, ∪) pour optimiser l'évaluation parallèle ou séquentielle.

Considérations Additionnelles

Pour une bonne optimisation, il est important de prendre en compte d'autres facteurs comme :

- Chemins d'Accès aux Données
  - Différents algorithmes d'accès (séquentiel, indexé, haché) peuvent influencer le choix de l'opérateur.

- Statistiques de la Base de Données
  - Taille des tables, sélectivité des attributs et autres statistiques impactent directement les performances des requêtes.

Limitations

Bien que la réécriture algébrique soit cruciale, elle ne suffit pas toujours à garantir une optimisation optimale. D'autres modèles de coût et analyses spécifiques peuvent être nécessaires pour prendre des décisions optimales en fonction des conditions réelles de la base de données.

En résumé, les méthodes d'optimisation basées sur les arbres algébriques sont essentielles pour améliorer les performances des requêtes SQL en réorganisant efficacement les opérations algébriques tout en tenant compte des propriétés des opérateurs et des caractéristiques spécifiques de la base de données.

Exercices sur l'optimisation des requêtes par les arbres algébriques

Bien sûr ! Voici trois exercices d'optimisation de requêtes SQL en utilisant les arbres algébriques, avec des détails sur la solution pour chaque exercice :

Exercice 1 : Optimisation d'une Jointure avec Sélection

Considérons la requête suivante :

```sql
SELECT *
FROM Etudiant
JOIN Cours ON Etudiant.cours_id = Cours.id
WHERE Cours.nom = 'Mathématiques';
```

Solution :

1. Analyse de la Requête :
  - La requête effectue une jointure entre la table `Etudiant` et `Cours` sur la condition `Etudiant.cours_id = Cours.id`.
  - Ensuite, une sélection est appliquée sur la table `Cours` pour récupérer uniquement les cours avec le nom 'Mathématiques'.

$$\pi_{\text{Etudiant.*}}\left(\sigma_{\text{Etudiant.cours\_id} = \text{Cours.id} \wedge \text{Cours.nom} = \text{'Mathématiques'}}(\text{Etudiant} \bowtie \text{Cours})\right)$$

2. Optimisation :

- Pour optimiser cette requête, il est judicieux de pousser la condition de sélection (`Cours.nom = 'Mathématiques'`) aussi bas que possible dans l'arbre d'exécution.
- La jointure peut être réorganisée pour d'abord sélectionner les cours 'Mathématiques' et ensuite joindre les étudiants correspondants.

3. Réécriture de la Requête Optimisée :

$$\pi_{\text{Etudiant.*}}(\text{Etudiant} \bowtie (\sigma_{\text{nom} = \text{'Mathématiques'}}(\text{Cours})))$$

```sql
SELECT *
FROM Etudiant
JOIN (
    SELECT *
    FROM Cours
    WHERE nom = 'Mathématiques'
) AS CoursMath
ON Etudiant.cours_id = CoursMath.id;
```

- En utilisant une sous-requête (sous forme de vue ou de requête dérivée), nous limitons les données jointes aux seuls cours 'Mathématiques', réduisant ainsi le volume de données traitées lors de la jointure.

Exercice 2 : Optimisation avec Projection et Sélection

Considérons la requête suivante :

```sql
SELECT nom, prénom, age
FROM Etudiant
WHERE age > 20;
```

$$\pi_{\text{nom, prénom, age}}(\sigma_{\text{age} > 20}(\text{Etudiant}))$$

Solution :

1. Analyse de la Requête :

- Cette requête sélectionne le nom, prénom et âge des étudiants dont l'âge est supérieur à 20.

2. Optimisation :

- Pour optimiser cette requête, il est efficace de placer la condition de sélection (`age > 20`) le plus bas possible dans l'arbre d'exécution, avant de sélectionner spécifiquement les colonnes nécessaires (`nom`, `prénom`, `age`).

3. Réécriture de la Requête Optimisée :

$$\pi_{\text{nom, prénom, age}}(\text{EtudiantsAgeSup20})$$

```sql
SELECT nom, prénom, age
FROM (
    SELECT *
    FROM Etudiant
    WHERE age > 20
) AS EtudiantsAgeSup20;
```

- En utilisant une sous-requête, nous appliquons d'abord la sélection sur les étudiants dont l'âge est supérieur à 20, puis nous projetons uniquement les colonnes requises (`nom`, `prénom`, `age`), minimisant ainsi le nombre de lignes et de colonnes traitées.

Exercice 3 : Optimisation avec Union

Considérons deux requêtes simples :

```sql
SELECT nom, prénom
FROM Etudiant
WHERE age > 20
UNION
SELECT nom, prénom
FROM Enseignant
WHERE département = 'Informatique';
```

$$\pi_{\text{nom, prénom}}((\sigma_{\text{age} > 20}(\text{Etudiant})) \cup (\sigma_{\text{département} = \text{'Informatique'}}(\text{Enseignant})))$$

Solution :

1. Analyse de la Requête :

- Cette requête utilise l'opérateur `UNION` pour combiner les résultats de deux sélections de noms et prénoms d'étudiants et d'enseignants basées sur des conditions spécifiques.

2. Optimisation :

- Pour optimiser cette requête, il est préférable de pousser les conditions de sélection (`age > 20` pour les étudiants et `département = 'Informatique'` pour les enseignants) avant d'appliquer l'opérateur `UNION`.

3. Réécriture de la Requête Optimisée :

$$\pi_{\text{nom, prénom}}(\text{Personnes})$$

```sql
SELECT nom, prénom
FROM (
    SELECT nom, prénom
    FROM Etudiant
    WHERE age > 20
    UNION
    SELECT nom, prénom
    FROM Enseignant
    WHERE département = 'Informatique'
) AS Personnes;
```

- En utilisant une sous-requête combinée avec `UNION`, nous appliquons d'abord les sélections sur chaque table (étudiants et enseignants) séparément, puis nous combinons les résultats, minimisant ainsi le temps d'exécution en ne traitant que les données pertinentes.

Les optimisations de requêtes SQL basées sur les arbres algébriques sont essentielles pour améliorer les performances des bases de données en réorganisant efficacement les opérations algébriques. En utilisant les bonnes pratiques telles que la réécriture des requêtes, le placement stratégique des opérations de sélection et de projection, et l'utilisation d'opérateurs comme `UNION` de manière optimale, les requêtes peuvent être exécutées de manière plus efficace et avec moins de ressources.

Conclusion

En conclusion, l'optimisation des requêtes SQL par l'utilisation des arbres algébriques est une technique puissante pour améliorer les performances des SGBD. En exploitant les propriétés des opérateurs algébriques et en appliquant des règles de réécriture, il est possible de transformer les requêtes en expressions plus efficaces, réduisant ainsi le temps de traitement et les ressources utilisées. L'étude approfondie des triggers et des contraintes d'intégrité ajoute une dimension supplémentaire à cette optimisation, garantissant que les données restent cohérentes et conformes aux règles définies. La maîtrise de ces techniques est essentielle pour tout professionnel travaillant avec des bases de données, permettant de concevoir des systèmes plus robustes et performants. Les exercices et exemples présentés dans ce cours offrent une base solide pour comprendre et appliquer ces concepts dans des scénarios réels.