

## Chapitre 1 :Rappels des Bases de Données Avancées

Ce chapitre couvre les concepts fondamentaux des bases de données, en se concentrant sur la conception conceptuelle et relationnelle, l'utilisation des langages de manipulation des données, et les opérations avancées pour manipuler des données relationnelles.

### **1. Modèle Entité-Association (E/A)**

Le **modèle Entité-Association (E/A)** est utilisé pour concevoir des bases de données de manière conceptuelle. Ce modèle facilite la visualisation des données, des relations et des structures lors de la phase de conception.

- **Entité** : Représente un objet identifiable dans le monde réel. Par exemple, un **Livre** dans une bibliothèque avec des attributs comme *ISBN*, *Titre*, et *Date de Publication*.
- **Association** : Représente la relation entre plusieurs entités. Par exemple, l'association **Emprunt** entre les entités **Livre** et **Adhérent** montre qu'un adhérent peut emprunter un livre.
- **Attributs** : Décrivent les caractéristiques d'une entité ou d'une association (ex. : *Nom* et *Prénom* pour l'entité **Adhérent**).
- **Cardinalité** : Définit combien d'instances d'une entité peuvent être associées à une autre. Les types courants incluent :
  - **1** : Un livre est publié par une seule maison d'édition, mais une maison d'édition peut publier plusieurs livres.
  - **N** : Plusieurs auteurs peuvent écrire plusieurs livres.
  - **1:1** : Un passeport est associé à une seule personne, et inversement.

**Exemple :**

Dans une bibliothèque universitaire, les entités incluent **Livre**, **Adhérent**, **Auteur**, et **Maison d'Édition**, avec des relations comme **Emprunt** (adhérent emprunte un livre) et **Écrit** (auteur écrit un livre).

---

## **2. Modèle Relationnel**

Le **modèle relationnel** organise les données sous forme de **tables** (relations), constituées de **lignes** (tuples) et de **colonnes** (attributs). Chaque table correspond à une entité ou à une relation du modèle E/A.

- **Table (Relation)** : Structure de données sous forme de lignes et de colonnes. Par exemple, une table **Livre** peut contenir des colonnes pour *ISBN*, *Titre*, et *DatePublication*.

**Transformation des Associations selon les Cardinalités :**

1. **Association 1 (Un-à-Plusieurs)** :

- On ajoute une **clé étrangère** dans la table du côté "N" pour faire référence à la clé primaire de la table du côté "1". (pour faire plus simple le fils prend la clé étrangère).

2. **Exemple** :

- **Livre et Maison d'Édition** : Une maison d'édition publie plusieurs livres.
  - **Table Livre** : Ajoute *ID\_Maison* (clé étrangère pointant vers **Maison d'Édition**).

3. **Association N (Plusieurs-à-Plusieurs)** :

- On crée une **table d'association** avec des clés étrangères pour chaque entité associée.

4. **Exemple** :

- **Livre et Auteur** : Un livre peut être écrit par plusieurs auteurs, et un auteur peut écrire plusieurs livres.
  - **Table d'association Écrit** : Contient *ISBN* et *ID\_Auteur* comme clés étrangères, avec une clé primaire composée (*ISBN*, *ID\_Auteur*).

5. **Association 1:1 (Un-à-Un)** :

- On peut ajouter une clé étrangère dans l'une des deux tables, ou créer une table séparée si l'association possède des attributs spécifiques.

6. **Exemple** :

- **Personne et Passeport** : Chaque personne possède un passeport unique.
  - **Table Personne** : Ajoute *ID\_Passeport* (clé étrangère pointant vers **Passeport**).

## **Les Clés :**

- **Clé primaire** : Identifie de manière unique chaque ligne d'une table (ex. : *ISBN* dans **Livre**).
- **Clé étrangère** : Établit une relation avec une autre table (ex. : *ID\_Maison* dans **Livre**, référant **Maison d'Édition**).

## **Contraintes d'intégrité :**

Elles garantissent la cohérence et l'exactitude des données :

- **UNIQUE** : Les valeurs doivent être uniques (ex. : *ISBN*).
- **NOT NULL** : Les champs obligatoires ne peuvent pas être vides.
- **CHECK** : Imposer des règles, comme vérifier que la date de naissance soit après une certaine année.

## **3. Langage SQL:**

Le **Structured Query Language (SQL)** est un langage standard pour interagir avec des bases de données relationnelles. Il se divise en deux grandes catégories : **LDL** et **LMD**.

- **Langage de Définition de Données (LDL)** : Utilisé pour créer et modifier la structure des bases de données.
  - **CREATE** : Crée des objets comme des tables (ex. : `CREATE TABLE Livre`).
  - **ALTER** : Modifie une table existante (ex. : `ALTER TABLE Livre ADD DatePublication DATE`).
  - **DROP** : Supprime des objets (ex. : `DROP TABLE Livre`).
- **Langage de Manipulation de Données (LMD)** : Pour manipuler les données.
  - **SELECT** : Récupère des données (ex. : `SELECT * FROM Livre WHERE DatePublication > 2000`).
  - **INSERT** : Ajoute de nouvelles données (ex. : `INSERT INTO Livre VALUES ('123456789', 'Le Petit Prince', '1943-04-06')`).

- **UPDATE** : Modifie des données existantes (ex. : `UPDATE Livre SET Titre = 'Le Grand Prince' WHERE ISBN = '123456789'`).
- **DELETE** : Supprime des données (ex. : `DELETE FROM Livre WHERE ISBN = '123456789'`).

### Fonctions d'Aggrégation :

Les fonctions d'agrégation permettent d'effectuer des calculs sur un ensemble de lignes. Voici les principales fonctions d'agrégation :

- **COUNT** : Compte le nombre total de lignes correspondant à une condition.  
`SELECT COUNT(*) FROM Livre;`
- **SUM** : Calcule la somme totale des valeurs d'une colonne numérique.  
`SELECT SUM(Prix) FROM Livre;`
- **AVG** : Calcule la moyenne des valeurs d'une colonne numérique.  
`SELECT AVG(Prix) FROM Livre;`
- **MIN** : Renvoie la plus petite valeur d'une colonne.  
`SELECT MIN(DatePublication) FROM Livre;`
- **MAX** : Renvoie la plus grande valeur d'une colonne.  
`SELECT MAX(Prix) FROM Livre;`

### Filtrage et Tri :

Pour affiner et trier les données lors des requêtes SQL :

- **WHERE** : Filtre les enregistrements en fonction d'une condition.  
`SELECT * FROM Livre WHERE Auteur = 'Antoine de Saint-Exupéry' ;`
- **ORDER BY** : Trie les enregistrements selon une ou plusieurs colonnes.

- **ASC** (ordre ascendant, par défaut) :  
`SELECT * FROM Livre ORDER BY DatePublication ASC;`
- **DESC** (ordre descendant) :  
`SELECT * FROM Livre ORDER BY DatePublication DESC;`

### **Contraintes d'intégrité en SQL :**

Les **contraintes d'intégrité** sont des règles qui assurent la validité et la cohérence des données dans une base de données relationnelle. Elles sont essentielles pour maintenir la qualité des données tout au long de leur cycle de vie.

Les principales contraintes d'intégrité sont :

- **Clé primaire (PRIMARY KEY)** :

- Assure que chaque ligne dans une table est unique. Aucun doublon n'est autorisé dans la colonne ou l'ensemble de colonnes désigné comme clé primaire, et cette clé ne peut pas être nulle.

`CREATE TABLE Livre (`

`ISBN VARCHAR2(13) PRIMARY KEY,`

`Titre VARCHAR2(100),`

`DatePublication DATE);`

- **Contrainte d'unicité (UNIQUE)** :

- Garantit que toutes les valeurs d'une colonne (ou d'un ensemble de colonnes) sont uniques. Cette contrainte empêche les doublons.

`CREATE TABLE MaisonEdition (`

`ID_Maison NUMBER PRIMARY KEY,`

`Nom_MaisonEdition VARCHAR2(255) UNIQUE);`

- **Contrainte de champ obligatoire (NOT NULL) :**

- Indique que la colonne ne peut pas accepter de valeurs nulles. Cela garantit que chaque ligne de la table aura une valeur dans cette colonne.

`CREATE TABLE Livre (`

`ISBN VARCHAR2(13) PRIMARY KEY,`

`Titre VARCHAR2(100) NOT NULL);`

- **Contrainte de domaine (CHECK) :**

- Définit une condition que les données d'une colonne doivent satisfaire. Par exemple, la contrainte suivante vérifie que l'ISBN est compris entre certaines valeurs.

`CREATE TABLE Livre (`

`ISBN NUMBER CHECK (ISBN > 99999 AND ISBN < 10000000));`

- **Intégrité référentielle (FOREIGN KEY) :**

- Garantit que les valeurs de la clé étrangère existent réellement dans la table référencée, maintenant la cohérence des relations entre les tables.

`CREATE TABLE Emprunt (`

`ID_Empunt NUMBER PRIMARY KEY,`

`ISBN VARCHAR2(13),`

`FOREIGN KEY (ISBN) REFERENCES Livre(ISBN));`

Il est également possible de nommer toutes ces contraintes en utilisant le mot clé

**CONSTRAINT**, et ce pour des raisons de clarté et de maintenance.

`CREATE TABLE Auteur (`

`ID_Auteur NUMBER,`

`....`

`CONSTRAINT pk_auteur PRIMARY KEY( ID_Auteur);`

## **4. Langage Algébrique:**

Le **langage algébrique** ou **algèbre relationnelle** est un ensemble d'opérations mathématiques permettant de manipuler et interroger les données dans une base de données relationnelle. Il constitue la base théorique des requêtes SQL et permet d'exécuter des opérations sur des ensembles de tables.

Les principales **opérations de l'algèbre relationnelle** sont :

- **Sélection ( $\sigma$ ) :**
  - Permet de filtrer les lignes d'une relation (table) en fonction d'une condition.
  - Exemple : Trouver tous les livres publiés par "Gallimard" :  
 $\sigma \text{ MaisonEdition} = \text{'Gallimard'} (\text{Livre})$
- **Projection ( $\pi$ ) :**
  - Extrait certaines colonnes d'une relation, éliminant les doublons s'il y en a.
  - Exemple : Récupérer uniquement les titres et auteurs des livres :  
 $\pi \text{ Titre, Auteur } (\text{Livre})$
- **Union ( $\cup$ ) :**
  - Combine les lignes de deux relations, en éliminant les doublons.
  - Exemple : Combiner les livres de deux catégories différentes (fiction et non-fiction) :  
 $\text{LivresFiction} \cup \text{LivresNonFiction}$
- **Intersection ( $\cap$ ) :**
  - Renvoie les lignes qui apparaissent à la fois dans les deux relations.
  - Exemple : Trouver les livres qui sont à la fois dans la catégorie **Fiction et Historique** :  
 $\text{LivresFiction} \cap \text{LivresHistoriques}$
- **Différence ( $-$ ) :**
  - Renvoie les lignes présentes dans la première relation mais pas dans la seconde.
  - Exemple : Trouver les livres qui ne sont pas dans la catégorie **Fiction** :  
 $\text{Livres} - \text{LivresFiction}$
- **Produit cartésien ( $\times$ ) :**
  - Combine toutes les combinaisons possibles de tuples (lignes) de deux relations.

- Exemple : Si nous avons une relation **Auteur** et une relation **Livre**, le produit cartésien génère toutes les combinaisons possibles d'auteurs et de livres.

**Auteur × Livre**

## **5. Les jointures:**

Les **jointures** sont des opérations qui permettent de combiner les données de deux ou plusieurs tables en fonction d'une condition, généralement une colonne commune entre les tables. Elles sont essentielles pour interroger des bases de données normalisées où les informations sont réparties sur plusieurs tables.

Les principaux types de jointures sont :

- **Jointure interne (Inner Join) :**

- Retourne uniquement les lignes ayant des correspondances dans les deux tables.
- Exemple : Récupérer tous les livres avec leurs maisons d'édition associées :

```
SELECT * FROM Livre INNER JOIN MaisonEdition ON
Livre.ID_Maison = MaisonEdition.ID_Maison;
```

- **Jointure externe gauche (Left Outer Join) :**

- Retourne toutes les lignes de la table de gauche, même celles sans correspondance dans la table de droite. Les colonnes de la table de droite sans correspondance seront **NULL**.
- Exemple : Afficher tous les livres, même ceux sans maison d'édition :

```
SELECT * FROM Livre LEFT JOIN MaisonEdition ON
Livre.ID_Maison = MaisonEdition.ID_Maison;
```

- **Jointure externe droite (Right Outer Join) :**

- Retourne toutes les lignes de la table de droite, même si elles n'ont pas de correspondance dans la table de gauche. Les colonnes de la table de gauche seront **NULL** pour ces lignes.
- Exemple : Récupérer toutes les maisons d'édition, même celles n'ayant publié aucun livre :

```
SELECT * FROM MaisonEdition RIGHT JOIN Livre ON
Livre.ID_Maison = MaisonEdition.ID_Maison;
```

- **Jointure externe complète (Full Outer Join) :**

- Combine toutes les lignes des deux tables, avec des **NULL** pour les valeurs manquantes dans l'une ou l'autre table.

- Exemple : Afficher tous les livres et toutes les maisons d'édition, même si certaines maisons n'ont publié aucun livre et certains livres n'ont pas de maison d'édition associée :

```
SELECT * FROM Livre FULL OUTER JOIN MaisonEdition ON
Livre.ID_Maison = MaisonEdition.ID_Maison;
```

- **Jointure croisée (Cross Join) :**

- Produit le produit cartésien des deux tables, c'est-à-dire toutes les combinaisons possibles de lignes entre les deux tables.
- Exemple : Récupérer toutes les combinaisons possibles entre les auteurs et les livres :

```
SELECT * FROM Auteur CROSS JOIN Livre;
```

- **Jointure naturelle (Natural Join) :**

- Jointure basée sur les colonnes ayant le même nom dans les deux tables. Elle est similaire à une **jointure interne**, mais automatique sur les colonnes communes.
- Exemple : Si les tables **Livre** et **MaisonEdition** ont une colonne commune **ID\_Maison**, la jointure naturelle les liera automatiquement.

```
SELECT * FROM Livre NATURAL JOIN MaisonEdition;
```

## **6. Les vues:**

Une **vue** est une table virtuelle qui présente les données issues d'une ou plusieurs tables de manière organisée et filtrée. Les vues ne stockent pas directement les données, mais elles permettent d'interroger les tables sous-jacentes comme si elles étaient une seule table.

### **Types de vues :**

- **Vue simple :**

- Une vue créée à partir d'une requête simple sur une ou plusieurs tables. Elle permet de simplifier les requêtes complexes en les pré-enregistrant.
- Exemple : Créer une vue des livres publiés après l'an 2000 :

```
CREATE VIEW LivresRecents AS
```

```
SELECT * FROM Livre WHERE DatePublication > '2000-01-01';
```

- **Vue matérialisée :**

- Contrairement aux vues normales, les **vues matérialisées** stockent physiquement le résultat d'une requête. Cela permet une récupération plus rapide des données, mais ces vues doivent être

- actualisées manuellement lorsque les données sources changent.
- Exemple : Créer une vue matérialisée des auteurs ayant écrit plus de trois livres :

```
CREATE MATERIALIZED VIEW AuteursPopulaires AS
```

```
SELECT Auteur.Nom, COUNT(*) as Nombre_Livres  
FROM Auteur JOIN Livre ON Auteur.ID_Auteur = Livre.ID_Auteur  
GROUP BY Auteur.Nom HAVING COUNT(*) > 3;
```

### Utilité des vues :

- **Simplification des requêtes complexes** : Les vues permettent de masquer des requêtes SQL complexes derrière une vue plus simple à interroger.
- **Sécurité** : On peut restreindre l'accès aux données sensibles en créant des vues qui n'exposent qu'une partie des données de la table originale.
- **Réutilisation** : Les vues permettent de réutiliser des requêtes fréquemment utilisées sans avoir à les réécrire.

### Manipulation des vues :

- On peut interroger une vue comme une table normale :  
`SELECT * FROM LivresRecents;`
- On peut également supprimer une vue avec **DROP VIEW** :  
`DROP VIEW LivresRecents;`

## Résumé : Chapitre 1 - Rappels des Bases de Données Avancées

---

### 1. Modèle Entité-Association (E/A)

- **Entités** : Représentent des objets du monde réel (ex. : Livre, Auteur).
  - **Associations** : Relations entre entités (ex. : Emprunt entre Livre et Adhérent).
  - **Attributs** : Décrivent les caractéristiques des entités (ex. : ISBN pour Livre).
  - **Cardinalités** : Définissent le nombre d'instances dans une association (1:N, N:M, 1:1).
  - **Exemple** : Une maison d'édition (1) peut publier plusieurs livres (N).
- 

### 2. Modèle Relationnel

- Les données sont organisées en **tables** (relations) composées de **lignes** (tuples) et **colonnes** (attributs).
  - **Transformation des associations** :
    - **1:N** : Clé étrangère dans la table du côté "N".(pour faire plus simple le fils prend la clé étrangère).
    - **N:M** : Table d'association avec clés étrangères et clé primaire composée.
    - **1:1** : Clé étrangère ou table séparée si nécessaire.
  - **Contraintes d'intégrité** :
    - **PRIMARY KEY** : Identifie de manière unique une ligne.
    - **FOREIGN KEY** : Maintient la cohérence des relations entre tables.
    - **NOT NULL, UNIQUE, CHECK**.
- 

### 3. Langage SQL

- **Langage de Définition de Données (LDD)** :
  - **CREATE** (Créer une table), **ALTER** (Modifier une table), **DROP** (Supprimer une table).

- **Langage de Manipulation de Données (LMD) :**
    - **SELECT** (Lire les données), **INSERT** (Ajouter), **UPDATE** (Modifier), **DELETE** (Supprimer).
  - **Fonctions d'agrégation :**
    - **COUNT, SUM, AVG, MIN, MAX.**
  - **Filtrage et tri :**
    - **WHERE** : Filtre les lignes.
    - **ORDER BY** : Trie les résultats (ASC/DESC).
- 

#### 4. Langage Algébrique (Algèbre Relationnelle)

- **Opérations de base :**
    - **Sélection ( $\sigma$ )** : Filtre les lignes.
    - **Projection ( $\pi$ )** : Extrait des colonnes.
    - **Union ( $\cup$ )** : Combine les lignes de deux relations (sans doublons).
    - **Intersection ( $\cap$ )** : Renvoie les lignes communes.
    - **Différence ( $-$ )** : Renvoie les lignes d'une relation absentes dans une autre.
    - **Produit cartésien ( $\times$ )** : Combine toutes les lignes de deux relations.
- 

#### 5. Les Jointures

- Permettent de combiner des données de plusieurs tables :
  - **INNER JOIN** : Lignes avec correspondances dans les deux tables.
  - **LEFT JOIN** : Toutes les lignes de la table de gauche, même sans correspondance.
  - **RIGHT JOIN** : Toutes les lignes de la table de droite, même sans correspondance.
  - **FULL OUTER JOIN** : Combine toutes les lignes des deux tables avec **NONE** pour les absences.
  - **CROSS JOIN** : Produit cartésien des tables.
  - **NATURAL JOIN** : Jointure basée sur des colonnes ayant le même nom.

---

## 6. Les Vues

- **Vue simple** : Table virtuelle basée sur une requête.
  - Exemple : Livres publiés après 2000.
- **Vue matérialisée** : Stocke physiquement les résultats pour des accès rapides.
  - Exemple : Auteurs ayant écrit plus de trois livres.
- **Utilité** :
  - Simplifie les requêtes complexes.
  - Renforce la sécurité en limitant l'accès aux données sensibles.
  - Réutilise des requêtes fréquentes.
- **Manipulation** :
  - **SELECT** pour interroger une vue.
  - **DROP VIEW** pour la supprimer.

## Chapitre 2 :Programmations Avancées

### 1. Modèle Entité-Association :

- Tables utilisées :
  - **Livre** (*ISBN, Titre, DatePublication, Genre, ID\_Auteur, ID\_Maison\**).
  - **Adhérent** (*ID\_Adhérent, Nom, Prénom, DateNaissance, Adresse*).
  - **Auteur** (*ID\_Auteur, Nom, Prénom, Nationalité*).
  - **MaisonÉdition** (*ID\_Maison, Nom, Adresse*).
  - **Emprunt** (*ID\_Emprunt, ISBN, ID\_Adhérent\**).
- Hypothèse : Chaque livre est écrit par un seul auteur.

### 2. Ordre d'utilisation des clauses SQL :

- **SELECT** (*obligatoire*) : Définit les colonnes à afficher.
- **FROM** (*obligatoire*) : Indique les tables à interroger.
- **JOIN** (*facultatif*) : Combine les données de plusieurs tables.
- **WHERE** (*facultatif*) : Filtre les lignes selon des critères.
- **GROUP BY** (*facultatif*) : Regroupe les données par colonnes.
- **HAVING** (*facultatif*) : Filtre les groupes créés par GROUP BY.
- **ORDER BY** (*facultatif*) : Trie les résultats par colonne(s).

### 3. Les Clauses IN et EXISTS:

#### 3.1. La clause IN

La clause **IN** permet de vérifier si une colonne contient une valeur parmi une liste spécifiée. Si une correspondance est trouvée, la ligne est incluse dans les résultats.

##### Syntaxe :

```
SELECT colonnes FROM table WHERE colonne IN (valeur1, valeur2, ...);
```

##### Exemple :

Pour sélectionner les livres de genres "Roman" ou "Biographie" :

```
SELECT * FROM Livre WHERE Genre IN ('Roman', 'Biographie');
```

##### a) Remplacement de l'opérateur OR :

La clause **IN** est une alternative concise à une série de conditions avec **OR**.

##### Exemple :

```
SELECT * FROM Livre WHERE Nom IN ('Jules', 'Victor', 'George');
```

- b) Utilisation avec sous-requêtes :

**IN** peut être utilisé avec une sous-requête pour comparer une colonne aux résultats de cette sous-requête.

### 3.2. Clause NOT IN

Permet de vérifier si une colonne **n'a pas** de correspondance dans une liste ou une sous-requête.

**Exemple** : Trouver les livres qui ne sont ni des romans ni des biographies :

```
SELECT * FROM Livre WHERE Genre NOT IN ('Roman', 'Biographie');
```

### 3.3. Clause EXISTS

Test si une sous-requête retourne au moins une ligne. Renvoie **vrai** si une correspondance existe.

**Syntaxe** :

```
SELECT colonnes FROM table WHERE EXISTS (sous-requête);
```

**Exemple** : Trouver les auteurs ayant écrit au moins un livre :

```
SELECT ID_Auteur, Nom_Auteur, Prénom_Auteur FROM Auteur A
```

```
WHERE EXISTS ( SELECT * FROM Livre L
```

```
WHERE L.ID_Auteur = A.ID_Auteur );
```

- Comparaison avec IN :

- **EXISTS** est souvent plus rapide que **IN** pour les sous-requêtes volumineuses, car il s'arrête dès qu'une correspondance est trouvée.
- **IN** vérifie toutes les valeurs d'un ensemble.

### 3.4. Clause NOT EXISTS

Test si une sous-requête ne retourne **aucune ligne**. Renvoie **vrai** si aucune correspondance n'est trouvée.

**Exemple** : Trouver les adhérents n'ayant jamais emprunté de livre :

```
SELECT * FROM Adherent A WHERE NOT EXISTS ( SELECT * FROM Emprunt E WHERE E.ID_Adherent = A.ID_Adherent );
```

### 3.5. IN et EXISTS par rapport aux jointures:

Critère	IN	EXISTS	Jointures
Utilisation	Compare une valeur à une liste de valeurs ou aux résultats d'une sous-requête.	Teste l'existence de lignes renvoyées par une sous-requête.	Récupère et combine des colonnes de deux ou plusieurs tables.
Performance	Plus performante avec une petite liste de valeurs. Peut être lente pour de grandes sous-requêtes.	Souvent plus rapide pour les grandes sous-requêtes car s'arrête dès la première correspondance trouvée.	Souvent très performante, en particulier avec des bases de données optimisées.
Complexité d'écriture	Souvent plus simple et plus lisible pour des requêtes simples.	Plus complexe, en particulier pour les utilisateurs moins expérimentés.	Peut être complexe, surtout pour des requêtes multi-tables complexes.
Flexibilité	Moins flexible que les jointures.	Moins flexible que les jointures.	Très flexible pour récupérer et manipuler des données de différentes tables.
Cas d'utilisation	Bon pour vérifier si une valeur appartient à un ensemble défini.	Bon pour vérifier l'existence de correspondances dans une autre table.	Idéal pour combiner des données de différentes tables.

## 4.Les requêtes imbriquées:

### 4.1. Définition :

Les requêtes imbriquées, ou sous-requêtes, sont des requêtes SQL incluses dans une autre requête. Elles retournent des données utilisées par la requête principale, notamment pour des conditions ou des calculs.

### 4.2. Requêtes imbriquées dans la clause WHERE

Utilisées pour filtrer les résultats de la requête principale en fonction des résultats d'une sous-requête.

**Exemple :** Trouver les livres écrits par des auteurs algériens :

```
SELECT ISBN, Titre FROM Livre WHERE ID_Auteur IN (SELECT  
ID_Auteur FROM Auteur WHERE Nationalité = 'Algérienne');
```

### 4.3. Requêtes imbriquées dans la clause FROM

Agissent comme des tables temporaires, souvent utilisées pour des opérations d'agrégation.

**Exemple :** Calculer la moyenne de livres écrits par auteur :

```
SELECT AVG(NombreLivres) FROM (SELECT ID_Auteur,  
COUNT(ISBN) as NombreLivres FROM Livre GROUP BY ID_Auteur);
```

#### **4.4. Requêtes imbriquées dans la clause SELECT**

Permettent de calculer des valeurs spécifiques pour chaque ligne de la requête principale.

**Exemple 1 :** Afficher le titre et le nom de la maison d'édition pour chaque livre

`SELECT Titre,`

`(SELECT Nom_MaisonEdition FROM MaisonEdition M WHERE  
M.ID_Maison = L.ID_Maison) as MaisonEdition`

`FROM Livre L;`

**Exemple 2 :** Afficher le nom et l'âge des adhérents :

`SELECT Nom_Adherent,`

`(YEAR(CURRENT_DATE) - YEAR(DateNaissance)) AS Age`

`FROM Adherent;`

#### **4.5. Utilisation avec les opérations de jointure**

Les sous-requêtes permettent de préparer des données complexes avant de les joindre.

**Exemple :** Trouver les livres empruntés par l'adhérent ayant l'ID "123" :

`SELECT L.Titre`

`FROM Livre L`

`JOIN (SELECT E.ISBN FROM Emprunt E WHERE E.ID_Adherent =  
'123') AS EmpruntsAdherent`

`ON L.ISBN = EmpruntsAdherent.ISBN;`

### **5.Agrégations et filtrage avec GROUP BY et HAVING:**

#### **5.1. Clause GROUP BY**

La clause GROUP BY permet de regrouper des lignes partageant les mêmes valeurs dans des colonnes spécifiques et d'appliquer des fonctions d'agrégation sur chaque groupe (comme COUNT, SUM, AVG, MAX, MIN).

## **Caractéristiques :**

- Peut regrouper selon une ou plusieurs colonnes.
- Les colonnes sélectionnées doivent être soit dans le GROUP BY, soit agrégées.
- Peut être combinée avec ORDER BY pour trier les groupes.
- Peut être combinée avec HAVING pour filtrer les groupes après l'agrégation.

## **Exemple :**

Trouver le nombre de livres écrits par chaque auteur :

```
SELECT ID_Auteur, COUNT(*) AS NombreLivres FROM Livre  
GROUP BY ID_Auteur;
```

## **5.2. Clause HAVING**

La clause HAVING filtre les groupes créés par GROUP BY, contrairement à WHERE, qui filtre les lignes avant le regroupement. HAVING peut utiliser des fonctions d'agrégation dans ses conditions.

## **Exemple :**

Trouver les auteurs ayant écrit plus de 5 livres :

```
SELECT ID_Auteur, COUNT(*) AS NombreLivres  
FROM Livre  
GROUP BY ID_Auteur  
HAVING COUNT(*) > 5;
```

## **5.3. Combinaison de GROUP BY, HAVING et fonctions d'agrégation**

En combinant ces éléments, il est possible de regrouper, appliquer des fonctions d'agrégation et filtrer les groupes selon des critères spécifiques.

**Exemple 1 :** Trouver les adhérents ayant emprunté plus de 5 livres, triés par nombre d'emprunts décroissant :

```
SELECT ID_Adherent, COUNT(*) AS NombreEmprunts
```

```
FROM Emprunt  
GROUP BY ID_Adhérent  
HAVING COUNT(*) > 5  
ORDER BY COUNT(*) DESC;
```

**Exemple 2 :** Ajouter les noms et prénoms des adhérents :

```
SELECT E.ID_Adhérent, A.Nom_Adhérent, A.Prénom_Adhérent,  
COUNT(*) AS NombreEmprunts
```

```
FROM Emprunt E  
JOIN Adhérent A ON E.ID_Adhérent = A.ID_Adhérent  
GROUP BY E.ID_Adhérent, A.Nom_Adhérent, A.Prénom_Adhérent  
HAVING COUNT(*) > 5  
ORDER BY COUNT(*) DESC;
```

## **6.Opérateurs algébriques en SQL:**

### **6.1. UNION et UNION ALL**

- **UNION** : Combine les résultats de deux requêtes en supprimant les doublons.
- **UNION ALL** : Semblable à UNION, mais conserve les doublons.
- **Conditions** : Les requêtes doivent avoir le même nombre de colonnes et des types de données compatibles.
- **Utilisation** : Combiner des résultats de différentes tables ou effectuer des sélections conditionnelles.
- **Exemple :**

Liste combinée des noms des auteurs et des adhérents :

```
SELECT Nom_Auteur AS Nom FROM Auteur
```

```
UNION
```

```
SELECT Nom_Adhérent AS Nom FROM Adhérent;
```

- **UNION vs OR :**
  - UNION exécute deux requêtes distinctes et fusionne leurs résultats.

- OR combine des conditions dans une seule requête. OR est souvent plus performant dans des cas simples.

## 6.2. INTERSECT

Renvoie les lignes communes à deux requêtes.

**Exemple :** Trouver les personnes qui sont à la fois auteurs et adhérents

`SELECT A.Nom_Auteur FROM Auteur A`

`INTERSECT`

`SELECT Ad.Nom_Adherent FROM Adherent Ad;`

## 6.3. EXCEPT (ou MINUS)

Renvoie les lignes de la première requête qui ne sont pas dans la seconde.

**Exemple :** Trouver les adhérents n'ayant jamais emprunté de livre :

`SELECT ID_Adherent FROM Adherent`

`MINUS`

`SELECT ID_Adherent FROM Emprunt;`

## 6.4. Performance et doublons

- Les opérateurs ensemblistes peuvent générer des doublons (notamment avec UNION ALL).
- Ils peuvent être coûteux en termes de performance sur de grands ensembles de données.
- **Optimisation** : Préférer des requêtes imbriquées ou des jointures lorsque c'est possible.

**Exemple :** Utiliser une requête imbriquée au lieu de MINUS :

`SELECT A.ID_Adherent, A.Nom_Adherent, A.Prenom_Adherent`

`FROM Adherent A`

`WHERE A.ID_Adherent NOT IN (`

`SELECT E.ID_Adherent FROM Emprunt E`

`);`

## Résumé : Chapitre 2 - Programmations Avancées

---

### 1. Modèle Entité-Association

Tables utilisées :

- **Livre** (*ISBN, Titre, DatePublication, Genre, ID\_Auteur, ID\_Maison*).
  - **Adhérent** (*ID\_Adhérent, Nom, Prénom, DateNaissance, Adresse*).
  - **Auteur** (*ID\_Auteur, Nom, Prénom, Nationalité*).
  - **MaisonÉdition** (*ID\_Maison, Nom, Adresse*).
  - **Emprunt** (*ID\_Emprunt, ISBN, ID\_Adhérent*).
- 

### 2. Ordre des Clauses SQL

1. **SELECT** : Définit les colonnes à afficher (*obligatoire*).
  2. **FROM** : Indique les tables à interroger (*obligatoire*).
  3. **JOIN** : Combine les données de plusieurs tables (*facultatif*).
  4. **WHERE** : Filtre les lignes selon des critères (*facultatif*).
  5. **GROUP BY** : Regroupe les données (*facultatif*).
  6. **HAVING** : Filtre les groupes après regroupement (*facultatif*).
  7. **ORDER BY** : Trie les résultats (*facultatif*).
- 

### 3. Clauses IN et EXISTS

- **IN** : Vérifie si une colonne contient une valeur dans une liste ou sous-requête.
    - **Exemple** : Livres de genres "Roman" ou "Biographie".
  - **NOT IN** : Vérifie qu'une colonne n'a pas de correspondance.
    - **Exemple** : Livres qui ne sont ni des romans ni des biographies.
  - **EXISTS** : Vérifie si une sous-requête retourne au moins une ligne.
    - **Exemple** : Auteurs ayant écrit au moins un livre.
  - **NOT EXISTS** : Vérifie si une sous-requête ne retourne aucune ligne.
    - **Exemple** : Adhérents n'ayant jamais emprunté de livre.
- 

### 4. Requêtes Imbriquées (Sous-requêtes)

- **Dans WHERE** : Filtront les résultats principaux.

- **Exemple** : Livres écrits par des auteurs algériens.
  - **Dans FROM** : Utilisées comme tables temporaires pour agrégations.
    - **Exemple** : Calculer la moyenne de livres par auteur.
  - **Dans SELECT** : Calcurent des valeurs spécifiques par ligne.
    - **Exemple** : Afficher les noms des maisons d'édition pour chaque livre.
  - **Avec jointures** : Préparent des données complexes avant de les joindre.
    - **Exemple** : Livres empruntés par un adhérent spécifique.
- 

## 5. Agrégations et Filtrage (GROUP BY et HAVING)

- **GROUP BY** : Regroupe les lignes par colonnes et applique des fonctions d'agrégation (*COUNT, SUM, AVG, MIN, MAX*).
    - **Exemple** : Nombre de livres par auteur.
  - **HAVING** : Filtre les groupes après regroupement (contrairement à WHERE).
    - **Exemple** : Auteurs ayant écrit plus de 5 livres.
  - **Combinaison GROUP BY + HAVING** : Permet des analyses avancées.
    - **Exemple** : Adhérents ayant emprunté plus de 5 livres, triés par nombre d'emprunts.
- 

## 6. Opérateurs Algébriques en SQL

- **UNION** : Combine les résultats de plusieurs requêtes (sans doublons).
- **UNION ALL** : Combine les résultats avec doublons.
  - **Exemple** : Liste des noms d'auteurs et d'adhérents.
- **INTERSECT** : Renvoie les lignes communes à deux requêtes.
  - **Exemple** : Personnes à la fois auteurs et adhérents.
- **EXCEPT (ou MINUS)** : Renvoie les lignes de la première requête absentes dans la seconde.
  - **Exemple** : Adhérents n'ayant jamais emprunté de livre.
- **Optimisation** : Préférer des requêtes imbriquées ou jointures pour améliorer la performance.

## Résumé PL/SQL

### ① Introduction à PL/SQL:

- PL/SQL est un langage procédural intégré à SQL permettant d'ajouter des mécanismes algorithmiques à SQL.
- Ses blocs PL/SQL contiennent : Déclaration de variables et constantes, instructions exécutables (commandes SQL, boucles, tests conditionnels), gérer des exceptions.
- Les blocs peuvent être anonyme ou nommés (procédures, fonctions déclencheurs).

### ② Structure d'un bloc PL/SQL

- Un bloc se compose de 3 parties :

Declarations: Variables, curseurs, constantes, exceptions.

Instructions: Commandes SQL et instructions algorithmiques.

Exception: Gestion des erreurs.

Syntaxe :

**DECLARE**

Declarations

**BEGIN**

Instructions

[**EXCEPTION**]

gestion des exception

**END;**

/

Exemple :

**DECLARE**

**V-agent**

employement **TYPE**

**BEGIN**

Select nom .into V-agent

from employe;

DBMS\_OUTPUT.PUTLINE

('Bonjour ' || V-agent);

**END;**

/

### ③ Déclarations:

#### \* Variables simples:

Définir des variables de types standards:

DECLARE

salaire NUMBER(8,2) := 1500;

nom VARCHAR2(50) := 'Dupont';

number tab met après la virgule

#### \* Variables références:

- %TYPE: Associe le Type à une colonne d'une table.

DECLARE

masSalaire employee.salary %TYPE;

%ROWTYPE: Associe la structure complète d'une table en un curseur.

DECLARE

employeeRec

employee %ROWTYPE;

#### \* Curseurs:

permettent de parcourir les résultats d'une requête SQL ligne par ligne.

#### Curseur avec FOR:

DECLARE

CURSOR nom\_curseur IS

SELECT colonne1, colonne2  
FROM table WHERE Condition;

BEGIN

FOR enregistrement IN nom\_curseur LOOP

// Instructions

ENDLOOP;

END;

CURSOR emp\_curseur IS

SELECT ename, salary  
FROM employee WHERE Dep = 10;

FOR employe IN emp\_curseur LOOP

DBMS\_OUTPUT.PUT\_LINE

(

ENDLOOP;

END;

- Il n'est pas nécessaire d'ouvrir ou de fermer le curseur.

- Il n'est pas nécessaire de faire un fetch pour lecture des lignes

Attribut du curseur  
%OPEN, %NOTFOUND  
%FOUND, %ROWCOUNT

## Curseur avec Boucle LOOP?

DECLARE

CURSOR nom curseur IS  
SELECT colonne1, ...

FROM table WHERE condition;  
nom - Chaque ligne de la table %ROWTYPE;

BEGIN

OPEN nom curseur;  
LOOP

FETCH nom curseur INTO "Enregistrement";

EXIT WHEN nom curseur %NOTFOUND;

// Instructions

END LOOP;

CLOSE nom curseur;

END;

- Il faut ouvrir  
le curseur.

(OPEN)

Faire un

FETCH pour  
lire les lignes

- Utiliser une  
condition explicite  
pour sortir de  
la boucle

EXIT WHEN  
curseur %NOTFOUND;

FERMER le curseur  
(CLOSE)

DECLARE

Curseur emp curseur IS

SELECT ename, salary

FROM employee WHERE DEP\_ID

= nom employee employee.ename %TYPE;

salaire employee.salary %TYPE;

BEGIN

OPEN emp curseur;  
LOOP

FETCH emp curseur INTO  
nom employee, salaries;

ENREGISTREMENT; EXIT WHEN  
emp curseur %NOT  
FOUND;

DBMS\_OUTPUT.PUT\_LINE('');

END LOOP;

CLOSE emp curseur;  
END;

## Cursor avec WHILE:

```
DECLARE
CURSOR nom_cursor IS
SELECT colonne1
FROM table WHERE condition;
enregistrement table%ROWTYPE;
BEGIN
OPEN nom_cursor;
FOR tuple
FETCH nom_cursor INTO enregistrement;
WHILE nom_cursor%FOUND LOOP
// Instructions
// Accès aux Tuples suivants
FETCH nom_cursor INTO enregistrement;
END LOOP;
CLOSE nom_cursor;
END;
```

## ④ Instructions PL/SQL :

→ Affectations :

Affecter un résultat SQL à des variables.

DECLARE

maxsal NUMBER;

BEGIN

SELECT MAX(salary) INTO maxsal FROM employee;

DBMS-OUTPUT.PUT-LINE ('Salaire max : ' || maxsal);

END;

→ Tests Conditionnels :

IF condition THEN

  // Instructions

ELSEIF autre- $C$  THEN

  // Instructions

ELSE

  // Instructions

ENDIF;

③

## \* Branches:

WHILE-LOOP:

```
WHILE condition LOOP  
  //Instructions  
END LOOP;
```

FOR-LOOP:

```
FOR i IN 1..10 LOOP  
  //Instructions  
END LOOP;
```

LOOP avec EXIT:

```
LOOP  
  //Instructions  
EXIT WHEN condition;  
END LOOP;
```

## ⑥ Traitement des exceptions:

### \* Exceptions système:

Existantes dans ORACLE (automatiquement déclenchées):

NO-DATA-FOUND: Aucun tuple trouvé.

TOO-MANY-ROWS: plus d'un tuple retourné.

ZERO-DIVIDE: Division par zéro.

CURSOR-ALREADY-OPEN: Curseur déjà ouvert.

INVALID-CURSOR: Curseur déjà fermé.

### \* Exceptions utilisateur:

Elle est déclarée par l'utilisateur et cette exception est soulevée par la rule RAISE dans la partie instruction.

PL/SQL: RAISE <nom-exception>;

DECLARE

tropGrand EXCEPTION;

BEGIN

IF salaire > 5000 THEN

RAISE tropGrand;

END IF;

EXCEPTION

WHEN tropGrand THEN

DBMS-OUTPUT.PUT-LINE ('Salaire trop grand');

END;

• Lancer une exception avec RAISE\_APPLICATION\_ERROR permet de définir un message d'erreur personnalisé:  
RAISE\_APPLICATION\_ERROR(-20001, 'Erreur personnalisée');

## ⑥ Procédures et fonctions:

### • Procédures:

Une procédure est un bloc PL/SQL nommé qui exécute une suite d'instructions et peut accepter des paramètres en entrée et/ou en sortie. Elle ne retourne pas de valeur.

```
CREATE OR REPLACE PROCEDURE nom_procedure(  
    param1 IN type, //paramètre d'entrée  
    param2 OUT type //paramètre de sortie
```

) IS

// Déclaration de variables (..., constantes, ...)

BEGIN

// Instructions exécutables (commandes SQL, affectations, ...)

EXCEPTION

// Gestion des exceptions (facultatif)

END;

/

```
CREATE OR REPLACE PROCEDURE calculer_salaire_bruit(  
    salaire_base IN NUMBER,  
    taux_bonus IN NUMBER,  
    salaire_bruit OUT NUMBER) IS
```

BEGIN

salaire\_bruit := salaire\_base + (salaire\_base \* taux\_bonous / 100);  
DBMS\_OUTPUT.PUT\_LINE('Salaire\_bruit: ' || salaire\_bruit);

END;

/

Procédure qui calcule un salaire brut à partir du salaire base et d'un taux de bonus

## \* Functions:

Une fonction est similaire à une procédure, mais elle retourne une valeur, ce qui permet d'utiliser dans des expressions SQL.

`CREATE OR REPLACE FUNCTION nom_fonction`

param1 IN type

) RETURN type IS

// Déclarations de variables, constantes, ...

BEGIN

// Instructions exécutables (commandes SQL, calculs, affectations)

RETURN valeur;

EXCEPTION

// Gestion des exceptions (facultatif)

END;

`CREATE OR REPLACE FUNCTION convertir_euro(`  
montant IN NUMBER)

RETURN NUMBER IS

taux\_conversion CONSTANT NUMBER := 2.20

BEGIN

RETURN montant / taux\_conversion;

END;

une fonction  
qui convertit  
un montant  
en dollars vers  
des euros.

## ⑦ Déclencheurs TRIGGERS :

Un trigger est un bloc PL/SQL qui s'exécute automatiquement lorsqu'un événement spécifique (comme INSERT, UPDATE ou DELETE) se produit sur une table ou une vue. Il permet de gérer des règles de gestion qui ne pourraient pas être implémentées par des contraintes simples.

```
CREATE OR REPLACE TRIGGER nom-trigger
  {BEFORE|AFTER} // Déclencheur avant ou après l'événement
  {INSERT|UPDATE|DELETE} // type d'événement
  ON nom-table // la table ou la vue sur laquelle le trigger
  [FOR EACH ROW] // Déclencheur de ligne (optionnel, mais souvent)
  DECLARE
    // Déclaration de variables (facultatif)
    // Logique du trigger (instructions à exécuter)
  EXCEPTION
    // Gestion des erreurs
  END;
```

### → Types de déclencheurs (Triggers):

#### \* Déclencheur de ligne (ROW Trigger):

Un déclencheur de ligne s'exécute pour chaque ligne affectée par l'événement. Il utilise les pseudo-enregistrements : **:NEW** et **:OLD** pour accéder aux valeurs des colonnes avant et après l'événement.

**:NEW:** utilisé pour accéder aux nouvelles valeurs après un INSERT ou un UPDATE.

**:OLD:** utilisé pour accéder aux anciennes valeurs après un UPDATE ou un DELETE.

**Exemple:** Trigger qui vérifie si un salaire est supérieur à 1000 lors de l'insertion dans la table employee. Si c'est le cas, il empêche l'insertion et lance une exception.

**CREATE OR REPLACE TRIGGER verifier\_salaire  
BEFORE INSERT ON employee  
FOR EACH ROW**

BEGIN

IF :NEW.salary > 1000 THEN

RAISE\_APPLICATION\_ERROR(-20001)

END IF;

END;

• **l'insertion** + Déclencheur d'état (FOR EACH non utilisée).

Un déclencheur d'état se déclenche une seule fois, peu importe le nombre de lignes affectées par l'événement. Il ne peut pas accéder aux valeurs spécifiques individuelles (pas de :**NEW** ou :**OLD**)

**Exemple:** Ce déclencheur vérifie si une suppression est faite sur une table mais empêche toutes les suppressions de se produire.

**CREATE OR REPLACE TRIGGER interdire\_suppression  
BEFORE DELETE ON employee**

BEGIN

RAISE\_APPLICATION\_ERROR(-20009, 'Les suppressions sont interdites sur la table employee');

END;

**FOR EACH ROW**  
Ce déclencheur s'applique à chaque ligne insérée dans la table **employee**. Si le salaire est supérieur à 1000, une erreur est lancée pour empêcher l'insertion.

**BEFORE DELETE**  
Ce déclencheur déclenche avant une suppression et bloque l'opération pour toute ligne affectée.

## Chapitre 4 :Optimisations des SGBD

### I.Introduction:

L'optimisation des bases de données vise à :

- **Améliorer les performances** : Réduire les temps de réponse pour les requêtes.
- **Minimiser les coûts** : Utiliser moins de ressources (mémoire, disques, processeurs).
- **Gérer la scalabilité** : Assurer un fonctionnement efficace malgré l'augmentation des utilisateurs ou des données.

### Types d'optimisation :

1. **Optimisation physique** : Adapte le schéma interne pour optimiser l'exécution réelle des requêtes.
2. **Optimisation logique** : Restructure les requêtes pour améliorer leur efficacité théorique.

### II.Optimisation du schéma interne:

L'optimisation des schémas internes vise à améliorer les performances en s'appuyant sur les fonctionnalités des SGBD et des principes généraux applicables.

### Facteurs clés à considérer :

- **Taille des tuples** : Réduire leur taille (ex. utiliser **VARCHAR** au lieu de **CHAR**) pour économiser les ressources.
- **Nombre de tuples** : Supprimer les données obsolètes pour alléger les tables actives.
- **Fréquence d'exécution des requêtes** : Optimiser les colonnes utilisées dans les requêtes fréquentes (ajouter des index).
- **Complexité des requêtes exécutées** : Simplifier les jointures et optimiser les chemins d'accès via des index ou la dénormalisation.
- **Fréquence des mises à jour** : Éviter les index sur des colonnes mises à jour fréquemment.
- **Temps de réponse attendu** : Utiliser des vues matérialisées pour les cas nécessitant des performances critiques.

- **Distribution des données** : Partitionner les données pour équilibrer la charge et éviter les goulets d'étranglement.

## Évaluation des performances :

1. **Évaluation théorique** : Calculer le coût des opérations (temps, mémoire, E/S).
2. **Évaluation empirique** : Tester un sous-ensemble du schéma avec des requêtes simulées en variant les paramètres (volume de données, utilisateurs).

## Techniques principales d'optimisation :

1. **Indexation** : Créer des index pour accélérer les recherches et réduire les parcours complets.
2. **Partitionnement** : Diviser les tables en partitions pour optimiser les accès ciblés.
3. **Vues concrètes** : Stocker les résultats des requêtes complexes pour réduire les calculs.
4. **Dénormalisation** : Regrouper des tables pour limiter les jointures fréquentes.

### 1. Organisation des fichiers :

Les bases de données stockent les données dans des fichiers divisés en pages fixes contenant les enregistrements.

**Fichiers** : Unités de stockage physiques utilisées sur des supports comme les disques durs ou clés USB.

**Pages (ou blocs)** : Unités logiques de taille fixe organisant les données dans les fichiers, accessibles en une seule opération E/S.

**Enregistrements** : Données logiques (ex. noms, adresses) stockées dans les pages de fichiers.

**Remarque:** Lire ou écrire un enregistrement nécessite l'accès à toute la page contenant cet enregistrement.

## 2. Les méthodes d'accès aux données :

### 2.1. Accès séquentiel :

- **Principe** : Parcours de tous les enregistrements d'un fichier jusqu'à trouver ceux recherchés.
- **Cas d'utilisation** : Lecture complète (ex. `SELECT *`) ou fichiers non indexés.
- **Avantages** : Simple et sans besoin de structures supplémentaires.
- **Inconvénients** : Lent pour les recherches spécifiques et inefficace sur de grandes tables.

### 2.2. Accès par hachage :

**Principe du hachage** : Utilisation d'une fonction de hachage pour calculer l'adresse d'une donnée (par exemple, `hash(ID_Client) = ID_Client % 5`). La clé est transformée en adresse ou bucket où la donnée est stockée.

**Gestion des collisions** : Les collisions se produisent lorsque plusieurs clés mènent au même bucket. Le système de chaînage est utilisé pour gérer cela. Chaque bucket contient une liste chaînée qui stocke les enregistrements en conflit.

#### Avantages :

- Accès rapide pour les recherches exactes.
- Répartition des données dans les buckets, limitant l'espace de stockage supplémentaire.

#### Inconvénients :

- Gestion des collisions peut ralentir les performances.
- Inefficace pour les recherches par plage (`BETWEEN, <, >`).
- Une mauvaise fonction de hachage peut provoquer une répartition inégale des données (création de points chauds).

### 2.3. Accès par index :

Dans l'accès indexé, les données sont organisées avec des index permettant un accès rapide aux enregistrements selon leur clé, sans

parcourir tout le fichier. Les index peuvent être basés sur une ou plusieurs clés et être stockés en mémoire ou sur disque.



## Différences accès par index et hachage :

### Accès par hachage :

Utilise une fonction de hachage pour transformer une clé en une position spécifique, permettant un accès direct aux données. Il est adapté aux requêtes précises (par exemple, `WHERE nom = 'Ali'`), mais ne fonctionne pas avec des plages (`BETWEEN`) ou des tris (`ORDER BY`).

### Accès par index :

Crée une structure auxiliaire qui associe des clés à leurs positions. L'index permet de localiser les données et est utilisé pour les requêtes par plage et les tris (`WHERE nom BETWEEN 'Ahmed' AND 'Zinedine'`, `ORDER BY`).

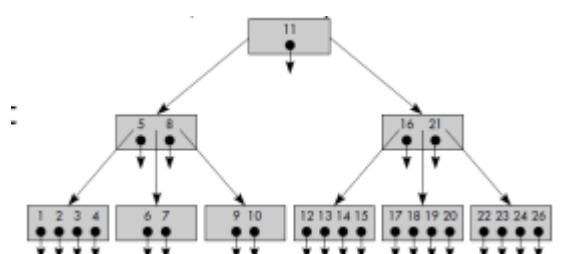
## 3. Les techniques principales d'optimisation :

### 3.1. Indexation (fait partie de l'accès par index) :

Un index est une structure auxiliaire qui associe une valeur de clé à l'emplacement des enregistrements dans un fichier, permettant un accès rapide aux données sans parcourir toute la table. Il utilise des structures optimisées comme les arbres B pour améliorer les performances des requêtes.

#### B-tree:

Le B-tree est une structure arborescente utilisée en bases de données pour



organiser les données de manière hiérarchique. Chaque nœud contient des clés triées et des pointeurs vers des sous-arbres ou des données. Il permet une recherche, insertion et suppression efficaces, même dans de grandes bases de données, en traversant l'arbre suivant les intervalles de valeurs.

Les index sont utilisés sur les tables fréquemment recherchées, associées à des attributs souvent utilisés dans des restrictions ou jointures, très discriminants et rarement modifiés.

### Syntaxe :

- Pour créer un index :

```
CREATE INDEX nom_index ON table (colonne_cle_1,  
colonne_cle_2, ...);
```

- Pour supprimer un index :

```
DROP INDEX idx_nom;
```

### Avantages de l'Index B-Tree :

- Accès rapide aux recherches exactes et par plage, avec une performance stable sur de grandes tables.
- Idéal pour les tris et recherches par plage.
- Maintient l'équilibre automatiquement lors des modifications.

### Inconvénients :

- Rééquilibrages coûteux lors d'insertion/suppression fréquentes.
- Consomme de l'espace supplémentaire dans la base de données.

### Opérateurs physiques Oracle :

1. **TABLE ACCESS FULL** : Parcourt toute la table pour récupérer les données.(séquentiel)
2. **INDEX RANGE SCAN, INDEX UNIQUE SCAN, INDEX FULL SCAN** : Recherche dans un index B-Tree pour récupérer des valeurs ou des plages.
3. **TABLE ACCESS BY INDEX ROWID** : Utilise un index pour localiser les ROWIDs des enregistrements.(opérateur complémentaire)

Exemple de plan d'exécution :

- **INDEX UNIQUE SCAN** : Utilise l'index B-Tree pour trouver les ROWIDs.
- **TABLE ACCESS BY INDEX ROWID** : Utilise les ROWIDs pour accéder aux lignes correspondantes.

### 3.2/Dénormalisation :

La **normalisation** optimise un modèle logique pour éliminer la redondance, fragmentant les données dans plusieurs tables liées par des références. En revanche, la **dénormalisation** regroupe ces tables en une seule, réalisant les jointures statiquement pour améliorer les performances en lecture. Cependant, la dénormalisation doit être utilisée avec précaution pour contrôler la redondance, en respectant trois principes :

1. Avoir une raison valable d'introduire la redondance (par exemple, pour améliorer les performances).
2. Documenter la redondance.
3. Contrôler la redondance via des mécanismes comme les triggers pour éviter l'incohérence.

### Exercice :

On vous demande de remplacer deux créations de tables liées par une clé étrangère par une seule instruction.

### Tables initiales :

```
CREATE TABLE T2 (C char(10) Primary Key,E char(10));
```

```
CREATE TABLE T1 (A char(10) Primary Key,B char(10),C  
char(10) References T2(C),D char(10));
```

### Solution :

Fusionner les deux tables en une seule en ajoutant la colonne E directement dans **T1**, et en supprimant la référence à **T2** :

```
CREATE TABLE T1 (  
A char(10) Primary Key,B char(10),C char(10),
```

```
D char(10),E char(10));
```

### **Explication :**

La solution combine les données de **T2** dans **T1**, éliminant la nécessité de la table séparée et de la clé étrangère. Cela simplifie la structure, mais introduit de la redondance.

### **3.3. Partitionnement de tables :**

**Partitionnement vertical** : Divise une table en isolant les attributs peu utilisés des plus utilisés, ce qui améliore les performances pour les requêtes sur les attributs fréquents. Cela ralentit les jointures et est inutile pour les tables avec peu d'attributs.

**Partitionnement horizontal** : Divise une table en sous-tables basées sur des critères (ex. date, région), améliorant les performances sur les enregistrements fréquents. Cela ralentit les opérations d'union et est inutile pour les petites tables.

### **Exercice :**

Optimiser la requête suivante :

```
SELECT ref1 FROM Tref WHERE (ref2 = 'XXX' OR ref2 =  
'YYY' OR ref2 = 'ZZZ');
```

### **Solution :**

1. **Partitionnement horizontal** : Diviser **Tref** en sous-tables selon les critères de la requête (**ref2** = 'XXX', 'YYY', 'ZZZ'), permettant de ne travailler qu'avec les enregistrements pertinents.
2. **Partitionnement vertical** : Ne conserver que les attributs **ref1** et **ref2**, les seuls nécessaires à la requête, pour éviter de traiter les autres attributs volumineux.

### **Tables partitionnées :**

- **TrefPartition12\_OK** : Enregistrements avec les critères de **ref2** et les colonnes **ref1** et **ref2**.

- `TrefPartition134567_OK` : Contient les autres colonnes de `Tref`, mais non utilisé dans la requête.
- `TrefPartition12_nonOK` et `TrefPartition134567_nonOK` : Enregistrements ne correspondant pas aux critères de `ref2`.

**Requête optimisée :**

```
SELECT ref1 FROM TrefPartition12_OK;
```

Cela réduit la taille des données traitées et améliore les performances.

### 3.4.Vues concrètes :

**Vues concrètes :**

Une vue est une table virtuelle, créée à partir d'une requête SELECT. Elle est définie par :

```
CREATE VIEW nom_view AS SELECT ...;
```

Une vue peut être utilisée comme une table dans des requêtes (SELECT, INSERT, UPDATE, DELETE). Pour la supprimer :

```
DROP VIEW nom_view;
```

**Vue concrète ou matérialisée :**

Contrairement à la vue classique (virtuelle), une vue concrète stocke statiquement les résultats de la requête dans une table. Elle permet d'éviter de recalculer la requête à chaque consultation, offrant ainsi des performances accrues. Cependant, elle nécessite une actualisation régulière pour éviter que les données deviennent obsolètes, souvent via des recalcults programmés ou déclenchés par des événements spécifiques.

Création d'une vue matérialisée :

```
CREATE MATERIALIZED VIEW nom_materialized_view AS  
SELECT ...;
```

## **4.Optimisation des requêtes :**

### **4.1.introduction :**

Une **requête SQL** décrit le besoin de l'utilisateur sans spécifier comment récupérer les données.

Le **SGBD** détermine la méthode d'exécution en s'appuyant sur :

- La structure des données (index, tables, partitions).
- Les statistiques (taille des tables, cardinalité).
- Les coûts estimés des approches possibles.

Ce processus produit un **plan d'exécution** optimal.

#### **4.1.1.Qu'est-ce qu'un plan d'exécution ?**

Un **plan d'exécution** détaille les étapes nécessaires pour exécuter une requête SQL, décrivant :

- Les opérateurs utilisés (ex. : TABLE ACCESS FULL, INDEX RANGE SCAN).
- L'ordre des opérations (filtrage, tri).
- Les chemins d'accès (parcours séquentiel, index).

**Une requête SQL peut avoir plusieurs plans possibles :**

- Accès séquentiel pour parcourir toute la table.
- Utilisation d'un index pour accéder rapidement aux données.

**Exemple :**

Pour `SELECT * FROM commandes WHERE client_id = 123` :

- **Plan 1** : TABLE ACCESS FULL (parcours complet).
- **Plan 2** : INDEX UNIQUE SCAN + TABLE ACCESS BY INDEX ROWID (indexation).

## **Évaluation et choix du meilleur plan:**

Le SGBD évalue et choisit le meilleur plan d'exécution en minimisant le coût total, basé sur :

- Lectures disque, mémoire utilisée, et temps CPU.

## **Étapes du choix d'un plan SQL :**

1. **Écriture** : L'utilisateur rédige la requête SQL.
2. **Analyse** : Traduction en opérations d'algèbre relationnelle.
3. **Génération** : Création de plusieurs plans possibles.
4. **Choix** : Sélection du plan au coût estimé le plus faible.
5. **Exécution** : Suivi du plan pour retourner le résultat.

### **4.1.2. Étapes d'optimisation (en général) :**

1. **Décomposition** : La requête SQL est analysée et traduite en opérations d'algèbre relationnelle (sélections, projections, jointures) pour une optimisation logique.
2. **Optimisation** : Transformation des expressions algébriques en plans d'exécution.
  - **Optimisation logique** : Réorganiser les opérations (ex. : appliquer les sélections avant les jointures).
  - **Optimisation physique** : Choisir les structures (index, parcours séquentiel) et algorithmes optimaux.
3. **Évaluation/Exécution** : Le SGBD exécute le plan choisi pour retourner les données demandées.

## **Exemple :**

- Plan 1 : TABLE ACCESS FULL (parcours complet).
- Plan 2 : INDEX RANGE SCAN (recherche via index pour accélérer).

## 4.2. Étapes d'optimisation (en détail) :

### 4.2.1. Décomposition :

#### Analyse syntaxique :

- Vérifie la conformité grammaticale et détecte les erreurs (ex. : parenthèses non fermées).
- Simplifie les requêtes pour réduire le coût d'exécution.

#### Bonnes pratiques :

- Préciser les colonnes dans `SELECT` au lieu d'utiliser `SELECT *`.
- Ajouter des conditions précises dans `WHERE`.
- Remplacer les sous-requêtes par des jointures si possible.

*Exemple :*

- Sous-requête : `SELECT nom FROM Clients WHERE id_client IN (SELECT id_client FROM Commandes);`
- Jointure : `SELECT c.nom FROM Clients c JOIN Commandes cmd ON c.id_client = cmd.id_client;`

#### Analyse sémantique :

- Vérifie la validité des opérations (types cohérents, pas d'incohérences logiques).
- Simplifie les conditions inutiles (ex. : A or not B and B devient A and B).

#### Optimisation algébrique :

- Traduit la requête en algèbre relationnelle pour optimiser.
  - **Sélection ( $\sigma$ )** : Filtrer les lignes (`WHERE`).
  - **Projection ( $\pi$ )** : Sélectionner les colonnes (`SELECT`).
  - **Jointure ( $\bowtie$ )** : Combiner les tables.
  - **Union/Intersection ( $\cup, \cap$ )** : Fusionner des ensembles.
- Le résultat peut être représenté sous forme d'un arbre algébrique pour une meilleure lisibilité et optimisation.

## Exemple : Traduction en algèbre relationnelle

Pour répondre à la question "Quels films commencent au REX à 20 h ?", voici la traduction de la requête SQL donnée en algèbre relationnelle :

**Données :**

- **Relations :**

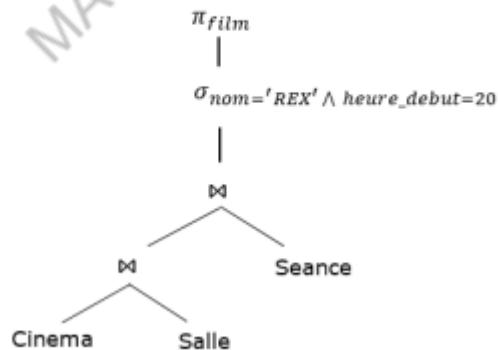
- $Cinema(ID_{cinema}, nom, adresse)$
- $Salle(ID_{salle}, ID_{cinema}, capacite)$
- $Seance(ID_{salle}, heure\_debut, film)$

**Requête SQL :**

```
SELECT film FROM Cinema, Salle, Seance WHERE
Cinema.nom = 'REX' AND Seance.heure_debut = 20 AND
Cinema.ID_cinema = Salle.ID_cinema AND Salle.ID_salle
= Seance.ID_salle;
```

**Traduction algébrique + arbre algébrique :**

$$\pi_{film} \left( \sigma_{nom='REX' \wedge heure\_debut=20} ((cinema \bowtie salle) \bowtie seance) \right)$$



### 4.2.2.Optimisation :

L'optimisation des requêtes SQL vise à trouver l'expression algébrique la plus efficace en réorganisant les opérations (sélections, projections, jointures). L'optimiseur évalue le coût de chaque expression (CPU, mémoire, disque) et choisit la meilleure. Cela se fait en appliquant des

règles de réécriture basées sur l'algèbre relationnelle pour générer des expressions optimisées avant de déterminer le plan d'exécution.

#### 4.2.2.1.Règles de réécriture :

- ❖ Commutativité et l'associativité de la jointure :

$$E_1 \bowtie E_2 = E_2 \bowtie E_1$$

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

- ❖ Cascade de projections (projections imbriquées)

$$\pi_{A1,\dots,An}(\pi_{B1,\dots,Bm}(E)) = \pi_{A1,\dots,An}(E)$$

- ❖ Cascade de restrictions

$$\sigma_{F1}(\sigma_{F2}(E)) = \sigma_{F1 \wedge F2}(E)$$

- ❖ Commutation projections et restriction

- Si F porte sur A<sub>1</sub>,...,A<sub>n</sub>

$$\pi_{A1,\dots,An}(\sigma_F(E)) = \sigma_F(\pi_{A1,\dots,An}(E))$$

- Si F porte aussi sur B<sub>1</sub>,...,B<sub>m</sub>

$$\pi_{A1,\dots,An}(\sigma_F(E)) = \pi_{A1,\dots,An} \left( \sigma_F(\pi_{A1,\dots,An,B1,\dots,Bm}(E)) \right)$$

#### 4.2.2.2.Algorithme de restructuration :

L'algorithme de restructuration vise à optimiser les requêtes SQL en réduisant la taille des données traitées le plus tôt possible. Voici les étapes clés :

1. Séparer les restrictions à plusieurs prédictats en restrictions à un seul prédictat.
2. Descendre les restrictions dans l'arbre pour les appliquer le plus bas possible.
3. Regrouper les restrictions sur une même relation pour simplifier les calculs.

4. Descendre les projections dans l'arbre pour éliminer rapidement les attributs inutiles.
5. Regrouper les projections sur une même relation.

L'idée est de d'abord appliquer les restrictions pour réduire les données, puis les projections pour éliminer les attributs inutiles, et enfin réaliser les jointures, qui sont coûteuses, une fois que la taille des données est réduite au minimum.

#### 4.2.2.3. Arbre algébrique :

Un arbre algébrique est une représentation hiérarchique d'une requête SQL, où chaque nœud représente une opération relationnelle (comme projection, sélection, ou jointure) et les feuilles correspondent aux tables ou relations. L'objectif est de structurer les opérations nécessaires pour répondre à la requête et d'aider à l'optimisation en identifiant les parties coûteuses.

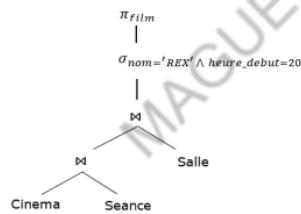
##### Exemple de restructuration

En reprenant l'expression algébrique de la requête retournant les films projetés au REX à 20h :

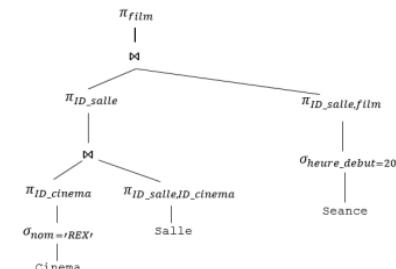
Requête algébrique:

$$\pi_{film} \left( \sigma_{nom='REX'} \wedge heure_debut=20 ((cinema \bowtie salle) \bowtie seance) \right)$$

Arbre algébrique:



##### Exemple de restructuration (suite)



#### 4.2.3. Jointures et optimisation :

La jointure est une opération coûteuse en termes de performance, donc son optimisation est essentielle. Voici trois principaux algorithmes de jointure :

1. **Boucles imbriquées (Nested Loop)** : Chaque ligne de la première table est comparée avec chaque ligne de la seconde table.
2. **Tri-fusion (Sort-Merge)** : Les tables sont d'abord triées selon l'attribut de jointure, puis fusionnées.

3. **Hachage (Hash-Join)** : Les deux tables sont hachées, puis jointes en utilisant la technique de hachage.

Ces algorithmes sont utilisés pour améliorer les performances de la jointure en fonction des caractéristiques des données.

#### **4.2.3.1.Jointure par boucles imbriquées (nested loop) :**

La jointure par boucles imbriquées (nested loop) consiste à parcourir deux tables, R et S, pour trouver les tuples qui satisfont une condition. Pour chaque tuple de R, on parcourt tous les tuples de S. Le coût de cette jointure est mesuré en termes d'entrées/sorties (E/S) :

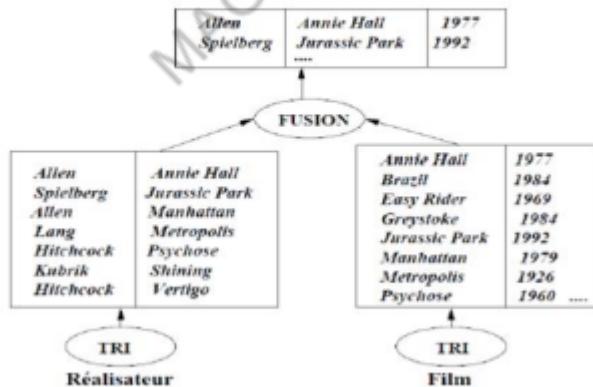
1. On parcourt toutes les pages de R (M pages).
2. Pour chaque tuple de R, on parcourt tous les tuples de S ( $p_R * M * N$ ).

Le total des E/S est donc  $M + p_R * M * N$ .

#### **4.2.3.2.Jointure par tri-fusion (sort-merge) :**

**Idée** : trier les 2 relations sur l'attribut de jointure puis rechercher les tuples satisfaisant la condition de jointure en fusionnant les 2 relations

*Réalisateur (nom, titre) ▷◁ Film (titre, année)*



#### **4.2.3.2.Jointure par hachage (hash join) :**

La jointure par hachage (Hash join) se déroule en plusieurs étapes :

1. Hachage de la table la plus petite sur la clé de jointure pour créer une table de hachage.

2. Parcours de la table la plus grande et hachage de chaque tuple sur la même clé.
3. Recherche dans la table de hachage pour trouver les tuples correspondants et les joindre.
4. Retour des tuples joints.

Les tuples sont organisés en "buckets" (partitions) selon leur valeur de hachage. Lors de la jointure, seul le bucket correspondant est recherché, ce qui évite de parcourir toute la table et améliore ainsi les performances.

#### **4.2.4.Les opérateurs physiques d'oracle :**

Opérateur Algébrique	Opérateur Physique Oracle	Description
Projection	PROJECTION	Sélection de certaines colonnes d'une ou plusieurs tables
Sélection	TABLE ACCESS / INDEX SCAN	Recherche des enregistrements d'une table qui satisfont une condition
Jointure	SORT MERGE JOIN, HASH JOIN	Combinaison de deux tables selon une condition de jointure
Groupe et Agrégat	SORT GROUP BY, HASH GROUP BY	Agrégation des enregistrements selon une ou plusieurs colonnes et calcul de fonctions d'agrégation (par exemple, SUM, COUNT, AVG, MAX, MIN)
Tri	SORT	Ordonnancement des enregistrements selon une ou plusieurs colonnes
Union	CONCATENATION	Combinaison des résultats de deux ou plusieurs requêtes
Intersection	HASH JOIN SEMI	Retourne les enregistrements communs à deux tables
Différence	HASH JOIN ANTI	Retourne les enregistrements d'une table qui ne se trouvent pas dans une autre table

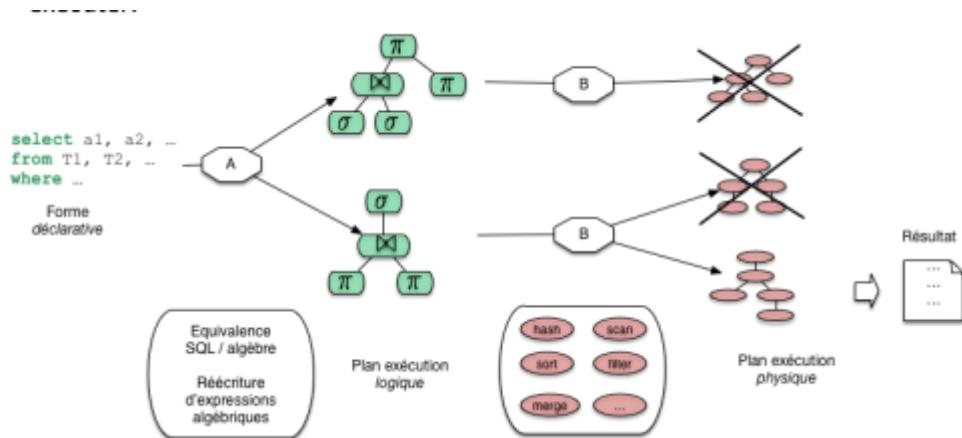
#### **4.2.5.Les phases de l'optimisation :**

Le passage de SQL à un plan se fait en deux étapes :

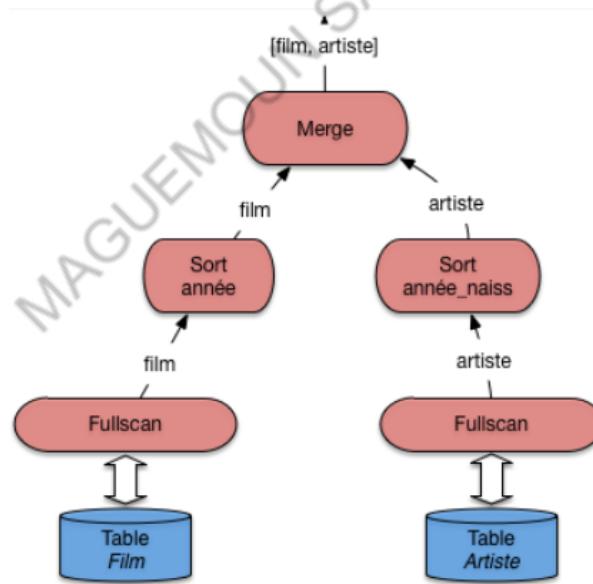
1. **Expression de la requête en langage algébrique** : La requête SQL est transformée en un plan d'exécution logique, avec plusieurs expressions équivalentes possibles grâce à l'algèbre relationnelle.
2. **Choix des opérateurs physiques** : On sélectionne les algorithmes et opérateurs physiques à exécuter pour créer le plan

d'exécution physique, avec plusieurs options disponibles pour chaque choix d'algorithme et d'opérateur.

Chaque étape offre donc plusieurs alternatives possibles pour optimiser l'exécution.



#### 4.2.6. Plan d'exécution physique :



#### 4.3. Outils d'analyse et d'optimisation sous oracle :

**EXPLAIN PLAN** : Génère un plan d'exécution d'une requête sans l'exécuter. Syntaxe :

```
EXPLAIN PLAN FOR SELECT * FROM Clients WHERE age > 30;SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

**DBMS\_XPLAN.DISPLAY** : Affiche le plan d'exécution sous un format lisible, fournissant des détails sur les opérations, les coûts et la cardinalité.

**AUTOTRACE** : Affiche les statistiques d'exécution après l'exécution réelle d'une requête. Activation :

```
SET AUTOTRACE ON; SELECT * FROM Clients WHERE age > 30;
```

## Résumé du chapitre 4 : Optimisations des SGBD

---

### I. Introduction

L'optimisation des bases de données vise à :

- **Améliorer les performances** : Réduire les temps de réponse.
- **Minimiser les coûts** : Utiliser moins de ressources.
- **Gérer la scalabilité** : Maintenir l'efficacité avec des volumes de données croissants.

### Types d'optimisation :

1. **Physique** : Adapter le schéma interne pour optimiser l'exécution.
  2. **Logique** : Restructurer les requêtes pour les rendre plus efficaces.
- 

### II. Optimisation du schéma interne

- **Facteurs clés** :

1. Réduire la taille et le nombre de tuples.
2. Optimiser les colonnes des requêtes fréquentes (index).
3. Simplifier les jointures (dénormalisation).
4. Partitionner les données pour équilibrer les charges.

- **Techniques principales** :

1. **Indexation** : Accélère les recherches en créant des index.

2. **Partitionnement** : Divise les tables pour optimiser les accès ciblés.
  3. **Vues concrètes** : Stocke les résultats de requêtes complexes.
  4. **Dénormalisation** : Regroupe les tables pour limiter les jointures.
- 

### III. Organisation des fichiers

- Les données sont stockées dans des fichiers divisés en pages fixes, contenant les enregistrements.
  - Lire/écrire un enregistrement implique l'accès à toute la page.
- 

### IV. Méthodes d'accès aux données

1. **Accès séquentiel** : Parcours complet des enregistrements.
    - **Avantages** : Simple, sans structure supplémentaire.
    - **Inconvénients** : Lent et inefficace pour de grandes tables.
  2. **Accès par hachage** : Utilise une fonction de hachage pour un accès direct aux données.
    - **Avantages** : Rapide pour des recherches exactes.
    - **Inconvénients** : Inefficace pour les plages ou tris, et sensible aux collisions.
  3. **Accès par index** : Utilise des structures auxiliaires comme les arbres B-Tree pour un accès rapide.
    - **Avantages** : Supporte les recherches par plage et les tris.
    - **Inconvénients** : Nécessite un espace supplémentaire et un rééquilibrage coûteux.
- 

### V. Techniques avancées d'optimisation

1. **Indexation** :
  - Utilisation des arbres B pour des recherches et tris rapides.
  - Syntaxe :

- Création : `CREATE INDEX nom_index ON table (colonne);`
- Suppression : `DROP INDEX nom_index;`

## 2. Dénormalisation :

- Regroupe les tables liées pour améliorer les performances en lecture.
- Nécessite un contrôle des redondances (ex. triggers).

## 3. Partitionnement :

- **Vertical** : Séparer les attributs peu utilisés.
- **Horizontal** : Diviser les enregistrements selon des critères spécifiques.

## 4. Vues concrètes :

- Stocke statiquement les résultats d'une requête complexe.
- Requiert une actualisation régulière pour rester pertinent.
- Syntaxe :

- Création : `CREATE MATERIALIZED VIEW nom_view AS SELECT ...;`
  - Suppression : `DROP VIEW nom_view;`
- 

## VI. Optimisation des requêtes

### 1. Plan d'exécution :

- Décrit les étapes nécessaires pour exécuter une requête SQL.
- **Exemple :**
  - Plan 1 : Parcourir toute la table (`TABLE ACCESS FULL`).
  - Plan 2 : Utiliser un index (`INDEX UNIQUE SCAN`).

### 2. Étapes d'optimisation :

- **Décomposition** : Traduire les requêtes en algèbre relationnelle (projection, sélection, jointure).
- **Optimisation logique** : Réorganiser les opérations pour réduire les coûts.
- **Optimisation physique** : Choisir les structures et algorithmes optimaux.

### 3. Jointures :

- **Boucles imbriquées** : Comparaison ligne par ligne.
  - **Tri-fusion** : Trie puis fusionne les tables.
  - **Hachage** : Utilise des buckets pour accélérer la jointure.
- 

## VII. Outils d'analyse et d'optimisation sous Oracle

### 1. EXPLAIN PLAN :

- Génère le plan d'exécution d'une requête sans l'exécuter.

Syntaxe :

```
EXPLAIN PLAN FOR SELECT * FROM table WHERE condition;
```

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

### 2. DBMS\_XPLAN.DISPLAY :

- Affiche le plan d'exécution avec les coûts et cardinalités.

### 3. AUTOTRACE :

- Affiche les statistiques réelles après l'exécution d'une requête.

Activation :

```
SET AUTOTRACE ON;
```

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