BASE DE DONNEES AVANCEE

Dr BATOUMA Narkoy

Chapitre 10

- **Contenu:**
  - Vues;
  - Séquences;
  - Indexes;
  - Déclencheurs;
  - Langage de programmation: les fonctions PL/PGSQL

Les Vues

- La vue est la représentation d'une requête sous la forme d'une table. Cette vue s'actualise automatiquement à chaque **UPDATE** ou **INSERT** ou **DELETE**.

- Dans le contexte des bases de données, une vue est un terme assez générique qui désigne le résultat d’une requête.

- En somme, tout résultat dérivé à partir des données originales peut être considéré comme une vue.

Les Vues

- Les vues en deux mots : tables virtuelles
- Les vues en une phrase : une vue est une table qui est le résultat d'une requête (SELECT) à laquelle on a donné un nom
- Le nom d'une vue peut être utilisé partout où on peut mettre le nom d'une table : SELECT, UPDATE, DELETE, INSERT, GRANT

Pourquoi les Vues?

Les applications des vues sont nombreuses:

- Tout d'abord, on peut **créer des vues pour des questions de sécurité**.
- La vue ne contiendra alors que **les données pertinentes** et elle omettra **les données confidentielles.**
- On peut aussi créer des vues pour simplifier le développement logiciel.
- La vue sert alors de variable intermédiaire.

Importance de la création des Vues

- **Efficacité dans le traitement des données :**
  - Les vues peuvent simplifier considérablement les requêtes en encapsulant les jointures et les filtres complexes.
  - Cela permet non seulement de rendre les données plus accessibles, mais aussi de réduire le risque d'erreurs dans la formulation des requêtes
- **Amélioration de la sécurité :**
  - En contrôlant la visibilité des informations de la base de données grâce aux vues, les données sensibles sont mieux protégées contre les accès non autorisés.
  - Cette technique utilise efficacement le concept du moindre privilège.

Importance de la création des Vues

- **Abstraction des données :**
  - Les vues fournissent un niveau d'abstraction, te permettant de présenter les données dans un format qui est le plus utile pour tes applications, indépendamment de la façon dont les données sont structurées dans les tables sous-jacentes.

  - Cela peut être particulièrement utile dans les scénarios où le schéma de la base de données est susceptible de changer, en offrant un tampon qui maintient le code de l'application cohérent.

- **Code réutilisable :**
  - Une fois qu'une vue est créée, elle peut être réutilisée dans diverses requêtes et applications, ce qui rationalise les efforts de développement et assure la cohérence de l'accès aux données.

Mais attention!

- Bien que les vues offrent des avantages considérables, il est également important de reconnaître leurs limites.

- Les vues opèrent sur les données sous-jacentes en temps réel, ce qui signifie que la performance peut être un problème avec des vues complexes sur de grandes bases de données.

- De plus, comme les vues font abstraction des tables sous-jacentes, elles peuvent parfois masquer les détails des relations entre les données.

- Il est essentiel de comprendre ces nuances pour exploiter efficacement les vues SQL dans le cadre de ta stratégie de base de données.

Etapes de la création d'une vue en SQL

La création d'une vue en SQL peut être décomposée en une série d'étapes simples, garantissant la clarté et la précision de ta stratégie de gestion de base de données :

1) Identifie le besoin d'une vue en fonction des exigences de requêtes répétées ou des considérations de sécurité des données.

2) Décide des colonnes et des données à inclure dans la vue à partir des tables sous-jacentes.

3) Utilise la syntaxe **CREATE VIEW** pour définir ta vue, en sélectionnant les colonnes nécessaires et en appliquant les filtres souhaités.

4) Exécute l'instruction SQL pour créer la vue dans ton système de gestion de base de données.

5) Teste la vue pour t'assurer qu'elle répond à tes exigences et qu'elle fonctionne comme prévu.

Création des vues

CREATE [OR REPLACE] [TEMP | TEMPORARY] [RECURSIVE] VIEW nom [(nom_colonne [, ...])]

[WITH (nom_option_vue [= valeur_option_vue] [, ...])] ----paramètre(s) optionnels pour l'éxécution de la vue

AS requête

[WITH [CASCADED | LOCAL] CHECK OPTION]

Création des vues

CREATE [OR REPLACE]

[FORCE | NOFORCE] VIEW

nom-de-vue[(attr1, ..., attrn)]AS requete

[WITH CHECK OPTION

[CONSTRAINT nom-contrainte]]

[WITH READ ONLY]]

Vue constituant une restriction de la table EMP aux employés du département 10 :

```
CREATE VIEW EMP10 AS
SELECT * FROM EMP
WHERE deptno = 10
```

Vue comme table

Utilisation d'une vue comme si elle était une table

SELECT ...

FROM nom-de-vue

WHERE ...

Suppression d’une vue

DROP VIEW nom-de-vue

- La suppression d’une vue n’entraîne pas la suppression des données

- Les vues figurent dans les tables systèmes
  - **ALL_CATALOG,**
  - **USER_VIEWS**
  - **et ALL_VIEWS**

Renommer une vue

RENAME ancien-nom **TO** nouveau-nom

Exemple d'une vue simple

- Imagine une base de données qui stocke les détails des employés dans une table nommée *Employés*.

- **Pour accéder fréquemment aux noms et aux départements des employés actifs sans avoir à écrire la requête complète à chaque fois**, tu peux créer une vue :

```
CREATE VIEW ActiveEmployees AS
        SELECT Name, Department
                FROM Employees
                        WHERE Status = 'Active' ;
```

- Cette vue *ActiveEmployees* **récupère les noms et les départements de tous les employés marqués comme "Actifs"** dans la colonne de statut, ce qui simplifie l'accès aux données pour les détails des employés actifs.

Exemple d’une vue complexe (1)

Pour mieux comprendre :

- Pour créer une vue plus complexe, considère un scénario impliquant une base de données pour une librairie.
- Cette base de données comporte des tables pour les livres, les auteurs et les ventes.

- L'objectif est de créer une vue qui donne un aperçu des ventes de livres, y compris le titre du livre, le nom de l'auteur et le total des ventes, en filtrant uniquement les livres dont les ventes dépassent un certain seuil.

- **La complexité vient du fait qu'il faut joindre plusieurs tables et utiliser des fonctions d'agrégation.**

Exemple d’une vue complexe (2)

Voici un guide étape par étape pour y parvenir :

- Identifie les tables concernées et les relations entre elles.
- Décide des données que tu veux inclure dans ta vue. Pour notre exemple, il s'agirait du titre du livre, du nom de l'auteur et du total des ventes.
- Construis ta requête SQL, en veillant à joindre correctement les tables et à appliquer les filtres et les fonctions d'agrégation nécessaires.
- Crée la vue basée sur ta requête.

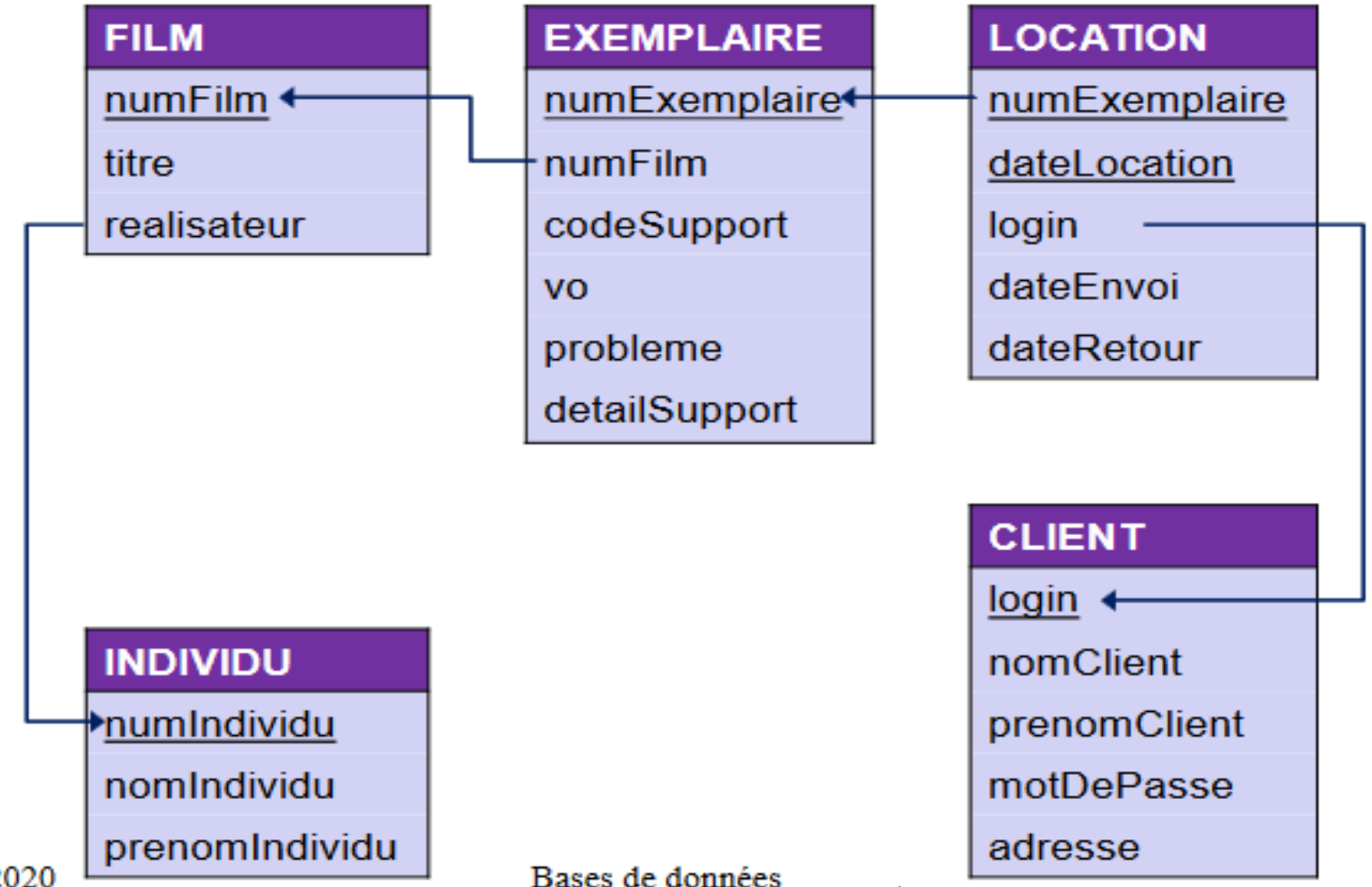
Exemple d'une vue complexe (3)

- L'instruction SQL correspondante pourrait ressembler à ceci :

```
CREATE VIEW BookSalesOverview AS
        SELECT b.Title, a.Name, SUM(s.Quantity) AS TotalSales
                FROM Books b
                        JOIN Authors  a ON b.AuthorID = a.ID JOIN Sales s ON b.ID = s.BookID
                                GROUP BY b.Title, a.Name
                                        HAVING SUM(s.Quantity) > 50;
```

- Cette vue, ***BookSalesOverview***, fournirait un ensemble de données soigneusement compilées montrant les performances des différents livres en termes de ventes, démontrant ainsi la puissance des vues complexes pour extraire et résumer des informations essentielles provenant de plusieurs tables

Vidéothèque



Vidéothèque

CREATE OR REPLACE VIEW *exemplairePlus*

(num, vo, titre, real, support) **AS**

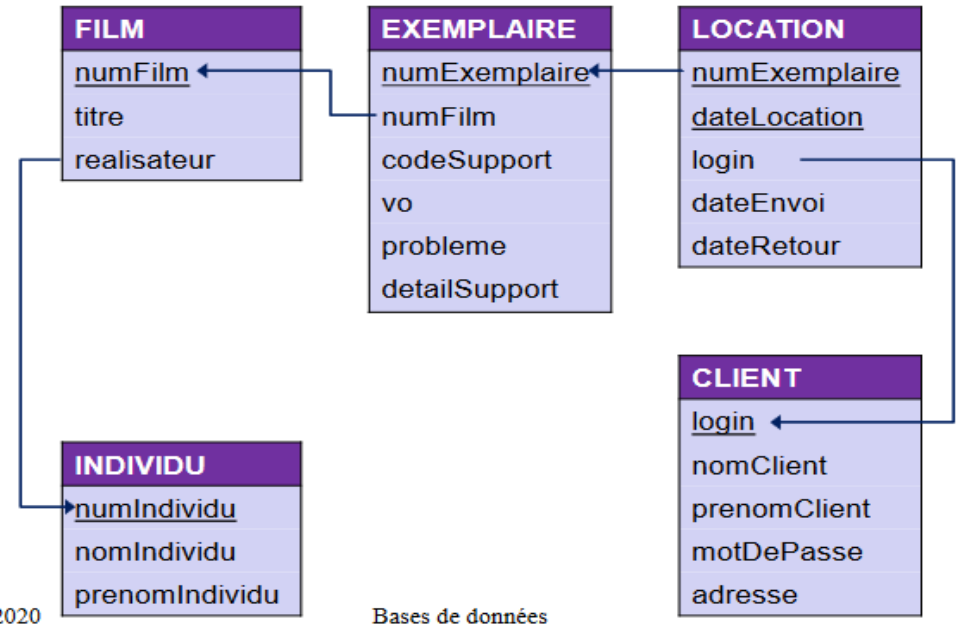
SELECT numExemplaire, vo, titre, nomIndividu, codesupport

FROM ExemplaireE, FilmF, Individu

WHERE E.numFilm= F.numFilm

AND realisateur= numIndividu

AND probleme **IS NULL**;



2019-2020 Bases de données

Vidéothèque

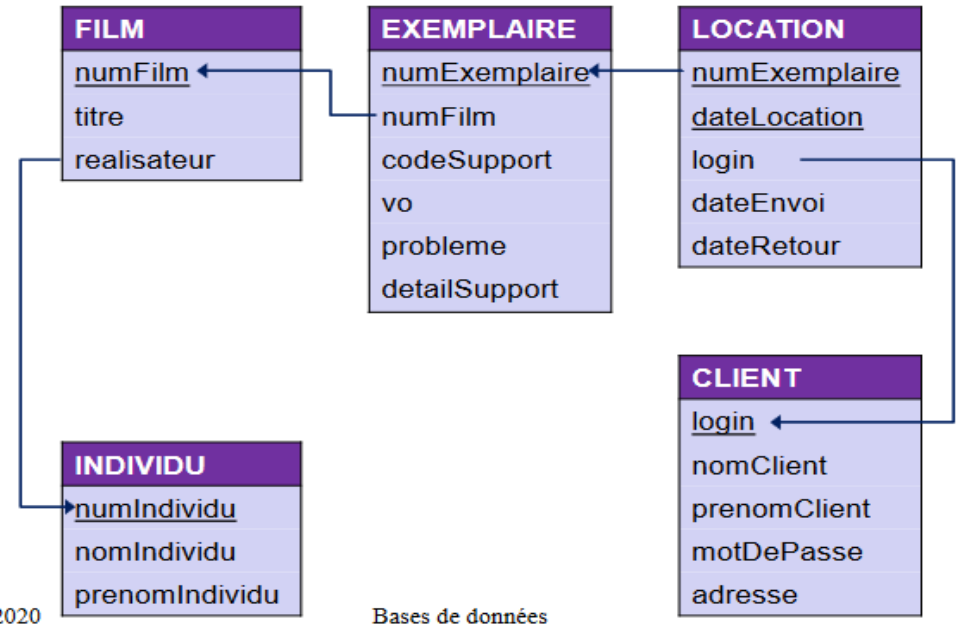
SELECT num, titre, dateLocation, login

FROM *exemplairePlus*, Location

WHERE num= numExemplaire

AND real= ‘ABAKAR‘

AND dateRetour IS NULL;

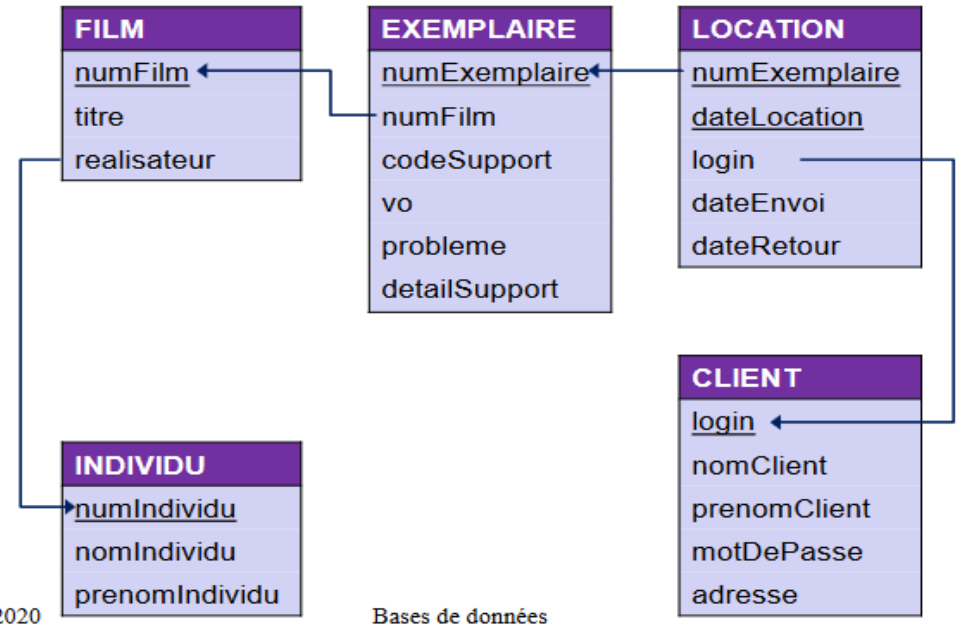


Vidéothèque

INSERT INTO exemplairePlus (num, support)

VALUES (150346, 'DVD');

DROP VIEW exemplairePlus;



Conseils pratiques pour une création des vues efficaces

- **Planifie à l'avance :**
  - Avant de créer une vue, évalue son objectif et la façon dont elle s'intègre dans ton plan global de gestion des données.
  - Cela permet de s'assurer que la vue répond à un besoin pratique et qu'elle est conçue de façon optimale
- **Optimise les performances :**
  - Bien que les vues simplifient l'accès aux données, elles peuvent potentiellement avoir un impact sur les performances.
  - Il faut veiller à n'inclure que les colonnes nécessaires et à utiliser des conditions de filtrage efficaces.

Conseils pratiques pour une création des vues efficaces

- **Maintenir la simplicité :**
  - Les vues complexes sont plus difficiles à maintenir et à comprendre.
  - Les vues simples et bien définies sont préférables pour la facilité d'utilisation et la clarté
- **Révise régulièrement :**
  - Au fur et à mesure que ta base de données évolue, tes vues doivent aussi évoluer.
  - Révise périodiquement tes vues pour t'assurer qu'elles restent pertinentes et performantes

Raisons pour utiliser les vues

1) **Effet macro** :

Remplacer une requête compliquée par des requêtes plus simples

2) **Confidentialité**

Exemple: **CREATE VIEW** emprunteurRestreint **AS**

SELECT login, nomClient, prenomClient

FROM client

Raisons pour utiliser les vues

3) Contraintes d'intégrité (CHECK OPTION) : exemple

```
CREATE VIEW anciensExemplaires
        AS SELECT * FROM Exemplaire WHERE numExemplaire< 2000
                WITH CHECK OPTION;
UPDATE anciensExemplaires SET numExemplaire= 3812
        WHERE numExemplaire= 1318;
```

- **Sans 'WITH CHECK OPTION', c'est possible.**
- **Avec 'WITH CHECK OPTION', c'est impossible.**

4) Augmenter l'indépendance logique

Les applications utilisant les tables de la base ne doivent pas être modifiées si on change le schéma de la base

Conditions de mises a jour pour les vues

Pour UPDATE, DELETE, INSERT la vue ne doit pas contenir :

- Un opérateur ensembliste **(UNION, MINUS, INTERSECT**)
- Un opérateur **DISTINCT**
- Une **fonction d'agrégation** comme attribut
- Une clause **GROUP BY**
- **Une jointure** (la vue doit être construite sur une seule table)

Conditions de mises a jour pour les vues

Pour UPDATE, DELETE, INSERT:

- Les colonnes résultats de l'ordre SELECT doivent être des colonnes réelles d'une table de la base et non des expressions

- Si la vue est construite à partir d'une autre vue, cette dernière doit elle-même vérifier les conditions ci-dessus

Exemples de Vues avec WITH OPTION

- Une vue définie avec l'option **WITH CHECK OPTION** applique toutes les lignes qui sont modifiées ou insérées dans l'instruction SELECT pour cette vue.

- Les vues avec l'option de vérification sont également appelées ***vues symétriques***.

- Par exemple, une vue symétrique qui ne **renvoie que les employés du service 10 n'autorise pas l'insertion d'employés dans d'autres services**.

- Cette option, par conséquent, **garantit l'intégrité des données en cours de modification dans la base de données, en renvoyant une erreur si la condition est violée lors d'une opération INSERT ou UPDATE**.

Exemple 1

- Voici un exemple de définition de vue utilisant WITH CHECK OPTION. Cette option est requise pour s'assurer que la condition est toujours vérifiée. **La vue garantit que le DEPT est toujours 10.** Cela limitera les valeurs d'entrée pour la colonne DEPT. Lorsqu'une vue est utilisée pour insérer une nouvelle valeur, l'option WITH CHECK OPTION est toujours appliquée:

```
CREATE VIEW EMP_VIEW2 (EMPNO, EMPNAME, DEPTNO, JOBTITLE, HIREDATE) AS

SELECT ID, NAME, DEPT, JOB, HIREDATE FROM EMPLOYEE

WHERE DEPT=10 WITH CHECK OPTION;
```

- Si cette vue est utilisée dans une instruction INSERT, la ligne est rejetée si la colonne DEPTNO n'est pas la valeur 10.

Exemple 2

- Dans une vue, vous pouvez mettre un sous-ensemble de données tabulaires à la disposition d'un programme d'application et valider les données à insérer ou à mettre à jour.

- Une vue peut avoir des noms de colonne différents de ceux des colonnes correspondantes dans les tables d'origine.

- Exemple :

```
CREATE VIEW <name> (<column>, <column>, <column>)

    SELECT <column_name>

        FROM <table_name> WITH CHECK OPTION
```

Exemple 3

- L'utilisation des vues offre une certaine souplesse dans la manière dont vos programmes et vos requêtes d'utilisateur final peuvent examiner les données tabulaires.

- L'instruction SQL suivante crée **une vue sur la table EMPLOYEE qui répertorie tous les employés du service A00 avec leurs employés et numéros de téléphone:**

```
CREATE VIEW EMP_VIEW (DA00NAME, DA00NUM, PHONENO) AS

    SELECT LASTNAME, EMPNO, PHONENO

        FROM EMPLOYEE

            WHERE WORKDEPT = 'A00' WITH CHECK OPTION
```

Exemple 4

- La clause WITH CHECK OPTION indique que **toute ligne mise à jour ou insérée dans la vue doit être vérifiée par rapport à la définition de la vue et rejetée si elle n'est pas conforme**. Cela améliore l'intégrité des données mais nécessite un traitement supplémentaire. **Si cette clause est omise, les insertions et les mises à jour ne sont pas vérifiées par rapport à la définition de la vue.**

- L'instruction SQL suivante crée la même vue sur la table EMPLOYEE à l'aide de la clause SELECT AS:

```
CREATE VIEW EMP_VIEW SELECT LASTNAME AS DA00NAME, EMPNO AS DA00NUM, PHONENO

    FROM EMPLOYEE

        WHERE WORKDEPT = 'A00' WITH CHECK OPTION
```

Les Vues

- Il est possible de créer **une vue temporaire qui persistera uniquement pendant la session de l'utilisateur**.

- Le but des vues est de masquer une complexité, qu'elle soit du côté de la structure de la base ou de l'organisation des accès

  - Dans le premier cas, elles permettent de fournir un accès qui ne change pas même si les structures des tables évoluent.

  - Dans le second cas, elles permettent l'accès à seulement certaines colonnes ou certaines lignes.

Les Vues

Les vues peuvent en effet utiliser deux algorithmes différents :

- **MERGE** : les clauses de la requête sur la vue (WHERE, ORDER BY…) sont fusionnées à la requête définissant la vue ;

- **TEMPTABLE** : une table temporaire est créée avec les résultats de la requête définissant la vue, et la requête de sélection sur la vue est ensuite exécutée sur cette table temporaire.

Les Vues

- Avec l'algorithme **MERGE**, tout se passe comme si l'on exécutait la requête directement sur les tables contenant les données. On perd un petit peu de temps à fusionner les clauses des deux requêtes, mais c'est négligeable.

- Par contre, avec **TEMPTABLE**, non seulement on exécute deux requêtes (une pour créer la vue temporaire, l'autre sur cette dernière), mais en plus, **la table temporaire ne possède aucun index**, contrairement aux tables normales. Une recherche dans la table temporaire peut donc prendre plus de temps que la même recherche sur une table normale, pour peu que cette dernière possède des index.

Les Vue matérialisées

- Finalement, on peut choisir **de matérialiser une vue pour des raisons de performance.**
- Les vues matérialisées sont des objets assez utiles, permettant un **gain de performance** relativement important lorsqu'ils sont bien utilisés
- C’est le type d’application qui nous intéresse ici.
- Dans ce sens, la vue devient alors une forme d’index : elle sert essentiellement à accélérer les opérations.
- Comme leur nom l'indique, les vues matérialisées sont **des vues dont les données sont matérialisées**, c'est-à-dire stockées

Les Vue matérialisées

- Si on matérialise la vue, la question de sa *mise à jour* se pose.

- Certains systèmes mettent à jour automatiquement les vues lorsque les données originales sont mises à jour.

- On dit alors qu'on a une *vue dynamique*. Le résultat net est que la mise à jour des données prend alors plus de temps, mais on n'a pas à se soucier de la cohérence et de l'exactitude des données

- Il existe aussi des formes hybrides de mises à jour : par exemple, un entrepôt pourra ne mettre à jour les vues qu'une fois par jour, même si de nouvelles données apparaissent plus fréquemment.

Les Vue matérialisées

- **DEFINITION** : Comme la vue, c'est la représentation d'une requête sous la forme d'une table

- **CRÉER UNE VUE MATERIALISEE**

CREATE **MATERIALIZED VIEW** [IF NOT EXISTS] nom_table

[(nom_colonne [, ...])]

[WITH (paramètre_stockage [= valeur] [, ...])] [TABLESPACE nom_tablespace]

AS requête [WITH [NO] DATA].

- **Les données ne sont actualisées qu'avec la commande :**

REFRESH MATERIALIZED VIEW [CONCURRENTLY] nom [WITH [NO] DATA]

Exemple

```
CREATE TABLE facture (
    no_facture    integer         PRIMARY KEY,
    no_vendeur    integer,        -- identifiant du vendeur
    date_facture  date,           -- date de la vente
    mtt_facture   numeric(13,2)   -- montant de la vente
);
```

```
CREATE MATERIALIZED VIEW resume_ventes AS
  SELECT
      no_vendeur,
      date_facture,
      sum(mtt_facture)::numeric(13,2) as mtt_ventes
    FROM facture
    WHERE date_facture < CURRENT_DATE
    GROUP BY
      no_vendeur,
      date_facture
    ORDER BY
      no_vendeur,
      date_facture;

CREATE UNIQUE INDEX ventes_resume_vendeur
  ON resume_ventes (no_vendeur, date_facture);
```

Une tâche de fond pourrait être planifiée pour mettre à jour les statistiques chaque nuit en utilisant cette requête SQL :

```
REFRESH MATERIALIZED VIEW resume_ventes;
```

… et les vues non matérialisées…

- **Vues non matérialisées**:
  - Il s'agit de la forme standard des vues qui ne stockent pas de données physiques. Au lieu de cela, elles agissent comme des requêtes sauvegardées sur la base de données, exécutées dynamiquement pour récupérer des données. Elles offrent une certaine flexibilité et un accès aux données en temps réel, mais peuvent entraîner des surcoûts de performance pour les requêtes complexes.

- **Vues matérialisées**:
  - Contrairement à leurs homologues non matérialisés, les vues matérialisées stockent le résultat de la requête sous la forme d'une table physique, qui peut être rafraîchie périodiquement. Cela permet une récupération plus rapide des données, mais nécessite un espace de stockage et une gestion supplémentaires pour s'assurer que les données stockées restent à jour.

Séquences

❑Utilité d'une séquence

- Dans certains cas, nous avons besoin de générer des nombres séquentiels et uniques, le plus souvent pour les insérer dans des clés primaires.
- Une séquence est un objet BD qui stocke une valeur qui s'incrémente à chaque fois où on la consulte.

❑ Création d'une séquence

```
CREATE SEQUENCE nom_seq
 MINVALUE min_val MAXVALUE max_val
 START WITH val INCREMENT BY val_inc
 [CYCLE|NOCYCLE]
```

Création et suppression des Séquence

CREATE SEQUENCE nom –séquence

[**INCREMENT BY** (1 | valeur)]

[**START WITH** valeur]

[**MAXVALUE** valeur| **NOMAXVALUE**]

[**MINVALUE** valeur| **NOMINVALUE**]

[**CYCLE | NOCYCLE**]

[**CACHE** (valeur| 20) | **NOCACHE**]

Pré-génération des valeurs (nombre des valeurs stockées en mémoire)

Suppression:

DROP SEQUENCE nom -séquence

Séquence en oracle

- Nous allons créer une **SEQUENCE** qui se nomme **SEQ_EXP** qui va commencer par **1**, se terminer par **9999999** et incrémenter par **1** :

```
CREATE SEQUENCE SEQ_EXP
        MINVALUE 1
        MAXVALUE 9999999
        INCREMENT BY 1
        START WITH 1;
```

  - **MINVALUE :** valeur minimale qui peut prendre la SEQUENCE.
  - **MAXVALUE :** valeur maximale qui peut prendre la SEQUENCE.
  - **INCREMENT BY :** nombre avec lequel la SEQUENCE va s'incrémenter.
  - **START WITH :** le nombre avec lequel la SEQUENCE va commencer.

Exemple

❑**Exemple:**

```
CREATE SEQUENCE emp_seq
 MINVALUE 100 MAXVALUE 1000
 START WITH 500 INCREMENT BY 1 CYCLE;
```

- Pour utiliser une séquence, on utilise les pseudo-colonnes `nom_seq.curval` (renvoie la valeur courante de la séquence), et `nom_seq.nextval` (valeur suivante de la séquence).
- Pour insérer un nouvel employé, on peut utiliser la séquence `emp_seq` pour dériver une valeur séquentielle et unique de la clé primaire `empno`. Exemple:

```
INSERT INTO emp(empno,ename,sal)
 VALUES(emp_seq.nextval,'PATRICK',1200);
```

Exemple de séquence

```
INSERT INTO film(numFilm, titre)
        VALUES(masequence.NEXTVAL, 'Kill Bill')

INSERT INTO exemplaire (numExemplaire, numFilm)
        VALUES (290870, masequence.CURRVAL)
```

Exemple de séquence

CREATE SEQUENCE ORDER_SEQ

START WITH 500

INCREMENT BY 1

MAXVALUE 1000

NOCYCLE

CACHE 24;

CREATE SEQUENCE masequence

START WITH 1000

INCREMENT BY 30

NOMAXVALUE

NOCYCLE

Observons les choses!!

CREATE SEQUENCE ORDER_SEQ
START WITH 500
INCREMENT BY 1

MAXVALUE 1000 **NOCYCLE CACHE 24**;

CREATE TABLE ORDERS (ORDERNO **SMALLINT NOT NULL,** CUSTNO **SMALLINT);**

INSERT INTO ORDERS (ORDERNO, CUSTNO) **VALUES** (**NEXT VALUE FOR** ORDER_SEQ, 12);

SELECT * FROM ORDERS;

N ° de commande	NON PERSONNALISE
500	12

INSERT INTO ORDERS (ORDERNO, CUSTNO) **VALUES** (**NEXT VALUE FOR** ORDER_SEQ, 12);

N ° de commande	NON PERSONNALISE
500	12
501	12

Observons les choses!!

modifiez l'incrément des valeurs de la séquence ORDER de 1 à 5

ALTER SEQUENCE ORDER_SEQ **INCREMENT BY** 5;

Exécutez à nouveau l'instruction INSERT, puis SELECT

INSERT INTO ORDERS (ORDERNO, CUSTNO) **VALUES** (**NEXT VALUE FOR** ORDER_SEQ, 12);

N ° de commande	NON PERSONNALISE
500	12
501	12
528	12

Lorsque l'instruction ALTER SEQUENCE est émise, le système supprime les valeurs affectées et redémarre avec la valeur disponible suivante ; dans ce cas, le 24 d'origine qui a été mis en cache, plus l'incrément suivant, 5.

Utilité des Indexe

- Afficher les employés dont le job est CLERK:

```
SELECT * FROM emp
     WHERE job='CLERK';
```

- A l'exécution, on doit visiter chaque ligne de **EMP** et évaluer la condition `job='CLERK'`. Le temps d'exécution dépendra du nombre de lignes de EMP !

- Si les utilisateurs de notre base consultent fréquemment EMP avec le critère de recherche JOB, il nous convient de créer un index sur la table EMP sur la colonne JOB.

- Créer un outil d'accès pour les champs les plus fréquemment utilisés pour accélérer leur accès et leur recherche.

Création des Indexes

```
CREATE INDEX nom_ind ON tab(col1,col2,…);
```

Cette requête crée un indexe sur la table `tab` sur les colonnes spécifiées entre parenthèses.

Exemple:

```
CREATE INDEX emp_job ON emp(job);
```

- **Inconvénients d'un index**

1. Consomme de l'espace disque
2. Nécessite une maintenance (mise à jour) à chaque fois où la table indexée est mise à jour.

Indexe: concept de base

- Un tel index inclut une liste d'indices pour chaque valeur distincte de JOB. Les indices référencent les lignes de la table EMP.

CLERK	1,11,12,14
ANALYST	8,13
MANAGER	4,6,7
PRESIDENT	9
SALESMAN	2,3,5,10

- Maintenant, pour exécuter cette requête

```
SELECT * FROM emp
        WHERE job='CLERK';
```

- L'accès se fait directement à partir de l'index !