Les Vues

Les Vues sont un outil du langage SQL qui permet de visualiser les données d'une base de données. Elles permettent également de contrôler les accès aux données.

1. Visualisation des données à travers les vues

Dans une base de données relationnelle, les données sont stockées dans des tables dont la structure permet de garantir leur intégrité en éliminant toutes les redondances et les incohérences. Toutefois, cette structuration de ces données n'est pas toujours adaptée à la façon dont les utilisateurs veulent les visualiser. Pour cette raison, il est possible de créer des vues (ou vues immatérielles) qui permettent d'obtenir une visualisation de données adaptée aux besoins des utilisateurs et qui sont générées à partir des données qu'il y a dans les tables.

Une vue est créée à l'aide d'une requête `SELECT` qui porte sur des tables ou sur d'autres vues existantes. Une vue est considérée comme une table virtuelle car elle n'a pas d'existence propre. Seule sa structure (la requête SQL sur laquelle elle est construite) est stockée. Une vue ne stocke pas les données qui résultent de l'exécution de sa requête. Cela signifie qu'une vue doit se recharger chaque fois qu'elle est invoquée dans une requête et qu'elle ne permet donc pas de diminuer le temps d'exécution d'une requête.

1.1 Création d'une vue

Syntaxe :

```
CREATE [OR REPLACE] VIEW nomvue [(alias1, alias2 …)]
AS requêteSQL
[WITH READ ONLY]
[WITH CHECK OPTION]
```

- `alias` : désigne le nom de chaque colonne de la vue. Si l'alias n'est pas présent, la colonne prend implicitement le nom de l'expression renvoyée par la requête de définition.
- `WITH READ ONLY` : déclare la vue non modifiable par des `INSERT`, `UPDATE` ou `DELETE`. On ne pourra alors faire que des `SELECT` sur la vue.
- `WITH CHECK OPTION` : empêche un ajout ou une modification non conforme à la définition de la vue.

Exemple : Soit le schéma relationnel suivant :

CLIENTS (idClient, nomClient, prenomClient, villeClient)
COMMANDES (idCommande, dateCommade, montantCommande, idClient#)

On veut réaliser une vue *ClientsActifs* contenant pour chaque client qui a passé une ou des commandes, l'identifiant du client ainsi que son nom, son prénom, sa ville et le nombre de commandes passées.

```
CREATE OR REPLACE VIEW ClientsActifs(idClient, nomClient, prenomClient, villeClient,
                       nbCommandes) AS
SELECT cl.idClient, nomClient, prenomClient, villeClient, COUNT(idCommande)
FROM Clients cl
     JOIN Commandes co ON cl.idClient = co.idClient
GROUP BY cl.idClient, nomClient, prenomClient, villeClient;
```

Ou alors (on ne nomme pas les colonnes, elles sont déterminées de manière implicite par la requête)

```
CREATE OR REPLACE VIEW ClientsActifs AS
SELECT cl.idClient, nomClient, prenomClient, villeClient,
       COUNT(idCommande) AS nbCommandes
FROM Clients cl
     JOIN Commandes co ON cl.idClient = co.idClient
GROUP BY cl.idClient, nomClient, prenomClient, villeClient;
```

On veut réaliser une vue *Prospect* contenant les clients qui n'ont pas passé de commande.

```
CREATE OR REPLACE View PROSPECTS AS
SELECT idClient, nomClient, prenomClient, villeClient
FROM Clients cl
WHERE NOT EXISTS (SELECT *
                  FROM Commandes co
                  WHERE co.idClient = cl.idClient);
```

1.2 Utilisation des Vues

Dans une requête d'extraction de données (requêtes `SELECT`), une vue peut être utilisée dans le `FROM` comme s'il s'agissait d'une table normale.

On veut réaliser une requête qui retourne le nom des clients montpelliérains qui ont passé plus de 3 commandes

```
SELECT nomClient
FROM ClientsActifs
WHERE villeClient = 'Montpellier'
AND nbCommandes > 3;
```

On veut réaliser une requête qui retourne le nombre de prospects de la ville de Nîmes

```
SELECT COUNT(*)
FROM Prospects
WHERE villeClient = 'Nimes';
```

On veut réaliser une vue *ProspectParVille* contenant le nombre de prospects pour chaque ville (où il y a des prospects). Pour cela, il est possible de construire cette vue à partir de la vue *Prospects*.

```
CREATE OR REPLACE VIEW ProspectsParVille(ville, nbProspects) AS
SELECT villeClient, COUNT(*)
FROM Prospects
GROUP BY villeClient;
```

1.3 Mise à jour des données à travers une vue

Il est parfois possible d'insérer, modifier ou supprimer des données à travers une vue. On dit alors que la vue est modifiable. Les données ne seront pas mises à jour dans la vue (qui nous le rappelons ne contient pas de données) mais dans la table concernée par la vue. Toutefois, toutes les vues ne sont pas modifiables. Pour qu'une vue soit modifiable, il faut qu'une ligne de la table corresponde à une seule ligne de la vue ; et inversement. Le fait qu'une vue soit modifiable ou non dépend donc de sa structure. Et les choses sont assez différentes selon que l'on ait affaire à une vue construite à partir d'une seule table (vue monotable) ou si la vue est construite avec une ou plusieurs des jointures (vue multitables).

1.3.1 Modifications à travers une Vue monotable

- Pour qu'une vue soit modifiable (`INSERT, UPDATE` ou `DELETE`) :
  - elle ne doit pas être créée avec l'option `WITH READ ONLY.`
  - il ne doit pas y avoir de `DISTINCT` ou de fonction (`AVG, SUM...`) dans le `SELECT.`
  - il ne doit pas y avoir de `GROUP BY, ORDER BY` ou `HAVING` dans la requête de définition.

  Soit la vue `ClientsParVille` contenant le nombre de clients par ville

  ```
  CREATE OR REPLACE VIEW ClientsParVille(ville, nbClients) AS
  SELECT villeClient, COUNT(*)
  FROM Clients
  GROUP BY villeClient;
  ```

  Il n'est pas possible de réaliser des insertions, des modifications ou des suppressions à travers cette vue (car il y a un `GROUP BY` et donc, une ligne de la vue est construite à partir de plusieurs lignes de la table). La vue est donc non modificable. Par exemple,

  ```
  INSERT INTO ClientsParVille VALUES('Montpellier',12);
  ```

  Retournerait l'erreur suivante : ORA-01733: les colonnes virtuelles ne sont pas autorisées ici

De même, la vue `TousLesClientsNimois` contenant les clients Nîmois

```
CREATE OR REPLACE VIEW TousLesClientsNimois
AS SELECT idClient, nomClient, prenomClient
   FROM Clients
   WHERE villeClient = 'Nimes'
WITH READ ONLY;
```

Cette vue n'est pas modifiable car elle possède l'option **`WITH READ ONLY`**

- Pour pouvoir réaliser un `INSERT` à travers une vue, la clé primaire de la table source (ainsi que tous les attributs non `NULL`) doit être inclue dans la vue.

  Par exemple, avec la vue suivante :

  ```
  CREATE OR REPLACE VIEW TousLesClients(nom, prenom) AS
  SELECT nomClient, prenomClient
  FROM Clients;
  ```

  Il ne sera pas possible d'ajouter un client à travers la vue `TousLesClients` car la clé primaire de la table source ne serait pas renseignée. Mais il sera possible de modifier les colonnes à cette vue.

  ```
  UPDATE TousLesClients
  SET prenom = 'Judas'
  WHERE nom = 'Bricot';
  ```

  ```
  DELETE FROM TousLesClients WHERE nom = 'Stico';
  ```

- Si une vue est déclarée avec l'option `WITH CHECK OPTION` il ne sera pas possible d'insérer à travers la vue des données qui ne respectent pas la condition de sélection de la vue (dans le `WHERE`). Dans le cas contraire, ces insertions seront possibles. Elles ne seront alors pas visibles dans la vue, mais elles seront malgré tout présentes dans la table.

  Par exemple, avec la vue suivante :

  ```
  CREATE OR REPLACE VIEW TousLesClientsMontpellierains AS
  SELECT idClient, nomClient, prenomClient, villeClient
  FROM Clients
  WHERE villeClient = 'Montpellier'
  ```

  A travers cette vue, il sera possible d'insérer dans la table Clients des clients qui ne sont pas de Montpellier.

  ```
  INSERT INTO TousLesClientsMontpellierains
  VALUES ('C1', 'Coptère', 'Elie', 'Nîmes');
  ```

  Par contre, cela ne sera pas possible si la vue a été déclarée avec l'option `CHECK OPTION`

  ```
  CREATE OR REPLACE VIEW TousLesClientsMontpellierains AS
  SELECT idClient, nomClient, prenomClient, villeClient
  FROM Clients
  WHERE villeClient = 'Montpellier'
  WITH CHECK OPTION;
  ```

1.3.2 Modifications à travers une Vue multitables

Une vue mutitables est une vue qui contient, dans sa définition, plusieurs tables (avec une ou des jointures). Pour ces vues, les mises à jour sont plus complexes.

Création de la vue `CommandesEtClients` contenant les Commandes avec les informations sur le client.

```
CREATE OR REPLACE VIEW CommandesEtClients  AS
SELECT idCommande, montantCommande, com.idClient, nomClient,
       prenomClient, villeClient
FROM Commandes com
     JOIN Clients cl ON com.idClient = cl.idClient;
```

Dans les vues multitables, on ne peut modifier que les données des tables dites protégées par sa clé. Une table est dite protégée par sa clé (*key preserved*), si chaque valeur de sa clé primaire (ou d'une clé unique) est aussi unique dans la vue. En gros, lorsque la clé de la table pourrait jouer le rôle de clé dans la vue (si la vue pouvait avoir une clé).

Pour qu'une vue multitables soit modifiable il faut donc que toutes les colonnes de la vue proviennent de la table préservée par la clé. Il est à noter qu'il ne faut pas que la vue soit déclarée avec l'option `WITH CHECK OPTION` (sinon on aura le message d'erreur : `les colonnes virtuelles ne sont pas autorisées ici`).

Dans la vue `CommandesEtClients`, la table préservée est la table `Commandes`, car la colonne `idCommande` identifie chaque enregistrement de la vue de façon unique et pourrait donc jouer le rôle de clé primaire de la vue. Par exemple, si on essaye de réaliser les instructions suivantes sur la vue, on aura les messages qui suivent :

```
INSERT INTO CommandesEtClients
VALUES ('COM125',1000,'C10','Assin','Marc','Montpellier');
```
// Impossible car on essaie d'insérer dans les deux tables en même temps. Or la table Clients n'est pas protégée par sa clé.

```
INSERT INTO CommandesEtClients (idClient, nomClient, prenomClient, villeClient)
VALUES ('C10','Assin','Marc','Montpellier');
```
// Impossible d'insérer des lignes dans la table Clients car non protégée par sa clé.

```
INSERT INTO CommandesEtClients(idCommande, montantCommande)
VALUES ('COM125',1000);
```
// Ligne ajoutée dans la table Commandes car protégée par sa clé.

```
INSERT INTO CommandesEtClients(idCommande, montantCommande, idClient)
VALUES ('COM130', 2000,'C2');
```
// Ligne ajoutée dans la table Commandes car protégée par sa clé.

Attention : pour pouvoir ajouter à travers la vue une commande dans laquelle on renseigne le `idClient`, il faut dans la requête de définition préciser que l'attribut `idClient` provient de la table `Commandes`. Si on indique que le `idClient` provient de la table `Clients`, l'insertion sera impossible.

```
UPDATE CommandesEtClients
SET montantCommande = 1500
WHERE idCommande = 'COM125';
```
// Données modifiées dans la table Commandes car protégée par sa clé.

```
DELETE FROM CommandesEtClients
WHERE idCommande = 'COM120' ;
```
// Supprime la commande 'COM120' dans la table Commandes.

```
UPDATE CommandesEtClients
SET villeClient = 'Nimes'
WHERE nomClient = 'Bricot';
```
// Impossible car la table Clients n'est pas protégée par sa clé.

```
DELETE FROM CommandesEtClients
WHERE nomClient = 'Bricot';
```
// Supprime dans la table Commandes toutes les commandes passées par le client `'Bricot'`

Sous Oracle, afin de savoir dans quelle mesure les colonnes d'une vue sont modifiables, on peut interroger la vue système `USER_UPDATABLE_COLUMNS`.

```
SELECT *
FROM USER_UPDATABLE_COLUMNS
WHERE table_name='COMMANDESETCLIENTS';
```

OWNER	TABLE_NAME	COLUMN_NAME	UPDATABLE	INSERTABL	DELETABLE
PALLEJAN	COMMANDESETCLIENTS	NUMCOMMANDE	YES	YES	YES
PALLEJAN	COMMANDESETCLIENTS	DATECOMMANDE	YES	YES	YES
PALLEJAN	COMMANDESETCLIENTS	CODECLIENT	YES	YES	YES
PALLEJAN	COMMANDESETCLIENTS	NOMCLIENT	NO	NO	NO
PALLEJAN	COMMANDESETCLIENTS	PRENOMCLIENT	NO	NO	NO
PALLEJAN	COMMANDESETCLIENTS	VILLECLIENT	NO	NO	NO

2. Contrôle de l'accès aux données à travers les vues

Sous Oracle, un utilisateur de la base de données possède un schéma (du nom de l'utilisateur) contenant tous ses objets : ses tables, ses vues, ses index, ses procédures, …
Les utilisateurs sont propriétaires des objets de leur schéma. Ils ont donc tous les privilèges sur leurs objets. Ils peuvent même accorder des privilèges sur les objets de leur schéma à d'autres utilisateurs.

2.1 Privilèges objets

Un privilège objet est un droit pour réaliser des actions sur un objet (d'un autre schéma). Les actions possibles sur les objets de type table ou de type vue sont `DELETE`, `INSERT`, `SELECT`, `UPDATE` (`ALL` permet d'englober toutes ces actions). Ces privilèges peuvent être attribués à un ou plusieurs utilisateurs, à un rôle ou à `PUBLIC`. Les privilèges sont attribués ou révoqués grâce aux commandes `GRANT .. TO` et `REVOKE .. FROM`.

Soit la table `CLIENTS(idClient, nomClient, prenomClient, villeClient)` faisant partie du schéma de l'utilisateur `Palleja`. L'utilisateur `Palleja` souhaite donner à l'utilisateur `Nemard` le droit de sélectionner la table `Clients` et le droit de modifier les données des champs prénom et ville de cette même table.

```
GRANT UPDATE(prenomClient,villeClient), SELECT
ON Clients
TO Nemard;
```

Maintenant, l'utilisateur `Nemard` peut exécuter la requête suivante

```
SELECT *
FROM Palleja.Clients;          -- [schema].nomTable
```

L'utilisateur `Palleja` peut révoquer le droit de faire l'action `SELECT` à l'utilisateur `Nemard` avec l'instruction suivante.

```
REVOKE SELECT
ON Clients
FROM Nemard;
```

2.2 Gestion des accès concurrents

Des problèmes d'incohérence ou de perte de mise à jour peuvent apparaître lorsque plusieurs personnes (ou transactions) essayent d'accéder simultanément aux mêmes données d'un schéma (*accès concurrents*). De plus, une suite d'instructions successives qui doivent toutes être exécutées (ou pas du tout) peut être interrompue par une *erreur* ou une *panne*.

Un SGBD est un système transactionnel qui assure la cohérence des données en cas de mise à jour de la base de données, même si plusieurs utilisateurs lisent ou modifient les mêmes données simultanément.

Prenons l'exemple d'un transfert d'argent depuis votre compte épargne vers votre compte courant. Imaginez qu'après une panne votre compte épargne a été débité de 1000€ sans que votre compte courant ait été crédité du même montant !

```
Début transaction
UPDATE CompteEpargne SET montant = montant - 1000 WHERE codeClient = 10 ;
UPDATE CompteCourant SET montant = montant + 1000 WHERE codeClient = 10 ;
Fin transaction ;
```

Le système transactionnel empêche cet incident en invalidant toutes les instructions faites depuis le début de la transaction si une panne survient au cours de cette même transaction.

- La commande **`COMMIT`** permet de valider la transaction en cours. Les modifications deviennent définitives et visibles à tous les utilisateurs.
- La commande **`ROLLBACK`** permet d'annuler la transaction en cours. Toutes les modifications depuis le début de la transaction sont alors défaites.

Il n'existe pas d'ordre SQL qui marque le début d'une transaction. Une transaction débute à la première commande SQL rencontrée ou dès la fin de la transaction précédente et se termine explicitement par les instructions `COMMIT` ou `ROLLBACK`.
Une transaction se termine aussi à la fin d'une session. Si la session se termine normalement (déconnexion de l'utilisateur), la transaction est validée par un `COMMIT` de façon implicite. Si la session se termine anormalement (suite une panne), la transaction est invalidée par un `ROLLBACK`. Sous Oracle, il est possible de travailler en mode autocommit. A ce moment-là, toutes les requêtes seront implicitement suivies d'un `COMMIT`.

En cours de transaction, seul l'utilisateur ayant effectué les modifications les voit. Ce mécanisme est utilisé par les SGBD pour assurer l'intégrité de la base en cas de fin anormale d'une transaction. Oracle utilise un mécanisme de verrouillage pour empêcher deux utilisateurs d'effectuer des transactions incompatibles et régler les problèmes pouvant survenir. Les verrous sont libérés en fin de transaction.
Pour que les autres utilisateurs puissent voir les modifications réalisées par un utilisateur donné, il faut donc que celui-ci valide sa transaction est réalisant un `COMMIT`.

2.3 Utilisation des Vues pour affiner les privilèges

Les vues permettent également d'assurer la confidentialité des données. En effet, un utilisateur va pouvoir montrer certaines lignes ou colonnes d'une table à travers une vue, tout en dissimulant les autres lignes ou colonnes de la table.

Création de la vue `TousLesClientsNimois` contenant l'identifiant, nom, et prénom de tous les clients qui habitent à `Nimes`

```
CREATE OR REPLACE VIEW TousLesClientsNimois AS
SELECT idClient, nomClient, prenomClient
FROM Clients
WHERE villeClient = 'Nimes';
```

Maintenant, le créateur de la vue (`Palleja`) peut donner le droit à l'utilisateur `Nemard` de voir la vue `TousLesClientsNimois` (mais pas la table `Clients`).

```
GRANT SELECT
ON TousLesClientsNimois
TO Nemard ;
```

Et l'utilisateur `Nemard` pourra maintenant utiliser cette vue comme une table.

```
SELECT nomClient, prenomClient
FROM Palleja.TousLesClientsNimois;
```

2.4 Synonymes

Un synonyme est un alias d'un objet (table, vue, procédure, …). Il est possible d'attribuer plusieurs noms à un même objet.
Les avantages d'utiliser les synonymes sont les suivants :

- simplifier l'accès aux objets en abrégeant le nom des tables,
- masquer le vrai nom des objets ou la localisation des objets distants,
- améliorer la maintenance des applications qui l'utilisent (le synonyme garde le même nom tout en référençant un nouvel objet). Il est possible d'utiliser les synonymes pour simplifier l'accès aux vues des autres schémas.

exemple : création d'un synonyme de la vue précèdente `TousLesClientsNimois` de `Palleja`

```
CREATE SYNONYM LesClientsNimois
FOR Palleja.TousLesClientsNimois;
```

La localisation de la vue `TousLesClientsNimois` de `Palleja` est maintenant masquée. L'utilisateur `Nemard` devra utiliser le synonyme `LesClientsNimois` pour accéder à cette vue.

```
SELECT nomClient, prenomClient
FROM LesClientsNimois;
```