Contrôle de l'accès aux données et gestion des transactions

L'accès aux informations et à la base de données doit être contrôlé à des fins de sécurité et de cohérence. Nous allons voir quels sont les aspects du langage SQL qui permettent de contrôler l'accès aux données puis nous verrons comment assurer la cohérence des données grâce aux transactions.

1. Contrôle de l'accès aux données

1.1 Gestion des utilisateurs

Un utilisateur est identifié au niveau de la base par son nom et peut se connecter puis accéder aux objets de la base sous réserve d'avoir reçu un certains nombres de privilèges.
Le schéma est une collection nommée (du nom de l'utilisateur qui en est le propriétaire) d'objets (tables, vues, index, procédures, fonctions…).

Dans la plupart des SGBD, on distingue deux types d'utilisateurs :

- Le DBA (DataBase Administrator) qui a tous les privilèges :
  - Installation et mise à jour de la base de données et des outils,
  - Gestion de l'espace disque et des espaces pour les données (tablespaces),
  - Gestion des utilisateurs et de leurs objets,
  - Optimisation des performances,
  - Sauvegardes, restaurations et archivages…
- Les utilisateurs qui sont propriétaires des objets de leur schéma. Ils ont donc tous les privilèges sur leurs objets. Ils peuvent donner des privilèges sur leurs objets à d'autres utilisateurs.

Exemple de création de l'utilisateur `Palleja` dont le mot de passe est `Pipo`. Ses objets sont stockés (pas plus de 10 Mo) dans le tablespace `Users`. Il devra changer son mot de passe à la première connexion.

```
CREATE USER Palleja
IDENTIFIED BY Pipo
DEFAULT TABLESPACE USERS
QUOTA 10M ON USERS
PROFILE ENSEIGNANT
PASSWORD EXPIRE ;
```

Le profil regroupe des caractéristiques système (ressources CPU, nombre de sessions concurrentes, nombre de tentatives de connexion, nombre de jours de validité du mot de passe…) qu'il est possible d'affecter à un ou plusieurs utilisateurs. Le profil `DEFAULT` est affecté par défaut à un utilisateur s'il n'est pas précisé.

1.2 Privilèges

Un privilège est un droit d'exécuter une certaine instruction SQL (privilège système) ou un droit d'accéder à un certain objet d'un autre schéma (privilège objet). Les privilèges peuvent être attribués ou révoqués grâce aux commandes `GRANT .. TO` et `REVOKE .. FROM`.

1.2.1 Privilèges système

Il existe une centaine de privilèges système. Les principaux sont les privilèges de créer des utilisateurs (`CREATE USER`), créer et supprimer des tables (`CREATE / DROP TABLE`), créer des espaces de travail (`CREATE TABLESPACE`), sauvegarder des tables (`BACKUP ANY TABLE`).

Ces privilèges peuvent être attribués à un ou plusieurs utilisateurs ou à un rôle. Un rôle est un ensemble nommé de privilèges (système ou objets). Il est accordé à un ou plusieurs utilisateurs, voire à tous (utilisation de PUBLIC). Ce mécanisme facilite la gestion des privilèges.

exemple : Palleja peut créer des tables dans son schéma et peut retransmettre ce privilège à un tiers

```
GRANT CREATE TABLE
TO Palleja
WITH ADMIN OPTION;
```

1.2.2 Privilèges objets

Les privilèges objets sont relatifs aux données de la base et aux actions sur les objets (table, vue, procédure…). Les actions sur les tables et vues sont `DELETE`, `INSERT`, `SELECT`, `UPDATE` (`ALL` permet d'englober toutes ces actions) et pour les tables uniquement, `INDEX` et `REFERENCES`. Pour les programmes PL/SQL (dont les procédures stockées), le seul privilège peut être `EXECUTE`. Ces privilèges peuvent, tout comme les privilèges système, être attribués à un ou plusieurs utilisateurs, à un rôle ou au `PUBLIC`.

exemple : soit la table `CLIENTS (codeClient, 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 champs prénom et ville de cette même table. L'utilisateur `Nemard` peut retransmettre les privilèges reçus à un tiers.

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

`Nemard` ne peut plus lire la table `Clients`.

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

1.3 Vues

Une vue est considérée comme une table virtuelle car elle n'a pas d'existence propre. Seule sa structure est stockée. Une vue est créée à l'aide d'une instruction `SELECT` sur des tables ou des vues existantes. Une vue se recharge chaque fois qu'elle est interrogée.
Les vues permettent essentiellement d'assurer la confidentialité des données mais elles peuvent aussi simplifier la formulation de requêtes complexes.
Une fois créée, on peut utiliser la vue comme s'il s'agissait d'une table. On peut donc lui attribuer des privilèges, ce qui améliore la sécurité des informations stockées.

1.3.1 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 le nom de l'expression renvoyée par la requête de définition.
- `WITH READ ONLY` : déclare la vue non modifiable par `INSERT`, `UPDATE` ou `DELETE`.
- `WITH CHECK OPTION` : empêche un ajout ou une modification non conforme à la définition de la vue.

exemple : création de la vue `TousLesClients` contenant le nom et prénom des `Clients`

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

exemple : création de la vue `TousLesClientsMontpellierains` contenant toutes les informations sur les clients habitant `Montpellier`

```
CREATE VIEW TousLesClientsMontpellierains AS
SELECT *
FROM Clients
WHERE villeClient = 'Montpellier'
WITH CHECK OPTION;
```

exemple : création de la vue `TousLesClientsNimois` non modifiable contenant le code, nom, et prénom de tous les clients habitant à `Nimes`

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

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;
```

1.3.2 Synonymes

Un synonyme est un alias d'un objet (table, vue, procédure, fonction…). 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'attribuer plusieurs noms à un même objet.

exemple : création d'un synonyme de la table `Clients` de `Palleja`

```
CREATE SYNONYM ClientsDePalleja
FOR Palleja.Clients ;
```

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;
```

1.3.3 Mise à jour des données à travers une vue

Il est possible d'insérer, modifier ou supprimer des données à travers une vue. Mais toutes les vues ne sont pas modifiables. Cela dépend de la structure de la vue.

1.3.3.1 Vue monotable

Pour qu'une vue soit modifiable (`INSERT, UPDATE` ou `DELETE`):

- elle ne doit pas être créée avec l'option `WITH READ ONLY`. Par exemple la vue `TousLesClientsNimois` ne sera pas modifiable.
- il ne doit pas y avoir de `DISTINCT` ou de fonction (`AVG, SUM, MIN…`) dans le `SELECT`.
- il ne doit pas y avoir de `GROUP BY, ORDER BY` ou `HAVING` dans la requête de définition.

exemple : création de la vue `ClientsParVille` contenant le nombre de Clients par ville

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

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

ORA-01733: les colonnes virtuelles ne sont pas autorisées ici

- la clé primaire de la table source doit être incluse dans la vue.

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

```
UPDATE TousLesClients
SET prenomClient = 'Judas'
WHERE nomClient = 'Bricot';
DELETE FROM TousLesClients WHERE nomClient = 'Stico';
```

1.3.3.2 Vue multitables

Les vues mutitables sont des vues qui contiennent, dans sa définition, plusieurs tables. Pour ces tables, les mises à jour sont plus complexes.

exemple : création de la vue `CommandesEtClients` contenant les Commandes avec les informations sur le client.

```
CREATE VIEW CommandesEtClients AS
SELECT numCommande, montantCommande, com.codeClient, nomClient,
       prenomClient, villeClient
FROM Commandes com
JOIN Clients cl ON com.codeClient = cl.codeClient;
```

Dans les vues multitables, on ne peut modifier que les tables dite protégées par sa clé. Une table est dite protégée par sa clé (*key preserved*), si la clé primaire est préservée dans la clause de jointure et se retrouve en tant que colonne de la vue mutitables (peut jouer le rôle de clé primaire de la vue).

Dans la vue `CommandesEtClients`, la table préservée est la table `Commandes`, car la colonne `numCommande` identifie chaque enregistrement de la vue de façon unique et pourrait donc jouer le rôle de clé primaire de la vue.

Exemples :

```
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
(codeClient, 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(numCommande, montantCommande)
VALUES ('COM125',1000);
```
// Ligne ajoutée dans la table Commandes car protégée par sa clé.

```
INSERT INTO CommandesEtClients(numCommande, montantCommande, codeClient)
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 `codeClient`, il faut dans la requête de définition préciser que l'attribut `codeClient` provient de la table `Commandes`. Si on indique que le `codeClient` provient de la table `Clients`, l'insertion sera impossible.

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

```
DELETE FROM CommandesEtClients
WHERE numCommande = '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'`

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. Gestion des transactions

Des problèmes d'incohérence ou de perte de mise à jour peuvent apparaître lorsque plusieurs personnes 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.

Une transaction est un bloc d'instructions (requêtes SQL) faisant passer la base de données d'un état cohérent à un autre état cohérent. Si une erreur ou une panne survient au cours d'une transaction, aucune des instructions contenues dans la transaction n'est effectuée, quelque soit l'endroit de la transaction où est survenue l'erreur.

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.

2.1 Propriétés (ACID) d'une transaction

Une transaction est une suite de requêtes dépendantes qui vérifient les propriétés d'atomicité, de cohérence, d'isolation et de durabilité (ACID).

- **Atomicité** : une transaction est considérée comme une seule opération (principe du tout ou rien)
- **Cohérence** : la transaction doit faire passer la base de données d'un état cohérent à un autre.
- **Isolation** : les résultats d'une transaction doivent être visibles aux autres transactions qu'une fois la transaction validée. En effet, les opérations internes de la transaction peuvent entraîner un état temporairement incohérent de la base. Cette incohérence temporaire doit être cachée aux autres transactions exécutées par d'autres utilisateurs au même moment.
- **Durabilité** : dès qu'une transaction est validée, les mises à jour perdurent dans la base de données même si une panne se produit après la transaction.

2.2 Instructions SQL pour gérer les transactions

- 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.