Institut Supérieur d'Informatique et de Gestion de Kairouan

Cours 7 : Le langage SQL (Le DDL)

Riadh ZAAFRANI
2ème année Licence Fondamentale : Sciences de l'informatique

Décembre 2009 – Janvier 2010

La définition des données : Le DDL

- Avant de parler précisément des ordres de description de données (le DDL) précisons brièvement les **types de données** autorisés.
- Les types de données que l'on peut représenter dans un SGBD sont fortement dépendants de l'architecture de l'ordinateur sur lequel il tourne.
- Il existe donc de nombreuses variantes de types selon les SGBD. Néanmoins certains types se retrouvent dans la plupart des SGBD.

Les types de données

- Voici les principaux types de données :
- **Le type alphanumérique :**
- **CHAR (n) :** Longueur fixe de n caractères. n max: 16 383
- **VARCHAR(n) :** Longueur variable, n représente le maximum
- **Le type numérique :**
- **NUMBER (n,[d])** : Champ de longueur variable de n chiffres dont **d** après la virgule, acceptant des nombres négatifs. Le nombre d'octets nécessaires pour stocker une valeur de ce type équivaut approximativement à la moitié du nombre de chiffres significatifs qui constituent cette valeur. Nombre de n chiffres dont **d** après la virgule.

Les types de données

- **SMALLINT :**Mot signé de 16 bits (-32768 a 32767)
- **INTEGER :** Double mot signé de 32 bits (-2E31 a 2E31-1)
- **FLOAT :**Numérique flottant
- **Le type gestion du temps:**
- **DATE :** Champ date (ex 24/02/1997)
- **TIME :** Champ heure (ex 14:45:10.95)
- **TIMESTAMP :** regroupe DATE et TIME

La création de tables

- Une fois les types de données définis il est alors possible de créer les tables. La commande **CREATE TABLE** permet de définir des colonnes, de leur associer un type de données et d'y ajouter des contraintes a vérifier.
- La syntaxe la plus simple est la suivante :
- **CREATE TABLE <nom-de-table>**
  (<nom-de-colonne> <Type de données>, ...);
- Dans la plupart des SGBD, le nom de la table doit commencer par une lettre et on autorise 254 colonnes maximum par table.

La création de tables

- **R44 : créer la table département avec un numéro (3 chiffres) et un nom (longueur fixe de 10 caractères) ?**
- **CREATE TABLE département**
  (numdépt NUMBER(3),
  nomdépt CHAR(10));

La création de tables

- Il est aussi possible de créer une table en insérant directement des lignes à la création.
- **CREATE TABLE <nom-de-table>**
  (<nom-de-colonne> <Type de données>, ...)
  AS SELECT <nom-de-champ>,... FROM <nom-de-table>
  WHERE <prédicat>;
- Dans le cas d'une telle requête, seules les données du SELECT sont insérées dans la nouvelle table. Il est bien évident que les contraintes de types, de clé ou d'intégrités de l'ancienne table ne sont pas recopiées dans la nouvelle.

La création de tables

- **R45 : créer la table bonus avec insertion des noms et des salaires des chefs de services de la table employé ?**
- **CREATE TABLE BONUS (nom,salaire)**
  AS SELECT nom,salaire FROM employé
  WHERE métier = 'Chef de service';

La création de tables

- Les SGBD contiennent rapidement de très nombreuses tables. Il est alors difficile de connaître les structures de toutes les tables. La description d'une table peut être obtenue par l'ordre **DESCRIBE** qui affiche à l'écran la structure complète de la table choisie.
- **R46 : Afficher la structure de la table Fournisseurs ?**
- **DESCRIBE fournisseurs;**

Expression des contraintes d'intégrité

- Il est très important de spécifier les contraintes dans le schéma, afin que le SGBD puisse les prendre en charge.
- Cela évite d'effectuer des contrôles de manière répétitive dans les applications qui accèdent à la base.

Dénomination des contraintes

- Dans le cas où aucun nom n'est affecté explicitement à une contrainte, le SGBD génère automatiquement un nom de la forme SYS_CXXXXXX(XXXXXX est un nombre entier unique).
- De tels noms ne sont pas parlants. Il est donc préférable de la fournir vous-même. L'emploi d'une stratégie pour affecter des noms permet de mieux les identifier et les gérer.

Dénomination des contraintes

- Lors de l'affectation explicite d'un nom à une contrainte, il est pratique d'utiliser la convention de dénomination suivante : **TABLE_COLONNE_TYPEDECONTRAINTE**.
- Où **TYPEDECONTRAINTE** est l'abréviation mnémonique associé au type de contrainte :
- - **NN** NOT NULL - **PK** PRIMARY KEY
- - **CK** CHECK - **FK** FOREIGN KEY
- - **UQ** UNIQUE
- Cela facilite la compréhension des messages, et permet de modifier ou de détruire une contrainte : **ALTER TABLE Salle DROP CONSTRAINT CINEMA_SALLE_NN**

Définir une valeur par défaut

- **DEFAULT** permet de préciser une valeur par défaut qui sera automatiquement introduite si l'utilisateur ne fournit pas de valeur par **INSERT**.
- Les valeurs suivantes sont généralement admises : une constante numérique ou alphanumérique, **USER** pour le nom de l'utilisateur, **NULL**, **SYSDATE** pour la date de saisie.



Types des contraintes

- Il existe 2 façons de spécifier des contraintes :
- **Contrainte de colonne :** Permet de définir une contrainte particulière sur une colonne, spécifiée dans la définition de la colonne : **COLONNE[CONSTRAINT <nom_contrainte>] Type_contrainte,**
- **Contrainte de table (portant sur plusieurs colonnes) :** Permet de définir une contrainte particulière sur une ou plusieurs colonnes, spécifiée à la fin d'une instruction **CREATE TABLE : …, [CONSTRAINT <nom_contrainte>] Type_contrainte(COLONNE1, COLONNE2, …),**



Expression des contraintes

- Il existe 5 types de contraintes :
- **1) Contrainte d'obligation : NOT NULL,**
- **2) Définition d'une clé secondaire : Tester l'unicité d'une valeur : UNIQUE,**
- **3) Tester la validité d'une valeur : CHECK,**
- **4) Définition d'une clé primaire : PRIMARY KEY,**
- **5) Définition d'une clé étrangère : FOREIGN KEY.**



Exemple introductif

```
CREATE TABLE UTILISATEUR (
   NO_UTILISATEUR NUMBER(6)
      CONSTRAINT UTILISATEUR_PK PRIMARY KEY,
   NOM_PRENOM     VARCHAR2(20)
      CONSTRAINT UTILISATEUR_NOM_PRENOM_NN NOT NULL,
   DATE_CREATION  DATE DEFAULT SYSDATE
     CONSTRAINT UTILISATEUR_ DATE_CREATION_NN NOT NULL,
   DATE_MISEAJOUR DATE DEFAULT SYSDATE NOT NULL,
   CONNECTIONS    NUMBER(6)
        CONSTRAINT UTILISATEUR_CONNECTIONS_CK
        CHECK (CONNECTIONS BETWEEN 1 AND 10),
   CONSTRAINT UTILISATEUR_ DATE_CREATION_DATE_MISEAJOUR_CK
        CHECK (DATE_CREATION <= DATE_MISEAJOUR) );
```



Expression des contraintes d'intégrité

- **1) Contrainte d'obligation (Refuser une valeur nulle) NOT NULL** permet de vérifier à la saisie ou à la modification d'un tuple si la valeur est effectivement renseignée. Par exemple :
- **CONSTRAINT UTILISATEUR_NO_UTILISATEUR_NN NOT NULL**
- Cette condition est automatiquement vérifiée pour les attributs d'une clé primaire.



Expression des contraintes d'intégrité

- **2) Définition d'une clé secondaire (Tester l'unicité d'une valeur)**
- On peut également spécifier que la valeur d'un attribut est unique pour l'ensemble de la colonne. Cela permet d'indiquer des **clés secondaires**.
- La clause **UNIQUE** vérifie que toutes les valeurs d'un attribut sont différentes au sein de la relation.



Expression des contraintes d'intégrité

- On peut par exemple indiquer que deux artistes ne peuvent avoir les mêmes nom et prénom avec l'option **UNIQUE**.
- **CREATE TABLE Artiste (id INTEGER,**
  nom VARCHAR (20) NOT NULL,
  prenom VARCHAR (20),
  anneeNaiss INTEGER,
  PRIMARY KEY (id),
  CONSTRAINT Artiste_nom_prenom_UQ
  UNIQUE (nom, prenom));

Définition d'une clé secondaire

- Il est facile de supprimer cette contrainte de clé secondaire par la suite. Ce serait beaucoup plus difficile si on avait utilisé la paire (nom, prenom) comme clé primaire puisqu'elle serait alors utilisée pour référencer un artiste dans d'autres tables.
- Voici un autre exemple d'utilisation d'une clé secondaire (contrainte sur la colonne : Ce n'est bien entendu possible que quand cette contrainte ne concerne qu'un seul attribut).
- **CREATE TABLE Cinema (nom VARCHAR (30),**
  adresse VARCHAR(50) CONSTRAINT Cinema_adresse_UQ UNIQUE,
  CONSTRAINT Cinema_PK PRIMARY KEY (nom));
- On indique qu'on ne peut pas trouver deux cinémas à la même adresse.

Expression des contraintes d'intégrité

- **3) Tester la validité d'une valeur :**
- La commande **CHECK (condition)** permet d'exprimer une contrainte consistant à restreindre un attribut à un ensemble de valeurs.
- La condition doit porter sur des colonnes de la même table.
- Cette contrainte n'est vérifiée que quand la valeur saisie est différente de NULL.

Expression des contraintes d'intégrité

- **Exemple simple :** on restreint les valeurs possibles des attributs **capacité** et **climatisée** dans la table ***Salle***.
- CREATE TABLE Salle
  (nomCinema VARCHAR (30) NOT NULL,
  no INTEGER,
  capacite INTEGER
  CONSTRAINT Salle_capacite_ck CHECK (capacite < 300),
  climatisee CHAR(1)
  CONSTRAINT Salle_climatisee_ck CHECK (climatisee IN ('O','N')),
  PRIMARY KEY (nomCinema, no),
  FOREIGN KEY nomCinema REFERENCES Cinema
  );

Expression des contraintes d'intégrité

- Il s'agit de contraintes portant sur des valeurs d'attributs.
- A chaque insertion d'un tuple, ou mise-à-jour de ce tuple affectant un des attributs contraints, le contrôle sera effectué. La règle est que la condition ne doit pas s'évaluer à **FALSE** (donc la valeur **UNKNOWN** est acceptée).
- Au lieu d'associer une contrainte à un attribut particulier, on peut la définir globalement. Dans ce cas la contrainte peut faire référence à n'importe quel attribut de la table et est testée tuple à tuple.

Expression des contraintes d'intégrité

- Exemple : **toute salle de plus de 300 places doit être climatisée:**
- CREATE TABLE Salle
  (nomCinema VARCHAR (30) NOT NULL,
  no INTEGER,
  capacité INTEGER,
  climatisée CHAR(1),
  PRIMARY KEY (nomCinema, no),
  FOREIGN KEY nomCinema REFERENCES Cinema,
  CONSTRAINT Salle_capacite_climatisée_ck CHECK
  (capacité< 300 OR Climatisée = 'O'));

Expression des contraintes d'intégrité

- **4) Définition d'une clé primaire :**
- **PRIMARY KEY(liste de cols)** permet de spécifier que la colonne définit la clé primaire de la table. La clé primaire peut porter sur plusieurs colonnes. Il n'y a bien sûr qu'une clause de ce genre par table.
- Il peut y avoir plusieurs clés (clés candidates) mais l'une d'entre elles doit être choisie comme clé primaire.
- Ce choix est important : la clé primaire est la clé utilisée pour référencer une ligne et une seule à partir d'autres tables. Il est donc assez délicat de la remettre en cause après coup. En revanche les clés secondaires peuvent être créées ou supprimées beaucoup plus facilement.

Définition d'une clé primaire

- Automatiquement les colonnes qui constituent la clé primaire ne peuvent plus être nulles et chaque clé doit être unique. Cela n'a pas vraiment de sens en effet d'identifier des lignes par des valeurs absentes.
- **Exemple :**

```
CREATE TABLE Internaute (email VARCHAR (50),
                         nom VARCHAR (20) NOT NULL,
                         prenom VARCHAR (20),
                         motDePasse VARCHAR (60) NOT NULL,
                         anneeNaiss INTEGER,
    CONSTRAINT Internaute_PK PRIMARY KEY (email));
```


Expression des contraintes d'intégrité

- **5) Définition d'une clé étrangère :**
- **FOREIGN KEY (liste-col1) REFERENCES table[(liste-col2)]** permet de spécifier que les colonnes liste-col1 de la table en cours de définition référencent la clé primaire liste-col2 de la table étrangère spécifiée.
- La clé étrangère peut porter sur plusieurs colonnes.
- Il peut bien sûr y avoir plusieurs clés étrangères dans une même table.

Expression des contraintes d'intégrité

```
CREATE TABLE Film
(                   idFilm INTEGER,
                    titre VARCHAR (50) NOT NULL,
                    annee INTEGER NOT NULL,
                    idMes INTEGER,
                    codePays INTEGER,
                    PRIMARY KEY (idFilm),
CONSTRAINT Film_idMes_FK
        FOREIGN KEY(idMes) REFERENCES Artiste,
CONSTRAINT Film_codePays_FK
        FOREIGN KEY (codePays) REFERENCES Pays
);
```


Définition d'une clé étrangère

- La commande **FOREIGN KEY (idMes) REFERENCES Artiste** indique que **idMes** référence la clé primaire de la table Artiste.
- Le SGBD vérifiera alors, pour toute modification pouvant affecter le lien entre les deux tables, que la valeur de **idMes** correspond bien à une ligne de Artiste. Ces modifications sont :
- 1. l'insertion dans Film avec une valeur inconnue pour **idMes** ;
  2. la destruction d'un artiste ;
  3. la modification de **id** dans Artiste ou de **idMes** dans Film.
- En d'autres termes le lien entre **Film** et **Artiste** est **toujours** valide. Cette contrainte est importante pour garantir qu'il n'y a pas de fausse référence dans la base, par exemple qu'un film ne fait pas référence à un artiste qui n'existe pas.

Définition d'une clé étrangère

- Il faut noter que l'attribut **idMes** n'est pas déclaré **NOT NULL**, ce qui signifie que l'on s'autorise à ne pas connaître le metteur en scène d'un film.
- La valeur **NULL** est donc acceptée dans les colonnes qui constituent la clé étrangère bien que cela soit certainement signe d'un défaut d'analyse au départ.
- Les contraintes de référence ne sont donc vérifiées pour un tuple que quand toutes les valeurs qui constituent la clé étrangère sont différentes de **NULL**.

Définition d'une clé étrangère

- **Que se passe-t-il quand la violation d'une contrainte d'intégrité est détectée par le système ?**
- Par défaut, la mise à jour est rejetée, mais il est possible de demander la répercussion de cette mise à jour de manière à ce que la contrainte soit respectée.
- Les événements que l'on peut répercuter automatiquement sont la modification ou la destruction de la ligne référencée, et on les désigne par **ON UPDATE** et **ON DELETE** respectivement.
- La répercussion elle-même consiste généralement soit à mettre la clé étrangère à NULL (option **SET NULL**), soit à appliquer la même opération aux lignes de l'entité composante (option **CASCADE**).

Définition d'une clé étrangère

- **ON DELETE {RESTRICT | CASCADE | SET NULL | SET DEFAULT}**
- **RESTRICT** donne un échec si la valeur effacée correspond à la clé de la table référencée,
- **CASCADE** signifie que les lignes correspondantes seront effacées en cascade en cas d'effacement d'une ligne de la table référencée,
- **SET NULL** place une valeur nulle dans la ligne de la table en cas d'effacement d'une ligne de la table référencée
- **SET DEFAULT** place dans ce cas la valeur par défaut

Définition d'une clé étrangère

- **ON UPDATE {RESTRICT | CASCADE | SET NULL | SET DEFAULT}**
- **RESTRICT** donne un échec si la valeur modifiée correspond a la clé de la table référencée,
- **CASCADE** signifie que les lignes correspondantes seront modifiées en cascade en cas de modification d'une ligne de la table référencée,
- **SET NULL** place une valeur nulle dans la ligne de la table en cas de modification d'une ligne de la table référencée
- **SET DEFAULT** place dans ce cas la valeur par défaut

Définition d'une clé étrangère

- Dans les deux cas, **RESTRICT** est l'option par défaut.
- **R47 : Indiquez que la destruction d'un metteur en scène déclenche la mise à NULL de la clé étrangère idMES pour tous les films qu'il a réalisé ?**

```
CREATE TABLE Film (titre VARCHAR (50),
                   annee INTEGER NOT NULL,
                   idMES INTEGER,
                   codePays  INTEGER,
                   PRIMARY KEY (titre),
    CONSTRAINT Film_idMes_FK  FOREIGN KEY (idMES)
    REFERENCES Artiste ON DELETE SET NULL,
            FOREIGN KEY (codePays) REFERENCES Pays);
```


Énumération des valeurs possibles

- Une autre manière de définir (autre que CHECK), dans la base, l'ensemble des valeurs autorisées pour un attribut (en d'autres termes, une codification imposée) consiste à placer ces valeurs dans une table et la lier à l'attribut par une contrainte de clé étrangère.
- C'est ce que nous pouvons faire par exemple pour la table Pays.

Énumération des valeurs possibles

```
CREATE TABLE Pays (code VARCHAR (4),
                   nom VARCHAR (30) NOT NULL,
                   langue VARCHAR (30) NOT NULL,
                   PRIMARY KEY (code));
```

```
INSERT INTO Pays VALUES (0, 'Inconnu', 'Inconnue');
INSERT INTO Pays VALUES (1, 'France', 'Français');
INSERT INTO Pays VALUES (2, 'USA', 'Anglais');
INSERT INTO Pays VALUES (3, 'Allemagne', 'Allemand');
INSERT INTO Pays VALUES (4, 'Angleterre', 'Anglais'); ...
```

- Si on ne fait pas de vérification automatique, soit avec **CHECK**, soit avec la commande **FOREIGN KEY**, il faut faire cette vérification dans l'application, ce qui est plus lourd à gérer.

Exercices

- **R48 : créer la table fournisseurs (numFrs, nom, adresse,ville) sachant que le numéro doit être strictement inférieur à 20, que le nom est obligatoire et que la ville par défaut est Kairouan ?**
- **create table fournisseurs**
  (numFrs integer
  constraint fournisseurs_numFrs_CK check (numFrs < 20),
  nom varchar(20) constraint fournisseurs_nom_NN not null,
  adresse varchar(50),
  ville varchar(10) default 'Kairouan',
  constraint fournisseurs_PK primary key (numFrs));

Exercices

- **R49 : créer la table produits (numProd,design,prix,poids,couleur) sachant que le numéro doit être strictement inférieur à 200, que le prix doit être compris entre 1000 et 9000, que le poids doit être strictement inférieur à 100, que la couleur doit être rouge, verte, jaune, bleue ou grise?**
- **create table produits**
  (numProd integer Constraint produits_numProd_CK check (numProd < 200) ,
  design varchar(10),
  prix integer Constraint produits_ prix_CK check (prix between 1000 and 9000),
  poids integer Constraint produits_ poids_CK check (poids < 100),
  couleur varchar(10) Constraint produits_ couleur_CK
  check (couleur in ('rouge','vert','jaune','bleu','gris')),
  constraint produits_PK primary key (numProd));

Exercices

- **R50 : créer la table commandes (numCde,numFrs,numProd,qté) sachant que le numéro de commande ne contient pas de doublons et est obligatoire?**
- **Create table commandes**
  (numCde integer
  constraint commandes_numCde_UQ Unique
  constraint commandes_numCde_NN NOT NULL,
  numFrs integer,
  numProd integer,
  qté integer,
  constraint commandes_PK primary key (numFrs,numProd),
  constraint commandes_numFrs_FK foreign key (numFrs)
  references fournisseurs on delete cascade on update cascade,
  constraint commandes_numProd_FK foreign key (numProd)
  references produits on delete cascade on update cascade));

Entité faible

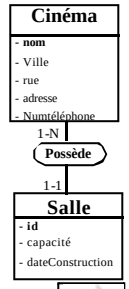
- Jusqu'à présent nous avons généralement considéré le cas d'entités **indépendantes** les unes des autres. Chaque entité, disposant de son propre identifiant, pouvait être considérée isolément.
- Il existe des cas où une entité ne peut exister qu'en étroite association avec une autre, et est identifiée relativement à cette autre entité. On parle alors d'**entité faible**.

Entité faible

- Prenons l'exemple d'un cinéma, et de ses salles. On peut considérer chaque salle comme une entité, dotée d'attributs comme la capacité, l'équipement en son Dolby, ou autre.
- Il est difficilement imaginable de représenter une salle sans qu'elle soit rattachée à son cinéma. C'est en effet au niveau du cinéma que l'on va trouver quelques informations générales comme l'adresse ou le numéro de téléphone.

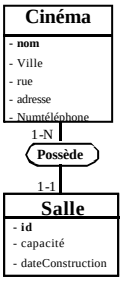
Entité faible

- Il est possible de représenter le lien entre un cinéma et ses salles par une association classique, comme le montre la figure ci-jointe.
- Cette cardinalité force la participation d'une salle à un lien d'association avec un et un seul cinéma.
- Cette représentation est correcte, mais présente un inconvénient : on doit créer un identifiant artificiel id pour le type d'entité Salle, et numéroter toutes les salles, indépendamment du cinéma auquel elles sont rattachées.



Entité faible

- On peut considérer qu'il est beaucoup plus naturel de numéroter les salles par un numéro interne à chaque cinéma.
- La clé d'identification d'une salle est alors constituée de deux parties :
- 1. **la clé de Cinéma**, qui indique dans quel cinéma se trouve la salle ;
- 2. **le numéro de la salle** au sein du cinéma.



Entité faible

- L'introduction d'entités faibles n'est pas une nécessité absolue puisqu'on peut très bien utiliser une association classique.
- La principale différence est que, dans le cas d'une entité faible, on obtient une identification composée qui est souvent plus pratique à gérer, et peut également rendre plus facile certaines requêtes.
- La présence d'un type d'entité faible B associé à un type d'entité A implique également des contraintes fortes sur les créations, modifications et destructions des instances de B et A car on doit toujours s'assurer que la contrainte est valide.

Entité faible

- Concrètement, en prenant l'exemple de **Salle** et de **Cinéma**, on doit mettre en place les mécanismes suivants :
- 1. Quand on insère une salle dans la base, on doit toujours l'associer à un cinéma ;
- 2. quand un cinéma est détruit, on doit aussi détruire toutes ses salles ;
- 3. quand on modifie la clé d'un cinéma, il faut répercuter la modification sur toutes ses salles.
- Pour respecter les règles de destruction/création énoncées, on doit mettre en place une stratégie dans le SGBD comme celle qui va permettre de définir l'entité Salle.

Définition d'une entité faible

- Dans le cas d'une **entité faible**, on décide en général de détruire le composant quand on détruit le composé.
- Par exemple, quand on détruit un cinéma, on veut également détruire les salles ; quand on modifie la clé d'un cinéma, on veut répercuter la modification sur ses salles.
- **CREATE TABLE Salle (nomCinema VARCHAR (30),**
  no INTEGER,
  capacite INTEGER,
  CONSTRAINT Salle_PK PRIMAR KEY (nomCinema, no),
  CONSTRAINT Salle_nomCinema_FK FOREIGN KEY (nomCinema) REFERENCES Cinema ON DELETE CASCADE ON UPDATE CASCADE);

Définition d'une entité faible

- Il est important de noter que **nomCinema** fait partie de la clé et ne peut donc pas être **NULL**. On ne pourrait donc pas spécifier ici **ON DELETE SET NULL**.
- La spécification des actions **ON DELETE** et **ON UPDATE** simplifie considérablement la gestion de la base par la suite : on n'a plus par exemple à se soucier de détruire les salles quand on détruit un cinéma.

Modification des attributs

- La création d'un schéma n'est qu'une première étape dans la vie d'une base de données. On est toujours amené par la suite à créer de nouvelles tables, à ajouter des attributs ou à en modifier la définition.
- La forme générale de la commande permettant de modifier une table est : **ALTER TABLE nomTable ACTION description**
- où **ACTION** peut être principalement **ADD**, **MODIFY**, **DROP** ou **RENAME**, et description est la commande de modification associée à **ACTION**.
- La modification d'une table peut poser des problèmes si elle est incompatible avec le contenu existant. Par exemple passer un attribut à **NOT NULL** implique que cet attribut a déjà des valeurs pour toutes les lignes de la table.

Modification du schéma

- Voici quelques exemples d'ajout et de modification d'attributs.
- On peut ajouter un attribut region à la table Internaute avec la commande :
- **ALTER TABLE Internaute ADD region VARCHAR(10);**
- S'il existe déjà des données dans la table, la valeur sera à NULL ou à la valeur par défaut.
- La taille de region étant certainement insuffisante, on peut l'agrandir avec MODIFY, et la déclarer NOT NULL par la même occasion : **ALTER TABLE Internaute MODIFY region VARCHAR(30) NOT NULL;**

Modification du schéma

- Il est également possible de diminuer la taille d'une colonne, avec le risque d'une perte d'information pour les données existantes.
- On peut même changer son type, pour passer par exemple de **VARCHAR** à **INTEGER**, avec un résultat imprévisible.
- L'option ALTER TABLE permet d'ajouter une valeur par défaut.
- **ALTER TABLE Internaute MODIFY region SET DEFAULT 'KAIROUAN';**
- Enfin on peut détruire un attribut avec DROP.
- **ALTER TABLE Internaute DROP region;**

Les vues

- Une des fonctionnalités les plus remarquables des SGBD relationnels est : les **vues**.
- Comme nous l'avons vu dans la partie consacrée à SQL, une requête produit toujours une relation. Cela suggère la possibilité d'ajouter au schéma des tables **'virtuelles'** qui ne sont rien d'autres que le résultat de requêtes stockées.

Les vues

- De telles tables sont nommées des ***vues*** dans la terminologie relationnelle. On peut interroger des vues comme des tables stockées.
- Une vue n'induit **aucun stockage** puisqu'elle n'existe pas physiquement. Seule sa description est stockée, sous la forme d'une requête faisant intervenir des tables de la base ou d'autres vues.

Création et interrogation d'une vue

- Une vue est tout point comparable à une table : en particulier on peut l'interroger par SQL.
- La grande différence est qu'une vue est le résultat d'une requête, avec la caractéristique essentielle que ce résultat est **réévalué** à chaque fois que l'on accède à la vue.
- En d'autres termes une vue est ***dynamique*** : elle donne une représentation fidèle de la base au moment de l'évaluation de la requête.
- Une **vue** est donc essentiellement une requête à laquelle on a donné un nom.

Création et interrogation d'une vue

- Les **vues** peuvent être utilisées pour :
- Répondre à des besoins de confidentialité;
- Maîtriser les mises à jour en assurant des contrôles de cohérence;
- Offrir plus de commodités aux utilisateurs dans la manipulation des données, en ne leur présentant de façon simplifiée que le sous-ensemble de données qu'ils ont à manipuler;
- Sauvegarder des requêtes dans le dictionnaire de données.

Création et interrogation d'une vue

- La syntaxe de création d'une vue est très simple :
- **CREATE [OR REPLACE] [FORCE | NOFORCE]**
  VIEW [SCHEMA.]<nom-vue>
  AS <requête> [WITH
  {CHECK OPTION [CONSTRAINT NOM_CONTRAINTE]
  | READ ONLY}];
- **FORCE** Cette clause permet la création de la vue même en présence d'une erreur, par exemple si la table n'existe pas ou si l'utilisateur n'a pas les droits correspondants.
- **NOFORCE** Cette clause ne permet pas la création de la vue en présence d'une erreur. C'est l'option par défaut.

Création et interrogation d'une vue

- **CHECK OPTION** Cette clause permet de vérifier que les mises à jour ou les insertions ne produisent que des lignes qui feront partie de la sélection de la vue.
- **READ ONLY** Cette clause interdit toute modification de données en utilisant le nom de la vue dans un ordre **INSERT**, **UPDATE** ou **DELETE**.
- **Exemple : on peut créer une vue qui ne contient que les cinémas tunisois :**
- **CREATE VIEW TunisCinemas**
  AS SELECT * FROM Cinema
  WHERE ville = 'Tunis';

Création et interrogation d'une vue

- **On peut aussi en profiter pour restreindre la vision des cinémas tunisois à leur nom et à leur nombre de salles.**
- **CREATE VIEW SimpleTunisCinemas**
  AS SELECT nom, COUNT(*) AS nbSalles
  FROM Cinema c, Salle s
  WHERE ville = 'Tunis'
  AND c.nom = s.nomCinema
  GROUP BY c.nom

Création et interrogation d'une vue

- Enfin un des intérêts des vues est de donner une représentation **dénormalisée** de la base, en regroupant des informations par des jointures.
- Par exemple on peut créer une vue ***Casting*** donnant explicitement les titres des films, leur année et les noms et prénoms des acteurs.
- **CREATE VIEW Casting (film, annee, acteur, prenom)**
  AS SELECT titre, annee, nom, prenom
  FROM Film f, Role r, Artiste a
  WHERE f.idFilm = r.idFilm AND r.idActeur = a.idArtiste;
- Remarque : on peut donner explicitement des noms d'attributs au lieu d'utiliser les attributs de la clause SELECT.

Création et interrogation d'une vue

- Maintenant, on peut utiliser les vues et les tables dans des requêtes SQL. **Par exemple la requête « Quels acteurs ont tourné un film en 2004 » s'exprime par :**
- **SELECT acteur, prenom**
  FROM Casting
  WHERE annee = 2004;
- **On peut ensuite donner des droits en lecture sur cette vue pour que cette information limitée soit disponible à tous.**
- **GRANT SELECT ON Casting TO PUBLIC**

Mise à jour d'une vue

- L'idée de modifier une vue peut sembler étrange puisqu'une vue n'a pas de contenu.
- En fait il s'agit bien entendu de modifier la table qui sert de support à la vue. Il existe de sévères **restrictions** sur les droits d'insérer ou de mettre-à-jour des tables à travers les vues.
- Un exemple suffit pour comprendre le problème. **Imaginons que l'on souhaite insérer une ligne dans la vue *Casting*.**
- **INSERT INTO CASTING (film, annee, acteur, prenom)**
  VALUES ('Titanic', 1998, 'DiCaprio', 'Leonardo');

Mise à jour d'une vue

- Cet ordre s'adresse à une vue issue de trois tables.
- Il n'y a clairement pas assez d'information pour alimenter ces tables de manière cohérente, et l'insertion n'est pas possible (de même que toute mise à jour).
- De telles **vues** sont dites ***non modifiables***.

Mise à jour d'une vue

- Les règles définissant les vues modifiables sont très strictes.
- **1.** La vue doit être basée sur **une seule table**.
- **2.** Toute colonne non référencée dans la vue doit pouvoir être mise à **NULL** ou disposer d'une **valeur par défaut**;
- **3.** On ne peut pas mettre à-jour un attribut qui résulte d'un calcul ou d'une opération;
- **4.** L'ordre **SELECT** utilisé pour définir la vue ne doit comporter ni **jointure**, ni clause **GROUP BY**, **CONNECT BY** ou **START WITH**.

Mise à jour d'une vue

- Il est donc tout à fait possible d'insérer, modifier ou détruire la table **Cinema** au travers de la vue ***Tunis-Cinema***.
- **INSERT INTO TunisCinema**
  VALUES ('Hannibal', 12, 'Assad Ibn Fourat', 'Kairouan')
- En revanche, il n'est pas possible d'insérer dans ***SimpleTunisCinemas*** (présence de **Count(*)**).
- L'insertion précédente illustre une petite subtilité : on peut insérer dans une vue, sans être en mesure de voir la ligne insérée au travers de la vue par la suite !

Mise à jour d'une vue

- Afin d'éviter ce genre d'incohérence, SQL2 propose l'option **WITH CHECK OPTION** qui permet de garantir que toute ligne insérée dans la vue satisfait les critères de sélection de la vue.
- **CREATE VIEW TunisCinemas**
  AS SELECT * FROM Cinema
  WHERE ville = 'Tunis'
  WITH CHECK OPTION;
- L'insertion donnée en exemple ci-dessus devient impossible.

Mise à jour d'une vue

- De même, la clause **WITH CHECK OPTION** interdit de modifier une ligne de telle sorte qu'avec les nouvelles valeurs, elle ne soit plus sélectionnée par la requête de définition de la vue.
- **UPDATE TunisCinema**
  SET VALUES VILLE = 'KAIROUAN'
  WHERE numéro = 12;
- Cette mise à jour donnée en exemple est aussi interdite.

Mise à jour d'une vue

- On peut renommer une vue avec la syntaxe courante SQL :
- **RENAME [SCHEMA.]ANCIEN_NOM**
  TO [SCHEMA.]NOUVEAU_NOM;
- Enfin, on détruit une vue avec la syntaxe courante SQL :
- **DROP VIEW TunisCinemas;**