

Chapitre 2: Définir et structurer les bases de données

Contraindre les données

Frédéric Flouvat

(dérivé du cours du Pr. Jeffrey Ullman, Stanford University)

Université de la Nouvelle-Calédonie

frederic.flouvat@univ-nc.nc



Contraintes et Triggers

- ☰ Une *contrainte* est une règle que doivent respecter les données et que le SGBD doit imposer.
 - **Exemple**: contraintes de clés.
- ☰ Les *Triggers* sont uniquement exécutés lorsqu'une condition prédéfinie apparaît, p.ex. insertion d'un tuple.
 - Plus facile à implémenter que des contraintes complexes.

Les types de contraintes

- Les clés (primaires).
- Les clés étrangères, ou contraintes d'intégrité référentielle.
- Les contraintes de valeurs.
 - contraintes de valeurs sur un attribut particulier.
- Les contraintes sur les tuples.
 - relations entre composants.
- Les assertions: n'importe quelle expression SQL booléenne.
 - contraintes booléennes sur les objets de la base de données.

Les clés primaires

- Clé primaire composé d'un seul attribut:
 - Mettre **PRIMARY KEY** ou **UNIQUE** après le type dans la déclaration de l'attribut.

```
CREATE TABLE Beers (  
    name CHAR(20) UNIQUE,  
    manf CHAR(20)  
);
```

- Différence **PRIMARY KEY** vs. **UNIQUE**
 - il peut y avoir une seule **PRIMARY KEY** pour une relation, mais plusieurs attributs **UNIQUE**.
 - Aucun attribut d'une **PRIMARY KEY** ne peut avoir la **NULL** pour un tuple, alors que les attributs déclarés **UNIQUE** peuvent prendre la valeur **NULL** et ceci plusieurs fois.

Les clés primaires

- Clé primaire composée de plusieurs attributs:
 - Mettre `PRIMARY KEY(<liste d'attributs>)` après le dernier attribut.

```
CREATE TABLE Sells (  
    bar          CHAR(20),  
    beer         VARCHAR(20),  
    price REAL,  
    PRIMARY KEY (bar, beer)  
);
```

Les clés étrangères

- Les valeurs prises par les attributs de la clé étrangère d'une relation doivent aussi apparaître ensemble au niveau des attributs d'une autre relation.
 - une clé étrangère est clé primaire dans une autre relation.
 - attributs utilisés pour les jointures.

■ Exemple:

Dans **Sells(bar, beer, price)**, les valeurs pour l'attribut **beer** apparaissent aussi toutes au niveau de l'attribut **name** de la relation **Beers(name, manf)**.

bar	beer	price
Pete	Bud	5
Pete	Bud lite	2.22
Kend	Man	1

name	manf
Bud	Anheuser-Busch
Bud lite	Anheuser-Busch
Man	Peterson
17	Al

Définir des clés étrangères

- Utiliser le mot clé REFERENCES, au choix :
 1. après un attribut (pour une clé composé d'un attribut).
 2. comme un élément de l'expression:
FOREIGN KEY (<list of attributes>
REFERENCES <relation> (<attributes>)

- Les attributs référencés doivent être déclarés PRIMARY KEY ou UNIQUE.

Définir des clés étrangères

Exemple avec un attribut:

```
CREATE TABLE Beers (  
    name CHAR(20) PRIMARY KEY,  
    manf CHAR(20)  
);
```

```
CREATE TABLE Sells (  
    bar CHAR(20),  
    beer CHAR(20) REFERENCES Beers(name),  
    price REAL  
);
```

Définir des clés étrangères

- Exemple en tant qu'élément du schéma de la relation:

```
CREATE TABLE Beers (  
    name CHAR(20) PRIMARY KEY,  
    manf CHAR(20)  
);
```

```
CREATE TABLE Sells (  
    bar CHAR(20),  
    beer CHAR(20),  
    price REAL,  
    FOREIGN KEY(beer) REFERENCES Beers(name)  
);
```

Violation des contraintes de clé étrangère

- Si une contrainte de clé étrangère est définie de la relation R vers la relation S , deux violations de cette contrainte sont possibles:

- Une insertion ou une mise à jour de R introduit des valeurs qui n'existent pas dans S .

bar	beer	price
Pete	Bud	5
Pete	Bud lite	2.22
Kend	Nber	1

name	manf
Bud	Anheuser-Busch
Bud lite	Anheuser-Busch
Man	Peterson
17	Al

- Une suppression ou mise à jour de S entraîne que des tuples de R deviennent "incomplets".

bar	beer	price
Pete	Bud	5
Pete	Bud lite	2.22
Kend	Man	1

name	manf
Bud	Anheuser-Busch
Bud lite	Anheuser-Busch
Man	Peterson
17	Al

Actions à prendre pour imposer les contraintes de clé étrangère

☰ Exemple:

- Supposons que $R = \text{Sells}$, $S = \text{Beers}$.
- Une insertion ou une mise à jour de **Sells** doit être rejetée lorsqu'elle implique la vente d'une bière n'existant pas.
- Une suppression ou une mise à jour de **Beers** qui enlève une valeur de bière utilisée dans certains tuples de **Sells** peut être traitée de trois façons.
 1. **Default** : rejet de la modification.
 2. **Cascade** : faire les mêmes modifications dans Sells.
 - **bière supprimée** : supprime des tuples de Sells.
 - **bière mise à jour**: change des valeurs dans Sells.
 3. **Set NULL** : remplace la bière par NULL.

Actions à prendre pour imposer les contraintes de clé étrangère

☐ Exemple: CASCADE

- Suppression du tuple de la bière Bud de la relation Beers:
 - Alors supprimer tous les tuples de Sells qui ont beer = 'Bud'.
- Mise à jour du tuple de la bière Bud en changeant 'Bud' par 'Budweiser':
 - Alors changer tous les tuples de Sells qui ont beer = 'Bud' par beer = 'Budweiser'.

☐ Exemple: SET NULL

- Suppression du tuple de la bière Bud de la relation Beers:
 - Alors changer tous les tuples de Sells qui ont beer = 'Bud' par beer = NULL.
- Mise à jour du tuple de la bière Bud en changeant 'Bud' par 'Budweiser':
 - même changement que pour la suppression.

Choisir une politique de validation de contrainte

- Quand une clé étrangère est déclarée, la politique de validation de la contrainte peut être SET NULL ou CASCADE indépendamment des suppressions et des mises à jours.

- Faire suivre la déclaration de clé étrangère par:
 ON [UPDATE, DELETE][SET NULL, CASCADE]
 - Les deux clauses UPDATE ET DELETE peuvent être utilisées.
 - Si aucune n'est définie, celle par défaut est utilisée (rejet).

Choisir une politique de validation de contrainte

☰ Exemple: définir une politique

```
CREATE TABLE Sells (  
  bar CHAR(20),  
  beer CHAR(20),  
  price REAL,  
  FOREIGN KEY (beer)  
    REFERENCES Beers (name)  
    ON DELETE SET NULL  
    ON UPDATE CASCADE  
);
```

Contraintes sur les attributs

- Contraintes sur la valeur d'un attribut.
- Ajouter CHECK(<condition>) après la déclaration de l'attribut.
- La condition peut être définie sur l'attribut constraints, mais tout autre relations ou attributs doivent être dans une sous-requête.

- Exemple:**

```
CREATE TABLE Sells (  
    bar    CHAR(20),  
    beer   CHAR(20)    CHECK ( beer IN  
        (SELECT name FROM Beers)),  
    price  REAL CHECK ( price <= 5.00 )  
);
```

Timing des vérifications

- Vérifications des valeurs effectuées uniquement quand une valeur pour l'attribut contraint est insérée ou mise à jour.

- **Exemple:**
 - `CHECK (price <= 5.00)`
 - vérifie chaque nouveau prix et refuse la modification (pour le tuple visé) si le prix est supérieur à 5\$.

 - `CHECK (beer IN (SELECT name FROM Beers))`
 - vérifie que chaque nouvelle bière soit une bière référencée dans `Beers`
 - pas vérifié si une bière est supprimée de `Beers`, contrairement aux clés étrangères.

Contraintes sur les tuples

- ☞ CHECK (<condition>) peut être ajoutée comme un élément de la définition du schéma de la relation.
- ☞ Dans ce cas, la condition peut traiter de tous les attributs de la relation.
 - mais l'utilisation d'un autre attribut ou d'une autre relation nécessite de faire une sous-requête
- ☞ Vérifiée à l'insertion ou lors de la mise à jour uniquement.
- ☞ **Exemple:** Seul les bars de Joe peuvent vendre de la bière à plus de 5\$.

```
CREATE TABLE Sells (  
    bar          CHAR(20) ,  
    beer        CHAR(20) ,  
    price REAL ,  
    CHECK (bar = 'Joe''s Bar' OR price <= 5.00)  
);
```

Les Triggers: Motivation

- Les contraintes sur les attributs et les tuples sont vérifiées à des instants connus, mais ne sont pas “puissantes”.
- Les triggers laissent l'utilisateur décider à quel moment une condition doit être vérifiée.

Des règles Événement-Condition-Action

- Un autre nom des “trigger” est *règles ECA*, ou règles *événement-condition-action*.
- Événement* : typiquement un type de modification de la base de données, p.ex. “insertion dans Sells.”
- Condition* : Toute expression booléenne SQL.
- Action* : Toute instruction SQL.
- Exemple**: En utilisant `Sells(bar, beer, price)` et une relation unaire `RipoffBars(bar)`, maintenir la liste des bars qui augmentent le prix d'une bière de plus de 1\$.

Définition d'un Trigger dans PostgreSQL

Exemple précédent:

```
CREATE OR REPLACE FUNCTION trig_price() RETURNS TRIGGER AS  
,  
BEGIN  
    INSERT INTO "RipoffBars" VALUES (NEW.bar);  
    RETURN NULL;  
END;  
,  
LANGUAGE 'plpgsql'
```

```
CREATE TRIGGER PriceTrig  
AFTER UPDATE ON "Sells"  
FOR EACH ROW  
WHEN ( NEW.price > OLD.price + 1.00 )  
EXECUTE PROCEDURE trig_price();
```

l'événement

la condition

l'action

Options: CREATE TRIGGER

```
CREATE TRIGGER PriceTrig
AFTER UPDATE ON "Sells"
FOR EACH ROW
WHEN ( NEW.price > OLD.price + 1.00 )
EXECUTE PROCEDURE trig_price();
```

CREATE TRIGGER <name>

OU

CREATE OR REPLACE TRIGGER <name>

- utile pour remplacer ou modifier un trigger.

Options: L' événement

```
CREATE TRIGGER PriceTrig
AFTER UPDATE ON "Sells"
FOR EACH ROW
WHEN ( NEW.price > OLD.price + 1.00 )
EXECUTE PROCEDURE trig_price();
```

- ☐ AFTER ou BEFORE
 - Egalement, INSTEAD OF, si la relation est une vue.
- ☐ INSERT, DELETE ou UPDATE
 - UPDATE peut être UPDATE OF <attribute list> ON ... pour cibler un ou plusieurs attributs.
 - Possibilité de préciser plusieurs événements déclencheurs avec **OR**
p.ex. AFTER INSERT **OR** UPDATE... ON "Sells"
- ☐ **Recommandations:**
 - BEFORE pour vérifier ou modifier les données insérées ou maj
 - AFTER pour propager des modifications sur d'autres tables

Options: FOR EACH ROW

```
CREATE TRIGGER PriceTrig
AFTER UPDATE ON "Sells"
FOR EACH ROW
WHEN ( NEW.price > OLD.price + 1.00 )
EXECUTE PROCEDURE trig_price();
```

- Les triggers sont “niveau ligne” ou “niveau instruction.”
- FOR EACH ROW indique un trigger niveau ligne; par défaut niveau instruction.
- Le triggers niveau ligne*: exécuté une fois pour chaque tuple modifié.
- Les triggers niveau instruction*: exécuté une fois pour chaque instruction SQL, peu importe le nombre de tuples modifiés.

Options: La Condition

```
CREATE TRIGGER PriceTrig
AFTER UPDATE ON "Sells"
FOR EACH ROW
WHEN ( NEW.price > OLD.price + 1.00 )
EXECUTE PROCEDURE trig_price();
```

- ☐ Toute condition à résultat booléen.
 - les sous-requêtes ne sont pas gérées
- ☐ Evaluée sur la base de données avant ou après l'événement, en fonction de BEFORE ou AFTER.
 - mais toujours avant que les changements prennent effet.
- ☐ Accède au nouveau/ancien tuple grâce aux variables **NEW** et **OLD**

Options: L' Action

```
CREATE TRIGGER PriceTrig
AFTER UPDATE ON "Sells"
FOR EACH ROW
WHEN ( NEW.price > OLD.price + 1.00 )
EXECUTE PROCEDURE trig_price();
```

- Il peut y avoir plus d'une instruction dans l'action et il s'agit nécessairement d'un appel de **fonction trigger**
- Attention: l'action peut déclencher à nouveau le trigger
 - Risque d'exécution infinie

Les fonctions trigger dans PostgreSQL

```
CREATE OR REPLACE FUNCTION trig_price() RETURNS TRIGGER AS
'
BEGIN
  INSERT INTO "RipoffBars" VALUES (NEW.bar);
  RETURN NULL;
END;
'
LANGUAGE 'plpgsql'
```

- Un trigger est obligatoirement associé à une fonction qui retourne un objet trigger
 - Nécessité de créer la fonction d'abord, puis de créer le trigger
 - Déclaration d'une fonction sans arguments
 - Possibilité de récupérer des arguments via le tableau TG_ARGV[] (la variable TG_NARGS donne le nombre d'arguments passés en paramètre)

Les fonctions trigger dans PostgreSQL

```
CREATE OR REPLACE FUNCTION trig_price() RETURNS TRIGGER AS
'
BEGIN
  INSERT INTO "RipoffBars" VALUES (NEW.bar);
  RETURN NULL;
END;
'
LANGUAGE 'plpgsql'
```

- Nécessité de retourner un objet trigger dépendant du mode d'exécution
 - Si **trigger niveau instruction**, RETURN NULL
 - Si **trigger niveau ligne**,
 - si AFTER événement, RETURN NULL
 - si BEFORE événement,
 - RETURN NULL annule l'opération sur la ligne courante
 - RETURN NEW pour valider INSERT/UPDATE
 - RETURN OLD pour valider DELETE

Les fonctions trigger dans PostgreSQL

```
CREATE OR REPLACE FUNCTION trig_price() RETURNS TRIGGER AS
'
BEGIN
  INSERT INTO "RipoffBars" VALUES (NEW.bar);
  RETURN NULL;
END;
'
LANGUAGE 'plpgsql'
```

- Plusieurs variables prédéfinies permettant de récupérer des informations sur le trigger déclenché (nom, événement déclencheur, table visée, etc)
 - TG_NAME, TG_WHEN, TG_LEVEL, TG_OP, TG_RELNAME,
- Possibilité d'accéder au tuple en cours de modification (celui sur lequel se fait l'action)
 - NEW dans la fonction pour INSERT/UPDATE
 - OLD dans la fonction pour DELETE/UPDATE

Un autre Exemple de trigger

- À la place d'utiliser une clé étrangère et de rejeter les insertions dans `Sells(bar, beer, price)` avec des bières inconnues, un trigger peut ajouter cette bière à `Beers`, en mettant la valeur NULL pour le fabricant.

```
CREATE TRIGGER BeerTrigger
```

```
AFTER INSERT ON "Sells"
```

```
FOR EACH ROW
```

```
EXECUTE PROCEDURE trig_beer();
```

événement: après une insertion dans Selles

considérer chaque insertion

les traitements sont réalisés dans la fonction `trig_beer()` (à déclarer avant)

Un autre Exemple de trigger

```
CREATE OR REPLACE FUNCTION trig_beer() RETURNS TRIGGER AS
$$
BEGIN
  IF ( NEW.beer NOT IN (SELECT name FROM Beers) ) THEN
    INSERT INTO "Beers"(name) VALUES (NEW.beer);
  END IF;
  RETURN NULL;
END;
$$
LANGUAGE 'plpgsql';
```

condition: si la bière n'est pas dans Beers

Rq: cette condition ne peut être exprimée dans le WHEN du trigger car elle dépend d'une sous requête

Remarque sur la visibilité des modifications

- Quelles sont les données que voit un trigger lorsqu'il s'exécute ?
 - Dans certains cas pas évident car la requête Q qui a déclenché le trigger peut être encore active et faire des modifications

- Trigger niveau instruction
 - Si *BEFORE* évènement: aucune des modifications de Q visibles
 - Si *AFTER* évènement: toutes les modifications de Q visibles

- Trigger niveau ligne (*FOR EACH ROW*)
 - Si *BEFORE* évènement: les modifications des lignes déjà traitées par Q sont visibles
 - pb: ordre de traitement des tuples pas prévisible
 - Si *AFTER* évènement: toutes les modifications de Q visibles

Les contraintes du point de vue théorique

- Différents concepts théoriques pour représenter les principaux types de contraintes (appelées aussi "classes de contraintes")
 - Dépendances fonctionnelles -> fondement théorique des clés et de la normalisation
 - Dépendances d'inclusions -> fondement théorique des clés étrangères
 - Dépendances multi-valuées, ...
- Mécanismes formels pour exprimer des propriétés attendues pour les données
- Dépendances utilisées pour
 - protéger les données contre certaines anomalies (p.ex. avec des triggers)
 - améliorer la conception/maintenance d'un schéma
 - pour améliorer les performances

Erreur de conception et Anomalies

■ Bon schéma relationnel:

- pas de redondance
 - redondance = plusieurs fois la même information
 - le fait que A.B. soit le fabricant de la Bud
- pas d'anomalies.
 - **Anomalie de mise à jour** : une occurrence d'une information est modifiée et pas les autres
 - si Janeway part pour l' *Intrepid*, pensera-t-on à changer tous les n-uplets?
 - **Anomalie de suppression** : une information pertinente est perdue en détruisant un n-uplet. :
 - si personne n'aime Bud, on perd le fait que son fabricant soit A.B.

name	addr	beersLiked	manf	favBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	Voyager	WickedAle	Pete's	WickedAle
Spock	Enterprise	Bud	A.B.	Bud

Dépendances Fonctionnelles

- $X \rightarrow A$ propriété d'une relation R si 2 n-uplets (tuples) sont égaux sur les attributs X alors ils sont égaux sur l'attribut A .
 - Quand c'est le cas, on dit que R satisfait la DF " $X \rightarrow A$ "

- **Conventions:**

- ..., X, Y, Z ensembles d'attributs; A, B, C, \dots attributs.
- On écrit ABC , plutôt que $\{A, B, C\}$.

- **Exemple:**

$Drinkers(name, addr, beersLiked, manf, favBeer)$

FD naturelles pour ce schéma:

1. $name \rightarrow addr$
2. $name \rightarrow favBeer$
3. $beersLiked \rightarrow manf$

Exemple

name	addr	beersLiked	manf	favBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	Voyager	WickedAle	Pete's	WickedAle
Spock	Enterprise	Bud	A.B.	Bud

Parce que name -> addr

Parce que name -> favBeer

Parce que beersLiked -> manf

DF à plusieurs attributs

- Plus d'un attribut à droite: pratique mais pas indispensable
 - Pratique comme raccourci pour plusieurs DF
 - Exemple:

```
name -> addr  
name -> favBeer
```

```
deviennent  
name -> addr favBeer
```

- Plus d'un attribut à gauche: essentiel.
 - Exemple: `bar beer -> price`

Clés d'une Relation

- K est une **clé** de R ssi pour tout attribut A de R on a la DF $K \rightarrow A$

- K est une **clé minimale** de R ssi
 - K est une clé,
 - et aucun sous ensemble strict de K n'est une clé de R

- Exemple: **Drinkers(name, addr, beersLiked, manf, favBeer)**
 - **{name, beersLiked}** est une clé: ces 2 attributs déterminent tous les autres.
 - $\text{name} \rightarrow \text{addr}$ favBeer et $\text{beersLiked} \rightarrow \text{manf}$
 - **{name, beersLiked}** est une **clé minimale**: ni {name}, ni {beersLiked} ne sont des clés
 - name ne détermine pas manf ; beersLiked ne détermine pas addr .
 - Il n'y a pas d'autre clé minimale, mais il y a beaucoup d'autres clés
 - tout ensemble d'attributs contenant **{name, beersLiked}**.

Les dépendances d'inclusions (DI)

- Autre type de dépendances appelées dépendances d'inclusion (DI) ou « contraintes d'intégrité référentielles ».
- Entre deux relations.
- Exemple:**
 - Tout titre projeté actuellement (présent dans la relation Programme) est le titre d'un film (c'est-à-dire apparaissant dans la relation Films).

$$\text{Programme}[\text{Titre}] \subseteq \text{Films}[\text{Titre}].$$

- Les DI peuvent faire intervenir des séquences d'attributs de chaque côté.

Différences DF versus DI

- Les DI se différencient des DF sur plusieurs points
 1. Peuvent être définies entre attributs de relations différentes
 2. Possèdent un caractère plus global (représentent les liens logiques entre des relations).
 3. Les DI sont définies non pas entre deux ensembles quelconques d'attributs, mais entre deux séquences d'attributs de même taille.
 - L'ordre des attributs est donc très important pour les DI !!!

Syntaxe et sémantique des DI

- Soit R un schéma de base de données. Une dépendance d'inclusion sur R est une expression de la forme

$$R[X] \subseteq S[Y],$$

- où $R, S \in R$, X et Y sont des séquences d'attributs distincts respectivement de R et de S , et $|X| = |Y|$.
- Une DI est satisfaite dans une base de données si toutes les valeurs prises par la partie gauche apparaissent dans la partie droite.
- Autrement dit,
 - Soit $d = \{r_1, r_2, \dots, r_n\}$ une base de données sur un schéma $R = \{R_1, \dots, R_n\}$. Une dépendance d'inclusion $R_i[X] \subseteq R_j[Y]$ sur R est satisfaite dans d , noté $d \models R_i[X] \subseteq R_j[Y]$, si $\forall t_i \in r_i, \exists t_j \in r_j$ tel que $t_i[X] = t_j[Y]$
 - de manière équivalente, $\pi_X(r_i) \subseteq \pi_Y(r_j)$.

Exemple

- Supposons des schémas de relation pour décrire les modules :

$$\text{MODULE} = \{\text{NUMMODULE} ; \text{INTITULE} ; \text{DESC} \}$$

- et un schéma de relation pour décrire les séances de cours :

$$\text{SEANCE} = \{\text{DATE} ; \text{NUMMODULE} ; \text{NUMSALLE} \}$$

- Pour forcer que les numéros de modules dans les séances soient bien des modules qui existent, on devra alors définir la contrainte :

$$\text{SEANCE} [\text{NUMMODULE}] \subseteq \text{MODULE} [\text{NUMMODULE}]$$

DI et clé étrangère

- Une contrainte d'intégrité référentielle est une DI dont la partie droite est une clé
 - Un attribut (ou ens. d'attributs) d'une relation apparaît comme clé d'une autre relation.
- La partie gauche d'une contrainte d'intégrité référentielle est appelée **clé étrangère**

Exemple

- les DI ne définissent pas toujours des clés étrangères !!!
- Il suffit d'imaginer qu'on souhaite imposer que tous les cours possèdent au moins une séance dans l'année.
On définira alors une DI :

$$\text{COURS}[\text{NUMCOURS}] \subseteq \text{SEANCE}[\text{NUMCOURS}]$$

- Tous les cours apparaîtront au moins une fois dans la relation des séances
- NUMCOURS n'est pas une clé de SEANCE (on imagine difficilement que tous les cours n'aient qu'une seule séance !)
- Donc ce n'est pas une clé étrangère

Les Dépendances : comment les trouver et pourquoi ?

☰ Comment les trouver ?

- L'analyse du problèmes donne des dépendances de bon sens
 - “jamais deux cours à la même heure dans la même salle”
heure salle -> cours.
- **Problème:**
 - des dépendances peuvent être impliquées de façon **implicite** par d'autres dépendances
nom -> adresse et adresse -> ville, donc nom -> ville
 - ces contraintes implicites peuvent échapper à la connaissance du concepteur
- Nécessité de méthodes permettant de déduire l'ensemble des dépendances induites par un ensemble de dépendances de départ
 - Inférence des DF, DI, ...

Les Dépendances : comment les trouver et pourquoi ?

- ☰ Pourquoi cette formalisation théorique des contraintes ?
 - Exhiber des propriétés théoriques
 - Définir des algorithmes permettant de découvrir automatiquement les dépendances
 - même celles implicites
 - Normaliser les schémas pour éviter les anomalies dans les données
 - normaliser = décomposer les schémas en fonction des dépendances