Langages d’interrogation et de manipulation

1. Algèbre relationnelle

- Ensemble d'opérations permettant de manipuler des relations comme des ensembles de tuples
- Chaque opération prend une ou deux relations en entrée et produit une relation en sortie.
- On peut composer des opérations
- Une requête est une expression algébrique qui s'applique à un ensemble de relations et produit une relation finale

Les opérateurs

5 opérateurs nécessaires et suffisants

1. la sélection σ
2. la projection π
3. le produit cartésien $\times$
4. l'union $\cup$
5. la différence -

- Les deux premiers sont des opérateurs unaires et les autres sont binaires.
- Possibilité de définir la jointure ∞, composition d'un produit cartésien et d'une sélection.

La sélection σ

La sélection $\sigma_F(R)$ extrait de la relation R les tuples satisfaisant un critère de sélection F (=, <, >, , ≤).

La sélection supprime des lignes, mais chaque ligne garde l'ensemble de ses attributs.

$\sigma_{\text{région=« Antilles »}}(\text{station})$

station

nomStation	capacité	lieu	région	tarif
Les boucaniers	300	Sainte Luce	Antilles	80
Ti village	150	Sainte Rose	Antilles	100
La montagne	300	Villard de lans	Auvergne-Rhone-Alpes	60
La forêt	400	La Palmyre	Nouvelle Aquitaine	100
Les tamarys	200	Antibes	Paca	90
Les flots	500	Paimpol	Bretagne	75

La projection π

La projection $\pi_{A1,\ldots,\ Ak}(R)$ ne garde que les attributs A_1, …, A_k de R.

Contrairement à la sélection, on supprime des colonnes.

Le résultat étant une relation, il ne peut pas y avoir deux lignes identiques. Le fait de supprimer des colonnes permet ce cas. On ne conserve alors qu'une seule des deux lignes identiques.

$\pi_{\text{nomStation, capacité}}(\text{station})$

station

nomStation	capacité	lieu	région	tarif
Les boucaniers	300	Sainte Luce	Antilles	80
Ti village	150	Sainte Rose	Antilles	100
La montagne	300	Villard de lans	Auvergne-Rhone-Alpes	60
La forêt	400	La Palmyre	Nouvelle Aquitaine	100
Les tamarys	200	Antibes	Paca	90
Les flots	500	Paimpol	Bretagne	75

Le produit cartésien ×

R×S crée une nouvelle relation où chaque tuple de R est associé à chaque tuple de S.

Le nombre de ligne est $|R| * |S|$.

Conflit de nom d'attribut : si les deux relations ont des attributs de même nom, on préfixe les attributs par le nom de la table de provenance.

On peut aussi faire un renommage noté $\rho_{A \to C,\ B \to D}(T)$.

Possibilité de faire × d'une relation avec elle-même. Le renommage est alors indispensable.

$$\pi_{\text{nomStation, capacité}}(\text{station}) \times \pi_{\text{libellé}}(\text{activité})$$

station

nomStation	capacité	…
Les boucaniers	300	
Ti village	150	
La montagne	300	
La forêt	400	
Les tamarys	200	
Les flots	500	

activité

libellé	**nomStation**	prix
catamaran	Ti village	40
ski de fond	La montagne	20
ski nautique	Ti village	100
surf	La forêt	20

nomstation	capacité	libellé
Les boucaniers	300	catamaran
Les boucaniers	300	ski de fond
…	…	…
Ti village	150	catamaran
…	…	…

L'union ∪

R∪S crée une relation comprenant tous les tuples existants dans l'une ou l'autre des relations R et S.

Les deux relations doivent avoir le même schéma (même nombre d'attributs, mêmes noms et mêmes types).

Comme pour la projection, les doublons sont supprimés.

$$\pi_{\text{région}}(\text{station}) \cup \pi_{\text{région}}(\text{client})$$

station

nomStation	région	…
Les boucaniers	Antilles	
Ti village	Antilles	
La montagne	Auvergne-Rhone-Alpes	
La forêt	Nouvelle Aquitaine	
Les tamarys	Paca	
Les flots	Bretagne	

client

idClient	région	solde	…
205	Ile de France	40	
389	Antilles	20	
1289	Paca	100	

région
Antilles
Auvergne-Rhones-Alpes
Nouvelle Aquitaine
Paca
Bretagne
Ile de France

La différence –

R-S a pour résultat tous les tuples de R qui ne sont pas dans S.

R et S doivent avoir le même schéma.

$\pi_{région}(station) - \pi_{région}(client)$

station

nomStation	région	…
Les boucaniers	Antilles	
Ti village	Antilles	
La montagne	Auvergne-Rhone-Alpes	
La forêt	Nouvelle Aquitaine	
Les tamarys	Paca	
Les flots	Bretagne	

client

idClient	région	solde	...
205	Ile de France	40	
389	Antilles	20	
1289	Paca	100	

région
Auvergne-Rhones-Alpes
Nouvelle Aquitaine
Bretagne

La jointure ∞

Elle rapproche les lignes de deux relations pour lesquelles les valeurs d'un (ou plusieurs) attributs sont identiques.

Souvent, ces attributs sont la clé primaire d'une des relations et la clé étrangère dans l'autre relation.

$R \infty_F S = \sigma_F(R \times S)$, F étant une opération de comparaison liant un attribut de R à un attribut de S.

Le renommage peut intervenir dans la jointure au même titre que pour le produit cartésien.

$$\pi_{idClient,nomStation}(séjour) \infty \pi_{idClient,région}(client)$$

séjour

idClient	nomStation	début	…
205	Ti village	11/2013	
205	Les flots	09/2016	
389	La forêt	08/2014	
1289	Ti village	01/2015	
1289	Les boucaniers	01/2017	

client

idClient	région	…
205	Ile de France	
389	Antilles	
1289	Paca	

idClient	nomStation	région
205	Ti village	Ile de France
205	Les flots	Ile de France
389	La forêt	Antilles
1289	Ti village	Paca
1289	Les boucaniers	Paca

La division ÷

X : ensemble des attributs de R

Y : ensemble des attributs de S

Z : ensemble des attributs communs à X et à Y

R÷S renvoie les tuples r de R sur les attributs X-Z qui vérifient que pour tout tuple s de S sur les attributs de Z, le tuple rs (sur les attributs de X) appartient à R.

$R \div S = \pi_{X-Z}(R) - \pi_{X-Z}((\pi_{X-Z}(R) \times \pi_{Z}(S)) - R)$

Expression de requêtes avec l'algèbre

- Application à la base « organisation de voyage »

  Station(**nomStation**, capacité, lieu, région, tarif)

  Activité(***nomStation*, libellé**, prix)

  Client(**idClient**, nom, prénom, ville, région, solde)

  Séjour(***idClient*, nomStation, début**, nbPlaces)

- Composition des opérations possibles par le fait que tout opérateur produit en sortie une relation sur laquelle on peut appliquer à nouveau des opérateurs.

Sélection généralisée

- Composition de plusieurs sélections = *conjonction*

  $\sigma_{capacité<200}(\sigma_{région='Antilles'}(Station)) \Leftrightarrow \sigma_{capacité<200 \wedge région='Antilles'}(Station)$

- Composition de la sélection et de l'union = *disjonction*

  $\sigma_{capacité<200}(Station) \cup \sigma_{région='Antilles'}(Station) \Leftrightarrow \sigma_{capacité<200 \vee région='Antilles'}(Station)$

- La différence permet d'exprimer la *négation* et d'éliminer des lignes.

  $\sigma_{capacité<200}(Station) - \sigma_{région='Antilles'}(Station) \Leftrightarrow \sigma_{capacité<200 \wedge région \neq 'Antilles'}(Station)$

L'union et la différence permettent de définir une sélection σ_F où F est une expression booléenne.

Requêtes conjonctives

- La plupart des requêtes courantes.
- Elles s'écrivent avec des π, des σ et des $\times$ (et donc indirectement avec des ∞).
- On utilise la jointure dès que les attributs nécessaires sont répartis dans plusieurs tables.

  « Nom et région des stations où on pratique la voile ? »

  $\pi_{nomStation,région}(Station \infty_{nomStation= nomStation} \sigma_{libellé='Voile'}(Activité))$

  « Nom des clients qui sont partis en vacances dans leur région ainsi que le nom de cette région ? »

  $\pi_{nom,client.région}(Client \infty_{idClient=idClient \wedge région=région}(Séjour \infty_{station=nomStation} Station))$

Requêtes avec -

- « tous les O qui ne satisfont pas p ».

  La requête A sélectionne tous les O, ensuite la requête B sélectionne tous les O qui satisfont p et finalement on fait A-B.

  « IdClient des clients qui ne sont pas allés aux Antilles ? »

  $\pi_{idClient}(Client) - \pi_{idClient}(\sigma_{région='Antilles'}(Station) \infty_{nomStation= station} Séjour)$

- La différence peut calculer le complément d'un ensemble.

  « IdClient des clients et les stations où ils ne sont pas allés ? »

  $(\pi_{idClient}(Client) \times \pi_{nomStation}(Station)) - \pi_{idClient,station}(Séjour)$

Quantification universelle

- Une propriété est vraie pour tous les éléments d'un ensemble ssi il n'existe pas un élément de cet ensemble pour lequel la propriété est fausse.
- On emploie la négation et la quantification existentielle.

  « Quelles sont les stations dont toutes les activités ont un prix supérieur à 100 ? »

  $\pi_{nomStation}(Station) - \pi_{nomStation}(\sigma_{prix<100}(Activité))$

Requêtes les plus complexes qui utilisent la division

« Quelles sont les activités qui sont dans toutes les stations ? »

Activité ÷ Station

« IdClient des clients qui sont allés dans toutes les stations ? »

(= « idClient des clients tels qu'il n'existe pas de station où ils ne soient pas allés »)

$\pi_{idClient}(Client) - \pi_{idClient}((\pi_{idClient}(Client) \times \pi_{nomStation}(Station)) - \pi_{idClient,station}(Séjour))$

2. SQL (Structured Query Language)

- Langage d'interrogation et de manipulation de données.
- Suit la syntaxe de la norme SQL2 implantée dans la plupart des SGBDR.
- Langage déclaratif permettant d'interroger une base de données sans se soucier de la représentation interne des données, de leur localisation, des chemins d'accès ou des algorithmes nécessaires.
- Utilisation de manière interactive mais aussi en association avec des interfaces graphiques ou des langages de programmation.

Requêtes simples

SELECT …

FROM …

WHERE …

- FROM indique la ou les tables dans lesquelles on trouve les attributs utiles à la requête,
- SELECT indique la liste des attributs pour le résultat (= projection),
- WHERE indique les conditions que doivent satisfaire les tuples pour faire partie du résultat (= sélection).

- Le résultat est une relation dont les attributs sont ceux spécifiés dans la clause SELECT.
- « Découpage » horizontal et vertical de la table indiquée dans le FROM (utilisation combinée de la sélection et de la projection).

Station(**nomStation**, capacite, lieu, region, tarif)

Activite(*nomStation*, **libelle**, prix)

Client(**idClient**, nom, prenom, ville, region, solde)

Sejour(*idClient*, **nomStation, debut**, nbPlaces)

- On peut renommer les attributs, appliquer des fonctions

```
SELECT libelle, prix/6,56 AS prixEnEuros
FROM Activite
WHERE nomStation = 'Santalba'
```

- Pour éviter deux tuples identiques, on utilise le mot-clé DISTINCT, cette opération peut être coûteuse.

```
SELECT DISTINCT libelle
FROM Activite
```

- On trie le résultat d'une requête avec la clause ORDER BY

  SELECT * (* : tous les attributs)

  FROM Station

  ORDER BY tarif, nomStation

- Pour trier en ordre descendant, on utilise le mot-clé DESC après la liste des attributs.
- Pour obtenir une recherche par intervalle, on utilise le mot-clé BETWEEN.

- SQL fournit des options de recherche par motif à l'aide de la clause LIKE

  '_' désigne n'importe quel caractère

  '%' n'importe quelle chaîne.

- Une date est spécifiée par le mot-clé DATE au format 'aaaa-mm-jj'

```
SELECT IdClient
FROM Sejour
WHERE debut BETWEEN DATE '1999-07-01' AND DATE '2006-07-01'
```

- La valeur de certains attributs est inconnue : NULL.
- Toutes les opérations appliquées à NULL retourne NULL.
- Toute comparaison avec NULL donne UNKNOWN.

  TRUE = 1, FALSE = 0 et UNKNOWN = ½.

  x AND y = min(x,y) ; x OR y = max(x,y)

  NOT x = 1-x
- La présence de NULL dans une comparaison renvoie FALSE. NULL peut avoir des effets surprenants.
- NULL est un mot-clé pas une constante. Pour tester l'absence de valeur : 'x IS NULL' (ou 'x IS NOT NULL').
- Il faut éviter NULL en spécifiant la contrainte NOT NULL ou en donnant une valeur par défaut.

Requêtes sur plusieurs tables

Jointure

- Liste des tables concernées dans la clause FROM et critères de rapprochement entre ces tables dans WHERE.
- On construit le produit cartésien des tables du FROM, en préfixant chaque attribut par le nom de sa table pour éviter les ambiguïtés.

  « Nom des clients avec le nom des stations où ils ont séjourné ? »

```
SELECT nom, station
FROM Client, Sejour
WHERE client.idClient = sejour.idClient
```

« Nom des clients habitant Paris, les stations où ils ont séjourné avec la date et le tarif hebdomadaire pour chaque station ? »

```
SELECT nom, nomStation, début, tarif
FROM Client, Sejour, Station
WHERE ville = 'Paris'
AND client.idClient = sejour.idClient
AND station.nomStation = sejour.nomStation
```

« Couples de stations situées dans la même région ? »

```
SELECT s1.nomStation, s2.nomStation
FROM Station s1, Station s2
WHERE s1.region = s2.region
```

Tout se passe comme si on faisait la jointure entre deux versions de la table Station.

Union, intersection et différence

- On construit deux requêtes dont les résultats ont même arité et on les relie par un des mots-clé UNION, INTERSECT ou EXCEPT.

  « Tous les noms des régions dans la base ? »

  ```
  SELECT region FROM Station
  UNION
  SELECT region FROM Client
  ```

  « Régions où on trouve à la fois des clients et des stations ? »

  ```
  SELECT region FROM Station
  INTERSECT
  SELECT region FROM Client
  ```

« Régions où on trouve des stations mais pas des clients ? »

```
SELECT region FROM Station
EXCEPT
SELECT region FROM Client
```

- La norme SQL2 spécifie que les doublons doivent être éliminés du résultat lors des trois opérations ensemblistes. Certains systèmes ne suivent pas cette norme. INTERSECT peut être exprimé avec une jointure et la différence avec des requêtes imbriquées.

Requêtes imbriquées

Conditions portant sur des relations

- Jointures pour lesquelles le résultat est constitué avec les attributs d'une seule table, l'autre ne servant que pour exprimer des conditions.

  « Nom des stations où ont séjourné des clients parisiens ? »

```
    SELECT nomStation
    FROM Client, Sejour
    WHERE ville = 'Paris'
    AND client.idClient = sejour.idClient
⇒   SELECT nomStation
    FROM Sejour
    WHERE idClient IN (SELECT idClient
                       FROM Client
                       WHERE ville = 'Paris')
```

- IN exprime la condition d'appartenance de idClient à la relation formée par la requête imbriquée.
- Sur une relation R construite avec une requête imbriquée, on peut exprimer les conditions suivantes :
  - EXISTS R, renvoie True si R n'est pas vide, False sinon (ou NOT EXISTS).
  - t IN R, renvoie True si le tuple t appartient à R, False sinon (ou NOT IN).
  - v cmp ANY R, renvoie True si la comparaison de l'attribut v avec un moins un des tuples de R est vérifiée, False sinon, cmp $\in$ {<, >, =, …}.
  - v cmp ALL R, renvoie True si la comparaison de l'attribut v avec tous les tuples de R est vérifiée, False sinon, cmp $\in$ {<, >, =, …}.
- La différence s'exprime avec NOT IN ou NOT EXISTS.

Sous-requêtes corrélées

- La sous-requête est basée sur une ou plusieurs valeurs issues des relations de la requête principale.

« Quels sont les clients (nom, prénom) qui ont séjourné à Santalba ? »

⇒

```
SELECT nom, prenom
FROM Client
WHERE EXISTS (SELECT *
                FROM Sejour
                WHERE nomStation = 'Santalba'
                      AND client.idClient = sejour.idClient)
```

« Dans quelle station pratique-t-on une activité au même prix qu'à Santalba ? »

⇒

```
SELECT nomStation
FROM Activite A1
WHERE EXISTS (SELECT *
                FROM Activite A2
                WHERE nomStation = 'Santalba'
                      AND A1.libelle = A2.libelle
                      AND A1.prix = A2.prix)
```

Fonctions d'agrégation

- Ces fonctions s'appliquent à une colonne en général de type numérique :
  - COUNT qui compte le nombre de valeurs non nulles,
  - MAX, MIN
  - AVG qui calcule la moyenne des valeurs de la colonne,
  - SUM qui effectue le cumul.

```
SELECT COUNT(nomStation), AVG(tarif), MIN(tarif), MAX(tarif)
FROM Station
```

« Combien de places a réservé M. Kerouac pour l'ensemble des séjours ? »

```
SELECT SUM(nbPlaces)
FROM Client, Sejour
WHERE nom = 'Kerouac'
AND client.idClient = sejour.idClient
```

- On ne peut pas utiliser simultanément dans la clause SELECT des fonctions d'agrégation et des noms d'attributs, sauf dans le cas d'un GROUP BY.

La clause GROUP BY

- On partitionne le résultat en groupes, en associant les tuples partageant la même valeur pour une ou plusieurs colonnes.

  « Afficher les régions avec le nombre de stations »

```
SELECT region, COUNT(nomStation)
FROM Station
GROUP BY region
```

  « Donner le nombre de places réservées par client »

```
SELECT nom, SUM(nbPlaces)
FROM Client, Sejour
WHERE client.idClient=sejour.idClient
GROUP BY idClient, nom
```

- COUNT et SUM n'ont ici de sens qu'avec le GROUP BY afin de compter pour chaque groupe.

La clause HAVING

- On peut faire porter des conditions sur les groupes avec la clause HAVING.
- La clause WHERE ne peut exprimer des conditions que sur les tuples pris un à un.

  « Nombre de places réservées par client, pour les clients ayant réservé plus de 10 places ? »

```
SELECT nom, SUM(nbPlaces)
FROM Client, Sejour
WHERE client.idClient=sejour.idClient
GROUP BY nom
HAVING SUM(nbPlaces) >= 10
```

Mises à jour

Insertion

INSERT INTO R(A_1, A_2, …, A_n) VALUES (v_1, v_2, …,v_n)

R est le nom de la relation, A_1, A_2, …, A_n sont les noms des attributs dans lesquels on veut placer une valeur et v_1, v_2, …,v_n sont les valeurs.

```
INSERT INTO Client (id, nom, prenom)
VALUES (40, 'Dupont', 'Jean')
```

Les attributs ville et region seront à NULL, solde sera à 0.

Il est également possible d'insérer dans une table le résultat d'une requête. La partie VALUES est remplacée par la requête.

Exemple : on crée une table Sites et on souhaite y copier les couples (lieu, region) déjà existant dans la table Station.

```
INSERT INTO Sites (lieu, region)
SELECT lieu, region FROM Station
```

Destruction

DELETE FROM R

WHERE condition

Condition est une condition valide pour toute clause WHERE.

« Destruction de tous les clients dont le nom commence par ‘M’ »

DELETE FROM Client

WHERE nom like ‘M%’

Modification

UPDATE R SET $A_1=v_1$, $A_2=v_2$, …, $A_n=v_n$

WHERE condition

« Augmenter les prix des activités de la station Passac de 10% »

```
UPDATE Activite
SET prix=prix*1.1
WHERE nomStation='Passac'
```

- Toutes les mises à jour ne deviennent définitives qu'à l'issue d'une validation par COMMIT. Entre-temps, elles peuvent être annulées par ROLLBACK.

Création de schémas relationnels

Utilisateurs

- L'accès à une BD est restreint à des utilisateurs connus et identifiés par un nom et un mot de passe. Chaque utilisateur a des droits sur les schémas et les tables.

  CONNECT utilisateur

- Le propriétaire d'un schéma a tous les droits sur ce schéma. Il définit les droits des autres utilisateurs sur ce schéma.

  GRANT <privilège>

  ON <élément du schéma>

  TO <utilisateur> [WITH GRANT OPTION]

  privilège = INSERT, UPDATE, SELECT, DELETE, REFERENCE (référence à une table dans une contrainte d'intégrité), USAGE (utilisation une définition ≠ table ou insertion).

- Pour accorder un privilège, il faut en avoir le droit soit parce qu'on est propriétaire du schéma, soit parce que le droit est accordé par WITH GRANT OPTION.

  GRANT SELECT ON Film TO Marc
- On désigne tous les utilisateurs avec PUBLIC, et tous les privilèges avec ALL PRIVILEGES.

  GRANT ALL PRIVILEGES ON Film TO PUBLIC
- On supprime un droit avec la commande

  REVOKE <privilège>

  ON <élément du schéma>

  FROM <utilisateur>

Définition d'un schéma

- Outre les tables, un schéma peut comprendre des vues, des contraintes de différents types, des triggers, etc.
- On crée un schéma en lui donnant un nom puis en donnant la liste des commandes créant les éléments.

  CREATE SCHEMA officiel_des_spectacles

  CREATE TABLE Film …

  CREATE VIEW …

  CREATE ASSERTION …

  CREATE TRIGGER …
- Pour choisir le schéma courant, on utilise la commande

  SET SCHEMA officiel_des_spectacles

Contraintes et assertions

- Restriction des attributs à un ensemble de valeurs, ou plus complexes faisant référence à d'autres relations. Une contrainte s'exprime par CHECK (condition).

```
CREATE TABLE Salle
(NomCine        VARCHAR (30) NOT NULL,
No              INTEGER,
Capacite        INTEGER CHECK (Capacite < 300),
Climatise       CHAR(1) CHECK (Climatise IN ('O', 'N')),
PRIMARY KEY (NomCine, No),
FOREIGN KEY NomCine REFERENCES Cinema)
```

- Au lieu d'associer une contrainte à un attribut particulier, on peut la définir globalement avec CONSTRAINT en donnant un nom à chaque contrainte.

  « Toute salle de plus de 300 places doit être climatisée ».

  CONSTRAINT clim CHECK (Capacite < 300 OR Climatise = 'O')

- On peut modifier ou détruire une contrainte :

  ALTER TABLE … DROP CONSTRAINT ...

  ALTER TABLE Salle DROP CONSTRAINT clim

Vues

Création et interrogation

- Ajouter au schéma des tables « virtuelles », résultats de requêtes stockées appelées « vues ».
- Résultat de la requête réévalué à chaque fois qu'on accède à la vue → dynamique.
- Représentation différente des tables.

```
CREATE VIEW nom_vue
AS <requête>
[WITH CHECK OPTION]
```

- « Vue ne contenant que les cinémas parisiens limités à leur nom et leur nombre de salles »

```
CREATE VIEW ParisCinemas
AS SELECT NomCine, COUNT(*) AS NbSalles
    FROM Cinema c, Salle s
    WHERE Ville = 'Paris'AND c.NomCine = s.NomCine
    GROUP BY c.NomCine
```

- « Vue donnant les titres des films, leur année et les noms et prénoms des acteurs »

```
CREATE VIEW Casting (film, annee, acteur, prenom)
AS SELECT Titre, Annee, NomArt, PrenomArt
    FROM Film f, Role r, Artiste a
    WHERE f.IdFilm = r.IdFilm AND a.IdArt = r.IdArt
```

- On peut utiliser des vues et des tables dans SQL.
- « Quels acteurs ont tourné un film en 1997 ? »

```
SELECT acteur, prenom
FROM Casting
WHERE annee = 1997
```

- On peut donner des droits en lecture sur cette vue.

```
GRANT SELECT ON Casting TO PUBLIC
```

Modification

- Modifier la table d'origine qui sert de support à la vue :
  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 calcule ou d'une opération.
- Option WITH CHECK OPTION (à la création de la vue) garantit que toute ligne insérée dans la vue satisfait les critères de sélection de la vue.
- Pour supprimer la vue, on utilise DROP

  DROP VIEW ParisCinemas

Triggers

- Procédure déclenchée par des événements de mise à jour spécifiés par l'utilisateur.
- Possibilité de manipuler simultanément les valeurs ancienne et nouvelle de la donnée modifiée (pour des tests)
- Base de donnée dynamique : une opération sur la base peut en déclencher d'autres, et ainsi de suite. Attention aux boucles sans fin.

```
CREATE TRIGGER nom-trigger
<quand> <événement> ON <table>
[FOR EACH ROW [WHEN <condition>]]
    BEGIN
    <action>
    END;
```

quand = BEFORE ou AFTER

événement = DELETE, UPDATE, ou INSERT (séparé par des OR)

FOR EACH ROW = en son absence, le trigger est déclenché une fois pour toute requête modifiant la table (utiliser new.attribut ou old.attribut).

action = Contient des ordres SQL mais pas de mise à jour de la table courante.

```
CREATE TRIGGER CumulCapaciteGlobal
AFTER UPDATE OR INSERT OR DELETE ON Salle
    BEGIN
        UPDATE Cinema c SET Capacite = (SELECT SUM (capacite) FROM Salle s
    WHERE c.NomCine = s.NomCine) ;
    END;
```