SQL et les bases de données relationnelles

SQL : au-delà du calcul relationnel

Guillaume Raschia — Nantes Université

originaux de Philippe Rigaux, CNAM et Jeff Ullman, Stanford

Dernière mise-à-jour : 7 décembre 2023

Plan de la session

À propos du bloc SFW (S5.1)

Agrégats (S5.3)

Création de schéma (S7.1)

Vues (S7.3)

Mises-à-jour (S5.4)

Expression de contraintes et déclencheurs (S7.1-7.2, S8.3)

Procédures stockées (S8.1)

À propos du bloc SFW (S5.1)

Récapitulatif SQL : le bloc `select-from-where`

Dans cette section : présentation de quelques extensions pratiques au SQL fondamental.

- Les valeurs nulles
- La jointure externe
- L'anti-jointure
- Le tri

Valeurs nulles

Une valeur nulle, ou plus précisément valeur à `null` est une valeur manquante.
Ne pas confondre avec la valeur « null » ou « ».

Dans notre table des occupants, le prénom de Prof est à `null`.

id	prénom	nom	profession	idAppart
1		Prof	Enseignant	202
2	Alice	Grincheux	Cadre	103
3	Léonie	Atchoum	Stagiaire	100
4	Barnabé	Simplet	Acteur	102
...	...	...	...	...

La présence de valeurs à `null` fausse le résultat attendu des requêtes.

Comparaisons avec valeurs nulles

On ne sait pas comparer un valeur manquante, ni lui appliquer une fonction.

```
select p.profession
from   Personne as p
where  not p.prénom = 'Alice';
```

Prof – sans prénom – n'est pas trouvé.

```
select * from Personne p where p.prénom like '%';
```

Prof n'est pas trouvé non plus !

Une comparaison avec `null` ne donne ni *Vrai*, ni *Faux*, mais une troisième valeur de vérité, `unknown`.

Calculs avec valeurs à null

Tout calcul appliqué à une valeur à `null` renvoie `null` !

```
select p.prénom || ' ' || p.nom as 'nomComplet'
from   Personne p
```

nomComplet
`null`
Alice Grincheux
Léonie Atchoum
Barnabé Simplet
…

Le test `is null`

Seule approche correcte :

- il faut tester explicitement l'absence de valeur avec `is null`.

En SQL :

```
select * from Personne
where  prénom like '%'
or     prénom is null
```

Attention le test `prénom = null` ne marche pas.

Conclusion

éviter autant que possible les valeurs à `null` en les interdisant (dans le schéma).

La jointure externe

On veut la liste des appartements avec leurs occupants.

```
select idImmeuble as IdI, no, niveau, surface, nom, prénom
from   Appart as a, Personne as p where p.idAppart = a.id
```

idI	no	niveau	surface	nom	prénom
2	2	2	250	Prof	null
1	52	5	50	Grincheux	Alice
1	1	14	150	Atchoum	Léonie
1	51	2	200	Simplet	Barnabé
2	1	1	250	Joyeux	Alphonsine
1	43	3	75	Timide	Brandon
2	10	0	150	Dormeur	Don-Jean

Il manque l'appartement 34 qui n'a pas d'occupant.

L'opérateur `outer join`

L'opérateur algébrique `left [outer] join`

- Renvoie tous les nuplets de la table directrice (celle de gauche)
- Associe à chaque nuplet un nuplet de la table de droite si un tel nuplet existe
- Sinon, les attributs provenant de la table de droite sont affichés à `null`

```
select idImmeuble as IdI, no, niveau, surface, nom, prénom
from   Appart as a
left outer join Personne as p
on     (p.idAppart = a.id)
```

On obtient, en plus de la jointure standard :

1	34	3	50	null	null

L'anti jointure

Un opérateur utile, combinant la jointure externe et la comparaison à `null`

Exemple

Donner la description des logements non occupés par des enseignants.

```
-- exemple d'anti-jointure
select a.idImmeuble, a.no, a.niveau, a.surface
from   Appart as a
left outer join     /* jointure avec les logements d'enseignants */
       (select idAppart
        from   Personne
        where  profession = 'Enseignant'
       ) as e
on     (e.idAppart = a.id)
/* sélection de ceux qui n'ont pas été appairés */
where  e.idAppart is null
```

Le tri, `order by`

On peut demander explicitement le tri du résultat sur une ou plusieurs expressions avec la clause `order by`

```
select   *
from     Appart
order by surface, niveau
```

En ajoutant des clauses sur l'ordre du tri (`ascending` ou `descending`)

```
select   *
from     Appart
order by surface desc, niveau desc
```

À retenir

Les fondements du langage SQL – logique ou algèbre – sont une base sur laquelle beaucoup d'extensions pratiques sont possibles.

- Valeurs à `null`, jointures externes, anti-jointures, tris
- Mais aussi des fonctions pour manipuler les valeurs typées, bien souvent spécifiques à chaque système
- Ne change en rien l'interprétation du langage, que vous devez maintenant maîtriser.

Agrégats (S5.3)

SQL, agrégation

Cette section présente les agrégats en SQL. Elle consiste à regrouper des lignes et à appliquer à chaque groupe une fonction d'agrégation.

Contenu :

- La clause `group by`
- Fonctions d'agrégation
- La clause `having`

Principe général

On définit des groupes de lignes ayant en commun une ou plusieurs valeurs.

On ramène chaque groupe à une seule valeur en appliquant une fonction d'agrégation, parmi `MIN`, `MAX`, `SUM`, `AVG` et même `COUNT`.

Cas le plus simple : un seul groupe, obtenu par un bloc `select-from-where`.

```
select count(*) as nbPersonnes,
       count(prénom) as nbPrénoms,
       count(nom) as nbNoms
from   Personne
```

nbPersonnes	nbPrénoms	nbNoms
7	6	7

Le `group by`

La clause `group by att1, ..., attn` partitionne le résultat d'un bloc `select-from-where` en fonction des `att1, ..., attn`

Chaque groupe contient les lignes qui partagent les mêmes valeurs pour `att1, ..., attn`.

```
select   idAppart, sum(quotePart) as totalQP
from     Propriétaire
group by idAppart
```

Procède en deux étapes : d'abord on groupe, puis on agrège.

Décomposons : l'étape de regroupement

On obtient une structure intermédiaire, avec autant de lignes que de valeurs distinctes pour les attributs de regroupement (ici, `idAppart`).

idAppart	Groupe (idPersonne, idAppart, quotePart)
100	{(1, 100, 33), (5, 100, 67)}
101	{(1, 101, 100)}
102	{(5, 102, 100)}
103	{(2, 103, 100)}
104	{(2, 104, 100)}
201	{(5, 201, 100)}
202	{1, 202, 100)}

Ce n'est pas une table en première forme normale.

L'étape d'agrégation

La fonction d'agrégation ramène un groupe à une valeur

idAppart	SUM(quotePart)
100	SUM (33, 67) = 100
101	SUM (100) = 100
102	SUM (100) = 100
103	SUM (100) = 100
104	SUM (100) = 100
201	SUM (100) = 100
202	SUM (100) = 100

- Cette fois c'est une table en première forme normale.
- La clause `select` ne contient que des valeurs agrégées ou des attributs de regroupement.

La clause `having`

Exprime un critère de sélection sur le résultats de la fonction d'agrégation.

Bien distinguer de la clause `where` qui s'applique aux nuplets

```
select   idAppart, count(*) as nbProprios
from     Propriétaire
group by idAppart
having   count(*) >= 2
```

idAppart	nbProprios
100	2

À retenir

Agrégats = extension de SQL.

- S'applique au résultat d'une requête standard
- Partitionne en groupes de nuplets partageant les mêmes valeurs de regroupement
- Réduit chaque groupe à une valeur grâce à une fonction d'agrégation
- On peut filtrer les groupes obtenus avec `having`

Attention aux valeurs à `null` et aux doublons.

Création de schéma (S7.1)

Création d'un schéma relationnel

La partie *Data Definition Language* (DDL) définit les commandes de création d'un schéma relationnel.

Dans cette section :

- La commande `create table`
- Les types de données
- La déclaration des clés

La commande `create table`

Un premier exemple : la table `Internaute`.

```
create table Internaute (email  varchar (40),
                         nom    varchar (30),
                         prénom varchar (30),
                         région varchar (30)
                         );
-- la contraposée, pour supprimer une table
drop table Internaute;
```

Quelques choix à effectuer : conventions de nommage, accents, etc.

Les types de données en SQL

Les types les plus usités :

- `int` or `integer` (synonymes)
- `real` ou `float` (synonymes)
- `char(n)` : chaîne de caractères de taille n fixe
- `varchar(n)` : chaîne de caractères de taille variable ($\leq n$).
- `date` et `time`

Les valeurs en SQL

- Les nombres entiers (5) et réels (3.0)
- Les chaînes de caractères entourées de guillemets simples
- Les dates sont par défaut 'yyyy-mm-dd'
- Les heures sont 'hh:mm:ss'
- Toute valeur peut être mise à `null`

De très nombreuses fonctions permettent de manipuler les valeurs en SQL
cf. les ressources du site web sql.sh.

La contrainte `not null`

Les valeurs à `null` sont source de problèmes : on peut les interdire avec `not null` ou donner une valeur par défaut.

```
create table Cinéma (idCinéma integer not null,
                     nom      varchar(30),
                     adresse  varchar(255) default 'Inconnue'
                     );
```

Le SGBD rejettera alors toute tentative d'insérer un nuplet avec une valeur `idCinéma` manquante.

La clé primaire

Elle est spécifiée avec la clause `primary key`.

```
create table Pays (code  char(4) primary key,
                   nom    varchar(30),
                   langue varchar(30),
                   );
```

Il devrait toujours y avoir une et une seule clé primaire dans une table et le système garantit

- l'existence de ses valeurs (`not null`),
- l'unicité de ses valeurs.

Clé primaire composée

Une clé primaire peut comprendre plusieurs attributs.

```
create table Notation (idFilm      integer,
                       email       varchar (40),
                       note        integer not null,
                       primary key (idFilm, email));
```

Tous les attributs qui composent une clé primaire sont `not null`.

Clé primaire, clés « secondaires »

On peut définir d'autres clés avec la clause `unique`.

```
create table Artiste  (idArtiste   integer,
                       nom         varchar (30),
                       prénom      varchar (30),
                       annéeNaiss  integer,
                       primary key (idArtiste),
                       unique      (nom, prénom))
```

- `unique` peut également figurer dans la déclaration de l'attribut.
- Une contrainte `unique` ne garantit pas l'absence de valeur à `null`.

Permet d'avoir un identifiant « abstrait », non modifiable, et d'ajouter des contraintes flexibles sur les attributs descriptifs.

À retenir

Après la phase de conception, la création du schéma ne présente aucune difficulté.

- Connaître les principaux types SQL
- Connaître la commande `create table`
- Ajouter les contraintes
  - de clé (`unique` et `primary key`),
  - et d'existence (`not null`).

La spécification des contraintes n'est pas une lourdeur. Elle garantit que la base est saine, et évite beaucoup d'ennuis.

Vues (S7.3)

Les vues

Toute requête produit une relation. Nommer cette requête c'est nommer la relation résultat et pouvoir l'interroger. Une vue est une requête nommée.

C'est aussi une table calculée au moment où on l'interroge.

Dans cette session :

- La commande `create view`
- Interrogation d'une vue
- Mise à jour d'une vue

La commande `create view`

```
create view Sillon as
        select   nom, adresse, count(*) as nb_logts
        from     Immeuble as i, Logement as l
        where    i.nom = 'Sillon'
        and      i.id = l.idImmeuble
        group by i.id, nom, adresse
```

Le résultat de la requête est réévalué à chaque fois que l'on accède à la vue.

Interrogation d'une vue

On interroge une vue comme n'importe quelle table.

```
select * from Sillon
```

nom	adresse	nb_logts
Sillon	1 Av. de l'Angevinière	780

Une vue peut répondre à des objectifs de simplification et/ou de restriction d'accès et de confidentialité.

Vue dénormalisée

Une vue peut présenter une base dénormalisée.

```
create view LogtSillon as
       select no, surface, niveau, i.nom as immeuble, adresse,
              concat(p.prénom,  ' ', p.nom) as occupant
       from   Immeuble as i, Logement as l, Personne as p
       where  i.id=l.idImmeuble
       and    l.id=p.idLogement
       and    i.nom='Sillon'
```

no	surf.	niv.	immeuble	adresse	occupant
1	150	29	Sillon	1 Av. de l'Angevinière	Léonie Atchoum
51	77	7	Sillon	1 Av. de l'Angevinière	Barnabé Simplet
52	45	12	Sillon	1 Av. de l'Angevinière	Alice Grincheux
43	58	3	Sillon	1 Av. de l'Angevinière	Brandon Timide

Insertion dans une vue

Beaucoup de restrictions :

- la vue doit être basée sur une seule table ;
- toute colonne non référencée dans la vue doit pouvoir être mise à `null` ou disposer d'une valeur par défaut ;
- on ne peut pas mettre à jour un attribut qui résulte d'un calcul ou d'une opération.

```
create view PropriétaireAlice as
    select * /* idProp, idLogt, quotePart */ from Propriétaire
    where idPersonne=2

insert into PropriétaireAlice values (2, 100, 20)
insert into PropriétaireAlice values (3, 100, 20)
```

La clause `check option`

Sur la vue précédente, la requête :

```
select * from PropriétaireAlice
```

ne montre pas le propriétaire 3 que l'on vient d'insérer !

La clause `check option` permet de n'insérer que des nuplets que l'on peut ensuite sélectionner.

```
create view PropriétaireAlice as
    select * from Propriétaire
    where  idPersonne = 2
with check option
```

À retenir

Les vues sont des requêtes nommées que l'on peut traiter comme des tables.

Elles permettent de restructurer « virtuellement » une base.

- Pour simplifier l'accès (jointures pré-définies)
- Pour restreindre la visibilité des données (on ne donne accès qu'à la vue)
- Les modifications dans les vues sont limitées et offrent peu d'intérêt

Mises-à-jour (S5.4)

Mises-à-jour des données

Des commandes SQL sont dévolues aux mises-à-jour des données des relations : l'insertion, la suppression et la modification.

Les commandes de mise-à-jour ne produisent aucune relation (à l'inverse des requêtes), mais modifient l'état de la base de données.

Dans cette section :

- La commande `insert`
- La commande `delete`
- La commande `update`

L'insertion, `insert`

```
insert into <relation> values ( <liste de nuplets> );
```

Exemple

Ajouter à la table `Aime(buveur, bière)` le fait que Alice aime la Titan.

```
insert into Aime values ('Alice', 'Titan');
```

`insert`, avec la liste des attributs

- Il est possible de spécifier une liste d'attributs
- Pour deux raisons :
  1. On ne respecte pas l'odre des attributs
  2. On ne fournit pas de valeur pour certains attributs dont on souhaite que le système complète à `null` ou une valeur par défaut

Exemple

```
insert into Aime(bière, buveur) values ('Titan', 'Alice');
```

`insert`, avec valeurs par défaut

Lors de la création de table, si une valeur par défaut est déclarée pour un attribut, celle-ce sera affectée à chaque insertion sans valeur.

Exemple

```
create table Buveur(
    nom     char(30) primary key,
    adresse varchar(50) default '123 rue C. Pauc',
    tel     char(10));

insert into Buveur(nom) values ('Alice')
```

Le nuplet créé est Buveur('Alice', '123 rue C. Pauc', `null`).

`insert`, par lot

Insertion du résultat d'une sous-requête :

```
insert into <relation> values ( <sous-requête> );
```

Exemple

À partir de la table `Fréquente(buveur, bar)`, on insert dans la relation `Rencontre(nom)` tous les buveurs qui fréquentent au moins un bar qu'Alice fréquente.

```
insert into Rencontre values (
    select f2.buveur
    from   Fréquente f1, Fréquente f2
    where  f1.buveur = 'Alice'
    and    not f2.buveur = 'Alice'
    and    f1.bar = f2.bar);
```

`delete`, pour la suppression

Supprimer des nuplets qui remplissent une condition :

```
delete from <relation> where <condition>;
```

Exemple

```
-- Alice n'aime plus la Titan
delete from Aime where buveur = 'Alice' and bière = 'Titan';

-- Plus aucun buveur n'aime de bière!
delete from Aime;
```

`delete`, pour plusieurs nuplets

Supprimer de la table `Bière(nom, brasserie)`, toutes les bières pour lesquelles il en existe déjà une de la même brasserie.

```
delete from Bière b where exists (
    select name from Bière bb
    where  bb.brasserie = b.brasserie
    and not bb.nom = b.nom );
```

delete, sémantique

```
delete from Bière b where exists (
    select name from Bière bb
    where  bb.brasserie = b.brasserie and not bb.nom = b.nom);
```

- Supposons que :
  - la Brasserie du Bouffay ne brasse que la Titan et la Moustache ;
  - la Titan soit la première bière affectée à la variable b ;
- La sous-requête n'est pas vide, puisqu'il existe la Moustache, donc la Titan est supprimée.
- Ensuite, vient l'examen de la Moustache. Doit-on supprimer cette dernière ?

`delete`, sémantique (suite)

Il faut supprimer la Moustache également.

La suppression opère en effet comme suit :

1. marquage de tous les nuplets qui remplissent la condition
2. suppression des nuplets marqués.

Modifications

L'instruction `update` change les valeurs d'attributs pour certains nuplets de la relation.

```
update    <relation>
    set   <liste d'affectations par attribut>
    where <condition sur les nuplets>
```

Exemple

Modifie le numéro de téléphone de Bob en 06.01.02.03.04

```
update Buveur set tel = '06.01.02.03.04' where nom = 'Bob';
```

Permet de modifier plusieurs nuplets : fixe un prix maximum pour les bières.

```
update Carte set prix = 4.00 where prix > 4.00;
```

À retenir

Les mises-à-jour de données en SQL sont circonscrites aux trois commandes

- `insert` : insertion individuelle ou par lot à l'aide d'une sous-requête
- `delete` : sur condition via la clause `where`
- `update` : idem.

Expression de contraintes et déclencheurs (S7.1-7.2, S8.3)

Les contraintes

Cette section détaille les formes d'expression de contraintes et les déclencheurs disponibles en SQL.

Les points abordés sont les suivants :

- clé étrangère
- contrainte locale et contrainte globale
- déclencheur

Contraintes et déclencheurs

Une **contrainte** est une propriété sur les données que le SGBD doit garantir.

Exemple

`unique`, `not null`, `primary key`

Un **déclencheur** (ou **trigger** en anglais) est un traitement réalisé à chaque fois qu'un événement survient, tel que l'insertion d'un nuplet.

- C'est une forme d'exigence parfois plus facile à exprimer qu'une contrainte complexe.

Les types de contraintes

- clé, existence, unicité
- clé étrangère, ou contrainte d'intégrité référentielle
- contrainte de domaine
  - locale
  - propre à un seul attribut
- contrainte de nuplet
  - globale
  - spécifie des relations entre plusieurs attributs
- assertion : toute expression booléenne formulée en SQL

Révision : clé simple et clé composite

Placer le mot-clé (sic !) `primary key` ou `unique` en fin de déclaration d'un attribut

Exemple

```
create table Bière(
    nom       varchar(20) unique,
    brasserie varchar(20));

-- La clé de la table Carte est composée du bar et de la bière
create table Carte(
    bar   varchar(20),
    bière varchar(20),
    prix  real,
    primary key (bar, bière));
```

Clé étrangère

Les valeurs d'un attribut doivent être piochées dans l'ensemble des valeurs d'un attribut de référence présent dans une autre relation.

Exemple

Dans la relation `Carte(bar, bière, prix)` on suppose que les bières figurent parmi les valeurs de l'attribut `Bière.nom`.

Expression d'une clé étrangère

À l'aide du mot-clé `references` :

- soit immédiatement après la déclaration d'attribut,
- soit comme un nouvel élément du schéma.

```
create table Bière (
    nom       varchar(20) primary key,
    brasserie varchar(20));

create table Carte (
    bar   varchar(20),
    bière varchar(20) references Bière(nom),
    prix  real
    /* alt.
    foreign key (bière) references Bière(nom) */
    );
```

Les garanties d'une clé étrangère

Supposons une clé étrangère dans S qui se rapporte à des valeurs dans R;

Violation de contrainte

- l'insertion ou la modification d'une valeur dans S, absente de R
- la suppression ou la modification d'une valeur de référence dans R, qui produit des nuplets « orphelins » dans S

Exemple

Avec S = `Carte` et R = `Bière`, l'ajout d'un triplet à la `Carte` composé d'une bière inconnue (dans la table `Bière`) constitue une violation de contrainte.

De même, la suppression d'une bière (de la table `Bière`) qui est encore disponible à la `Carte` pose problème.

Traitement des violations de contrainte

Il existe 3 façons de traiter ces problèmes.

Les 3 options

1. `default` : rejeter la mise-à-jour
2. `cascade` : propager la mise-à-jour
   - suppression d'une bière : supprimer les nuplets correspondant à la carte
   - modification d'une bière : modifier la valeur sur la carte.
3. `set null` : fixer la valeur à `null`

Choix d'une option

À la déclaration d'une clé étrangère, on choisit parmi les 3 options, indépendamment pour la suppression et la modification.

Exemple

```
create table Carte (
    bar   varchar(20),
    bière varchar(20),
    prix  real,
    foreign key (bière)
        references Bière(nom)
        on delete set null
        on update cascade
);
```

En l'absence de spécification, le comportement par défaut (`default`) prévaut.

Contrainte de domaine

Elle s'applique sur les valeurs d'un attribut

- Ajouter `check (<condition>)` à la déclaration de l'attribut
- La condition peut utiliser le nom de l'attribut
- Tout·e autre attribut ou relation doit figurer dans une sous-requête

Exemple

```
create table Carte (
    bar   varchar(20),
    bière varchar(20)
        check (bière in (select nom from Bière)),
    prix  real
        check (prix <= 5.00)
);
```

Temps du `check`

La vérification a lieu seulement à l'insertion ou la modification d'un nuplet.

Exemple

- `check (prix <= 5.00)` teste chaque nouvelle valeur de prix et rejette la mise-à-jour (pour le nuplet incriminé seulement) si le prix est supérieur à 5
- `check (bière in (select nom from Bière))` n'est pas vérifiée si une bière est supprimée de la table `Bière`!

Contrainte de nuplet

La contrainte `check (<condition>)` peut être définie comme un élément du schéma.

- qui est exprimée à l'aide d'un ou plusieurs attribut(s) de la relation
- mais qui requiert une sous-requête pour toute référence à d'autres relations
- et qui est vérifiée à l'insertion et à la modification seulement.

Exemple

Seul le LAB[1] peut vendre de la bière à plus de 5€.

```
create table Carte (
    bar   varchar(20),
    bière varchar(20),
    prix  real,
    check (bar = 'LAB' OR prix <= 5.00)
);
```

1. Little Atlantique Brewery

Assertion

Une assertion est un élément du schéma de la base de données, comme une relation ou une vue.

```
create assertion <nom> check (<condition>);
```

La condition peut porter sur un élément quelconque du schéma de toute la base de données.

Exemple

Chaque bar doit proposer une carte dont la moyenne des prix ne dépasse pas 5€.

```
create assertion bonMarché check (
    not exists (select bar from Carte
                group by bar having 5.00 < AVG(prix))
);
```

Temps du `assertion`

En principe, la vérification a lieu à chaque mise-à-jour (`insert`, `update`, `delete`) d'une relation quelconque de la base de données.

Une analyse plus fine permettrait de s'apercevoir que nombre d'actions n'ont aucun impact sur l'assertion…

Exemple

Une mise-à-jour de `Bar` ou de `Bière` est sans conséquence sur l'assertion `bonMarché`.

En définitive, seule une mise-à-jour de `Carte` qui induit un nouveau prix supérieur à 5€ doit faire l'objet d'une vérification !

Les déclencheurs

Motivation

- Les assertions sont puissantes, néanmoins le SGBD ne comprend pas toujours très bien quand il est utile de les vérifier
- les `check` (contraintes de domaine et de nuplet), sont mieux réglés mais moins puissants
- les déclencheurs (*trigger* en anglais) offrent la possibilité de paramétrer l'événement déclencheur ainsi que de définir une condition complexe.

Règle Événement-Condition-Action

ECA : un autre nom pour les déclencheurs :

- Événement : le plus souvent un type de mise-à-jour, tel que « insérer dans `Carte` »
- Condition : toute expression booléenne en SQL
- Action : toute instruction SQL

Exemple liminaire

À la place de la clé étrangère sur `Carte.bière` qui rejette l'insertion d'un nuplet (`bar`, `bière`, `prix`) avec une bière inconnue, un déclencheur peut automatiquement ajouter la bière dans la table `Bière`, avec une valeur à `null` pour la brasserie.

```
create trigger BièreDec
    after insert on Carte   /* l'événement */
    referencing new row as nuplet for each row
    when (nuplet.bière not in   /* la condition */
        (select nom from Bière))
    insert into Bière(nom)      /* l'action */
            values (nuplet.bière);
```

Les options d'un déclencheur

La création

`create trigger <nom>` ou `create or replace trigger <nom>`

- utile pour modifier la définition d'un déclencheur existant

L'événement

- `after`, peut aussi être `before`
- ou encore `instead of` pour les vues seulement

L'action

- `insert`, peut être `delete` ou `update`
  - et `update` ou `update...on` un attribut spécifique

Les options (suite)

La portée

- `for each row` : exécuté à chaque mise-à-jour de nuplet
- par défaut : exécuté à chaque ordre SQL, quel que soit le nombre de nuplets mis-à-jour

Les variables `referencing`

- `insert` produit un nouveau nuplet (`for each row`) ou une nouvelle table (ensemble des nuplets-résultats de l'ordre SQL)
- `delete` suggère l'existence d'un·e (futur·e-ancien·ne) nuplet ou table
- `update` couvre les deux situations précédentes

```
[new | old] [tuple | table] as <nom>
```

Les options (fin)

La condition

- L'expression booléenne est évaluée sur l'image de la base de données telle qu'elle est avant (`before`) ou après (`after`) l'événement
- Mais **toujours** avant que la mise-à-jour prenne effet
- L'accès aux nouvelles et anciennes valeurs se fait via les variables de la clause `referencing`

Le bloc d'actions

- Il est possible de définir une suite d'ordres SQL, séparés par `;`
- Dans ce cas, le bloc débute par `begin` et termine par `end`
- Néanmoins, les requêtes `select-from-where` ne sont pas éligibles dans le bloc d'actions

Autre exemple de déclencheur

Maintenir à jour une liste de bars `ÀÉviter(bar)` qui ont augmentés leur prix de plus de 1€.

```
create trigger PrixDec
    /* l'événement: seulement une MàJ du prix */
    after update of prix on Carte
    referencing old row as ooo new row as nnn
    for each row
    when (nnn.prix > ooo.prix + 1.00)
    insert into ÀÉviter values (nnn.bar)
```

À retenir

`check`, `assertion` et `trigger` sont les mots-clés SQL pour déclarer des contraintes et des déclencheurs.

Les contraintes et déclencheurs en SQL permettent de formuler un grand nombre de propriétés à vérifier sur les données.

Une fois déclarés, ils sont automatiquement pris en charge par le SGBD.

C'est une économie pour le programmeur et une assurance que les données restent saines.

Procédures stockées (S8.1)

Pourquoi un langage de programmation ?

SQL n'est pas un langage de programmation (quoi que...)

- SQL est un langage pensé pour permettre l'interrogation d'une base de manière déclarative, et non procédurale
- Pas de variable, pas de boucle, pas de condition

Pour écrire des applications, SQL est associé à un langage de programmation

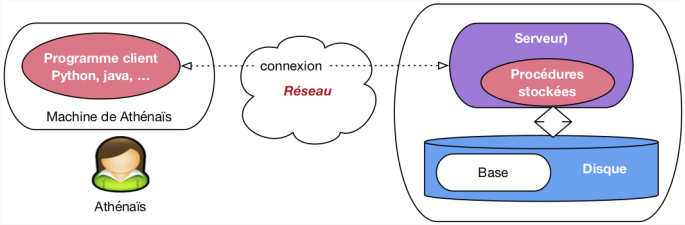
L'association :

- SQL pour les accès à la base
- Python, TypeScript, Go, Rust, Java, PHP, C++, Haskell, etc. pour tout le reste

Deux architectures possibles

Premier cas (majoritaire, à gauche) :

Les requêtes SQL sont intégrées à un programme client et transmises au serveur de données.



Second cas (à droite) :

Les requêtes SQL sont intégrées au serveur – via des procédures stockées – et induisent donc moins d'échanges réseaux, plus de sécurité.

Premier exemple, mode interactif (PL/SQL)

```
DECLARE                                                    -- Quelques variables
    v_nbFilms   INTEGER;
    v_nbArtistes INTEGER;

BEGIN
    SELECT COUNT(*) INTO v_nbFilms FROM Film;       -- nb films
    SELECT COUNT(*) INTO v_nbArtistes FROM Artiste; -- nb artistes

    -- Affichage des résultats
    DBMS_OUTPUT.PUT_LINE ('Nombre de films: ' || v_nbFilms);
    DBMS_OUTPUT.PUT_LINE ('Nombre d''artistes: ' || v_nbArtistes);
END;
```

Notez : les variables, la clause `into`.

Procédure stockée

```
CREATE OR REPLACE PROCEDURE InsereGenre (p_genre VARCHAR) AS
    v_genre_majuscules VARCHAR(20);
    v_count INTEGER;
BEGIN
    v_genre_majuscules := UPPER(p_genre);       -- en majuscule

    SELECT COUNT(*) INTO v_count
    FROM Genre WHERE code = v_genre_majuscules; -- existe ?

    IF (v_count = 0) THEN                       -- insertion cond.
        INSERT INTO Genre (code) VALUES (v_genre_majuscules);
    END IF;
END;
```

Notez : la condition.

Second exemple, fonction et itérateur

```
CREATE OR REPLACE FUNCTION MesActeurs(v_idFilm INTEGER) RETURN VARCHAR IS
    resultat VARCHAR(255);
BEGIN
    FOR art IN (SELECT Artiste.* FROM Role, Artiste
                WHERE idFilm = v_idFilm  AND idActeur=idArtiste)
    LOOP                                    -- boucle sur les acteurs du film
        IF (resultat IS NOT NULL) THEN
            resultat := resultat || ', ' || art.prenom || ' ' || art.nom;
        ELSE
            resultat := art.prenom || ' ' || art.nom;
        END IF;
    END LOOP;
    return resultat;
END;
```

Notez : le type automatique de `art`, la boucle

Utilisation

- En ligne de commande interactive

```
SQL> start StatsFilms.sql
```

- Avec l'ordre `execute`, placé dans un autre langage (C, Java, PHP, etc.)

```
execute insereGenre('Policier')
```

- Dans une requête SQL

```
SELECT titre, MesActeurs(idFilm)
FROM Film WHERE idFilm=5;

TITRE        MESACTEURS(IDFILM)
------------ ----------------------------
Volte/Face   John Travolta, Nicolas Cage
```

Dérivation de types depuis le schéma

Deux exemples pour illustrer.

- `Film.titre%TYPE` est le type de l'attribut `titre` de la table `Film`;
- `Artiste%ROWTYPE` est un type `RECORD` correspondant aux attributs de la table `Artiste`.

Beaucoup plus difficile avec un langage de programmation externe.

À retenir

Principales difficultés SQL/programmation : typage et conversion

Typage et conversion des nuplets

- Définir des variables du langage (Python, Java) correspondant aux types SQL
- Convertir le résultat d'une requête en variables du langage

Typage et conversion des tables

- Ce sont des ensembles – ou séquences si `order by` –, on doit les parcourir avec des boucles.