SQL Comme langage de requêtes

Contenu du chapitre:

- SQL: propriétés générales
- SQL: Langage de Définition de Données
- SQL: Langage de Manipulation des Données
- Opérations ensemblistes
- Sous-requêtes
- Jointures
- Agrégats
- Quelques outils techniques

- ✎ Découvrir la syntaxe de SQL
- ✎ Savoir écrire des requêtes simples
- ✎ Connaissances de bases sur jointures et agrégats
- ✎ Créer& interroger une base de donnée sous PostgreSQL

Table des matières

SQL: Introduction

SQL = Structured Query Language

Version originelle (ancêtre) SEQUEL: Boyce et Chamberlain (IBM).
Norme ANSI *(American National Standards Institute)*
puis standard ISO *(International Organisation for Standardization)*.

Avantages:

- ✔ langage standard d'interrogation: supporté (en partie) par principaux SGBDs
- ✔ portabilité des applications, interopérabilité

Inconvénients ::

- ✘ évolution du langage par "extensions"
  SQL1 : 1989, SQL2 : 1992, SQL3 : 1999
  SQL:2003, SQL:2008 ...
- ✘ frein à l'émergence d'un nouveau langage
- ✘ style de programmation désuet

La casse en SQL

Mots-clefs (`SELECT, FROM...`) insensibles mais souvent majuscule

Nom de table, colonne généralement insensibles, peuvent être sensible
(option paramétrable, valeur défaut dépend du SGBD/de l'OS...)

Valeurs des chaines de caractères sensibles: 'aa'$\neq$ 'Aa'
(apparaissent typiquement dans conditions `WHERE, HAVING`)

Types de données en SQL

- chaînes de caractères de taille fixe, complétées à droite par des espaces: `CHAR(taille)`
- chaînes de taille variable: `VARCHAR(taille_max)`
- entiers sur 32 bits: `INTEGER`
- nombres en précision fixe (nb chiffres dont nbDecimales après virgule): `NUMERIC(nb, nbDecimales)`
- nombres virgule flottante: `REAL`/`DOUBLE PRECISION`).
- types date/heure: `DATE`/`TIME`/`TIMESTAMP`

NULL en SQL

`NULL` représente une valeur inconnue.

3 valeurs possibles pour expressions booléennes SQL:

`Vrai, Faux, Inconnu`.

- 2=2: `Vrai`
- 2=NULL: `Inconnu`
- NULL= NULL: `Inconnu`

SQL utilise *logique à 3 valeurs* pour évaluer conditions:

- NOT `Inconnu`= `Inconnu`
- OR retourne `Vrai` si un des arguments est `Vrai`, `Faux` si les deux arguments sont `Faux`, `Inconnu` sinon
- AND retourne `Vrai` si les deux arguments sont `Vrai`...

Pour tester si un attribut est défini: `attr IS NULL/IS NOT NULL`

Les expressions arithmétiques mettant en jeu un null s'évaluent à null:

- 2+NULL renvoie NULL

NULL in SQL: traps

Réécrire les requêtes suivantes

SELECT * FROM t WHERE (name <> 'Jean') OR name='Jean';
SELECT * FROM t WHERE name = name;

Table des matières

SQL: LDD

Création de schéma

plus exactement, création d'une table vide.

```
CREATE TABLE Film (
Titre CHAR(20),
MeS CHAR(20),
Acteur VARCHAR(20))
```

Contraintes

- NOT NULL
- valeurs uniques
- clefs primaires
- clefs étrangères
- condition de valeurs (check)

http://www.postgresql.org/docs/9.0/static/ddl-constraints.html

Contraintes: NOT NULL

```
CREATE TABLE Film (
Titre CHAR(20) NOT NULL,
Acteur VARCHAR(20),
Realisateur CHAR(20) NOT NULL DEFAULT 'Kurosawa',
)
```

Film

TITRE	ACTEUR	REALISATEUR
Les petits mouchoirs	Cotillard	Canet
La plage	Canet	
Mon Idole	Canet	Canet
Bienvenus chez les ch'tis	Boon	Boon

↪ Instance impossible car la contrainte sur réalisateur n'est pas respectée.

Contraintes: clef /clef primaire

créent un index implicite.

```
CREATE TABLE Prog (
Nom_Cine CHAR(20),
Titre CHAR(20),
Salle INT,
Horaire TIME,
UNIQUE (Nom_Cine,Titre, Horaire))
```

Deux lignes ne peuvent pas avoir la même valeur de clef (sauf NULL)

```
CREATE TABLE Cine (
Nom_Cine CHAR(20) PRIMARY KEY,
Adresse  VARCHAR(60),
Telephone CHAR(8))
```

Clef primaire=UNIQUE+NOT NULL
1 seule par table

```
CREATE TABLE Cine (
NomCine CHAR(20),
Adresse  VARCHAR(60),
Telephone CHAR(8),
CONSTRAINT pk_nc PRIMARY KEY (Nom_Cine))
```

http://troels.arvin.dk/db/rdbms/#constraints-unique

Contraintes: clef étrangère

La clef étrangère doit être clé primaire (ou unique) dans la table référencée.

```
CREATE TABLE Prog (
Nom_Cine CHAR(20) REFERENCES Cine(Nom_Cine),
Titre CHAR(20),
Salle INT,
Horaire TIME,
CONSTRAINT fk_titre FOREIGN KEY (Titre) REFERENCES Film(Titre))
```

Chaque titre de film au programme doit apparaître dans une des lignes de la table `Film`.

⚠ Vérifié à chaque modification de `Prog` ET chaque modification de `Film`

Récapitulatif des contraintes d'intégrité

Example

Considérons les contraintes :

```
-- dans Cine:
Telephone CHAR(8) NOT NULL
CONSTRAINT pk_nc PRIMARY KEY (Nom_Cine))
-- dans Prog:
CONSTRAINT fk_cine FOREIGN KEY (Nom_Cine) REFERENCES Cine(Nom_Cine)
UNIQUE (Nom_Cine,Titre, Horaire)
```

Les tables ci-dessous violent des contraintes (donc instance impossible) :

Cine

NOM_CINE	ADRESSE	TELEPHONE
ugc bercy	3 rue...	06043494
le champo	17 av...	01049059
ugc bercy	18 bd...	

Prog

NOM_CINE	TITRE	SALLE	HORAIRE
ugc bercy	Le discours d'un roi	1	20h00
ugc bercy	Le discours d'un roi	3	20h00
ugc bercy	Le discours d'un roi	2	22h00
ugc bercy	Inception	1	14h00
le champo	Le discours d'un roi	1	18h00
le campo	Inception	1	20h00

N.B. les symboles (✎,☆,...) n'ont rien de standard: je les utilise juste ici comme légende des violations

Contraintes: restriction du domaine

Évalue une condition; retourne une erreur si le résultat est `Faux`:

```
CREATE TABLE Prog (
Nom_Cine CHAR(20),
Titre CHAR(20),
Salle INT NOT NULL CHECK (1<=salle AND salle<10),
Horaire TIME)
```

La condition peut porter sur plusieurs colonnes de la table, faire appel à une requête SQL sur une autre table, etc.:

```
CREATE TABLE Prog (
...,
CHECK ('France'= SELECT pays FROM Cine WHERE Prog.Nom_Cine=Cine.nom))
```

 Vérifié à chaque modification de `Prog`, mais pas de `Cine`.

Modifier/Détruire des tables (au niveau du schéma)

```
DROP TABLE Prog
-- Erreur si d'autres objets dépendent de la table
```

En présence de clefs étrangères, respecter l'ordre pour détruire les tables, ou bien:

```
DROP TABLE Prog CASCADE
-- Détruit aussi les objets faisant référence à Prog
```

```
ALTER TABLE Prog ADD COLUMN jour: DATE
```

Table des matières

LMD

Langage de Manipulation de Données

( DML: Data Manipulation Language)

- `SELECT` : pour interroger (cf chap 4)
- `INSERT` : pour insérer des tuples dans une table
- `UPDATE` : pour modifier des données dans une table
- `DELETE` : pour supprimer des tuples dans une table

L'instance de la base est (définitivement) modifiée

. . . pourvu que l'on ait fait un "commit".

 Les contraintes d'intégrité peuvent empêcher `INSERT`, `UPDATE`, `DELETE`.

UPDATE

Mises à jour

```sql
UPDATE Prog
SET Heure = '21h00'
WHERE Nom_Cine LIKE 'ugc%' AND Titre= 'Dersou Ouzala';

-- Autres expressions possibles:
UPDATE Prog SET Heure = Heure+1; -- les séances seront retardées
UPDATE Produits SET PrixTTC = PrixHT * TVA;
```

Example

Prog avant

```
NOM_CINE   TITRE          SALLE HEURE
---------- ------------- ----- -----
ugc bercy  Dersou Ouzala  1    20h00
ugc bercy  Dersou Ouzala  2    22h00
ugc bercy  Kagemusha      1    14h00
le champo  Dersou Ouzala  1    18h00
le champo  Kagemusha      1    20h00
```

Prog après mise à jour

```
NOM_CINE   TITRE          SALLE HEURE
```

INSERT

Insertions

```sql
INSERT INTO Prog (Nom_Cine, Titre, Salle, Horaire)
VALUES ('ugc', 'Dersou Ouzala', 1, '20h00');
-- ne pas oublier les ' ' pour les chaînes de caractères
```

Préciser les noms de colonnes n'est pas nécessaire ici, mais permet en général

- d'insérer la valeur par défaut (NULL sauf mention explicite d'un autre DEFAULT) pour les colonnes non spécifiées.
- de donner les valeurs dans le désordre.

Selon le SGBD, on trouve des commandes mixtes Update-insert

```sql
INSERT INTO produit(id,prix) VALUES (1,10.5)
    ON CONFLICT (id) DO UPDATE SET prix = 10.5; -- PostgreSQL
...
    ON DUPLICATE KEY UPDATE prix=10.5; -- MariaDB
```

DELETE

Suppressions

```sql
DELETE FROM Prog WHERE Heure >= '20h00';

DELETE FROM Prog
WHERE Titre in (SELECT Titre FROM Film WHERE LOWER(Acteur) Like 'Jean-%');

-- Comportements possible en cas de Foreign Key: échec ou cascade.
```

Pour vider complètement une table, TRUNCATE est plus rapide.

```sql
TRUNCATE Prog;
```

Ne parcourt pas la table. À utiliser avec prudence !

(pas d'entrée dans le journal⇒ perturbe MVCC, ne déclenche pas les triggers).

Langage de requête

syntaxe d'une requête SQL (simplifiée)

SELECT *<liste d'attributs>*	attributs du schéma cible $\leftrightarrow \pi$
FROM *<liste de relations>*	relations du schéma source
WHERE *<condition>*	conditions de sélection $\leftrightarrow \sigma$

Clause `WHERE` optionnelle. Garde les tuples évaluant condition à "`Vrai`".
Clause `FROM` aussi optionnelle pour PostgreSQL (pas Oracle: `FROM DUAL`)

```
SELECT Nom_Cine
FROM Prog
WHERE Titre='Marion'
```

```
SELECT 2+3
```

 Les doublons ne sont pas éliminés

SQL: requêtes simples

Select Project Join (SPJ)

```
SELECT * FROM FILM WHERE Acteur='Lonsdale'
```

$*$: liste de tous les attributs

$\leftrightarrow \sigma_{Acteur='Lonsdale'}(FILM)$

```
SELECT Titre
FROM FILM
WHERE Acteur='Lonsdale' OR Acteur='Astaire'
```

$\leftrightarrow \pi_{titre}(\sigma_{Acteur='Lonsdale' \vee Acteur='Astaire'}(FILM))$

Sémantique formelle (cas mono-relation):

SELECT $A_1, \ldots, A_k$
FROM R
WHERE C

$\leftrightarrow \quad \pi_{A_1,\ldots,A_k}\sigma_C(R)$

Illustration de requête SQL: cas mono-relation

Example

Prog

NOM_CINE	TITRE	SALLE	HORAIRE
ugc bercy	Le discours d'un roi	1	20h00
ugc bercy	Le discours d'un roi	2	22h00
ugc bercy	Inception	1	14h00
le champo	Le discours d'un roi	1	18h00
le champo	Inception	1	20h00

```
SELECT Nom_cine, Titre
FROM Prog
WHERE Nom_cine = 'ugc bercy'
    AND HORAIRE < 23h00
    AND HORAIRE > 10h00
```

NOM_CINE	TITRE
ugc bercy	Le discours d'un roi
ugc bercy	Le discours d'un roi
ugc bercy	Inception

SQL: requête simple

Select Project Join (SPJ)

```
SELECT Nom_Cine, Film.Titre, Horaire
FROM Film, Prog
WHERE Film.Titre = Prog.Titre
    AND Acteur = 'M.Freeman'
```

$\leftrightarrow \pi_{Nom\_Cine, Film.titre, Horaire}(\sigma_{Acteur='M.Freeman'}(Film \bowtie Prog))$

Sémantique formelle (cas multi-relation):

SELECT $A_1, \ldots, A_k$
FROM $R_1, \ldots, R_p$
WHERE C

$\leftrightarrow$ $\pi^*_{A_1,\ldots,A_k}(\sigma_C(R_1^* \times \cdots \times R_p^*))$

-projection multiensembliste
-tous attributs distincts

SQL: requête simple

Select Project Join (SPJ): ambiguïté sur la table d'un attribut

```
SELECT Nom_Cine, Film.Titre, Horaire
FROM Film, Prog
WHERE Film.Titre = Prog.Titre
    AND Acteur = 'M.Freeman'
```

- Table ajoutée automatiquement par le système pour chaque attribut
- Inférence impossible si l'attribut apparait dans plusieurs tables: en ce cas l'utilisateur doit impérativement spécifier la table.
  ↪ *sinon erreur*

Illustration de requête SQL: cas multi-relation

Example

Prog

NOM_CINE	TITRE	SALLE	HORAIRE
ugc bercy	Le discours d'un roi	1	20h00
ugc bercy	Le discours d'un roi	2	22h00
ugc bercy	Inception	1	14h00
le champo	Le discours d'un roi	1	18h00
le champo	Inception	1	20h00

Cine

NOM_CINE	ADRESSE	TELEPHONE
ugc bercy	3 rue...	06043494
le champo	17 av...	01049059
nef chava	18 bd...	04387953

```
SELECT Telephone, Horaire
FROM Prog, Cine
WHERE Prog.Nom_cine = Cine.Nom_cine
    AND Titre = 'Inception'
```

TELEPHONE	HORAIRE
06043494	14h00
01049059	20h00

Renommage d'attributs

Attr `AS` Attr_alias

Intérêt:

- nommer une colonne résultant d'un calcul,
- renommer une colonne de façon plus explicite
- permettre union...

Toutes les personnes ayant participées au tournage du film "Marion":

```
SELECT Acteur AS Personne FROM Film WHERE Titre = 'Marion'
UNION
SELECT MeS AS Personne FROM Film WHERE Titre = 'Marion'
```

Renommage de table

Table Table_alias

Intérêt:

- distinger plusieurs copies d'une même table
- donner un nom à une table résultant d'un calcul intermédiaire (sous requête) ...

Les films avec leur MeS et leurs acteurs dans lesquels joue M-F Pisier ?

$\pi_{\text{Titre}}(\sigma_{\text{Actrice}=\text{M}-\text{F.Pisier}}(\textbf{film})) \bowtie \textbf{film}$

```
SELECT F2.Titre , F2.MeS, F2.Acteur
FROM  FILM  F1, FILM  F2
WHERE F1.Titre = F2.Titre  AND
    F1.Acteur = 'M-F. Pisier'
```

Interprétations de F1 et F2:

- F1 et F2 sont des copies *virtuelles* de **film**
- F1 et F2 sont des variables utilisées pour désigner n'importe quel couple de n-uplets de **film**

Élimination des doublons

SELECT DISTINCT

Liste des films projetés dans chaque cinéma:

```
SELECT Nom_Cine, Titre
FROM Prog
```

```
NOM_CINE   TITRE
---------- --------------------
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Inception
le champo  Le discours d'un roi
le champo  Inception
```

```
SELECT DISTINCT Nom_Cine, Titre
FROM Prog
```

```
NOM_CINE   TITRE
---------- --------------------
ugc bercy  Le discours d'un roi
ugc bercy  Inception
le champo  Le discours d'un roi
le champo  Inception
```

Opérations sur les attributs

Dans la clause SELECT ou dans les conditions une expression peut être:

- une constante
- un attribut
- *expr1* * *expr2* (ou +,-,/)
- une opération sur les chaînes de caractères
  - ↪ `ch1 || ch2` : concaténation
  - ↪ `LOWER(ch)` : met en minuscules
  - ↪ `UPPER(ch)` : met en majuscules
  - ↪ `SUBSTR(ch,i,j)` : extrait la sous chaîne de longueur *j* débutant à l'indice *i*
- `CASE WHEN` condition `THEN` expr `ELSE` expr `END`
- ...

```sql
SELECT 3*montant FROM Ventes;
SELECT SUBSTR('ABCDEFG',3,4) FROM DUAL;  -- (oracle): 'CDEF'
SELECT substring('Thomas' from 2 for 3); -- (postgresql): 'hom'
```

Contenu de la clause WHERE

- *attr* `IS NULL`
- *expr* `op` *expr* $\quad$ `op`$\in (=, <, <=, >=, <>)$
- *expr* `BETWEEN` *val1* `AND` *val2*
- *expr* `IN` *(val1,val2,...)*
- *expr* `LIKE` *string-pattern*
  - ↪ `%` : n'importe quelle chaîne de caractère
  - ↪ `_` : n'importe quel caractère

`Films`

```
TITRE                      ACTEUR
-------------------------  ------
Rien à Declarer            Boon
La plage                   Canet
Bienvenus chez les ch'tis  Boon
```

```
SELECT Titre
FROM Films
WHERE LOWER(Titre) LIKE '%bien%'
    OR Titre LIKE '%declarer%'
--LOWER: convertir en minuscule
```

```
TITRE
---------------------------
Bienvenus chez les ch'tis
```

Piège 1: produit cartésien

Sémantique de la requête ci-dessous $\stackrel{?}{=} R.A \cap (S.A \cup T.A)$?

```
SELECT R.A
FROM R,S,T
WHERE R.A=S.A OR R.A=T.A
```

Piège 2: NULLs

Sémantique de la requête ci-dessous?

```
SELECT P.name
FROM Personnes P
WHERE P.age > 10 OR P.age <= 10
```

Trier le résultat de la requête

Relation = multi-ensemble $\Longrightarrow$ ordre d'affichage dans SQL arbitraire si il n'est pas spécifié.

Liste des films projetés dans chaque cinéma triés par titre:

```
SELECT Nom_Cine, Titre
FROM Prog
ORDER BY Titre
```

```
NOM_CINE   TITRE
---------- --------------------
ugc bercy  Inception
le champo  Inception
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
le champo  Le discours d'un roi
```

```
SELECT Nom_Cine, Titre
FROM Prog
ORDER BY Titre DESC, Nom_Cine ASC
```

```
NOM_CINE   TITRE
---------- --------------------
le champo  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
ugc bercy  Le discours d'un roi
le champo  Inception
ugc bercy  Inception
```

- tri croissant ( ASCending) par défaut.
- ordre naturel de l'attribut dépend de son type (entier, chaîne de caractères...)
- tri sur plusieurs colonnes: lexicographique

Table des matières

Opérations ensemblistes

Union, Intersection, Différence

- opérations ensemblistes: éliminent par défaut les doublons
- les relations doivent avoir même schéma
- ajouter `ALL` pour garder les doublons

Les personnes ayant travaillé sur le film 'Marion':

```
SELECT Acteur AS Personne FROM Film WHERE Titre = 'Marion'
UNION
SELECT MeS AS Personne FROM Film WHERE Titre = 'Marion'
```

$$\leftrightarrow \rho_{MeS \to Personne}(\pi_{MeS}(\sigma_{Titre='Marion'}(Film))) \cup \rho_{Acteur \to Personne}(\pi_{Acteur}(\sigma_{Titre='Marion'}(Film)))$$

Les personnes ayant travaillé sur le film 'Marion', comptées 1 fois par rôle:

```
SELECT Acteur AS Personne FROM Film WHERE Titre = 'Marion'
UNION ALL
SELECT MeS AS Personne FROM Film WHERE Titre = 'Marion'
```

Opérations ensemblistes (2)

Union, Intersection, Différence

Les titres des films à l'affiche dans lesquels a joué M-F Pisier:

```
SELECT Titre FROM Film WHERE Acteur = 'M-F. Pisier'
    INTERSECT
SELECT Titre FROM Prog
ORDER BY Titre
```

Les titres des films qui ne sont pas à l'affiche:

```
SELECT Titre FROM Film
    EXCEPT
SELECT Titre FROM Prog
```

Remarque: MySQL supporte seulement UNION. Certains SGBD utilisent `MINUS` au lieu de `EXCEPT`.

Table des matières

Sous-requête

Principe: utiliser le résultat d'une requête comme relation ou comme valeur dans une autre requête. Une sous-requête peut apparaître dans les clauses

- WHERE et HAVING
- FROM (en renommant la relation résultante)
- SELECT (à condition d ne retourner qu'une ligne pour chaque ligne de la requête principale)

Supposons que `FILM-DEB(Titre,Acteur)` stocke le titre du premier film de chaque acteur.

Les acteurs du premier film joué par M-F. Pisier:

```
SELECT Acteur FROM  FILM
WHERE Titre = (SELECT Titre FROM FILM-DEB
    WHERE Acteur = 'M-F. Pisier')
```

 La sous-requête doit retourner *exactement un* résultat sinon erreur à l'exécution de la requête.

Sous-requête

Sous-requête dans le FROM

Les acteurs ayant joué avec M-F.Pisier dans le même film:

```
SELECT Acteur
FROM  FILM,  (SELECT Titre
    FROM Film Where Acteur = 'M-F. Pisier') F2
WHERE Film.Titre = F2.titre
```

Sous-requête avec opérateur

Sous-requête à plusieurs lignes dans la clause `WHERE/HAVING`

Opérateurs exprimant des conditions avec des sous requêtes pouvant retourner plusieurs lignes:

- *attr* `IN` *sous-requête*
  vrai si la valeur apparaît dans le résultat de la sous-requête
- `EXISTS` *sous-requête*
  vrai si le résultat de la sous-requête contient au moins un tuple
- *attr* `op ANY` *sous-requête*
  vrai si il existe une valeur dans le résultat de la sous-requête qui satisfait la comparaison
- *attr* `op ALL` *sous-requête*
  vrai si toutes les valeurs dans le résultat de la sous-requête satisfont la comparaison

avec op$\in \{=, <, >, <=, >=\neq\}$

aussi : `NOT IN, NOT EXISTS`

⚠ Ne pas en abuser: optimiseur souvent inefficace

Sous-requêtes

Opérateur IN

Les titres des films dont un des MeS est acteur (pas forcément dans le même film):

```
SELECT DISTINCT Titre
FROM  FILM
WHERE MeS IN (SELECT Acteur FROM FILM)
```

Formulation équivalente sans sous-requête?

Sous-requêtes

Opérateur EXISTS

Les films dirigés par au moins deux metteurs en scène:

```
SELECT DISTINCT F1.Titre
FROM  FILM F1
WHERE EXISTS
    (SELECT F2.MeS
    FROM FILM F2
    WHERE F1.Titre=F2.Titre AND NOT F1.MeS=F2.MeS)
```

Remarque: on a ici une requête *corrélée*.

Formulation équivalente sans sous-requête?

Sous-requêtes

Comparaison avec ALL

Les films projetés à l'UGC plus tard que tous les films projetés au Trianon.

```
SELECT DISTINCT Titre
FROM  PROG
WHERE Nom_Cine='UGC' AND
Horaire > ALL
    (SELECT  Horaire
    FROM PROG
    WHERE Nom_Cine='Trianon')
```

Formulation équivalente sans sous-requête??

Sous-requêtes

Comparaison avec ANY

Le téléphone des cinémas proposant une programmation après 23h:

```
SELECT Telephone FROM CINE AS C1
WHERE 23< ANY (
    SELECT Horaire FROM PROG
    WHERE C1.Nom_Cine=PROG.Nom_Cine)
```

Sous-requêtes

Précisions sur la sémantique de IN/NOT IN/=ANY

attr `IN` {*x*,*y*,*z*} ↔ *attr*=x OR *attr*=y OR *attr*=z

attr `NOT IN` {*x*,*y*,*z*} ↔ *attr* != x AND *attr* != y AND *attr* != z

⚠ résultat en présence de NULLs peut surprendre. . .

Est-ce que l'opérateur `IN` est équivalent à `=ANY`? Pouvez vous exprimer `NOT IN` en fonction de ALL?

Table des matières

Jointures en SQL

Différents types de jointure

Rappel:

- Produit cartésien:

```
SELECT R.a,S.b FROM R,S
```

- Jointure:

```
SELECT R.a,S.b FROM R,S WHERE R.c=S.c
```

On peut utiliser d'autres types de jointures!

equi-jointure : condition de jointure est une égalité

↪ *c'est le cas le plus courant: ± tous les examples de ce cours.*

non equi-jointure : condition de jointure n'est pas une égalité:

$>, \ldots,$ LIKE

Très souvent lorsque l'on parle de jointures on parle des équijointures.
Même pour des équijointures, il existe plusieurs formulations en SQL.

Jointures en SQL

Différents types de jointure en SQL

Instructions SQL pour spécifier le type de jointure :

- jointures internes : `(INNER) JOIN`
- jointures externes : `LEFT/RIGHT /FULL(OUTER) JOIN`
- produit cartésien : `CROSS JOIN`

Instructions SQL pour spécifier les conditions de jointure :

- ... `JOIN` ... `ON`: le plus général, condition quelconque (pas forcément equi-jointure), colonnes non fusionnées.
- ... `JOIN` ... `USING` : jointure naturelle restreinte aux attributs listées, colonnes fusionnées.
- `NATURAL JOIN` ... : jointure naturelle sur toutes les colonnes de même nom.

Syntaxe jugée plus lisible qu'un produit cartésien suivi d'une sélection dans la clause WHERE, et (à peu près) indispensable pour les jointures externes.

Jointures en SQL

Jointure interne

```
SELECT F1.Titre, F2.Acteur
FROM  FILM  F1 INNER JOIN FILM  F2
ON F1.Titre = F2.Titre
WHERE F1.Acteur = 'M-F. Pisier'
```

Le mot-clé `INNER` est facultatif : lorsque l'on écrit une jointure sans en préciser le type(`JOIN`) c'est bien `INNER JOIN` qui est calculé.

Jointures en SQL : plusieurs tables

Jointure interne

```
SELECT Film.Titre, Pers.nom, Pers.taille
FROM  Film
JOIN Programme Prog ON Film.Titre = Programme.Titre
JOIN Personne Pers ON Film.acteur = Pers.nom
```

Pourquoi la requête ci-dessous ne calcule-t-elle pas les triplets d'acteurs distincts qui ont collaboré sur un film?

```
SELECT F1.Titre, F1.Acteur, F2.Acteur, F3.Acteur
FROM  FILM  F1
JOIN FILM  F2 ON F1.Titre = F2.Titre
JOIN FILM  F3 ON F1.Titre = F3.Titre
WHERE F1.Acteur != F2.Acteur AND F1.Acteur != F3.Acteur
```

Jointures en SQL

Jointure naturelle

```sql
SELECT Nom_Cine,Film.Titre, Horaire
FROM Film NATURAL JOIN Prog
WHERE Acteur = 'M.Freeman'
```

Jointure naturelle sur tous les attributs communs.

On peut restreindre la jointure naturelle à un sous ensemble $(attr\_a, attr\_b, \ldots, attr\_c)$ des attributs communs avec la syntaxe:

```sql
-- préciser les attributs d'une jointure naturelle est recommandé
-- (par ex: le schéma risque de changer par la suite)
SELECT attr_a, attr_b, nom_table1.attr1...
FROM nom_table1 INNER JOIN nom_table2
USING(attr_a,attr_b,...,attr_c)
```

Jointures en SQL

Jointure externe

```
SELECT attr1, attr2,...
FROM nom_table1 LEFT OUTER JOIN nom_table2
USING(attr_a,attr_b,...,attr_c)
```

Permet d'afficher additionnellement les lignes de la table gauche (à gauche du mot clé JOIN) qui n'ont pas de ligne correspondante à droite pour la jointure, en complétant par des `NULL` à droite.

`RIGHT OUTER JOIN`: affiche les lignes de la table droite sans correspondance
`FULL OUTER JOIN`: affiche les lignes des tables gauche ou droite sans correspondance

Jointures en SQL

Jointure externe

Roles

```
TITRE                     ID_ACT PERSONNAGE
------------------------- ------ ---------------
It's a Wonderful Life         35 George Bailey
It's a Wonderful Life            Mary Bailey
It's a Wonderful Life         36 Clarence Odbody
Rear Window                   35 LB Jeffries
The Shawshank Redemption      40 red
```

Acteur

```
  ID NOM         PRENOM
--- ----------- -------
  35 Stewart     James
  40 Freeman     Morgan
  50 Serrault    Michel
```

```
SELECT titre, personnage, nom
FROM roles LEFT OUTER JOIN acteurs ON id_act=id;
```

```
TITRE                     PERSONNAGE      NOM
------------------------- --------------- --------
Rear Window               LB Jeffries     Stewart
It's a Wonderful Life     George Bailey   Stewart
The Shawshank Redemption  red             Freeman
It's a Wonderful Life     Mary Bailey
It's a Wonderful Life     Clarence Odbody
```

Jointures en SQL

Jointure externe

Roles

```
TITRE                       ID_ACT PERSONNAGE
--------------------------- ------ --------------
It's a Wonderful Life           35 George Bailey
It's a Wonderful Life              Mary Bailey
It's a Wonderful Life           36 Clarence Odbody
Rear Window                     35 LB Jeffries
The Shawshank Redemption        40 red
```

Acteur

```
 ID NOM        PRENOM
--- ---------- -------
 35 Stewart    James
 40 Freeman    Morgan
 50 Serrault   Michel
```

```
SELECT titre, personnage, nom
FROM roles FULL OUTER JOIN acteurs ON id_act=id;
```

```
TITRE                       PERSONNAGE      NOM
--------------------------- --------------- --------
It's a Wonderful Life       George Bailey   Stewart
Rear Window                 LB Jeffries     Stewart
The Shawshank Redemption    red             Freeman
It's a Wonderful Life       Mary Bailey
It's a Wonderful Life       Clarence Odbody
                                            Serrault
```

in both

in LEFT OUTER JOIN

in RIGHT OUTER JOIN

Table des matières

Fonctions d'agrégation

(Aggregates)

But: résumer les données en regroupant les tuples.

Exemple: calculer le nombre de film pour chaque réalisateur.

SELECT *<liste d'attributs>*	attributs du schéma cible
FROM *<liste de relations>*	relations du schéma source
WHERE *<condition>*	conditions de sélection
GROUP BY *<liste d'attributs>*	attributs de regroupement
ORDER BY *<liste d'attributs>*	ordre d'affichage

Fonctions les plus courantes:

- `COUNT()`: cardinalité du multiensemble
- `SUM()`: somme
- `MAX/MIN()`: valeur max/min
- `AVG()`: moyenne

Fonctions d'agrégation: principe du regroupement

Film

TITRE	ACTEUR
Rien à Declarer	Boon
Brice de Nice	Cornillac
La vie de Chantier	Boon
Ensemble c'est tout	Canet
Mon Idole	Canet
The dark knight rises	Cotillard
Les petits mouchoirs	Cotillard
La plage	Canet
Inception	Cotillard
Bienvenus chez les ch'tis	Boon

```
SELECT acteur, COUNT(*)
FROM film
GROUP BY acteur
```

ACTEUR	COUNT(*)
Canet	3
Cotillard	3
Boon	3
Cornillac	1

Les tuples ayant même valeur de groupement sont regroupés sur une ligne

- la fonction est calculée pour chaque groupe indépendamment
- si fonction d'agrégation sans `GROUP BY`: 1 groupe=table entière
- les attributs hors du `GROUP BY` ne peuvent apparaître dans `SELECT` qu'à l'intérieur d'une fonction d'agrégation

Sinon: erreur "`not a group by expression`".

Fonctions d'agrégation (2)

Film

```
TITRE                ACTEUR QT
-------------------- -------- --
Rien à Declarer      Boon 2
Brice de Nice        Cornillac 5
La vie de Chantier   Boon 7
Ensemble c'est tout  Canet 2
Mon Idole            Canet 3
```

Exemple de requête avec agrégat sans GROUP BY...

Attention: la clause WHERE filtre les données *avant* la création des groupes.

```sql
SELECT COUNT(*) c, SUM(QT) s
FROM film
```

```
 c | s
---+---
 5 | 19
```

```sql
SELECT
 SUBSTR(MAX(Titre),0,4) mx,
 SUM(QT)/COUNT(*) s
FROM t
WHERE Titre>'L'
```

```
 mx  | s
-----+---
 Rie | 4
```

Fonctions d'agrégation : ordre

Ordre d'évaluation intuitif des clauses:

1. FROM
2. WHERE
3. GROUP BY
4. HAVING
5. SELECT
6. ORDER BY

Donner le résultat de la requête:

```
SELECT acteur, nomrealisateur, COUNT(*) NB
FROM film
WHERE acteur < 'D'
GROUP BY acteur, nomrealisateur
ORDER BY NB
```

Film

TITRE	ACTEUR	NOMREALISATEUR
Les petits mouchoirs	Cotillard	Canet
La plage	Canet	Boyle
Mon Idole	Canet	Canet
The dark knight rises	Cotillard	Nolan
La vie de Chantier	Boon	Boon
Brice de Nice	Dujardin	Dujardin
Ensemble c'est tout	Canet	Canet
Gran Torino	Eastwood	Eastwood
Inception	Cotillard	Nolan
Brice de Nice	Cornillac	Dujardin
Rien à Declarer	Boon	Boon

COUNT

Que compter?

`COUNT(*)` retourne le nombre de lignes.
`COUNT(a)` retourne le nombre de lignes non nulles sur colonne *a*.
`COUNT(DISTINCT a)` retourne le nombre de valeurs *distinctes* sur la colonne *a*.

Film

titre	acteur	nomrealisateur
Green Book	Mortensen	Farrelly
Seigneur des Anneaux	Mortensen	Jackson
Coda	Jones	Heder
Nomadland		Zhao

```sql
SELECT COUNT(*) A,
 COUNT(acteur) B,
 COUNT(DISTINCT acteur) C
FROM film
WHERE Titre != 'Coda'
```

a	b	c
3	2	1

Compter en SQL les acteurs que chaque réalisateur a encadré :

Autres fonctions d'agrégation

`Nb_Films_Joués`

```
ACTEUR       NB_FILMS
---------- ----------
Canet               3
Cotillard           3
Boon                3
Cornillac           1
```

Calculer dans une même requête le nombre maximal et moyen de films par acteur.

Résultat attendu:

```
    NB_MAX     NB_MOY
---------- ----------
         3        2.5
```

Conditions sur les groupes

HAVING

SELECT *<liste d'expressions1>*	attributs du schéma cible
FROM *<liste de relations>*	relations du schéma source
WHERE *<conditions1>*	conditions de sélection
GROUP BY *<liste d'attributs>*	attributs de regroupement
HAVING *<conditions2>*	conditions sur les groupes
ORDER BY *<liste d'expressions2>*	ordre d'affichage

avec *<liste d'expressions1>* incluse dans *<liste d'attributs>* sauf pour les colonnes calculant une fonction d'agrégation.

`HAVING` permet de ne garder que les groupes satisfaisant une condition:

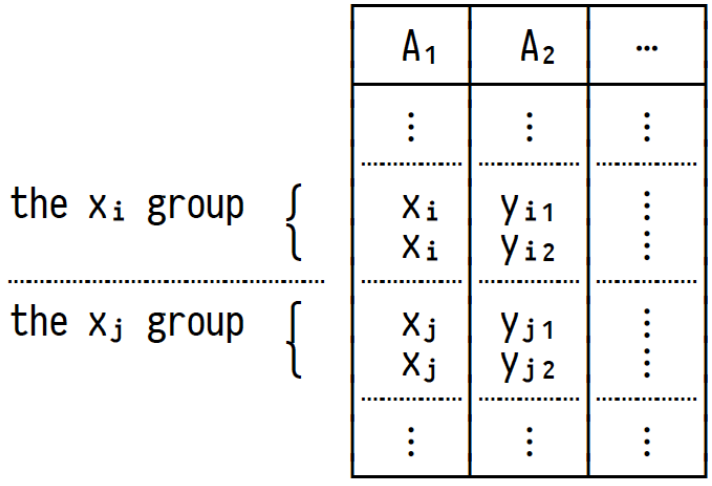
```
SELECT acteur, COUNT(*) NB
FROM film
GROUP BY acteur
HAVING COUNT(*) > 2
```

```
ACTEUR      NB
---------- ---
Canet        3
Cotillard    3
Boon         3
```

Agrégats SQL: sémantique (1)

```
SELECT ...
FROM   ...
GROUP BY A1
HAVING ...
```

evaluated once per **group** (**not** per **row**)



Grouping partitions the row bindings:

- there are no empty groups
- each row belongs to exactly one group

source: cours de Torsten Grust

Agrégats SQL: sémantique (2)

Agrégat: opérateur commutatif et associatif (à qques détails près).
Pour calculer agg (comme *fold* en programmation fonctionnelle): on initialise un accumulateur à la valeur z^{agg}.

Aggregate agg	$\varnothing^{agg}$	z^{agg}	$\oplus^{agg}(a, x)$
COUNT	0	0	a + 1
SUM	NULL[7]	0	a + x
AVG[8]	NULL	⟨0, 0⟩	⟨a.1 + x, a.2 + 1⟩
MAX	NULL	$-\infty$	$max_2(a, x)$
MIN	NULL	$+\infty$	$min_2(a, x)$
bool_and	NULL	true	a ∧ x
bool_or	NULL	false	a ∨ x
⋮	⋮	⋮	⋮

source: cours de Torsten Grust

Agrégats SQL: exemples

```sql
SELECT a, b, count(*), max(c)
FROM t
GROUP BY a, b
-- nothing special there
```

```sql
SELECT a, count(*)
FROM t
GROUP BY a, b
-- sometimes it makes sense not to display b
```

```sql
SELECT a, sum(a), max(b+d)
FROM t
GROUP BY a
/* we can group *and* aggregate on a,
though it seldom makes sense */
```

```sql
SELECT a, b, sum(a)
FROM t
GROUP BY a
/* Incorrect query:
b is not a group by expression */
```

When there is a functional dependency $a \rightarrow b$, some DBMS (ex: PostgreSQL) may tolerate it, but it is not safe (optional part of SQL standard) so you should include b in the grouping, if this is your intent.

NULLs in SQL aggregates

Aggregates (SUM, AVG, MAX) ignore null values.

Propose a table where c1 and c2 will differ:

```
SELECT SUM(a)+SUM(b) c1,
    SUM(a+b) c2
FROM t;
```

Table des matières

Guillemets: simple ou double

```sql
SELECT a FROM t; -- la colonne nommée a
SELECT 'a'; -- constant string 'a'
CREATE TABLE "ma table" ("select" int); -- identificateur entre guillemets
SELECT "select" from "ma table"; -- faisable mais à éviter.
```

Intérêt d'un identificateur entre guillemets ( "quoted identifier"):

- sensible à la casse
- permet d'utiliser des espaces
- permet d'échapper des mots clés réservés par le SGBD

Mais je ne recommande pas de les utiliser, et recommande d'éviter les mélanges.

Remarque: les identifiants qui ne sont pas entre guillements sont par défaut convertis en minuscules sous postgresql, en majuscule sous oracle.

Dans une requête SQL, le symbole d'échappement est ':

```sql
SELECT Titre FROM Film WHERE Acteur = 'O''Brien'; -- cherche O'Brien
```

Mais les applications utilisent plus souvent une requête préparée avec paramètres.

Pour plus d'informations (ex: comment échapper des caractères):

https://docs.postgresql.fr/10/sql-syntax.html#sql-syntax-lexical

Les Schémas sous PostgreSQL

Un cluster de BD postgresql contient une ou plusieurs BD. Chaque BD comporte un ou plusieurs schémas.

```sql
CREATE SCHEMA mon_schema;
CREATE TABLE mon_schema.t(id int) -- voire: base.schema.table
SELECT a FROM mon_schema.t;

CREATE SCHEMA mon_schema2;
CREATE TABLE mon_schema2.t(id int) -- valide car pas dans le même schéma
-- DROP SCHEMA mon_schema CASCADE; -- pour effacer un schéma non-vide
```

- Par défaut, schéma `public`
- En général on évite les noms qualifiés. Le schéma utilisé sera le premier dans le chemin de parcours:

```sql
SET search_path TO mon_schema, mon_schema2, public;
SHOW search_path; -- SELECT * FROM t utilisera la table dans mon_schema
```

```
 search_path
--------------
mon_schema, mon_schema2, public;
```

https://docs.postgresql.fr/10/ddl-schemas.html

Les séquences

Compteur que l'on va pouvoir incrémenter pour générer une suite de nombres.
Sert en particulier pour générer des identifiants.

```
CREATE SEQUENCE serie INCREMENT BY 100 START 101;
SELECT nextval('serie'); -- 101
SELECT currval('serie'); -- le résultat du dernier nextval: ici, 101
```

Outils similaires sous Oracle, MariaDB, MySQL, SQL Server...

Colonne autoincrémentée:

```
CREATE TABLE t (
    a SERIAL,
    b string
);
-- revient à
CREATE SEQUENCE t_a_seq;
CREATE TABLE t (
    a NOT NULL DEFAULT nextval('t_a_seq'),
    b string
);
```

Spécificités de certains SGBDs (non exhaustif)

Je déconseille fortement d'utiliser ces particularités:

- PostgreSQL:

```
SELECT country
FROM cities
GROUP BY city; -- valid if city is PK. Or more generally FD city -> country
-- Optional extensions of standard
```

- MySQL, MariaDB (InnoDB engine):

```
cities(id, city, country)
... FOREIGN KEY REFERENCES (cities.country)
-- FK referencing a non unique column!
-- Non standard
```

- PostgreSQL, MySQL, MariaDB

```
SELECT a AS bbb
FROM t
GROUP BY bbb;
```