Bases de Données

(G3SI2BD)

AIR1

Manel Zarrouk

zarrouk@lipn.univ-paris13.fr

Institut Galilée
UNIVERSITÉ PARIS 13

2019–2020

Bases de Données

Plan du cours

Part 1 - Introduction

Part 2 - L'algèbre relationnelle

Part 3 - SQL

Part 4 - Modélisation

Part 5 - Programmes et transactions

Bases de Données

Sources des transparents:

MOOC Bases de données relationnelles : apprendre pour utiliser

Philippe Rigaux et Serge Abiteboul

Lien:

https://www.fun-mooc.fr/courses/course-v1:CNAM+01041+session01/about

Je vous encourage fortement à regarder les vidéos de ce MOOC car ils sont complémentaires aux transparents

Bases de Données
Part 3 - SQL

- Démonstration de la base Voyageurs
- Requêtes mono-tables
- Jointures
- Requêtes imbriquées
- La négation
- Les agrégats
- Les vues

Bases de Données
Part 3 - SQL

- Démonstration de la base Voyageurs
- Requêtes mono-tables
- Jointures
- Requêtes imbriquées
- La négation
- Les agrégats
- Les vues

Démonstration de la base Voyageur

Le schéma est résumé ci-dessous.

- Voyageur (**idVoyageur**, nom, prénom, ville, région)
- Séjour (**idSéjour**, *idVoyageur*, *codeLogement*, début, fin)
- Logement (**code**, nom, capacité, type, lieu)
- Activité (***codeLogement***, **codeActivité**, description)

En **gras**, les clés primaires, en *italiques* les clés étrangères.

Remarque: Nommage

Totalement libre. Il est préférable d'utiliser des conventions homogènes.

Démonstration de la base Voyageur
Table de Voyageurs et des Logements

idVoyageur	nom	prénom	ville	région
10	Fogg	Phileas	Ajaccio	Corse
20	Bouvier	Nicolas	Aurillac	Auvergne
30	David-Néel	Alexandra	Lhassa	Tibet
40	Stevenson	Robert Louis	Vannes	Bretagne

code	nom	capacité	type	lieu
ca	Causses	45	Auberge	Cévennes
ge	Génépi	134	Hôtel	Alpes
pi	U Pinzutu	10	Gîte	Corse
ta	Tabriz	34	Hôtel	Bretagne

Démonstration de la base Voyageur
Table des séjours

idSéjour	*idVoyageur*	*codeLogement*	début	fin
1	10	pi	20	20
2	20	ta	21	22
3	30	ge	2	3
4	20	pi	19	23
5	20	ge	22	24
6	10	pi	10	12
7	30	ca	13	18

Remarque: Rôle des clés étrangères

Chaque nuplet référence un voyageur et un logement. Mais un voyageur ou un logement peut être référencé plusieurs fois.

Démonstration de la base Voyageur
Table des activités

codeLog.	codeAct.	description
ca	Randonnée	Sorties d'une journée en groupe
ge	Piscine	Nage loisir non encadrée
ge	Ski	Sur piste uniquement
pi	Plongée	Baptèmes et préparation des brevets
pi	Voile	Pratique du dériveur et du catamaran

Remarque: Clé primaire et étrangère

Une clé étrangère peut être une partie d'une clé primaire.

Démonstration de la base Voyageur
À retenir

Comprendre un schéma relationnel.

- Identifier la clé primaire de chaque relation (il doit toujours y en avoir une)
- Identifier les clés étrangères ; comprendre quelle clé primaire elles référencent.
- Comprendre comment les nuplets sont liés transitivement les uns aux autres.

Ces deux bases sont accessibles en ligne :

http://deptfod.cnam.fr/bd/tp

Bases de Données
Part 3 - SQL

- Démonstration de la base Voyageurs
- Requêtes mono-tables
- Jointures
- Requêtes imbriquées
- La négation
- Les agrégats
- Les vues

Requêtes mono-tables
Forme de base d'une requête SQL

La forme de base est celle d'un "bloc" constitué de trois clauses.

```
select expressions
from  une_table
[where conditions]
```

L'interprétation est toujours la suivante (dans l'ordre) :

- `from` : l'espace de recherche ; c'est toujours une table
- `where` : conditions imposées aux nuplets du `from`
- `select` : construit un nuplet-résultat à partir de chaque nuplet du `from` qui satisfait le `where`

Dans un premier temps : on suppose que l'espace de recherche est une des tables de la base

Requêtes mono-tables
Illustration

Un premier exemple très simple.

```
select prénom, nom from Voyageur where idVoyageur=10
```

Donne une table avec un nuplet-résultat.

prénom	nom
Phileas	Fogg

- L'espace de recherche est la table `Voyageur`
- La condition est une égalité sur un attribut (`idVoyageur=10`)
- On construit le nuplet-résultat à partir d'un nuplet-voyageur

Requêtes mono-tables
D'où vient SQL?

SQL ne sort pas du chapeau de quelques bricoleurs. Il a pour fondement deux langages complémentaires.

- L'algèbre relationnelle (déjà vu) est un ensemble d'opérations applicables à des tables ;
- La logique formelle (à voir en complément) donne à SQL son aspect déclaratif (ce que l'on veut, et pas comment on veut le calculer).

Nous avons choisi de baser nos explications sur l'algèbre.
La requête précédente correspond à l'expression :

$$\pi_{prénom,nom}(\sigma_{idVoyageur=10}(Voyageur))$$

Requêtes mono-tables
Clause where = opérateur σ en AR

Les conditions sont celles déjà vues pour σ

- Comparaison ($=, <, >, !=$) entre un attribut et une constante
- Comparaison entre un attribut et un autre attribut

Le `where` est un formule propositionnelle (avec `and`, `or`, `not` et le parenthésage) combinant ces conditions.

```
select * from Logement
where lieu = 'Corse' or (capacité > 50 and type='Hôtel')
```

Requêtes mono-tables
Clause select = opérateur π et ρ en AR

La projection $\pi_{A_1,\cdots,A_n}$ correspond à `select A1, .., An`

Le renommage (ρ en algèbre) s'exprime avec `as`.

```
select idVoyageur as id, prénom as p,  nom as n
from Voyageur
where idVoyageur < 30
```

id	p	n
10	Phileas	Fogg
20	Nicolas	Bouvier

Requêtes mono-tables
Attention aux doublons

Pour éviter le tri, SQL n'élimine pas les doublons.

Si on le souhaite, on peut les éliminer avec `distinct`.

```
select type
from Logement
```

type
Auberge
Hôtel
Gîte
Hôtel

```
select distinct type
from Logement
```

type
Auberge
Hôtel
Gîte

Requêtes mono-tables
À retenir

Nous savons déjà faire beaucoup de choses : toutes les requêtes exprimables avec σ, π et ρ

- Retenir l'interprétation d'une requête : d'abord le `from`, puis le `where`, et enfin le `select`
- SQL présente quelques extensions à l'algèbre (SQL permet d'utiliser des fonctions par exemple : voir la documentation de votre système)
- Le `distinct` est à réserver aux cas où les doublons sont à éliminer

Bases de Données
Part 3 - SQL

- Démonstration de la base Voyageurs
- Requêtes mono-tables
- **Jointures**
- Requêtes imbriquées
- La négation
- Les agrégats
- Les vues

Jointures

Jointure = produit cartésien composé avec une sélection.

```
select expressions
from  table1 cross join table2
[where condition_jointure]
```

Interprétation : *comme avant*, l'espace de recherche (`from`) est maintenant la table résultat de $table_1 \times table_2$

Remarque: autres syntaxes

SQL propose plusieurs syntaxes équivalentes pour la jointure. Nous donnons les autres options plus tard.

Jointures
Cas extrême : le produit cartésien

Sans clause `where`, on effectue un produit cartésien.

```
select * from Logement cross join Activité
```

code	nom	capacité	type	codeLog.	codeActivité
ca	Causses	45	Auberge	ca	Randonnée
ge	Génépi	134	Hôtel	ca	Randonnée
pi	U Pinzutu	10	Gîte	ca	Randonnée
ta	Tabriz	34	Hôtel	ca	Randonnée
ca	Causses	45	Auberge	ge	Piscine
ge	Génépi	134	Hôtel	ge	Piscine
pi	U Pinzutu	10	Gîte	ge	Piscine
...	...	...	...	...	...

Quels nuplets nous intéressent particulièrement ?

Jointures
En ajoutant la sélection

Le plus souvent, la jointure est "naturelle" : la clé primaire d'une table doit être égale à la clé étrangère d'une autre.

```
select * from Logement cross join Activité
where code = codeLogement
```

code	nom	capacité	type	codeLog.	codeActivité
ca	Causses	45	Auberge	ca	Randonnée
ge	Génépi	134	Hôtel	ge	Piscine
ge	Génépi	134	Hôtel	ge	Ski
pi	U Pinzutu	10	Gîte	pi	Plongée
pi	U Pinzutu	10	Gîte	pi	Voile

Jointures
Plus de deux tables

Aucun problème : on les énumère dans le `from`.

Exemple : les voyageurs et les logements qu'ils ont visités.

```
select V.nom as nomVoyageur, L.nom as nomLogement
from Logement as L, Séjour as S, Voyageur as V
where code = S.codeLogement
and   S.idVoyageur = V.idVoyageur
```

Remarque 1 : dans la clause `from`, on utilise le plus souvent la virgule à la place de `cross join`.

Remarque 2 : le renommage `as` sert, notamment, à gérer les ambiguités soulevées par les noms d'attributs.

Jointures
D'autres syntaxes

SQL connaît la jointure naturelle.

```
select *
from Séjour natural join Voyageur
```

idV.	idS.	codeL.	début	fin	nom	prénom	ville
10	1	pi	20	20	Fogg	Phileas	Ajaccio
10	6	pi	10	12	Fogg	Phileas	Ajaccio
20	2	ta	21	22	Bouvier	Nicolas	Aurillac
20	4	pi	19	23	Bouvier	Nicolas	Aurillac
...	...	...	...	...	...	...	...

NB : l'attribut codeLogement apparaît une seule fois

Jointures
D'autres syntaxes

SQL connaît aussi la jointure "non naturelle" (les attributs de jointure n'ont pas le même nom).

```
select *
from Logement join Activité on (code=codeLogement)
```

Intérêt ? Aucun. Connaître une syntaxe suffit. Celle que nous avons présenté au début est la plus lisible et la plus neutre.

Remarque: La syntaxe ne compte pas !

Il existe 100 façons d'écrire une même requête : aucune n'est meilleure qu'une autre : le système décide seul de l'évaluation !

Jointures
À retenir

La jointure est obtenue en effectuant un produit cartésien dans le `from`.

- Tout se passe comme si on interrogeait une seule table, celle définie par $table_1 \times table_2$
- Le critère de jointure est exprimé dans le `where`, avec les critères (éventuels) de sélection
- Les synonymes (attributs ou tables) entrainent des ambiguités, et nécessitent des renommages

Beaucoup de syntaxes alternatives (dues aux deux paradigmes fondant SQL). En connaître une suffit.

Bases de Données
Part 3 - SQL

- Démonstration de la base Voyageurs
- Requêtes mono-tables
- Jointures
- Requêtes imbriquées
- La négation
- Les agrégats
- Les vues

Requêtes imbriquées

Exemple : Les logements où je peux faire du ski.

```
select distinct nom
from  Logement, Activité
where code = codeLogement
and codeActivité = 'Ski'
```

C'est la syntaxe classique, "à plat".

Cas particulier : je ne m'intéresse qu'aux attributs de la table Logement

Requêtes imbriquées
Avec quantification existensielle

Formulation légèrement différente : les logements où il existe une activité 'Ski'

```
select nom
from  Logement
where exists (select '' from Activité
       where code = codeLogement
       and codeActivité = 'Ski')
```

Note : la clause `select` de la requête imbriquée ne sert à rien. Je m'intéresse uniquement à l'existence (ou non) des nuplets de `Activité`.

Requêtes imbriquées
Avec condition d'appartenance

Autre formulation : Les logements qui font partie de ceux où on trouve une activité 'Ski'

```
select nom
from  Logement
where code in  (select codeLogement from Activité
                where codeActivité = 'Ski')
```

Note : correspondance entre les attributs code du bloc principal et codeLogement de la requête imbriquée

Requêtes dites corrélées à cause du lien établi entre les deux blocs.

Requêtes imbriquées
Quand utiliser l'imbrication

Condition : c'est une jointure, mais on construit le résultat avec les nuplets d'une seule relation. Dans l'algèbre, c'est une semi-jointure

L'imbrication n'apporte pas d'expressivité supplémentaire. Le choix est une question de goût.

Critères à prendre en compte :

- La lisibilité compte ! Au-delà d'un niveau d'imbrication, l'interprétation devient laborieuse
- On ne peut pas construire le nuplet-résultat avec les nuplets des sous-requêtes

Requêtes imbriquées
Critère de lisibilité : un exemple

Les voyageurs qui sont allés en Corse.

```
select prénom, nom
from  Voyageur as V
where  exists (select '' from Séjour as S
               where V.idVoyageur = S.idVoyageur
               and exists (select '' from Logement
                  where codeLogement=code
                  and lieu='Corse') )
```

La même requête, à plat.

```
select distinct V.prénom, V.nom
from  Voyageur as V, Séjour as S, Logement as L
where V.idVoyageur = S.idVoyageur
and   codeLogement=code
and lieu='Corse'
```

Requêtes imbriquées
À retenir

Tant qu'on n'exprime pas de négation, l'imbrication n'apporte que des manières alternatives d'exprimer une même requête.

- Bien comprendre les équivalences (étudier les exemples)
- La lisibilité est le principal critère : SQL, c'est apprendre à exprimer des requêtes avec élégance !
- Une requête imbriquée évite naturellement des doublons qu'il faut éliminer dans la version "à plat"

Une séquence pour rien ? Non car l'imbrication est indispensable pour les négations. À suivre.

Bases de Données
Part 3 - SQL

- Démonstration de la base Voyageurs
- Requêtes mono-tables
- Jointures
- Requêtes imbriquées
- La négation
- Les agrégats
- Les vues

La négation

On veut les logements qui ne proposent pas de Ski.

La requête suivante n'est pas la bonne solution.

```
select distinct nom
from  Logement, Activité
where code = codeLogement
and codeActivité != 'Ski'
```

Elle donne les logements où il existe une activité autre que 'Ski'

On veut ici exprimer une condition (négative) sur un ensemble.
"Il ne doit pas exister d'activité Ski parmi celles du logement."

La négation
La bonne formulation

"Il ne doit pas exister d'activité Ski parmi celles du logement."

C'est donc une condition existentielle (négative).

```
select nom
from Logement
where not exists (select ''
                  from Activité
                  where code = codeLogement
                  and codeActivité = 'Ski')
```

Rappel : le `select` du bloc imbriqué ne sert à rien

La négation
Le not in marche aussi

On peut toujours formuler le `(not) exists` de manière très proche avec `(not) in`

```
select nom
from Logement
where code not in (select codeLogement
                   from Activité
                   where codeActivité = 'Ski')
```

Autre syntaxe possible : avec la différence ensembliste ("tous les logements moins ceux où on fait du ski"). Peu pratique.

La négation
La quantification universelle

Je sais exprimer une condition existentielle. Quid de la quantification universelle ?

"Je veux les voyageurs qui sont allés dans tous les logements"

SQL peut le faire. Si je suis allé dans tous les logements, il n'existe pas de logement où je ne suis pas allé.

La quantification universelle s'exprime avec une double quantification existentielle et la négation (Ouf...).

La négation
La solution

Double imbrication, double quantification, double négation

```
select distinct v.prénom, v.nom
from Voyageur as v
where  not exists
     (select '' from Logement as l
      where not exists
           (select ''  from Séjour as s
            where l.code = s.codeLogement
            and   v.idVoyageur = s.idVoyageur))
```

Peu courant, mais montre la puissance du langage.

SQL couvre tout ce qui s'exprime dans la logique classique, dite "du premier ordre". Cf. le complément sur le calcul.

La négation
Bilan

Nous savons maintenant (presque) tout faire !

- Exprimer toutes les requêtes dites "positives", sélection, projection et jointure.
- Exprimer la négation avec un `not exists`
- Et même exprimer la quantification universelle.

SQL, langage de logicien, pas d'informaticien

Il reste à étudier quelques extensions pratiques.

Bases de Données
Part 3 - SQL

- Démonstration de la base Voyageurs
- Requêtes mono-tables
- Jointures
- Requêtes imbriquées
- La négation
- Les agrégats
- Les vues

Les agrégats
Principe général

On définit des groupes de lignes partageant une ou plusieurs valeurs.

On ramène chaque groupe à une seule valeur en appliquant une fonction d'agrégation.

Cas le plus simple : groupe = résultat d'une requête classique

```
select count(*) as nombre, sum(capacité) as totalCapacité
from Logement
where type='Hôtel'
```

nombre	totalCapacité
2	168

Les agrégats
Le group by

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

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

```
select type, count(*) as Nombre,
            sum(capacité)as totalCapacité
from Logement
group by type
```

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

Les agrégats
Décomposant : l'étape de regroupement

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

type	Groupe (Logement)
Auberge	(ca, Causses, 45, Auberge, Cévennes)
Gîte	(pi, U Pinzutu, 10, Gîte, Corse)
Hôtel	(ge, Génépi, 134, Hôtel, Alpes) (ta, Tabriz, 34, Hôtel, Bretagne)

Ce n'est pas une table en première forme normale (atomicité des valeurs).

Les agrégats
Décomposant : l'étape d'agrégation

La fonction d'agrégation ramène chaque groupe à une valeur

type	count(*)	sum(capacité)
Auberge	1	45
Gîte	1	10
Hôtel	2	168

Cette fois c'est une table en première forme normale.

Les agrégats
La clause having

Filtrage dans un deuxième temps, après calcul des agrégats, portant sur le résultat de la fonction d'agrégation.

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

```
select type, count(*) as nombre,
            sum(capacité) as totalCapacité
from Logement
group by type
having count(*) > 1
```

type	nombre	totalCapacité
Hôtel	2	168

Les agrégats
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 atomique grâce à une fonction d'agrégation
- On peut filtrer les groupes obtenus avec `having`

Bases de Données
Part 3 - SQL

- Démonstration de la base Voyageurs
- Requêtes mono-tables
- Jointures
- Requêtes imbriquées
- La négation
- Les agrégats
- Les vues

Les vues

Toute requête produit une relation.

Nommer cette requête c'est nommer la relation résultat.

Une vue est une requête nommée.

On peut la considérer comme une relation calculée et l'interroger comme les autres. Très utile pour :

- Filter une partie de la base
- Simplifier (et contrôler) les accès

Les vues
Création et interrogation d'une vue

Exemple : on veut créer une "vue" de la base par région.

```
create view LogementCorse as
        select *
        from Logement
        where lieu='Corse'
```

On peut interroger la vue comme n'importe quelle table.

```
select * from LogementCorse
```

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

Les vues
Vue = “macro” pour des requêtes complexes

On peut nommer des requêtes complexes souvent utilisées

```
create view VoyageursEnCorse as
select V.nom as nomVoyageur, L.nom as nomLogement
from LogementCorse as L, Séjour as S, Voyageur as V
where code = S.codeLogement
and   S.idVoyageur = V.idVoyageur
```

Notez : on construit des vues sur d’autres vues. Ici, on a tous les clients des logements corses.

Les vues
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.

On pourrait insérer dans `LogementCorse` mais pas dans `VoyageursEnCorse`

Les vues
À retenir

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

Elles permettent de restructurer "en intention" une base.

- Pour faciliter l'accès (jointures pré-définies)
- Pour restreindre la visibilité des données (on ne donne accès qu'à la vue)
- Les insertions dans les vues sont soumises à fortes restrictions .

Un mécanisme très utile pour "dénormaliser" une base sans en subir les incovénients. À suivre !