Chap. 8 : Algèbre relationnelle et SQL

1 Le modèle relationnel : son intérêt

1.1 La limite des structures de données de type liste pour la recherche d'information : un exemple

Ce qu'on veut faire :

• On veut stocker la liste des élèves de prépa. de notre lycée en renseignant leur prénom, leur nom, leur classe, leur lycée de terminale.
• On veut ensuite pouvoir *exploiter* facilement ces données i.e. rechercher facilement p.ex. :
— les élèves appartenant à une même classe,
— les élèves venant d'un même lycée en Terminale.

Quelle *structure* choisir pour nos données ?

Disons pour simplifier qu'on a des toutes petites classes et pour simplifier encore on se contentera de prénoms :
• MPSI 1, avec comme élèves :
— Alexis (terminale : J. Jaurès)
— Alexandre (terminale : A. Daudet)
• MPSI 2, avec comme élèves
— Noëlie (terminale : G. Pompidou)
— Arnaud (terminale : Nevers)
• PCSI 1, avec comme élèves
— Arthur (terminale : A. Lavoisier)
— Géraldine(terminale : G. Eiffel)
Avec ce que nous connaissons pour l'instant, en PYTHON, on peut *choisir* de rentrer ces données sous forme d'une liste de listes comme suit :

```
lycee=[ ["MPSI 1",[["Alexis","Jaurès"],["Alexandre","Daudet"]]],
["MPSI 2",[["Noëlie","Pompidou"],["Arnaud","Nevers"]]],
["PCSI 1",[["Arthur","Lavoisier"],["Géraldine","Eiffel"]]]
]
```

Avec ce choix, on voit que la liste (en info. en général on dit plutôt tableau) `Lycee` a trois entrées correspondantes aux trois classes [1] (MPSI 1, MPSI 2, PCSI 1).

La notion d'attribut

• Avec ce choix chaque classe (scolaire) a ce qu'on appelle des **attributs** ici au nombre de *deux* :
— son nom "MPSI 1" ou "MPSI 2" ou "PCSI 1"
— sa liste d'élève.
• De même chaque élève a *deux attributs* :
— son prénom "Alexis"
— son lycée d'origine "Jaurès".

L'exploitation des données : *data mining*

Avec la structure choisie :
• si on veut avoir la liste des élèves d'une classe c'est très facile :

```
def eleve_classe(nomcl):
    "renvoie la liste des élèves d'une classe"
    for classe in lycee:
```

1. au sens scolaire du mot classe, et pas celui de python !

```
        if classe[0]==nomcl:
            return classe[1]
    return None
```

• en revanche, si on voit avoir la liste des élèves ayant fait leur Terminale dans un lycée donné c'est plus compliqué :

```
def eleve_terminale(term):
    "renvoie la liste des élèves ayant fait leur terminale en term"
    resultat=[]
    for classe in lycee:
        for eleve in classe[1]:
            if eleve[1]==term:
                resultat.append(eleve)
    return resultat
```

Moralité :

Bien sûr, on aurait pu fabriquer la liste `lycée` autrement, en classant au départ par lycée d'origine, mais dans ce cas, c'est la première recherche (par classe) qui aurait été plus coûteuse.

Ce qu'il faut comprendre :

Dans cet exemple, un élève a des liens d'appartenance à la fois à une classe et à un lycée d'origine. Or la *représentation* des données sous forme d'une liste de listes en PYTHON (en informatique en général, on appelle plutôt cela un *tableau* de *tableaux*), *contraint* à privilégier un de ces liens. On va développer dans ce qui suit une *autre* façon de représenter les données fondée sur la notion de *relation*, qui n'a plus ce défaut [a].

a. mais nous ne dirons pas comment cette représentation relationnelle est effectivement implantée en machine

1.2 Représentation dans le modèle relationnel

Définition en deux temps d'une *relation*

La façon suivante de représenter notre problème repose sur un déclaration en deux temps. Elle ressemble à l'usage que vous pouvez avoir fait d'un tableur pour lesquel on a donné un nom aux colonnes.

• **1er temps : schéma ou intension de la relation `eleve`** On définit un élève comme un triplet *prénom, classe, lycée-de-term* : les trois entrées de ce triplet s'appellent les *attributs* de l'élève.

• **2ème temps : entrée de la table des élèves : extension de la relation `eleve`**

L'information sur les élèves peut être représentée par un tableau (ou table) où la première ligne n'apparaît pas normalement :

prénom	classe	lycée-de-term
"Alexis"	"MPSI 1"	"Jaures"
"Alexandre"	"MPSI 1"	"Daudet"
"Noëlie"	"MPSI 2"	"Pompidou"
"Arnaud"	"MPSI 2"	"Nevers"
"Arthur"	"PCSI 1"	"Lavoisier"
"Géraldine"	"PCSI 1"	"Eiffel"

Là, où, informatiquement, la relation est **plus** qu'un tableau est qu'elle a été déclarée avec des attributs au départ : autrement dit on ne fabrique pas seulement un tableau, mais un tableau où chaque colonne a déjà un nom. D'où le mot *relation* : entre chaque concept (attribut) et les valeurs prises.

Remarque 1. Bien sûr, informatiquement, on a besoin d'un langage qui permette cette déclaration (comme votre tableur). Le langage adapté à la création de cette structure s'appelle SQL, nous y reviendrons plus loin.

Le domaine des attributs :

Dans l'exemple précédent, pour la table `eleve`, chacun des attributs prend comme valeur une *chaîne de caractères*.

Si on considère une nouvelle table `eleve2` où les attributs sont (prénom,année-naissance,sexe, classe) qu'on remplit par exemple comme suit :

"Alexis"	1996	M	"MPSI 1"
"Alexandre"	1995	M	"MPSI 1"
"Noëlie"	1996	F	"MPSI 2 "
"Arnaud"	1996	M	"MPSI 2"

dans ce cas, lors de la déclaration des attributs, on va demander que le domaine de l'attribut année-naissance soit un *entier* et que le domaine de l'attribut sexe soit M ou F.

1.3 Le lien entre les tables : répercuter les modifications

Imaginons que Noëlie passe de MPSI 2 en MPSI 1 à un moment de l'année. Avec des tableaux python, il nous faudrait modifier "manuellement" les *deux* tables `eleve` et `eleve2`. Le principe des SGBD (systèmes de gestion de base de données) est de *relier les données des tables entre elles* : l'attribut `classe` de la table `eleve` et celui de la table `eleve2` devront être reliés pour qu'une modif. de l'un entraîne une modif. de l'autre. Ici, ils seront reliés grâce aux valeurs communes des attributs `nom` des élèves. Nous allons détailler cela dans ce qui suit.

2 Définitions plus générales (abstraites) sur les relations

2.1 Deux définitions : schéma relationnel et relation proprement dite

Défnition : [**Schéma relationnel ou *intension* d'une relation**] On considère un ensemble fini $\mathcal{A}$ dont les éléments sont appelés **attributs**. On appelle *schéma relationnel* (ou *intension d'une relation*), la donnée d'un n-uplet $S = (A_1, \dots, A_n)$ où $A_1, \dots, A_n$ sont des attributs deux à deux distincts, et, pour chaque $i \in [\![1, n]\!]$, d'un ensemble D_i appelé **domaine** de l'attribut A_i.

Remarque 2. Souvent, on note un schéma relationnel en précisant à côté de chaque attribut A_i son domaine D_i, ce qui donne l'écriture :

$$S = ((A_1, D_1), \dots, (A_n, D_n)).$$

Exemple Un schéma relationnel correspondant à la relation `eleve2` ci-dessus est :

```
Schemaeleve2=((prénom,string),(datenaissance,integer),(sexe,{M,F}),(classe,string))
```

Définition : [**Table ou *extension* d'une relation**] Etant donné un schéma relationnel S comme défini ci-dessus, une table (ou simplement une relation !) associée à ce schéma est la donnée d'un ensemble fini de n-uplets, éléments de $D_1 \times \cdots \times D_n$. On note $R(S)$ une telle relation pour dire qu'elle est associée à S. Le nombre m de n-uplets s'appelle le cardinal de la relation.

Exemple Au schéma relationnel S de l'exemple ci-dessus, on peut associer par exemple la table `eleve2` déjà donnée, qui est de cardinal 4.

"Alexis"	1996	M	"MPSI 1"
"Alexandre"	1995	M	"MPSI 1"
"Noëlie"	1996	F	"MPSI 2 "
"Arnaud"	1996	M	"MPSI 2"

Remarque 3. Ainsi, sur la table de la relation, le nombre n correspond au nombre de colonnes du tableau (nombre d'attributs) alors que le cardinal m de la relation correspond au nombre de lignes du tableau (on dit parfois le nombre de *valeurs*).

D'une manière générale, la table d'une relation $R(S)$ où $S = ((A_1, D_1), \ldots, (A_n, D_n))$ ressemblera donc à ceci (la première ligne ne figure pas dans la table, elle est là pour mémoire) :

A_1, D_1	$\ldots$	A_j, D_j	$\ldots$	A_n, D_n
$t_{1,1}$	$\ldots$	$t_{1,j}$	$\ldots$	$t_{1,n}$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$t_{i,1}$	$\ldots$	$t_{i,j}$	$\ldots$	$t_{i,n}$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$
$t_{m,1}$	$\ldots$	$t_{m,j}$	$\ldots$	$t_{m,n}$

avec $t_{i,j} \in D_j$ pour tout $i \in [\![1, m]\!]$ et $j \in [\![1, n]\!]$.

N.B. On a fixé un ordre sur les attributs par commodité ici, mais du point de vue de l'information cet ordre est sans importance, ce qui compte est que les attributs ont un *nom*.

Retenir : se donner une relation c'est se donner :
• son schéma S avec ses attributs (qui seront les noms des colonnes de la table qui n'existe pas encore)
• ses éléments (qui sont les lignes de la table que l'on crée).

2.2 Sous-ensembles du schéma S et de la relation $R(S)$:

Abus de notation : On a dit qu'un schéma S était un n-uplet $(A_1, \ldots, A_n)$ d'attributs (deux à deux distincts). Un tel uplet étant ordonné, on peut définir clairement chaque ligne de la relation $R(S)$ associée comme un n-uplet. Cependant, il est commode aussi (et indifférent du point de vue de l'information stockée) de considérer S comme l'ensemble $\{A_1, \ldots, A_n\}$ des attributs.

Moralité (appartenance et inclusion dans S) :
- On se permettra de noter $A \in S$ pour dire que A est un attribut du schéma S.
- De même si p.ex. S=(nom,prénom,adresse, code-postal,ville, tél) on pourra considérer un *sous-ensemble* X={nom,prénom, ville} d'attributs et noter (abusivement) $X \subset S$.

Les éléments de la relation $R(S)$: Avec la table donnée plus haut, le quadruplet e=("Alexis",1996,"M","MPSI 1") est un *élément* de la relation `eleve2`=R(Schemaeleve2)

On pourra noter $e \in R$(Schemaeleve2).

Composante d'un élément $e \in R(S)$ sur un attribut $A \in S$: si $e \in R(S)$ et $A \in S$, on note $e.A$ la composante du n-uplet e associée à l'attribut A. Par exemple si e=("Alexandre",1995,"M","MPSI 1") et A="sexe", alors $e.A$="M".

Concrètement si e est à la ligne i et A à la colonne j, alors $e.A$ est simplement l'entrée (i, j) du tableau.

Généralisation : composante d'un $e \in R(S)$ sur un sous-ensemble $\{A_{i_1}, \ldots A_{i_k}\}$ de S : si $X = (A_{i_1}, \ldots, A_{i_k}) \subset S$ (ordonnées comme dans S) et $e \in R(S)$ on note $e(X) = (e.A_{i_1}, \ldots, e.A_{i_k})$.

3 Notion de clé, clé primaire

Définition : [**Clé**] Soit $R(S)$ une relation de schéma S et $K \subset S$ un sous-ensemble d'attributs. On dit que K est une *clé* pour $R(S)$ si, et seulement si, pour toutes valeurs t_1 et t_2 de $R(S)$ telle que $t_1(K) = t_2(K)$ on a $t_1 = t_2$.

De manière plus informelle : deux lignes distinctes de $R(S)$ auront donc leur entrées correspondants à la clé qui seront distinctes.

N.B. Comme $R(S)$ est un *ensemble*, on ne répète jamais deux fois la même ligne.

Exemple Si on considère la relation R définie par la table suivante (la première ligne, qui donne le schéma de la relation, ne fait pas partie de la table) :

prénom	nom	datenaissance	telbureau
Arthur	duschmol	01042000	0983
gérard	machin	03081983	0332
Arthur	duschmol	15121965	0222
norbert	truc	15121965	0983

K=(prénom,nom) n'est pas une clé de R, aucun attribut a lui seul ne l'est ici, en revanche, K=(nom,datenaissance) ou K=(prénom,nom,telbureau) sont des clés.

Remarque : c'est un exemple tordu. Souvent, on pourra fabriquer une clé avec un seul attribut (par exemple votre numéro INSEE).

Définition : Clé primaire Lors de la confection d'un schéma relationnel S, on peut *imposer* qu'un certain sous-ensemble d'attributs $K \subset S$ soit toujours une *clé* pour les relations $R(S)$ qu'on va fabriquer à partir de S. Cette clé (unique) choisie s'appelle une *clé primaire*. On dit alors qu'on impose la contrainte de *clé primaire*.

Notation : lorsqu'on se donne un tel schéma avec une clé primaire, on *souligne* les attributs qui constitue la clé primaire.

Exemple : Si on se donne le schéma S=(prénom,nom,datenaissance, telbureau) alors la relation $R(S)$ définie par le tableau précédent *viole la contrainte de la clé primaire* . En revanche, si on enlève le deuxième Arthur duschmol du tableau c'est ok. En pratique, pour un tel schéma avec la contrainte de clé primaire imposée, on ne pourrait pas rentrer le deuxième Arthur duschmol dans la relation.

Remarque 4. Souvent on essaiera de choisir une clé primaire constituée d'un seul attribut.

4 Opérations sur les relations : algèbre relationnelle et SQL

4.1 Motivation (double)

• Pour l'instant nous n'avons pas introduit les opérations par lesquelles nous allons pouvoir *interroger* notre base de données : dans le langage des base de données on parlera de faire des *requêtes*.

C'est ce que nous allons développer dans ce qui suit, en commençant par les opérations les plus simples qui correspondent aux requêtes les plus simples.

• Ensuite comme nous avons définis deux tables `eleve` et `eleve2`, nous verrons dans un second temps au § 5 comme exploiter conjointement les données de deux tables.

4.2 Introduction aux requêtes à la SQL

Considérons la table `eleve` qui suit le schéma relationnel défini au § 1.2 :

"Alexis"	"MPSI 1"	"Jaures"
"Alexandre"	"MPSI 1"	"Daudet"
"Noëlie"	"MPSI 2"	"Pompidou"
"Arnaud"	"MPSI 2"	"Nevers"
"Arthur"	"PCSI 1"	"Lavoisier"
"Géraldine"	"PCSI 1"	"Eiffel"

Supposons qu'on veuille avoir la liste des *nom* des élèves qui viennent de "Daudet" ou qui sont en "PCSI 1". Dans le langage SQL (Structured Query Langage) que nous introduirons plus loin, en ayant déclaré pour la tables `eleve` le schéma `(nom, classe, lyceeterm)` cette liste s'obtiendra comme suit :

```
SELECT nom FROM eleve WHERE lyceeterm="Daudet" OR classe="PCSI 1"
```

Le langage renverra alors la table réduite

"Alexandre"
"Arthur"
"Géraldine"

Dans les deux paragraphes qui suivent, nous allons décomposer les opérations en jeu dans cette requête comme des *opérations sur les tables.*

4.3 Opérations sur une table : sélections et projections (algèbre relationnelle et SQL)

Définition : (sélection)

Soit $R = R(S)$ une relation (table) de schéma $S = (A_1, \ldots, A_n)$. Soit A un des attributs de S et a une valeur dans le domaine de A. On appelle *sélection de R selon $A = a$* la relation obtenue en sélectionnant dans R uniquement les valeurs e de R (i.e. les lignes de la table) pour lesquelles $e.A = a$.

Notation la relation obtenue est notée $\sigma_{A=a}(R)$.

On peut aussi écrire, avec les mêmes conventions qu'en maths. $\sigma_{A=a}(R) = \{e \in R | e.A = a\}$

Exemple Avec la table `eleve` du § 4.2 ci-dessus, $\sigma_{lyceeterm="Daudet"}(eleve)$ donne la relation :

"Alexandre"	"MPSI 1"	"Daudet"

Définition : (projection) Soit $R = R(S)$ une relation de schéma S. Soit $X \subset S$ un sous-ensemble d'attributs. On appelle *projection de R selon X* la relation $\pi_X(R)$ obtenue en ne gardant que les *composantes* des valeurs de R sur les attributs de X (cf. § 2.2).

Formellement : $\pi_X(R) = \{e(X) | e \in R\}$

Exemple Avec la table `eleve` § 4.2 ci-dessus, S=(nom, classe, lyceeterm), et par exemple X=(nom, lyceeterm), on obtient pour $\pi_X(eleve)$:

"Alexis"	"Jaures"
"Alexandre"	"Daudet"
"Noëlie"	"Pompidou"
"Arnaud"	"Nevers"
"Arthur"	"Lavoisier"
"Géraldine"	"Eiffel"

Remarque : Du coup le schéma relationnel de $\pi_X(R)$ est X.

Retenir :
- la sélection agit sur les lignes (choix des valeurs),
- la projection sur les colonnes (choix des attributs).

Remarque 5 (L'algèbre relationnelle vs SQL). Les notations $\sigma_{A=a}$ ou π_X sont les notations de **l'algèbre relationnelle**. Cette algèbre, que nous allons continuer à développer, va nous permettre de décomposer nos requêtes en processus élémentaires.

Concrètement, ces requêtes se feront avec le langage SQL dont les algorithmes internes resteront *cachés* pour nous ! Nous ne ferons donc aucune étude de complexité pour ces requêtes.

Dans le langage SQL la commande `SELECT` permet de faire les deux opérations de sélection et projection.

Ainsi :

- pour obtenir $\pi_X(eleve)$ on rentrera :

```
SELECT nom, lyceeterm FROM eleve
```

- pour obtenir $\sigma_{lyceeterm="Daudet"}(eleve)$ on rentrera :

```
SELECT * FROM eleve WHERE lyceeterm="Daudet"
```

où le joker `*` permet d'avoir tous les attributs.

N.B. Ainsi la commande `SELECT` est la fonction de base pour les requêtes d'interrogation de base de données en SQL. Noter que `SELECT * FROM eleve` nous affichera toute la table `eleve` par exemple.

4.4 Opérations ensemblistes simples sur deux tables ayant le même schéma

Les définitions suivantes sont immédiates : il s'agit des définitions ensemblistes usuelles. **Définition :** Soient $R_1(S)$ et $R_2(S)$ deux relations (tables) ayant le même schéma relationnel S. On peut définir leur réunion $R_1 \cup R_2$, leur intersection $R_1 \cap R_2$, leur différence $R_1 - R_2$. Il s'agit encore de relations associées au schéma S.

Par exemple

si R_1 est :

"Alexis"	"MPSI 1"	"Jaures"
"Alexandre"	"MPSI 1"	"Daudet"

et R_2 est :

"Alexandre"	"MPSI 1"	"Daudet"
"Noëlie"	"MPSI 2"	"Pompidou"
"Arnaud"	"MPSI 2"	"Nevers"

alors $R_1 \cup R_2$ est :

"Alexis"	"MPSI 1"	"Jaures"
"Alexandre"	"MPSI 1"	"Daudet"
"Noëlie"	"MPSI 2"	"Pompidou"
"Arnaud"	"MPSI 2"	"Nevers"

et $R_1 \cap R_2$ est :

"Alexandre"	"MPSI 1"	"Daudet"

et $R_1 - R_2$ est :

"Alexis"	"MPSI 1"	"Jaures"

4.5 Opérations de sélections composées avec des opérations ensemblistes

4.5.1 Retour sur l'exemple donné plus haut : la réunion en algèbre relationnelle et le `OR` en `SQL`

a) **Une requête SQL avec un OR** :
On peut reprendre l'exemple écrit en SQL au § 4.2.

```
SELECT *  FROM eleve WHERE lyceeterm="Daudet" OR classe="PCSI 1"
```

b) **Comment interpréter cette requête en terme d'algèbre relationnelle ?**

Il s'agit de fabriquer la table *réunion* des deux tables, chacune obtenue par sélection sur `eleve`

Ces deux tables sont : $\sigma_{lyceeterm="Daudet"}(eleve)$ et $\sigma_{classe="PCSI1"}(eleve)$.

La requête s'écrit donc en algèbre relationnelle :

c) **Une alternative SQL avec UNION au lieu de OR**
Le SQL est muni d'un opérateur `UNION` qui colle davantage avec l'algèbre relationnelle :

```
SELECT * FROM eleve WHERE lyceeterm="Daudet" UNION SELECT * FROM eleve WHERE classe="MPSI 1"
```

Cet exemple doit aussi faire comprendre que le *résultat* de chaque `SELECT` est *une table* et donc qu'on va pouvoir `imbriquer` les résultats de plusieurs `SELECT`.

d) **Attention aux parenthèses :** l'usage de parenthèses en SQL est un sujet délicat, qui peut provoquer facilement des erreurs de syntaxe. Ici, il n'en faut pas entre les deux requêtes reliées par un UNION. Voir le paragraphe 4.6 plus loin.

4.5.2 Plusieurs façons de réaliser une différence ensembliste en SQL

Par exemple si on veut les noms des élèves n'ayant pas fait leur terminale à J. Jaures :

Dans ce qui suit, on suppose *d'abord* que `nom` est une clé, autrement dit qu'il n'y a pas deux élèves ayant le même nom. Ensuite, on comparera les requêtes dans le cas où `nom` n'est pas un clé.

a) **Une façon assez intuitive avec WHERE NOT :** on voit ici les conditions de sélection qui sont après le WHERE comme des booléens.

b) **Par différence de deux tables en algèbre relationnelle :**

c) **La commande EXCEPT de SQL qui colle à la structure ensembliste :**
Ou encore avec `EXCEPT` qui a l'avantage de vraiment opérer entre deux tables :

```
SELECT nom FROM eleve  EXCEPT SELECT nom FROM eleve WHERE lyceeterm="Jaures"
```

d) **Une autre méthode avec une *sous-requête***

```
SELECT nom FROM eleve  WHERE nom NOT IN  (SELECT nom FROM eleve WHERE lyceeterm="Jaures")
```

‘

e) Comparer maintenant l'effet des requêtes précédentes si l'attribut `NOM` n'est pas une clé autrement dit il peut y avoir plusieurs élèves ayant le même nom.

4.6 Un mot sur les parenthèses en SQL

Attention : je ne prétends pas que les règles empiriques que je donne ici soient complètement générales.

Lorsqu'on veut mettre ensemble des requêtes (donc plusieurs SELECT) suivant l'articulation entre les requêtes , il faut ou ne faut pas mettre des parenthèses autour de ces requêtes sous peine d'erreur...

• **Cas de « conjonctions de coordination en requêtes »** : si les requêtes sont « de même niveau reliées par des `UNION, INTERSECT, EXCEPT` on ne met PAS de parenthèses ;

• **Cas des « conjonctions de subordination entre requêtes »** : pour les sous-requêtes introduites par exemple par `WHERE.. IN` ou `WHERE .. NOT IN` on met la sous-requêtes entre parenthèses.

4.7 Résumé des opérations de comparaison en SQL

On vient de voir que le langage SQL permet de coder l'algèbre relationnelle élémentaire et davantage avec la souplesse des `OR` en plus des `UNION` etc. Voici une petite synthèses des opérations entre valeurs dans les tables (et pas seulement entre tables) que permet SQL

— Les opérateurs de comparaisons mathématiques : `<,>,=,<>,>=,<=`
— Le test pour savoir si une case du tableau est *vide* : `IS NULL` ou `IS NOT NULL`
— Le test pour savoir si un élément est dans un tableau `IN`

5 Opérations entre tables de schémas différents : fabrication d'un lien entre les tables

Les opérations que nous allons définir maintenant permettent de croiser les informations présentes dans plusieurs tables.

5.1 Produit cartésien

Comme pour l'intersection, la réunion, la définition du produit cartésien de deux relations (deux tables) suit celle des ensembles avec toutefois une subtilité liée au suivi des attributs.

Union disjointe forcée

Définition : Soit $S = (A_1, \ldots, A_n)$ et $S' = (B_1, \ldots, B_{n'})$ deux schéma relationnels pouvant avoir des attributs communs. On appellera union disjointe et on note $S \uplus S'$ le schéma relationnel $(A_1, \ldots, A_n, B_1, \ldots, B_{n'})$.

N.B. *Même si S et S' ont des attributs communs, on va les distinguer dans $S \uplus S'$.* En pratique, cela passe par le renommage d'attribut.

Exemple Si S=(nom,classe,lyceeterm) et S' =(nom,âge,sexe) alors :

$S \uplus S'$ =(nom1,classe,lyceeterm,nom2,âge,sexe).

Les deux attributs qui portaient le même « nom » ont été renommés pour pouvoir les distinguer dans l'union (voulue disjointe) $S \uplus S'$.

Application à la déf. du produit cartésien

Définition : Soient $R = R(S)$ et $R' = R'(S')$ deux relations. On définit le produit cartésien $R \times R'$ comme l'ensemble des couples (e, e') pour $e \in R$ et $e' \in R'$.

Cependant, si $S = (A_1, \ldots, A_n)$ et $S' = (B_1, \ldots, B_{n'})$ alors chaque $e \in R$ est un n uplet $e = (a_1, \ldots, a_n)$ (représentant une ligne de la table de R) et chaque $e' \in R'$ est un n'-uplet $(b_1, \ldots, b_{n'})$ et on identifie naturellement (e, e') au $n + n'$-uplet $(a_1, \ldots, a_n, b_1, \ldots, b_{n'})$.

De la sorte $R \times R'$ est une relation associée au schéma $S \uplus S'$.

Exemple On reprend les exemples du § 1.2. On prend la relation R = *eleve* associée au schéma S=(nom,classe,lyceeterm) et définie par la table :

"Alexis"	"MPSI 1"	"Jaures"
"Alexandre"	"MPSI 1"	"Daudet"
"Noëlie"	"MPSI 2"	"Pompidou"

On prend la relation R' associée au schéma S'=(nom,annee-naissance,sexe) et définie par la table

"Alexis"	1996	M
"Alexandre"	1995	M
"Noëlie"	1996	F

Alors le produit cartésien $R \times R'$ est associé à l'union disjointe des schémas (en renommant les attributs de même nom) $S \uplus S'$=(nom1,classe,lyceeterm,nom2,datenaissance,sexe) et défini par la table

"Alexis"	"MPSI 1"	"Jaures"	"Alexis"	1996	M
"Alexis"	"MPSI 1"	"Jaures"	"Alexandre"	1995	M
"Alexis"	"MPSI 1"	"Jaures"	"Noëlie"	1996	F
"Alexandre"	"MPSI 1"	"Daudet"	"Alexis"	1996	M
"Alexandre"	"MPSI 1"	"Daudet"	"Alexandre"	1995	M
"Alexandre"	"MPSI 1"	"Daudet"	"Noëlie"	1996	F
"Noëlie"	"MPSI 2"	"Pompidou"	"Alexis"	1996	M
"Noëlie"	"MPSI 2"	"Pompidou"	"Alexandre"	1995	M
"Noëlie"	"MPSI 2"	"Pompidou"	"Noëlie"	1996	F

Remarque 6. Si R est de card. m (ce qui signifie que la table de R a m lignes) et R' est de cardinal m' alors $R \times R'$ est de cardinal

Scholie : A première vue, on se dit que ce produit cartésien fabrique des tables monstrueuses, pas très utiles. Si elles peuvent effectivement devenir monstrueuses [2], elles sont au moins utiles du

2. et donc si possible à éviter du point de vue de l'efficacité informatique

point de vue *conceptuel*, car à partir de celles-ci on va pouvoir sélectionner, projeter, pour fabriquer une table qui résultera vraiment d'un lien qu'on veut établir entre deux relations. C'est le sens de l'opération de *jointure* qu'on va expliquer ci-dessous.

5.2 Opération de jointure

Définition : Soient $R_1 = R_1(S_1)$ et $R_2 = R_2(S_2)$ deux relations et si A est un attribut de S_1 et B est un attribut de S_2 ayant le même domaine alors on définit la jointure de R_1 et R_2 selon $A = B$, comme :

$$R_1 \underset{A=B}{\bowtie} R_2 = \{(e, e') \in R_1 \times R_2 | e.A = e'.B\}$$

De manière moins formelle : pour fabriquer cette jointure, on prend la table $R_1 \times R_2$ et on ne garde que les lignes telles que l'entrée correspondant à l'attribut A coïncide avec celle correspondant à l'attribut B.

Exemple On va utiliser la jointure pour relier les deux relations `eleve` et `eleve2` que nous avons constituées pour l'instant. Bien sûr on va considérer la relation :

`eleve3=eleve` $\underset{eleve.nom=eleve2.nom}{\bowtie}$ `eleve2`.

On obtient comme table de la relation :

'Alexis"	"MPSI 1"	"Jaures"	"Alexis"	1996	M
"Alexandre"	"MPSI 1"	"Daudet"	"Alexandre"	1995	M
"Noëlie"	"MPSI 2"	"Pompidou"	"Noëlie"	1996	F

A ce stade pour avoir une relation `eleve3` plus jolie, on peut effectuer une projection :

`eleve3=`$\pi_{nom1,classe,lyceeterm,datenaissance,sexe}$`(eleve 3)`

ce qui enlèvera la colonne inutile

Code SQL pour la jointure (deux méthodes) Vous aurez noté qu' en algèbre relationnelle, on a mis les préfixes `eleve.nom` et `eleve2.nom` pour la jointure.

(M1) En SQL, la jointure peut s'effectuer ainsi :

```
SELECT eleve.nom, eleve.classe, eleve.lyceeterm, eleve2.datenaissance,eleve2.sexe
FROM eleve, eleve2
WHERE eleve.nom=eleve2.nom -- la condition de jointure
```

Remarque 7. On peut aussi introduire des abréviations pour citer les tables qu'on utilise, comme suit :

```
SELECT e.nom, e.classe, e.lyceeterm, e2.datenaissance,e2.sexe -- projection
FROM eleve e, eleve2 e2 -- introduction des abréviations séparée du mot par une espace
WHERE e.nom=e2.nom -- vous avez compris que les deux -- désignent le début d'un commentaire
```

(M2) On peut aussi utiliser l'opérateur `JOIN`

```
SELECT eleve.nom, eleve.classe, eleve.lyceeterm, eleve2.datenaissance,eleve2.sexe
FROM eleve JOIN eleve2 ON eleve.nom=eleve2.nom -- on pourrait mettre un WHERE après
```

Remarque 8. A quoi bon avoir cette commande `JOIN` alors qu'on peut tout faire avec le `WHERE` ? Une raison est la *lisibilité* des scripts. Lorsqu'on a des scripts avec beaucoup de jointures et des sélections, il est bon de savoir ce qui est jointure et ce qui est sélection.

Remarque 9. (Algèbre relationnelle vs SQL)

• Dans le cadre de l'algèbre relationnelle, la jointure se déduit logiquement des autres opérations (produit cartésien, sélection).

• En revanche, pour les logiciels de gestion de base de donnée, la jointure est programmée de manière optimisée sans passer par le calcul du produit cartésien plus coûteux. Pour avoir le produit cartésien en SQL, il suffit d'enlever le `WHERE` dans le code précédent.

6 Fonctions et agrégation : calculs sur les tables en SQL

6.1 Un exemple élémentaire de fonction : calcul de moyenne

Disons qu'on dispose d'une table `elevenote` avec les attributs (nom, classe,note) :

TABLE elevenote | Recherche | Tout voir | Ajouter une nouvelle entrée | Duplicate | Éditer les entrées sélectionnées | Supprimer le

rowid	nom	prenom	classe	note
1	truc	nono	1	19
2	chose	lea	1	10
3	zz	atchoum	1	5
4	toujy	jojo	2	18
5	qv	zoro	2	5

Si on veut la moyenne (*average* en anglais) des notes des élèves de la classe 1 :

```
SELECT avg(note) FROM elevenote WHERE classe=1
```

Le résultat étant un nombre, on peut aussi utiliser le résultat pour faire des comparaisons :

```
SELECT * FROM elevenote WHERE note>=(SELECT avg(note) FROM elevenote)
```

6.2 Cinq fonctions à connaître :

Pour les exercices, on n'utilisera que les cinq fonctions suivantes :

comptage	COUNT
max	MAX
min	MIN
somme	SUM
moyenne	AVG

On a vu un exemple avec `AVG` ci-dessus. Un exemple avec `COUNT` qui va compter le nombre de lignes du tableau réalisant une condition, comme ceci par exemple si on veut le nombre d'élèves ayant une note en dessous de la moyenne (strictement), avec `COUNT(*)` ou bien `COUNT(nom)` c'est pareil ici.

Précision sur le COUNT :

a) La différence entre `COUNT(*)` et `COUNT(nom d'attribut)` : une ligne où l'attribut correspondant ne sera pas renseigné (case vide) ne sera pas comptée.

b) L'intérêt du `COUNT(DISTINCT attribut)` : on ne comptera que les lignes correspondant à des valeurs distinctes des attributs.

Voir T.P. pour des exemples.

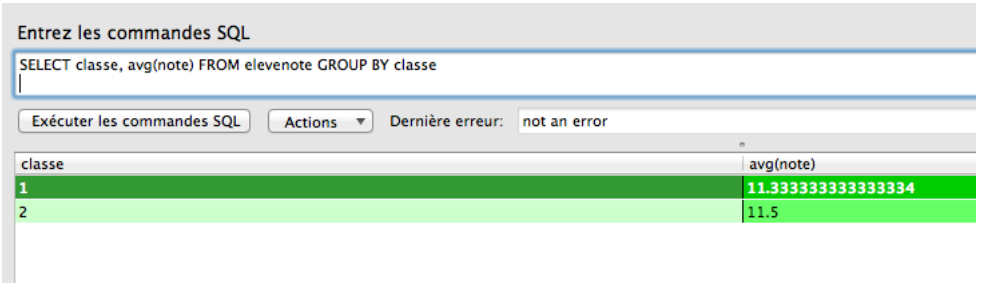
Remarque 10. Fonctions symétriques Toutes les fonctions qui s'appliquent à des tableaux sont symétriques i.e. le résultat ne change pas si on permute les variables en entrée.

6.3 Les agrégations : groupement `GROUP BY`

Dans l'exemple précédent, on veut maintenant faire la moyenne sur *chaque classe*. On doit donc *regrouper les élèves d'une même classe* puis faire une opération sur chacun de ces regroupements (ou agrégats).

En SQL cela donne :

```
SELECT classe, avg(note) FROM elevenote GROUP BY classe
```



On peut aussi donner un nom à la colonne `avg(note)` ainsi créée pour y faire référence par la suite, en utilisant l'instruction `AS` :

```
SELECT classe, avg(note) AS moyenne FROM elevenote GROUP BY classe
```

D'une manière plus générale : si on a une relation R avec (notamment) des attributs $A_1, \ldots, A_n$, et $B_1, \ldots, B_m$ et $f_1, \ldots, f_m$ des fonctions d'opération sur les attributs $B_1, \ldots, B_m$ respectivement, on peut :

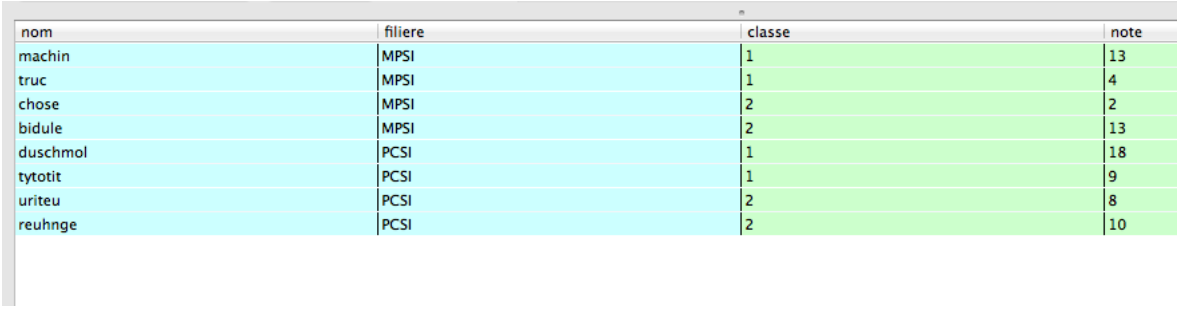
- regrouper les valeurs de R qui sont identiques sur les attributs $A_1, \ldots, A_n$
- puis définir des nouveaux attributs $f_i(B_i)$ pour ces valeurs regroupées.

En SQL la syntaxe sera :

```
SELECT A1,...,An, f1(B1),..,fn(Bm) FROM R GROUP BY A1,.., An
```

N.B. Comme indiqué dans la formule du cartouche : tous les attributs mis après le `GROUP BY` doivent aussi figurer dans le `SELECT`.

Voyons cela sur une table `NotesCB` plus compliquée :



nom	filiere	classe	note
machin	MPSI	1	13
truc	MPSI	1	4
chose	MPSI	2	2
bidule	MPSI	2	13
duschmol	PCSI	1	18
tytotit	PCSI	1	9
uriteu	PCSI	2	8
reuhnge	PCSI	2	10

Exercice :

(i) Pour avoir la moyenne des MPSI et des PCSI :

(ii) Pour avoir la moyenne de chaque classe au sens usuel "MPSI 1", etc...

(iii) Pour avoir la moyenne de la MPSI 1 et celle de la PCSI 1 :

(iv) Transition avec le paragraphe suivant : si on veut la liste des classes dont la moyenne est supérieure ou égale à 8 ?

```
SELECT * FROM  (SELECT filiere, classe, avg(note) AS moyenne
 FROM NotesCB GROUP BY filiere, classe)  WHERE moyenne >=8
```

6.4 Selection en aval : `HAVING`

Dans la dernière question de l'exercice précédent, on fait une requête avec une condition portant sur le *résultat* d'une fonction (ici `avg`). On a déjà rencontré cela à la fin du § 6.1, mais la différence ici est qu'on aurait envie d'écrire `WHERE avg(note)>8`. Mais `avg(note)` n'est pas une colonne de notre table à ce stade. On pourrait faire cela en deux temps, en créant une colonne `moyenne` comme expliqué au § 6.3.

Cela donne le résultat donné fin du (iv)

Une autre façon est d'utiliser le `HAVING` qui code ce qu'on appelle logiquement la *sélection en aval* (après l'application de la fonction).

Le code SQL correspondant est :

```
SELECT filiere, classe, avg(note) FROM NotesCB GROUP BY filiere,classe
HAVING avg(note)>8.
```

7 De l'importance des sous-requêtes

Le fait de pouvoir utiliser le résultat d'une requête dans une autre requête est évidemment essentiel. On notera qu'on a rencontré deux types de sous-requêtes :

7.1 Les sous-requêtes qui renvoient une nombre

Pour celles-ci, on peut utiliser leur résultat avec `=, >,>=,<=,<` ou même dans un calcul. C'était le cas avec

```
SELECT * FROM elevenote WHERE note>=(SELECT avg(note) FROM elevenote)
```

7.2 Les sous-requêtes qui renvoient une table

On utilise leur résultat avec `IN` ou bien `NOT IN`, ou bien avec `FROM` comme vu ci-dessus !

8 Memento SQL

8.1 Comparaison en SQL

— Les opérateurs de comparaisons mathématiques : `<,>,=,<>,>=,<=`
— Le test pour savoir si une case du tableau est *vide* : `IS NULL` ou `IS NOT NULL`
— Le test pour savoir si un élément est dans un tableau `IN`

8.2 Opérations ensemblistes

`UNION, INTERSECT, EXCEPT` à mettre en perspective avec `OR, AND, WHERE NOT, WHERE NOT IN`.

8.3 Jointure

`FROM table1, table2 WHERE table1.truc=table2.chose` ou bien
`FROM table1 JOIN table2 ON table1.truc=table2.chose`

8.4 Fonctions de calculs par agrégation

`COUNT, MAX, MIN, SUM, AVG`, à utiliser avec des GROUP BY si on ne veut pas tout compter.

8.5 Présentation des résultats :

8.5.1 SELECT DISTINCT

A la suite d'une projection, il se peut qu'en SQL, on ait plusieurs fois la même ligne qui apparaît. Exemple ?
Si on veut éviter cela, on peut utiliser SELECT DISTINCT.

8.5.2 ORDER BY

Si l'on veut afficher la table `elevenote` du § 6.3 par notes décroissantes, on peut faire.

```
SELECT * FROM NotesCB ORDER BY note DESC -- sans DESC l'ordre est croissant par défaut.
```

8.5.3 LIMIT

On peut demander de ne pas afficher plus de N valeurs avec `LIMIT N` à la fin de la requête.

Table des matières