BDD1

Initiation aux bases de données

Partie I

Limite des structures de données plates pour la recherche d'informations

1 Un petit exemple

Au lycée Kléber, chaque classe de prépa est identifiée par un couple unique (filière,numéro). De plus, chaque classe admet une équipe professorale qui lui est propre. Si l'on voulait stocker ces informations, on pourrait se dire qu'il suffirait d'une simple tableau comme par exemple le suivant qui stocke dans une liste, l'ensemble des informations nécessaires à la définition d'une classe, c'est-à-dire sa filière, son numéro et la liste des professeurs où l'on stocke des doublets (nom,matière).

De la sorte, il est très simple de déterminer l'équipe professorale d'une classe donnée connaissant sa filière et son numéro (complexité linéaire dans le nombre de classes). En revanche, il devient plus difficile de rassembler tous les professeurs d'une matière donnée (par exemple pour prévenir tous les professeurs de mathématiques de la tenue d'olympiades dans leur matière). Comme on le voit, la complexité sera globalement quadratique (nombre de classes que multiplie le nombre de matières dans chaque classe)

```python
lycee = [('PCSI',1,[('E. Bougnol',  'Mathematiques'),
                    ('JJ Fleck',    'Physique'),
                    ('JJ Fleck',    'Informatique'),
                    ('S. Velikonia','Francais')]),
         ('PCSI',3,[('M. Kostyra',  'Mathematiques'),
                    ('M. Heckmann', 'Physique'),
                    ('JJ Fleck',    'Informatique'),
                    ('S. Velikonia','Francais')]),
         ('MPSI',1,[('F. Cuvellier','Mathematiques'),
                    ('F. Cuvellier','Informatique'),
                    ('T. Meyer',    'Physique')])
         ]

def recherche_profs_classe(filiere,numero):
    for classe in lycee:
        # La filière est le premier élément stocké, le numéro est le second
        if classe[0] == filiere and classe[1] == numero:
            # Et la liste des profs/matières est le troisième. On la renvoie
            # directement car on suppose (logiquement) que les couples
            # (filiere,numero) sont uniques.
            return classe[2]
```

```
22
23 def recherche_profs_de(matiere):
24     L = []  # Pour le stockage
25     for classe in lycee:            # On itère sur la liste des classes
26         for prof in classe[2]:      # puis sur les profs de chaque classe
27             if prof[1] == matiere: # Si on trouve la bonne matière
28                 L.append(prof[0])  # on stocke le nom du prof
29     return L
```

```
>>> recherche_profs_classe('PCSI',1)
[('E. Bougnol', 'Mathematiques'), ('JJ Fleck', 'Physique'), ('JJ Fleck', 'Informatique')
>>> recherche_profs_de('Mathematiques')
['E. Bougnol', 'M. Kostyra', 'F. Cuvellier']
```

On privilégie naturellement un type de recherche et complique un autre type. Si on réorganise les données par matières par exemple, la seconde recherche sera facilitée alors que la première sera compliquée... En bref, il faudrait trouver une autre manière de faire les choses.

2 Le problème du stockage des données

On n'a jusqu'à présent parlé que de la *structure* des données et non pas de leur stockage physique sur le disque. On pourrait se dire que le plus simple serait de stocker le tout dans un fichier texte à raison d'un ligne par entrée, mais....

- Le format des fichiers n'est pas standardisé (chacun va créer le sien, par exemple en ce qui concerne les séparateurs)
- Il y aura une possible redondance des données (un professeur peut agir dans plusieurs classes) avec les difficultés de cohérence inhérents (mauvaise orthographe sur une ligne et voici encore un prof qui devient schizophrène !)
- Il va falloir écrire un programme (ou une fonction) spécifique à chaque type d'interrogation et il est impossible de changer la structure du stockage sans casser tous ces programmes.
- Partage et sécurisation des données sont difficiles. Il faut s'occuper de la gestion des pannes soi-même. Etc.

3 Les SGBD à la rescousse !

SGBD = Système de Gestion de Base de Données. Ils ont plusieurs avantages substanciels:

- Indépendance physique
  - Possibilité de modifier les structures de stockage sans modifier les programmes
  - Écriture des applications par des non-spécialistes des fichiers
  - Meilleure portabilité, indépendance vis-à-vis du matériel
- Indépendance logique
  - Possibilité d'ignorer les données d'autres applications
  - Possibilité d'enrichir les applications sans devoir tout réécrire
  - Possibilité de protéger (rendre confidentielles) certaines données
- Manipulation aisée par le biais d'un langage déclaratif (SQL, Structured Query Language)
- Conception aisée par le biais de méthodes utilisant des modèles simples à manipuler (Modèle Entité/Association)
- Exécution et optimisation
  - Les requêtes sont traduites en un langage procédural (algèbre relationnelle)
  - ... qui peut être optimisé automatiquement. (des années de recherche en BD...)

- Intégrité logique (Contraintes d'intégrité)
  - Par le biais d'un langage déclaratif
  - Détection de mises à jour erronées
  - Contrôle sur les données élémentaires et les relations
- Intégrité physique
  - Tolérance aux pannes (transactions, système (panne de courant), disque (crash))
- Partage de données
  - Isolation (chacun a l'air d'être seul)
  - Concurrence (tout le monde peut agir en même temps)
- Confidentialité
- Standardisation

Partie II

Bases de données relationnelles

1 Définitions

Domaine: type (c'est-à-dire entier, flottant, chaîne de caractère ou autre) des données qui seront stockées dans une colonne de la table considérée

Attribut: Couple nom/domaine qui permet de définir une colonne. Par exemple `Nom:String` ou `Age:Int`, etc.

Variable de relation: Ensemble ordonné d'attributs (sert d'entête à une table ou « relation »).
Par exemple `(Nom:String,Age:Int,Adresse:String)`

Tuple: Ensemble ordonné de couples attribut/valeur (ligne).
Par exemple `(Nom:Trump,Age:70,Adresse:WhiteHouse)`

Relation: Ensemble de tuples, soit une table en SQL

Nom	Age	Adresse
Trump	70	WhiteHouse
Macron	39	Élysée
Poutine	64	Kremlin

Schéma relationnel: Descriptif qui associe à chaque relation (=table) la liste de ses attributs:

```
COURS    (NC:int, CODE_COURS:string, INTITULE:string)
ETUDIANT (NE:int, NOM:string, PRENOM:string)
INSCRIT  (NE:int, NC:int, ANNEE:int)
```

2 Notion de clé primaire

Une clé de relation est un groupe d'attributs qui détermine un **tuple unique** dans une relation.

Dans un SGBD (Système de Gestion de Base de Données), pour chaque relation, une clé (la plus simple possible) est choisie et est appelée **clé primaire** de la relation. C'est particulièrement utile pour toutes les problématiques d'indexation (pour retrouver facilement une ligne donnée dans toute la table) et pour les jointures (la jonction de deux tables qui partagent un même attribut).

Dans les exemples précédents, `NC` est une clé pour la table `COURS` (représente le « numéro » du cours) alors que `NE` en est une pour la table `ETUDIANT`. En revanche, il n'existe pas de clé mono-attribut pour la table `INSCRIT`, même si l'ensemble des trois attributs devrait normalement constituer une clé (un même étudiant ne devrait pas pouvoir s'inscrire plusieurs fois au même cours une année donnée).

À noter qu'en SQL, on peut imposer qu'un attribut[1] soit considéré comme clé primaire de la table. De la sorte, la base de données refusera tout simplement qu'une nouvelle ligne soit créée avec une valeur redondante de la colonne (par exemple si on essaie de créer un étudiant qui aurait le même « numéro étudiant » qu'un autre).

Notation: en général, on souligne l'attribut choisi pour clé primaire lors de la déclaration d'une table dans un schéma relationnel. L'exemple précédent devient alors

```
COURS    (NC:int, CODE_COURS:string, INTITULE:string)
ETUDIANT (NE:int, NOM:string, PRENOM:string)
INSCRIT  (NE:int, NC:int, ANNEE:int)
```

À noter que ni `NE`, ni `NC` ne sont des clés primaires pour la table `INSCRIT` car un même étudiant peut être inscrit à plusieurs cours et un même cours rassemble (généralement !) plusieurs étudiants. En revanche, elles sont ce qu'on appelle des « clés étrangères », c'est-à-dire qu'elles pointent sur la clé primaire d'une autre table, ce qui permet d'établir des jointures[2] entre les différentes tables.

Partie III

Opérateurs sur le modèle relationnel

Dans tous les exemples qui suivent, nous présenterons successivement les notions abstraites, puis leur traduction en langage de l'algèbre relationnelle (notation mathématique permettant de s'affranchir des problèmes de syntaxe) et enfin leur traduction en langage SQL. Le programme spécifie que vous devez connaître (donc savoir définir et utiliser) les notions abordées et notamment traduire en langage de l'algèbre relationnelle et en langage SQL des requêtes simples du langage courant.

Notons dès à présent que le langage SQL, contrairement à Python, n'est pas sensible à la casse (c'est-à-dire aux majuscules et aux minuscules), mais par convention, on choisit de donner les instructions du langage en majuscules (comme `SELECT`, `FROM`, `WHERE`, etc.) alors que ce qui est du ressort des tables (autant les attributs que les noms des tables) en minuscules.

Notons enfin que SQL est un langage *déclaratif*, c'est-à-dire que l'on dit au SGBD ce que l'on veut faire, mais on ne lui dit pas (vraiment) *comment* il doit le faire. En interne, l'interpréteur va s'arranger pour optimiser la requête pour obtenir le résultat en un minimum de temps. Les notions de complexité abordées dans les précédents chapitre d'algorithmique n'ont donc pas vraiment de sens puisque même une requête « mal formulée » peut s'exécuter très rapidement grâce à ce qui se passe « sous le capot » (et que vous ne voulez pas connaître...)

1 Projection

Considérons la table `bbc` suivante qui contient certaines informations sur les pays du monde

```
SELECT * FROM bbc ;
```

nom	region	surface	population	PIB
Afghanistan	South Asia	652 225	26 000 000	NULL
Albania	Europe	28 728	3 200 000	6 656 000 000
Algeria	Middle East	2 400 000	32 900 000	75 012 000 000
...				

[1] Unique, d'où parfois la croyance qu'une clé primaire est forcément mono-attribut.
[2] Cf plus loin

Parmi tous les attributs de la relation, certains ne nous intéressent pas forcément. On va donc projeter la relation précédente sur certaines colonnes ou mélange de colonnes. Dans le langage de l'algèbre relationnelle, cette projection d'une table `table` sur certaines colonnes `colonnes` se note

$$\pi_{\texttt{colonnes}}(\texttt{table})$$

Ici, par exemple, on voudrait n'avoir que le nom du pays, la région du monde concernée et le nombre d'habitant au km^2. On va donc procéder à la projection $\pi_{\text{nom,region,population/surface}}(\texttt{bbc})$. En langage SQL, cela s'écrit

```
SELECT nom,region,population/surface FROM bbc ;
```

nom	region	population/surface
Afghanistan	South Asia	39
Albania	Europe	111
Algeria	Middle East	13
Andorra	Europe	136
Angola	Africa	11
...		

2 Sélection (ou restriction)

Très souvent, seule une partie de l'ensemble des lignes d'une table nous intéresse, on va donc sélectionner les lignes intéressantes d'après une certaine contrainte. En langage de l'algèbre relationnelle, une telle sélection se note

$$\sigma_{\texttt{contrainte}}(\texttt{table})$$

La contrainte peut être de toute sorte, comme une vérification d'égalité (`nom="France"`) ou une inégalité (`surface<100`). Par exemple, si l'on veut sélectionner tous les pays dont la densité surfacique de population est supérieure à 200 hab/km^2, cela s'écrirait $\sigma_{\text{population/surface}>200}(\texttt{bbc})$. Et en langage SQL, il suffit d'utiliser le mot-clé « `WHERE` »:

```
SELECT * from bbc WHERE population/surface>200 ;
```

nom	region	surface	population	PIB
Bahrain	Middle East	717	754 000	9 357 140 000
Bangladesh	South Asia	143 998	152 600 000	67 144 000 000
Barbados	Americas	430	272 000	2 518 720 000
Belgium	Europe	30 528	10 300 000	319 609 000 000
Burundi	Africa	27 816	7 300 000	NULL
...				

Bien entendu, très souvent, une sélection s'accompagne d'une projection. Si l'on combine les deux opérations suivantes, cela donne

$$\pi_{\texttt{nom,region,population/surface}}(\sigma_{\texttt{population/surface>200}}(\texttt{bbc}))$$

```
SELECT nom,region,population/surface FROM bbc WHERE population/surface>200 ;
```

nom	region	population/surface
Bahrain	Middle East	1051
Bangladesh	South Asia	1059
Barbados	Americas	632
Belgium	Europe	337
Burundi	Africa	262
...		

On peut avoir des requêtes plus restrictives en imposant des intervalles (pour éviter d'imposer un « supérieur à » puis un « inférieur à »)

```
SELECT nom,region,population/surface,pib FROM bbc
       WHERE population/surface BETWEEN 10 AND 20 ;
```

nom	region	population/surface	PIB
Algeria	Middle East	13	75 012 000 000
Angola	Africa	11	14 935 000 000
Argentina	South America	14	146 196 000 000
Belize	Americas	11	NULL
Equatorial Guinea	Africa	18	484 530 000
...			

ou encore faire un test de valeur manquante (`IS NULL`) ou justement non manquante (`IS NOT NULL`):

```
SELECT nom,region,population/surface,pib FROM bbc
       WHERE population/surface BETWEEN 10 AND 20
         AND pib IS NULL ;
```

nom	region	population/surface	PIB
Belize	Americas	11	NULL
Paraguay	South America	15	NULL
Somalia	Africa	16	NULL
Turkmenistan	Asia-Pacific	10	NULL
Uruguay	South America	19	NULL

D'autres opérations sont encore possibles comme la vérification d'appartenance à un autre ensemble[3] (mot-clé: `IN`), vérifier s'il existe une donnée ayant certaines propriétés (mot-clé: `EXISTS`), vérifier comment des données se comparent à toutes celles d'une autre table (mot-clé: `ALL`) ou seulement si la comparaison tient pour l'une ou l'autre des données d'une autre table (mot-clé: `ANY` ou `SOME`). Pour plus de détails sur ces possibilités, on pourra par exemple consulter le site `http://sql.sh`

[3] qui peut-être une sélection.

3 Renommage

Il peut être pénible d'emporter des colonnes définies de manière complexe à partir d'autres colonnes (ou tout simplement nommées avec des noms qui ne sont pas très parlants). On définit donc l'opération de renommage d'un attribut A en B sous la forme

$$\rho_{\mathrm{A}\to\mathrm{B}}(\texttt{table})$$

Dans la requête précédente, on pourrait choisir de baptiser `densite` la quantité `population/surface`, de sorte que l'on puisse la réécrire

$$\rho_{\texttt{population/surface}\to\texttt{densite}}(\pi_{\texttt{nom,region,population/surface}}(\sigma_{\texttt{population/surface>200}}(\texttt{bbc})))$$

L'intérêt d'une telle chose se ressent surtout dans l'écriture dans le langage SQL avec l'utilisation du mot-clé « `AS` ». Dans l'exemple suivant, on rajoute la clause « `ORDER BY` » qui permet de trier les résultats suivant l'ordre (par défaut ascendant) défini sur certaines colonnes (ici la colonne renommée en ordre descendant, d'où l'usage du mot clé « `DESC` »)

```
SELECT nom,region,population/surface AS densite FROM bbc
WHERE densite>200 ORDER BY densite DESC ;
```

nom	region	densite
Monaco	Europe	16 000
Singapore	Asia-Pacific	6 666
Malta	Europe	1256
The Maldives	South Asia	1134
Bangladesh	South Asia	1059
Bahrain	Middle East	1051
Barbados	Americas	632
...		

4 Opérateurs usuels sur les ensembles

Le résultat d'une requête est toujours un ensemble de lignes. Et ces lignes peuvent être arrangées avec les opérateurs ensemblistes que sont l'union, l'intersection et la différence. Pour illustrer ces trois concepts, considérons les deux requêtes suivantes (Exercices: traduire ce que font ces requêtes en langue française)

```
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE rayon_atomique < 100 ;
```

Z	Symb.	Nom	M (g/mol)	ρ (g/cm^3)	r_{atomique} (pm)	r_{ionique} (pm)	r_{covalent} (pm)
1	H	Hydrogen	1.00794	0.0708	79	154	32
2	He	Helium	4.002602	0.147	0	93	NULL
5	B	Boron	10.811	2.34	98	23	82
6	C	Carbon	12.011	2.25	91	NULL	77
7	N	Nitrogen	14.00674	0.808	92	NULL	75

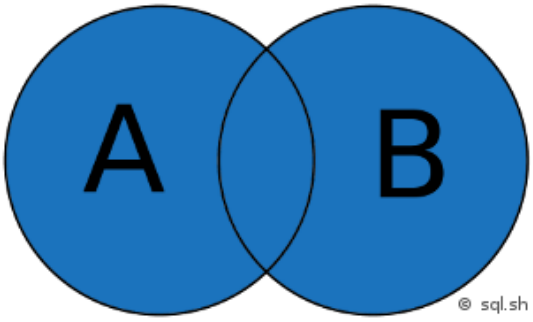
```
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE masse_volumique < 1 ;
```

Z	Symb.	Nom	M (g/mol)	ρ (g/cm^3)	r_{atomique} (pm)	r_{ionique} (pm)	r_{covalent} (pm)
1	H	Hydrogen	1.00794	0.0708	79	154	32
2	He	Helium	4.002602	0.147	0	93	NULL
3	Li	Lithium	6.941	0.534	155	68	163
7	N	Nitrogen	14.00674	0.808	92	NULL	75
11	Na	Sodium	22.989768	0.971	190	97	154
19	K	Potassium	39.0983	0.856	235	133	203

Pour des simplicités d'écriture, on supposera que la table `periodic` se limite aux colonnes présentées ci-dessus (même si elle est en vrai plus étendue), ce qui permettra d'écrire les deux requêtes en langage de l'algèbre relationnelle sous la forme

$$\text{A} = \sigma_{r_{\text{atomique}}<100}(\texttt{periodic}) \qquad \text{et} \qquad \text{B} = \sigma_{\rho<1}(\texttt{periodic})$$

a) Union $\text{A} \cup \text{B}$



L'union de deux ensembles renvoie tous les éléments qui appartiennent à l'un **ou** à l'autre de ces ensembles. Si un élément appartient aux deux à la fois, il n'est renvoyé qu'une seule fois. En langage SQL, on utilise le mot-clé « `UNION` » pour rassembler deux requêtes distinctes (qui doivent donc être compatibles, c'est-à-dire renvoyer le même nombre de colonnes avec les mêmes types et dans le même ordre), ou alors on peut utiliser l'instruction « `OR` » lors de la sélection

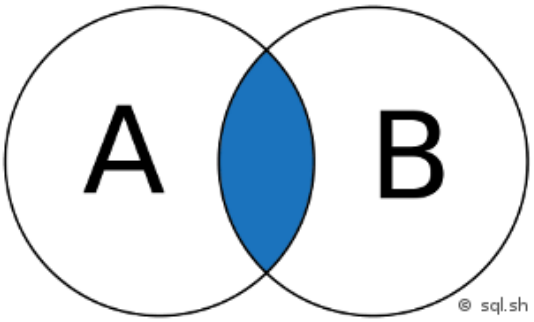
```
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE rayon_atomique < 100 OR masse_volumique < 1;
```

Z	Symb.	Nom	M (g/mol)	ρ (g/cm^3)	r_{atomique} (pm)	r_{ionique} (pm)	r_{covalent} (pm)
1	H	Hydrogen	1.00794	0.0708	79	154	32
2	He	Helium	4.002602	0.147	0	93	NULL
3	Li	Lithium	6.941	0.534	155	68	163
5	B	Boron	10.811	2.34	98	23	82
6	C	Carbon	12.011	2.25	91	NULL	77
7	N	Nitrogen	14.00674	0.808	92	NULL	75
11	Na	Sodium	22.989768	0.971	190	97	154
19	K	Potassium	39.0983	0.856	235	133	203

```
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE rayon_atomique < 100
UNION
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE masse_volumique < 1 ;
```

Z	Symb.	Nom	M (g/mol)	ρ (g/cm^3)	r_{atomique} (pm)	r_{ionique} (pm)	r_{covalent} (pm)
1	H	Hydrogen	1.00794	0.0708	79	154	32
2	He	Helium	4.002602	0.147	0	93	NULL
3	Li	Lithium	6.941	0.534	155	68	163
5	B	Boron	10.811	2.34	98	23	82
6	C	Carbon	12.011	2.25	91	NULL	77
7	N	Nitrogen	14.00674	0.808	92	NULL	75
11	Na	Sodium	22.989768	0.971	190	97	154
19	K	Potassium	39.0983	0.856	235	133	203

b) **Intersection** A ∩ B



L'intersection de deux ensembles renvoie tous les éléments qui appartiennent à l'un **et** à l'autre de ces ensembles. En langage SQL, on utilise le mot-clé « `INTERSECT` » pour rassembler deux requêtes distinctes (qui doivent donc être compatibles, c'est-à-dire renvoyer le même nombre de colonnes avec les mêmes types et dans le même ordre), ou alors on peut utiliser l'instruction « `AND` » lors de la sélection

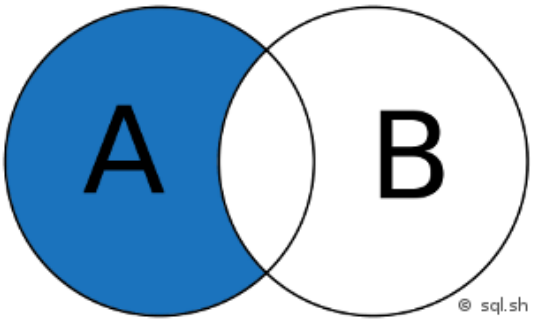
```
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE rayon_atomique < 100
INTERSECT
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE masse_volumique < 1 ;
```

Z	Symb.	Nom	M (g/mol)	ρ (g/cm^3)	r_{atomique} (pm)	r_{ionique} (pm)	r_{covalent} (pm)
1	H	Hydrogen	1.00794	0.0708	79	154	32
2	He	Helium	4.002602	0.147	0	93	NULL
7	N	Nitrogen	14.00674	0.808	92	NULL	75

```
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE rayon_atomique < 100 AND masse_volumique < 1;
```

Z	Symb.	Nom	M (g/mol)	ρ (g/cm^3)	r_{atomique} (pm)	r_{ionique} (pm)	r_{covalent} (pm)
1	H	Hydrogen	1.00794	0.0708	79	154	32
2	He	Helium	4.002602	0.147	0	93	NULL
7	N	Nitrogen	14.00674	0.808	92	NULL	75

c) Différence A − B (ou A\B)



La différence A − B de deux ensembles (aussi notée A\B) revient à renvoyer les éléments qui appartiennent à A mais pas à B (ou « à l'exception de ceux qui appartiennent aussi à B »). En langage SQL, le mot-clé utilisé dépend du SGBD utilisé. On utilise soit le mot-clé « `MINUS` » (MySQL, Oracle), soit le mot-clé « `EXCEPT` » (PostgreSQL, SQLite). Là encore, les requêtes doivent être compatibles (c'est-à-dire renvoyer le même nombre de colonnes avec les mêmes types et dans le même ordre) et on peut souvent s'en tirer en arrangeant les conditions dans l'instruction `WHERE` (ici, il suffit d'inverser la condition sur ρ).

```
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE rayon_atomique < 100
EXCEPT
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE masse_volumique < 1 ;
```

Z	Symb.	Nom	M (g/mol)	ρ (g/cm^3)	r_{atomique} (pm)	r_{ionique} (pm)	r_{covalent} (pm)
5	B	Boron	10.811	2.34	98	23	82
6	C	Carbon	12.011	2.25	91	NULL	77

```
SELECT numero_atomique,symbole,nom,
       masse_atomique,masse_volumique,
       rayon_atomique,rayon_ionique,rayon_covalent FROM periodic
       WHERE rayon_atomique < 100 AND masse_volumique > 1;
```

Z	Symb.	Nom	M (g/mol)	ρ (g/cm^3)	r_{atomique} (pm)	r_{ionique} (pm)	r_{covalent} (pm)
5	B	Boron	10.811	2.34	98	23	82
6	C	Carbon	12.011	2.25	91	NULL	77

5 Produit cartésien

Le produit cartésien a été défini en mathématiques. Si l'on prend deux ensembles $A = \{1, 2\}$ et $B = \{3, 4\}$, alors le produit cartésien $A \times B$ s'écrit $\{(1, 3), (1, 4), (2, 3), (2, 4)\}$ c'est-à-dire que l'on prend chaque élément de A et on l'apparie à tous les éléments de B. Bien sûr, avec des ensembles de taille conséquente, le nombre d'éléments du produit cartésien croît très très vite (si A est contient a éléments et B contient b éléments, alors $A \times B$ contient $a \times b$ éléments).

Supposons à présent que l'on dispose de deux tables `etudiants` et `villes` du type

Nom	Année de naissance	Ville
Alice	1997	Strasbourg
Bob	1998	Paris
Eve	1995	Strasbourg

et

NomV	Département
Strasbourg	67
Paris	75

Le produit cartésien de ces deux tables est noté simplement en algèbre relationnelle `etudiants` $\times$ `villes`. Quant à la traduction en SQL, elle revient simplement à requérir toutes les colonnes depuis les deux tables.

```
SELECT * FROM etudiants,villes ;
```

Nom	Année de Naissance	Ville	NomV	Département
Alice	1997	Strasbourg	Strasbourg	67
Alice	1997	Strasbourg	Paris	75
Bob	1998	Paris	Strasbourg	67
Bob	1998	Paris	Paris	75
Eve	1995	Strasbourg	Strasbourg	67
Eve	1995	Strasbourg	Paris	75

On voit bien que dans cet exemple, le produit cartésien n'a pas énormément de sens car certains étudiants se retrouvent avec deux villes différentes. Il va donc falloir faire une sélection parmi les résultats précédents, ce qui correspond très exactement à l'action de la jointure.

6 Jointure

La jointure revient à associer les lignes d'une table aux lignes d'une autre table en se basant sur la correspondance de valeurs dans deux colonnes. Dans l'exemple ci-dessus, on va vouloir joindre les tables `etudiants` et `villes` de manière à avoir, en plus du nom de la ville, l'information sur le département dans lequel se situe la ville (information qu'il est inutile de dupliquer 36 fois dans la table `etudiants`). Cela revient donc à faire un produit cartésien des deux tables suivi d'une sélection sur les lignes où il y a correspondance entre la colonne `Ville` de la table `etudiants` et la colonne `NomV` de la table `villes`.

Nom	Année de Naissance	Ville	NomV	Département
Alice	1997	Strasbourg	Strasbourg	67
~~Alice~~	~~1997~~	~~Strasbourg~~	~~Paris~~	~~75~~
~~Bob~~	~~1998~~	~~Paris~~	~~Strasbourg~~	~~67~~
Bob	1998	Paris	Paris	75
Eve	1995	Strasbourg	Strasbourg	67
~~Eve~~	~~1995~~	~~Strasbourg~~	~~Paris~~	~~75~~

En langage de l'algèbre relationnelle, la jointure des tables `table1` et `table2` selon une certaine contrainte sur les lignes a sa notation propre qui, selon les auteurs, peut s'écrire `table1[contrainte]table2` ou $\texttt{table1} \bowtie_{\texttt{contrainte}} \texttt{table2}$. Ainsi, en la regardant comme une sélection sur un produit cartésien, il vient

$$\texttt{table1} \bowtie_{\texttt{contrainte}} \texttt{table2} = \texttt{table1[contrainte]table2} = \sigma_{\texttt{contrainte}}(\texttt{table1} \times \texttt{table2})$$

Sur notre exemple avec les tables `etudiants` et `villes`, cela donnerait

$$\texttt{etudiants[etudiants.Ville} = \texttt{villes.NomV]villes} = \sigma_{\texttt{etudiants.Ville} = \texttt{villes.NomV}}(\texttt{etudiants} \times \texttt{villes})$$

ou peut-être de manière un peu plus légère en supposant que le premier champ vienne de la table de gauche et le second de la table de droite:

$$\texttt{etudiants} \bowtie_{\texttt{Ville} = \texttt{NomV}} \texttt{villes} = \texttt{etudiants[Ville} = \texttt{NomV]villes} = \sigma_{\texttt{Ville} = \texttt{NomV}}(\texttt{etudiants} \times \texttt{villes})$$

En SQL, on peut utiliser la construction `JOIN ... ON ...` ou alors la solution en terme de sélection sur un produit cartésien `SELECT * FROM table1,table2 WHERE ...`. Dans nos exemples, cela donne

```
SELECT * FROM etudiants,villes WHERE etudiants.Ville = villes.NomV ;
```

Nom	Année de Naissance	Ville	NomV	Département
Alice	1997	Strasbourg	Strasbourg	67
Bob	1998	Paris	Paris	75
Eve	1995	Strasbourg	Strasbourg	67

```
SELECT * FROM etudiants JOIN villes ON etudiants.Ville = villes.NomV ;
```

Nom	Année de Naissance	Ville	NomV	Département
Alice	1997	Strasbourg	Strasbourg	67
Bob	1998	Paris	Paris	75
Eve	1995	Strasbourg	Strasbourg	67

Bien sûr, on peut aussi rajouter une projection pour éviter d'avoir le champ commun présent deux fois

```
SELECT nom,naissance,ville,dpt
FROM etudiants JOIN villes ON Ville = NomV ;
```

Nom	Année de Naissance	Ville	Département
Alice	1997	Strasbourg	67
Bob	1998	Paris	75
Eve	1995	Strasbourg	67

Remarque: si tous les champs ont des noms différents, il n'est pas nécessaire de préciser de quelle table provient l'information, mais si jamais les noms des attributs ont été mal choisis (par exemple `Nom` et non `NomV` pour les villes), il faut préciser de quelle table on parle.

```
SELECT etudiants.nom,naissance,ville,dpt
FROM etudiants JOIN villes ON ville = villes.nom ;
```

Nom	Année de Naissance	Ville	Département
Alice	1997	Strasbourg	67
Bob	1998	Paris	75
Eve	1995	Strasbourg	67

7 Division cartésienne

La division cartésienne est le pendant du produit cartésien, tout comme la division euclidienne est le pendant du produit sur des entiers: si on note R et R′ deux relations (ou tables), alors R÷R′ est la « plus grande » relation (au sens de l'inclusion) telle que le produit cartésien avec R′ redonne (presque) R. Formellement, c'est la plus grande relation telle qu'il existe une relation R″ vérifiant

$$[(\mathrm{R}\div\mathrm{R}')\times\mathrm{R}']\cup\mathrm{R}''=\mathrm{R}\qquad\text{et}\qquad[(\mathrm{R}\div\mathrm{R}')\times\mathrm{R}']\cap\mathrm{R}''=\varnothing$$

Comme il n'existe pas en SQL d'opérateur permettant de l'implémenter directement, ce n'est pas la peine de se casser la tête à en implémenter un. Ce sera de toutes façons bien plus facile de répondre à des requêtes posées en langage courant sur de vraies tables que d'essayer d'en faire une architecture abstraite.

8 Fonctions d'agrégation

On a vu avec l'exemple de la densité de population qu'il était assez facile de calculer des quantités « sur une ligne » en combinant les informations que l'on peut y trouver. Néanmoins, il peut être intéressant de calculer des quantités « sur une colonne », comme par exemple déterminer la valeur moyenne des populations des pays du monde. Mieux, on peut être tenté de calculer cette moyenne, *en fonction* de la région du monde. Il va donc falloir définir des « groupes » de pays à l'aide de l'instruction « `GROUP BY` ». Un exemple valant mieux qu'un long discours

```
SELECT region,AVG(population) AS pop FROM bbc GROUP BY region ORDER BY pop;
```

Région	Population moyenne
Americas	3 716 450,0
Africa	14 852 802,659 6
Europe	17 404 680,851 1
Middle East	19 678 000,0
South America	30 992 500,0
Asia-Pacific	60 383 108,333 3
North America	144 466 666,667
South Asia	186 017 250,0

En notation de l'algèbre relationnelle, il faut traduire à la fois le `GROUP BY` et la fonction d'agrégation. Ici, la notation serait $\pi_{\texttt{groupe}}\gamma_{\text{fonction d'agrégation}}(\texttt{table})$. En particulier, la dernière requête s'écrirait

$$\pi_{\texttt{region}}\gamma_{\texttt{AVG(population)}}(\texttt{bbc})$$

D'autres fonctions d'agrégations sont disponibles telles que `MIN()`, `MAX()`, `COUNT()` et `SUM()`. On peut résumer tous les usages en une seule requête:

$\pi_{\texttt{region}}\gamma_{\texttt{COUNT()}, \texttt{AVG(population)}, \texttt{MAX(population)}, \texttt{MIN(population)}, \texttt{SUM(population)}}(\texttt{bbc})$

```
SELECT region,COUNT() AS nb,ROUND(AVG(population)),
MAX(population),MIN(population),SUM(population) AS tot
FROM bbc GROUP BY region ORDER BY tot ;
```

Région	Nb pays	Population moyenne	Maximale	Minimale	Total région
Americas	20	3 716 450,0	13 000 000	46 000	74 329 000
South America	12	30 992 500,0	182 800 000	442 000	371 910 000
Middle East	19	19 678 000,0	74 900 000	628 000	373 882 000
North America	3	144 466 667,0	295 000 000	32 000 000	433 400 000
Africa	47	14 852 803,0	130 200 000	76 000	698 081 725
Europe	48	17 404 681,0	141 500 000	27 000	818 020 000
South Asia	8	186 017 250,0	1 100 000 000	338 000	1 488 138 000
Asia-Pacific	36	60 383 108,0	1 300 000 000	9 900	2 173 791 900

La même chose sur le monde entier pour vérifier la « fraîcheur » de la base en question

```
SELECT COUNT(),ROUND(AVG(population)),
MAX(population),MIN(population),SUM(population)
FROM bbc ;
```

Nb pays	Population moyenne	Maximale	Minimale	Total mondial
193	33 497 670,0	1 300 000 000	9 900	6 431 552 625

Visiblement la version de la base de la BBC date un peu puisque nous sommes plus de 7 milliard d'individus sur Terre depuis octobre 2011. Et une dernière pour la route

```
SELECT region,ROUND(SUM(population)/SUM(surface)) AS pop,
              ROUND(AVG(population/surface)) FROM bbc
       GROUP BY region ORDER BY pop;
```

Région	Population moyenne/km^2	Moyenne population/km^2
North America	20.0	29.0
South America	20.0	20.0
Middle East	27.0	125.0
Africa	30.0	81.0
Europe	34.0	479.0
Asia-Pacific	75.0	320.0
Americas	101.0	186.0
South Asia	301.0	415.0

9 Requête agrégative

Pour imposer une contrainte alors que l'on a utilisé une fonction d'agrégation, il faut utiliser le mot-clé « `HAVING` » qui s'utilise comme `WHERE` mais permet d'inclure des fonctions d'agrégation. En effet, la condition demande à la base que la table entière ait été lue préalablement à l'évaluation sur chaque ligne, d'où l'usage d'un mot-clé différent.

On peut à présent se poser des question du type: pour chaque région, quel est le pays le plus peuplé ?

```
SELECT region,population,nom
FROM bbc GROUP BY region HAVING MAX(population) = population ;
```

Région	Maximum	Pays Max
Africa	130 200 000	Nigeria
Americas	13 000 000	Guatemala
Asia-Pacific	1 300 000 000	China
Europe	141 500 000	Russia
Middle East	74 900 000	Egypt
North America	295 000 000	United States of America
South America	182 800 000	Brazil
South Asia	1 100 000 000	India

Et finalement, pour récupérer à la fois le plus peuplé et le plus densément peuplé de chaque région, on utilise des sous-requêtes qui renvoient chacune des tables complètes sur lesquelles on fait une jointure avec les regions.

```
SELECT r1,m1,p1,m2,p2
FROM (SELECT region AS r1,population AS m1,nom AS p1 FROM bbc
      GROUP BY region HAVING MAX(population) = population)
   JOIN
     (SELECT region AS r2,population/surface AS m2,nom AS p2 FROM bbc
      GROUP BY region HAVING MAX(population/surface) = population/surface)
   ON r1=r2 ;
```

Région	Maximum	Pays Max	Densité max	Le plus dense
Africa	130 200 000	Nigeria	588	Mauritius
Americas	13 000 000	Guatemala	632	Barbados
Asia-Pacific	1 300 000 000	China	6 666	Singapore
Europe	141 500 000	Russia	16 000	Monaco
Middle East	74 900 000	Egypt	1051	Bahrain
North America	295 000 000	United States of America	54	Mexico
South America	182 800 000	Brazil	49	Ecuador
South Asia	1 100 000 000	India	1134	The Maldives

Partie IV

Concept de client-serveur

La plupart des bases de données sont organisées suivant le paradigme client/serveur. La base de donnée en elle-même est gérée par le serveur qui s'occupe du stockage et des recherches proprement dite. À l'autre bout de la ligne, les clients envoient des requêtes au serveur (simple interrogation ou écriture sur la base) et celui-ci répond. De la sorte, le serveur et le client n'ont pas à être sur la même machine. De plus, le client n'a pas à savoir comment le serveur s'arrange pour stocker les données, il suffit qu'il sache lui demander gentiment pour les obtenir...

Néanmoins, il apparaît de plus en plus une architecture en « trois tiers » à décomposer comme suit:

- Vous accéder depuis votre ordinateur (qui constitue le premier tiers) à un site web, disons `http://www.sdss.org` celui du Sloan Digital Sky Survey.
- Le serveur web du SDSS (qui constitue le deuxième tiers) vous permet d'interroger la base de données du SDSS. Votre navigateur ne discute qu'avec lui, mais...
- Le troisième tiers correspond au SGBD du SDSS qui lui ne discute qu'avec le serveur web du SDSS (qui est son client). C'est ce dernier tiers qui répond effectivement à la requête que le serveur web lui a envoyé après que vous-même lui ayez gentiment demandé...

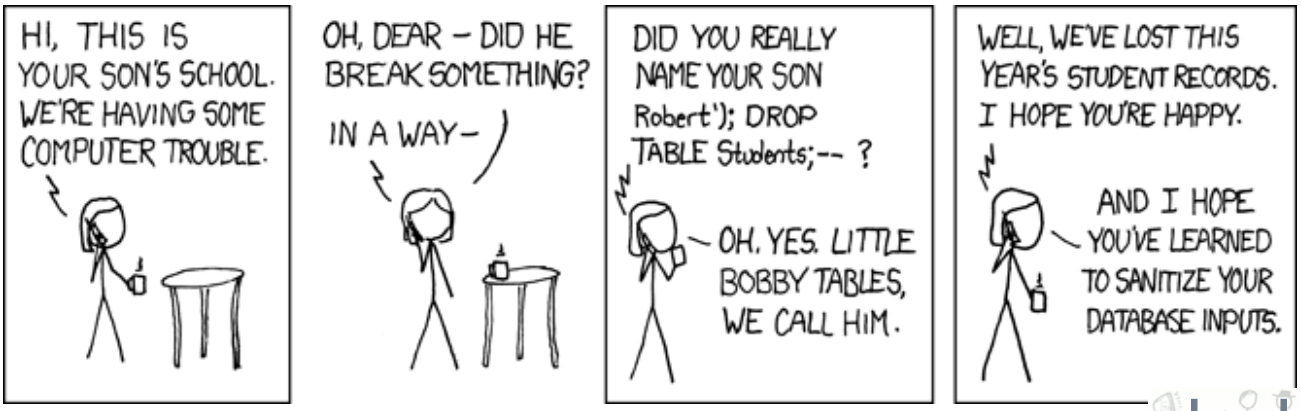
Partie V

Quelques exercices

Soit la BDD définie par le schéma suivant

```
ACTOR   (id:int, name:string)
MOVIE   (id:int, title:string, year:int, score:float, votes:int, director:int)
CASTING (movieid:int, actorid:int, ord:int)
```

1. Trouver les films dont le score est le plus élevé.
2. Sortir la liste des films ayant comptabilisé le moins de votes mais dont le score est supérieur à 8.
3. Trouver les réalisateurs qui comptabilisent les meilleurs scores moyens pour leurs films.
4. Pareil pour les acteurs.
5. Sortir le casting du film `"Star Wars"`
6. Sortir la liste des films dont le réalisateur est aussi acteur.



Her daughter is named Help I'm trapped in a driver's license factory.