BASE DE DONNEES

Dans de nombreux domaines de la vie économique, sociale ou associative, on est amené à manipuler des tables de données donnant des informations sur des produits, des clients, des adresses, des achats/ventes… Ces différentes données peuvent être liées entre elles : par exemple pour un achat d'un produit, on peut avoir gardé en mémoire la date de la transaction, le nom du vendeur, le prix unitaire, la quantité etc.… mais dans cette table recensant ces achats, il serait fastidieux de conserver toutes les informations ayant un lien de près ou de loin avec ces transactions : adresse du fournisseur, nom des contacts, chiffre d'affaire, quantité de tous les produits en stock, budget disponible, état des comptes à cet instant…. Il est donc nécessaire de séparer tous ces types d'informations dans des tables distinctes mais qui sont liées entre elles. C'est cette notion de multiplicité de tables reliées entre elles qui est le fondement des bases de données.

Ces bases de données existaient avant les ordinateurs : livre de compte, recensement… Cependant l'outil informatique a permis des gains en efficacité prodigieux dans l'organisation de ces bases de données.

I) Bases de données et modèle relationnel

Bases de données

Une **base de données** est une collection de données **structurées**, organisées par des **relations**, stockées au sein d'un **SGBD** (Système de Gestion de bases de données) permettant :

- la définition de bases de données structurées ;
- l'interrogation et la mise à jour cohérente des données ;
- un stockage pérenne et efficace de grandes quantités de données ;

Le SGDB est un système logiciel visant à simplifier la tâche des usagers en proposant un niveau d'abstraction (c'est-à-dire que le fonctionnement peut s'adapter à différents types de données). Il doit gérer les représentations **physique** (les "tables" de données) et **logique** (les liens entre les tables) des données. Pour cela il utilisera un langage dédié à la manipulation des données.

Plusieurs SGDB existent : ORACLE, ACCESS, OPEN OFFICE etc.…., mais ils utilisent tous le langage normalisé **SQL** (Structured Query Language) créé en 1974. Plusieurs SGDB (libres et gratuites…) ont d'ailleurs un nom construit sur cet acronyme : SQLite, MySQL, sqliteman…

Modélisation relationnelle des données

Les données sont organisées en **tables** (ou **relations**). Dans l'exemple suivant, nous avons d'abord la relation Regions puis la relation Villes

Nom	Capitale	Habitants
Alsace	25	1 861 020
Aquitaine	4	3 303 392
Auvergne	8	1 355 630
…	…	…

Cle	Nom	Region	Habitants	CodePostal
1	**Ajaccio**	Corse	66 809	20 000
2	**Amiens**	Picardie	133 327	80 000
3	**Besançon**	Franche-Comté	115 879	25 000
4	**Bordeaux**	Aquitaine	239 399	33 000
5	**Bourges**	Centre	66 602	18 000
…	….	…	…	…

Pour fixer les idées on travaillera avec les deux tables suivantes :

Regions		
Nom	**Capitale**	**Habitants**
Alsace	**25**	**1 861 020**
Aquitaine	**4**	**3 303 392**
Auvergne	**10**	**1 355 630**
Basse-Normandie	**6**	**1 479 242**
Bourgogne	**11**	**1 643 931**
Bretagne	**23**	**3 259 659**
Centre	**20**	**2 572 931**
Champagne-Ardenne	**7**	**1 333 497**
Corse	**1**	**322 120**
Franche-Comté	**3**	**1 177 906**
Haute-Normandie	**24**	**1 848 102**
Île-de-France	**21**	**11 978 363**
Languedoc-Roussillon	**18**	**2 727 286**
Limousin	**13**	**741 047**
Lorraine	**17**	**2 350 657**
Midi-Pyrénées	**26**	**2 946 507**
Nord - Pas-de-Calais	**12**	**4 052 156**
Pays de la Loire	**19**	**3 658 351**
Picardie	**2**	**1 924 737**
Poitou-Charentes	**22**	**1 792 159**
Provence-Alpes-Côte d'Azur	**16**	**4 937 445**
Rhône-Alpes	**15**	**6 393 470**

Villes				
Cle	**Nom**	**Region**	**Habitants**	**CodePostal**
1	**Ajaccio**	Corse	**66 809**	**20 000**
2	**Amiens**	Picardie	**133 327**	**80 000**
3	**Besançon**	Franche-Comté	**115 879**	**25 000**
4	**Bordeaux**	Aquitaine	**239 399**	**33 000**
5	**Bourges**	Centre	**66 602**	**18 000**
6	**Caen**	Basse-Normandie	**108 793**	**14 000**
7	**Chalons-en-Champagne**	Champagne-Ardenne	**45 153**	**51 000**
8	**Chartres**	Centre	**39 159**	**28 000**
9	**Chartres**	Bretagne	**7 304**	**35 131**
10	**Clermont-Ferrand**	Auvergne	**140 957**	**63 000**
11	**Dijon**	Bourgogne	**151 672**	**21 000**
12	**Lille**	Nord - Pas-de-Calais	**227 533**	**59 000**
13	**Limoges**	Limousin	**137 758**	**87 000**
14	**Luisant**	Centre	**6 812**	**28 600**
15	**Lyon**	Rhône-Alpes	**491 268**	**69 000**
16	**Marseille**	Provence-Alpes-Côte d'Azur	**850 636**	**13 000**
17	**Metz**	Lorraine	**119 962**	**57 000**
18	**Montpellier**	Languedoc-Roussillon	**264 538**	**34 000**
19	**Nantes**	Pays de la Loire	**287 845**	**44 000**
20	**Orléans**	Centre	**114 185**	**45 000**
21	**Paris**	Île-de-France	**2 249 975**	**75 000**
22	**Poitiers**	Poitou-Charentes	**87 906**	**86 000**
23	**Rennes**	Bretagne	**208 033**	**35 000**
24	**Rouen**	Haute-Normandie	**111 553**	**76 000**
25	**Strasbourg**	Alsace	**272 222**	**67 000**
26	**Toulouse**	Midi-Pyrénées	**447 340**	**31 000**

Les colonnes sont appelées les **champs** ou les **attributs** : dans l'exemple précédent, les attributs de la relation Regions sont Nom, Capitale et Habitants. Pour la relation Villes, les attributs sont Cle, Nom, Region, Habitants et CodePostal.

Les lignes sont les **entrées** ou **tuples** ou n-uplets. Ces tuples ne sont pas nommés.

Dans une colonne, les valeurs sont toutes du même type. L'attribut prend ses valeurs dans un **domaine** (entier, chaine de caractères,…). Sous SQL, un domaine est décrit par un type, auquel on peut ajouter des contraintes. Par exemple, le type de l'attribut Habitants est le type INT. Le type de l'attribut Cle de la table Villes est aussi le type INT, mais on rajoute la contrainte que les valeurs prises doivent être 2 à 2 distinctes. Voici les types les plus usuels :

- Le type VARCHAR (ou CHAR VARYING) permet de coder des chaînes de longueur variables, mais la longueur maximale est fixée. Par exemple VARCHAR(20) désigne le type des chaines alphanumériques formées d'au plus 20 caractères.

- CHAR désigne le type des chaînes alphanumériques de longueur fixe, par exemple CHAR(10) désigne le type des chaînes alphanumériques formées de 10 caractères.
- NUMERIC (ou DECIMAL ou DEC) qui permet de coder des nombres décimaux de façon exacte.
- Le type FLOAT qui permet de coder des décimaux en base 2 de façon approchée (mais les calculs seront plus rapides qu'avec le type NUMERIC).
- INT (ou INTEGER) qui permet de coder des entiers.
- TEXT qui permet de coder des textes (éventuellement longs) de longueur indéterminée.
- DATE et TIME pour la date et l'heure.

Il faut se souvenir que chaque attribut possède un type qui lui est propre (éventuellement son domaine peut être restreint par des contraintes). Il ne serait par exemple pas possible de mémoriser la valeur "Beaucoup" pour l'attribut Habitants.

Les attributs ne sont pas ordonnés : on ne peut pas demander le "premier attribut" de la table. Par contre chaque attribut possède un nom qui lui est propre (Nom, Capitale …) et qui permet d'identifier l'attribut de façon univoque (deux attributs distincts d'une même table ne peuvent pas avoir le même nom).

Les lignes non plus ne sont pas ordonnées. Par contre, deux lignes doivent nécessairement être différentes.

Clé primaire

Une **clé primaire** permet d'identifier de manière univoque chaque tuple d'une relation. C'est un sous-ensemble des attributs de la relation qui permet de distinguer chaque entrée de la relation, mais telle que, si on retire n'importe lequel des attributs de ce sous-ensemble, alors on ne peut plus distinguer certaines entrées de la relation. Par exemple dans la relation Villes de notre exemple, à laquelle on aurait enlevé l'attribut Cle, on aurait pu choisir le sous-ensemble constitué des attributs Nom et CodePostal : 2 villes peuvent avoir le même nom ou le même code postal mais jamais les deux en même temps.

En pratique la clé est souvent limitée à un seul attribut, qui prend des valeurs distinctes pour chaque entrée de la relation. Cet attribut sera identifié par le terme Cle ou Identifiant ou Id (éventuellement suivi de l'objet à identifier : Villes, Nom, Etudiant, Professeur….)

Clés étrangères

Une manière sommaire de mémoriser des données serait de regrouper toutes les informations dans une seule relation. Sur notre exemple, on pourrait imaginer une seule relation où les entrées seraient constituées par les villes et qui contiendrait les attributs NomRegion et HabitantsRegion.

Cle	Nom	Habitants	CodePostal	NomRegion	HabitantsRegion
1	Ajaccio	66 809	20 000	Corse	322 120
20	Orléans	114 185	45 000	Centre	2 572 931
8	Chartres	39 159	28 000	Centre	2 572 931
4	Bordeaux	239 399	33 000	Aquitaine	3 303 392
…	…	…	…	…	…

Cette méthode n'est pas bonne. Le principal problème est la gestion des informations redondantes. Par exemple les villes Orléans et Chartres contiendraient deux fois les informations Centre et 2572931 (pour les attributs NomRegion et HabitantsRegion). En cas de mise à jour de la valeur du nombre d'habitants de la région Centre, il faudra modifier chaque entrée de ville faisant partie de la région Centre : il y a des risques d'oubli. Et cela augmente la quantité d'informations à retenir (puisqu'on garde plusieurs places mémoires pour la même information).

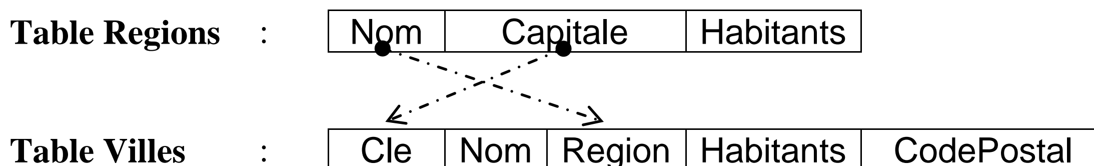
Pour éviter ces écueils, on créera plusieurs relations, certains attributs d'une relation faisant référence à des attributs d'autres relations. Une **clé étrangère** permet de référencer les entrées de certaines relations avec la clé (primaire) d'une autre relation : dans notre exemple, l'attribut Region de la relation Villes est une clé étrangère qui référence la clé primaire de la relation Regions, et l'attribut Capitale de la relation Regions est une clé étrangère qui référence la clé primaire Cle de la relation Villes.

Schéma relationnel

Le **schéma d'une relation** est la donnée du nom de la relation et de chacun de ses attributs (avec ou sans mention de leurs domaines).

Dans nos exemples, nous avons les deux schémas : Regions(Nom, Capitale, Habitants) et Villes(Cle, Nom, Region, Habitants, Code postal). Notez que l'on utilisera un symbole pour distinguer les clés primaires…

Le **schéma de base de données** (ou schéma relationnel) est la donnée des différentes relations de la base, chacune avec ses attributs, et des références entre clés étrangères et clés primaires des différentes relations d'une base de données. Dans notre exemple le schéma relationnel serait le suivant :



II) Opérations ensemblistes

Sélection et tri des tuples

La **sélection** est l'opération permettant de ne choisir que les entrées (donc les lignes) d'une relation vérifiant une certaine condition.

Par exemple si l'on cherche les villes de plus de 400 000 habitants, le code SQL permettant cette demande est : (on appelle cette demande **une requête SQL**)

```
SELECT * FROM Villes
WHERE Habitants > 400000
```

Cette requête s'écrit en algèbre relationnelle :

$\sigma_{\text{Habitants} > 400000}(\text{Villes})$

Ci-dessous voici la réponse pour notre base de données

Cle	Nom	Region	Habitants	CodePostal
15	Lyon	Rhône-Alpes	491 268	69 000
16	Marseille	Provence-Alpes-Côte d'Azur	850 636	13 000
21	Paris	Île-de-France	2 249 975	75 000
26	Toulouse	Midi-Pyrénées	447 340	31 000

On peut faire des tris sur la réponse donnée. Par exemple, pour un tri décroissant selon le nombre d'habitants puis, en cas d'égalité, par Cle : (ici réponse donnée par ACCESS….)

```
SELECT * FROM Villes
WHERE Habitants > 400000
ORDER BY Habitants DESC, Cle
```

DESC pour décroissant (descending)

Villes + de 400000 habitants				
Cle	Nom	Région	Habitants	Code Postal
21	Paris	Île-de-France	2 249 975	75 000
16	Marseille	Provence-Alpes-Côte d'Azur	850 636	13 000
15	Lyon	Rhône-Alpes	491 268	69 000
26	Toulouse	Midi-Pyrénées	447 340	31 000

Il est possible de rajouter en fin de requête une condition LIMIT d, n.
Dans ce cas, la base de donnée ne retournera que les uplets du numéro d + 1 à d + n

Projection

La **projection** est l'opération permettant de ne choisir que certains attributs (donc certaines colonnes) d'une relation.

Par exemple lorsque l'on projette la relation Regions en ne gardant que les attributs Nom et Habitants, la requête SQL est :

```
SELECT Nom, Habitants
FROM Regions
```

Requête	
Nom	Habitants
Alsace	1 861 020
Aquitaine	3 303 392
Auvergne	1 355 630
Basse-Normandie	1 479 242
Bourgogne	1 643 931
Bretagne	3 259 659
Centre	2 572 931

Cette requête s'écrit en algèbre relationnelle :

$$\pi_{\text{Nom, Habitants}} \text{ (Regions)}$$

Evidemment, on peut composer une projection et une sélection

```
SELECT Nom, CodePostal
FROM Villes
WHERE Region = "Centre"
```

On peut aussi utiliser des " " autour de Centre

VillesduCentre	
Nom	CodePostal
Bourges	18 000
Chartres	28 000
Luisant	28 600
Orléans	45 000

Cette requête s'écrit en algèbre relationnelle :

$$\pi_{\text{Nom, CodePostal}} (\sigma_{\text{Region = "Centre"}} \text{ (Villes)})$$

Union, intersection, différence

La **réunion** est l'opération algébrique classique. Elle concerne les entrées uniquement. Lors d'une requête, les attributs projetés doivent être les mêmes.

```
SELECT Nom, Region, Habitants  FROM  Villes
WHERE Region = "Centre"
UNION
SELECT Nom, Region, Habitants  FROM  Villes
WHERE Habitants < 100000
```

UNION		
Nom	Region	Habitants
Ajaccio	Corse	66809
Bourges	Centre	66602
Chalons-en-Champagne	Champagne-Ardenne	45153
Chartres	Centre	39159
Chartres	Bretagne	7304
Luisant	Centre	6812
Orléans	Centre	114185

Cette requête s'écrit en algèbre relationnelle :

$$\pi_{\text{Nom, Region, Habitants}} (\sigma_{\text{Region = "Centre"}} (\text{Villes})) \cup \pi_{\text{Nom, Region, Habitants}} (\sigma_{\text{Habitants} < 100000} (\text{Villes}))$$

On peut également écrire cette requête sous la forme :

```
SELECT Nom, Region, Habitants  FROM  Villes
WHERE ( Region = "Centre" )  OR  ( Habitants < 100000 )
```

L'**intersection** est l'opération algébrique classique. Elle concerne les entrées uniquement. Lors d'une requête, les attributs projetés doivent être les mêmes.

```
SELECT Nom, Region, Habitants  FROM  Villes
WHERE Region = "Centre"
INTERSECT
SELECT Nom, Region, Habitants  FROM  Villes
WHERE Habitants < 100000
```

INTERSECTION		
Nom	Region	Habitants
Bourges	Centre	66 602
Chartres	Centre	39159
Luisant	Centre	6812

Cette requête s'écrit en algèbre relationnelle :

$$\pi_{\text{Nom, Region, Habitants}} (\sigma_{\text{Region = "Centre"}} (\text{Villes})) \cap \pi_{\text{Nom, Region, Habitants}} (\sigma_{\text{Habitants} < 100000} (\text{Villes}))$$

On peut également écrire cette requête sous la forme :

```
SELECT Nom, Region, Habitants  FROM  Villes
WHERE ( Region = "Centre" )  AND ( Habitants < 100000 )
```

Remarque : dans certains SGDB, la fonction "INTERSECT" n'existe pas … mais le connecteur logique AND existe …

La **différence ensembliste** est l'opération algébrique classique. Elle concerne les entrées uniquement. Lors d'une requête, les attributs projetés doivent être les mêmes.

```
SELECT Nom, Region, Habitants  FROM  Villes
WHERE  (Region = "Centre")
AND NOT (Habitants < 100000)
```

DIFFERENCE ENSEMBLISTE		
Nom	Region	Habitants
Orléans	Centre	114 185

Cette requête s'écrit en algèbre relationnelle :

$$\pi_{\text{Nom, Region, Habitants}} (\sigma_{\text{Region = "Centre"}} (\text{Villes})) \setminus \pi_{\text{Nom, Region, Habitants}} (\sigma_{\text{Habitants} < 100000} (\text{Villes}))$$

Remarque : dans certains SGDB, les fonctions "EXCEPT" ou "MINUS" existent …

Renommage

Le **renommage** est l'opération algébrique consistant à donner un autre nom à une relation ou à un attribut.

```
SELECT Nom, Habitants  AS Population
FROM Regions
```

Renommage	
Nom	**Population**
Alsace	1 861 020
Aquitaine	3 303 392
Auvergne	1 355 630
Basse-Normandie	1 479 242
Bourgogne	1 643 931
Bretagne	3 259 659
Centre	2 572 931

Cette requête s'écrit en algèbre relationnelle :

$$\rho_{\text{Habitants}\rightarrow\text{Population}} \left(\pi_{\text{Nom, Habitants}} \text{ (Regions)} \right)$$

Fonctions d'agrégation

Les **fonctions d'agrégation** permettent de faire des calculs sur un groupe d'entrées sélectionnées.

Par exemple, si on veut calculer le nombre moyen d'habitants par région (arrondi à 2 chiffres après la virgule), on a la requête :

```
SELECT ROUND(AVG(Habitants), 2)
FROM Regions
```

Moyenne
2893618,55

Parmi les fonctions d'agrégation les plus usuelles, nous avons :

- AVG pour le calcul de la moyenne (Average)
- COUNT pour compter le nombre de tuples sélectionnés
- MAX et MIN pour le maximum et le minimum
- SUM pour le calcul d'une somme
- Certaines fonctions dont la valeur de retour est booléenne et qui servent à des critères de sélections d'autres requêtes : ALL, ANY (ou SOME), EXISTS, NOT EXISTS. Elles apparaissent dans des sous-requêtes que nous verrons plus loin.

III) Sous-requêtes

Il peut être interessant d'utiliser le résultat d'une requête R_1 à l'intérieur du critère d'une autre requête R_2 : la requête R_1 est une **sous-requête** de la requête R_2.

Exemple : si on veut le code postal et le nom de la ville du Centre ayant le plus grand nombre d'habitants, on peut utiliser la requête :

```
SELECT Nom, CodePostal FROM Villes
WHERE Habitants = (SELECT MAX(Habitants)
FROM Villes WHERE Region = 'Centre')
```

SousRequete1	
Nom	**CodePostal**
Orléans	45 000

Utilisations du résultat d'une sous-requête avec ALL, EXISTS, ANY ou IN

EXISTS

On souhaite connaitre toutes les villes qui se trouvent dans la même région qu'une commune nommée Chartres. Il y a donc deux requêtes portant sur la relation Villes.

En utilisant la fonction EXISTS, qui fournit le booléen Vrai si le résultat de la sous-requête est non vide et Faux sinon,

```
SELECT Nom, Region FROM Villes AS V1

WHERE

EXISTS (SELECT * FROM Villes AS V2
WHERE
V2.Region = V1.Region AND V2.Nom = 'Chartres')
```

Requête5	
Nom	**Region**
Bourges	Centre
Chartres	Centre
Chartres	Bretagne
Luisant	Centre
Orléans	Centre
Rennes	Bretagne

Autre version mais qui change le nom des colonnes…. (on peut régler le souci… en renommant V1.Nom AS Nom et V1.Region AS Region…)

```
SELECT V1.Nom, V1.Region
FROM Villes AS V2, Villes AS V1
WHERE
V2.Region = V1.Region
AND
 V2.Nom = 'Chartres'
```

Requête6	
V1.Nom	**V1.Region**
Orléans	Centre
Lunery	Centre
Bourges	Centre
Saint Florent	Centre
Ajaccio	Corse
Saint Florent	Corse

ALL

ALL signifie "vrai pour toutes les entrées".

Exemple : Recherche des villes plus peuplées que toutes les villes de la région Centre

```
SELECT *  FROM Villes
WHERE   Habitants > ALL (SELECT Habitants FROM Villes WHERE Region = 'Centre' )
```

ANY

ANY signifie "vrai pour au moins une entrée".

Exemple : Recherche des villes plus peuplées qu'au moins une ville de la Corse

```
SELECT * FROM Villes
WHERE   Habitants > ANY (SELECT Habitants FROM Villes WHERE Region = 'Corse' )
```

IN est un synonyme de =ANY et il existe aussi la négation NOT IN

<u>Remarque</u> : Dans SQLite, le ALL est omis : Habitants > (SELECT Habitants ….) fait le test Habitants supérieur à toutes les entrées de l'attribut…. et le ANY est à trasformer en NOT propriété contraire….

IV) Groupement de tuples

GROUP BY

Cette fonction permet de partitionner les tuples renvoyés par une requête SELECT en différents groupes pour appliquer une fonction d'agrégation à chaque groupe.

Exemple : si on veut calculer la somme des habitants des villes de chaque région. :

```
SELECT Region, SUM(Habitants)  AS Sh
FROM Villes
GROUP BY Region
ORDER BY SUM(Habitants)
```

Requête9	
Region	Sh
Champagne-Ardenne	45153
Corse	68431
Poitou-Charentes	87906
Basse-Normandie	108793

Autre exemple : moyenne des sommes des habitants des villes de chaque région :

```
SELECT   AVG (Sh)  FROM (SELECT SUM(Habitants)  AS Sh
                                      FROM Villes      GROUP BY Region  )
```

HAVING

Cette fonction effectue la même sélection que WHERE mais elle s'applique au groupe sélectionné via GROUP BY et non aux tuples eux-mêmes.

```
SELECT   Region   FROM Villes
GROUP BY Region
HAVING  MAX(Habitants) > 300000
```

Somme habitants >300000
Region
Île-de-France
Midi-Pyrénées
Provence-Alpes-Côte d'Azur
Rhône-Alpes

V) Sélection sur plusieurs relations

Produit cartésien

Soient des relations R_1, R_2, …, R_n. Le **produit cartésien** $R_1 \times R_2 \dots \times R_n$ est la relation contenant tous les uplets $v_1, \dots v_n$ où v_1 est un uplet de R_1, …, v_n est un uplet de R_n .

Dans le langage SQL, le produit cartésien de deux relations s'obtient en mettant les relations concernées, séparées par des virgules, just après le mot FROM. Par exemple, pour obtenir le produit cartésien des relations Region et Villes, on utilisera la requête :

```
SELECT * FROM Villes, Regions
```

On obtiendra 22 * 26 = 572 uplets dont les premiers sont :

Requête5							
Cle	Villes.Nom	Region	Villes.Habitants	CodePostal	Regions.Nom	Capitale	Regions.Habitants
1	Ajaccio	Corse	66 809	20 000	Alsace	25	1 861 020
1	Ajaccio	Corse	66 809	20 000	Aquitaine	4	3 303 392
1	Ajaccio	Corse	66 809	20 000	Auvergne	8	1 355 630
..	…	…	…	…	…	…	…

Le résultat n'est pas très pertinent du point de vue de la redondance de l'information mais surtout du point de vue de la cohérence de l'information : quel lien entre Ajaccio et l'Alsace ?

Pour éviter cet écueil, on va unifier la clé primaire Regions.Nom avec la clé étrangère Villes.Region en rajoutant une clause WHERE Regions.Nom = Villes.Region ce qui permet de ne garder effectivement que les uplets formant une unité logique significative.

Requête6							
Cle	Villes.Nom	Region	Villes.Habitants	CodePostal	Regions.Nom	Capitale	Regions.Habitants
1	Ajaccio	Corse	66 809	20 000	Corse	1	322 120
2	Amiens	Picardie	133 327	80 000	Picardie	2	1 924 737
3	Besançon	Franche-Comté	115 879	25 000	Franche-Comté	3	1 177 906
…	…	…	…	…	…	…	…

Cependant, cette manipulation est tellement importante en gestion de bases de données qu'elle a un statut particulier : il s'agit de la jointure.

Jointure

Soient deux relations R et S, ayant respectivement des attributs $r_1, \dots r_p$ et $s_1, \dots s_q$.

Soient deux indices k dans $[\![1,p]\!]$ et $h \in [\![1,q]\!]$. La **jointure symétrique** des relations R et S qui joint les attributs r_k et s_h est la relation qu'on obtiendrait si on effectuait successivement les opérations suivantes :

- former le produit cartésien des relations R et S ;
- sélectionner les uplets qui vérifient $r_k = s_h$.

L'écriture en SQL de cette manipulation est : SELECT … FROM R JOIN S ON $r_k = s_h$

Par exemple, pour afficher toutes les villes avec les informations disponibles sur la région à laquelle elles appartiennent :

```
SELECT * FROM Villes JOIN Regions ON Villes.Region = Regions.Nom
```

Requête7							
Cle	Villes.Nom	Region	Villes.Habitants	CodePostal	Regions.Nom	Capitale	Regions.Habitants
1	Ajaccio	Corse	66 809	20 000	Corse	1	322 120
2	Amiens	Picardie	133 327	80 000	Picardie	2	1 924 737
3	Besançon	Franche-Comté	115 879	25 000	Franche-Comté	3	1 177 906
…	…	…	…	…	…	…	…

Cette requête s'écrit en algèbre relationnelle : $\text{Villes} \bowtie_{\text{Region} = \text{Nom}} \text{Regions}$

(Remarque : dans ACCESS, il faut indiquer INNER JOIN pour la jointure symétrique)

Souvent, la condition de jointure porte sur l'unification d'une clé primaire avec une clé étrangère s'y référant, mais ce n'est pas toujours le cas. Voici par exemple une requête pour trouver les homonymies dans les noms de ville :

SELECT V1.Cle, V1.Nom, V1.Region, V2.Cle, V2.Nom, V2.Region
FROM Villes AS V1 JOIN Villes AS V2 ON V1.Nom = V2.Nom
WHERE V1.Cle < V2.Cle

Requête8					
V1.Cle	V1.Nom	V1.Region	V2.Cle	V2.Nom	V2.Region
8	Chartres	Centre	9	Chartres	Bretagne

Il s'agit d'une jointure interne : on utilise deux fois la relation Villes, en renommant à chaque fois la relation Villes (respectivement V1 et V2) pour faire la distinction.
La clause WHERE V1.Cle < V2.Cle permet de n'écrire qu'une seule fois chaque doublon.

Remarque : lorsque l'on veut faire une jointure symétrique entre deux tables dans laquelle la sélection de toutes les entrées s'effectue sur les attributs de même nom (et ayant les mêmes valeurs) dans les deux tables, on peut utiliser une jointure naturelle, avec la syntaxe :
SELECT * FROM Table1 NATURAL JOIN Table2

Division cartésienne

Soient deux relations R et S. On suppose que S a pour attributs $s_1, \dots s_q$ et que chacun de ces attributs s'identifie naturellement avec un attribut de R : les attributs de R se notent alors $r_1, \dots r_p$, $s_1, \dots s_q$.

La notation $U = (r_1 = v_1, \dots, r_p = v_p, s_1 = w_1, \dots, s_q = w_q)$ signifie que U est un uplet de la relation R et que $v_1, \dots v_p$, $w_1, \dots w_q$ sont les valeurs respectives prises par l'uplet U sur chacun des attributs.

La division de R par S est la relation D dont les attributs sont $r_1, \dots r_p$ et telle que pour tout uplet $V = (r_1 = v_1, \dots, r_p = v_p)$, on a :

$$V \in D \Leftrightarrow \forall\, W = (s_1 = w_1, \dots, s_q = w_q) \in S,\ (r_1 = v_1, \dots, r_p = v_p, s_1 = w_1, \dots, s_q = w_q) \in R$$

Exemple : S est l'ensemble des noms de villes de Bretagne ayant moins de 10000 habitants. On souhaite connaître les régions possédant une ville ayant le même nom que toutes les entrées de S. Ici la relation R est obtenue par :

SELECT Regions.Nom, Villes.Nom
FROM Regions JOIN Villes ON Regions.Nom = Villes.Region

alors que la relation S est obtenue par

SELECT Nom FROM Villes
WHERE Region = 'Bretagne' AND Habitants < 10000

Sur cet exemple, $p = q = 1$. L'attribut r_1 est le nom de la région et l'attribut s_1 est le nom de la ville.

Dire que D est le résultat de la division cartésienne de R par S, c'est dire que D est la plus grande relation possible (pour l'inclusion des uplets) telle que le produit cartésien $D \times S \subset R$

On notera en algèbre relationnelle : $\mathbf{D = R \div S}$

Récapitulatif des requêtes SELECT

SELECT	*attributs de projection*
FROM	*relation ou produit cartésien de relations ou jointure ou table dérivée*
WHERE	*critères de sélection des uplets*
GROUP BY	*critères de regroupement*
HAVING	*critères de sélection des groupes*
ORDER BY	*attributs de tri (éventuellelement* [DESC] *)*
LIMIT	*premier_uplet, nombre_uplet*

VI) Exercices

Exercice 1

On travaille avec la base de données constituées des 2 relations Villes et Regions vues dans le cours. Ecrire pour chaque question, une requête permettant de connaître :

1) les villes du Centre en ordonnant les résultats par ordre croissant d'habitants ;

2) le plus grand code postal des villes de Corse

3) les villes qui ne sont pas capitale de région

4) les villes du Centre plus peuplées que la moyenne des villes du Centre

5) les régions ayant le plus de villes (dans la base)

Exercice 2

On rajoute dans le schéma relationnel de notre base de données une relation Jumelages ayant exactement deux attributs Ville1 et Ville2, chacun de ces deux attributs étant une clé étrangère référant la clé primaire Cle de la relation Villes.

Par exemple, s'il existe l'entrée (Ville1 = 3, Ville2 = 6) dans cette relation Jumelages, cela signifie que Besançon et Caen sont jumelées.

On admettra qu'un jumelage n'apparait qu'une seule fois : si on a le couple (i, j) en entrée, on n'aura pas le couple (j, i).

1) Ecrire une requête permettant de connaître les villes jumelées à Lyon

2) Ecrire une requête permettant de connaître les villes jumelées à au moins une ville du Centre

3) Ecrire une requête permettant de connaitre les villes jumelées à toutes les villes de Corse.

4) Ecrire une requête permettant de vérifier que la condition sur l'unicité des jumelages est bien respectée dans la table.

ORACLE

VII) Correction des exercices

Exercice 1

1) les villes du Centre en ordonnant les résultats par ordre croissant d'habitants ;
```
SELECT Nom, Habitants FROM Villes
WHERE Region = 'Centre' ORDER BY Habitants
```

2) le plus grand code postal des villes de Corse
```
SELECT MAX('CodePostal') FROM Villes WHERE Region = 'Corse'
```

3) les villes qui ne sont pas capitale de région
```
SELECT * FROM Villes WHERE Cle NOT IN (SELECT Capitale FROM Regions)
```

4) les villes du Centre plus peuplées que la moyenne des villes du Centre
```
SELECT * FROM Villes
WHERE Region = 'Centre' AND Habitants > (SELECT AVG(Habitants) FROM Villes
                                          WHERE Region = 'Centre')
```

5) les régions ayant le plus de villes (dans la base)
```
SELECT Region FROM Villes
GROUP BY Region
HAVING COUNT (Cle) = (SELECT MAX(NB_Villes)
                      FROM (SELECT COUNT (Cle) AS NB_Villes FROM Villes
                            GROUP BY Region) )
```

Exercice 2 Jumelages

1) Ecrire une requête permettant de connaître les villes jumelées à Lyon
```
SELECT Nom FROM Villes JOIN Jumelages ON Cle = Ville1
WHERE Ville2 IN (SELECT Cle FROM Villes WHERE Nom = 'Lyon')
UNION
 SELECT Nom FROM Villes JOIN Jumelages ON Cle = Ville2
WHERE Ville1 IN (SELECT Cle FROM Villes WHERE Nom = 'Lyon')
```

2) Ecrire une requête permettant de connaître les villes jumelées à au moins une ville du Centre
```
SELECT Nom FROM Villes JOIN Jumelages ON Cle = Ville1
WHERE Ville2 IN (SELECT Cle FROM Villes WHERE Region = 'Centre')
UNION
 SELECT Nom FROM Villes JOIN Jumelages ON Cle = Ville2
WHERE Ville1 IN (SELECT Cle FROM Villes WHERE Region = 'Centre')
```

3) Ecrire une requête permettant de connaitre les villes jumelées à toutes les villes de Corse.
```
SELECT Nom FROM Villes AS V
WHERE NOT EXISTS (SELECT * FROM Villes AS W
                  WHERE Region = 'Corse'
                  AND NOT EXISTS (SELECT * FROM Jumelages
                       WHERE (Ville1 = W.Cle AND Ville2 = V.Cle)
                       OR (Ville2 = W.Cle AND Ville1 = V.Cle)))
```

4) Ecrire une requête permettant de vérifier que la condition sur l'unicité des jumelages est bien respectée dans la table.
```
SELECT COUNT(*) FROM Jumelages AS J1, Jumelages AS J2
WHERE J1.Ville1 = J2.Ville2 AND J1.Ville2 = J2.Ville1
```