Bases de données

Introduction

Une base de données (BD) est un fichier constitué d'un ensemble organisé et structuré d'informations sous la forme de tables (tableaux) indépendants et dont le but global est de modéliser un système réel. L'objectif est de gérer de manière efficace et structurée des données qui peuvent être de taille très importante.
Pour gérer les données enregistrées dans une base, on utilise un système de gestion de bases de données (SGBD). Ce sont des logiciels complexes qui permettent à des utilisateurs ou des programmes d'exprimer des requêtes pour interroger des bases de données ou pour les modifier.
Nous nous focaliserons ici sur les plus répandus d'entre ces systèmes, les systèmes relationnels, parmi lesquels nous trouvons des logiciels propriétaires (ORACLE, Microsoft SQL server, ...) et des logiciels libres très utilisés (MySQL, SQLite, ...). Python propose également un module (sqlite3) qui permet d'interroger des BD.
De gros efforts de standardisation ont été effectués et un langage de requêtes commun aux différents SGDB est né : **le langage SQL** (Structured Query Language). C'est ce langage que nous utiliserons;

1 Description d'une base de données

1.1 Schéma général

Le modèle relationnel est le principal modèle employé par les SGBD. Le but du modèle relationnel est de percevoir une réalité sous la forme de tableaux (tables) de valeurs à deux dimensions appelées relations :

1. une relation (table) a un nom ;
2. une colonne d'une relation est un **attribut** ;
3. une ligne (on dit aussi enregistrement) d'une relation est un **n-uplet** ou **tuple** ;

Pour illustrer nos propos, nous nous appuierons sur la base de données `commerce` contenant les trois tables `clients`, `commandes` et `pizzas`, que l'on figure ci-dessous :

Table `clients` :

Cle	Nom	Prenom	Adresse	Ville
100	Dubois	Pierre	18 rue des Roses	Dijon
101	Dubois	Amélie	18 rue des Roses	Dijon
102	Albert	Alberic	25 rue Devosges	Dijon
...	...	...	...	...

Table `pizzas` :

Cle	Nom	Prix
01	Margarita	12,5
02	Calzone	14
...	...	...

Table `commandes` :

Cle	Client	Pizza	Date	Livre	Paye
1	102	8	12/07/2015	1	1
2	107	5	12/05/2015	1	1
3	115	7	09/04/2015	1	0
...	...	...	...	...	...

1.2 Les attributs et leurs domaines

Seuls les attributs sont nommés : dans la table (on dit aussi relation) `pizzas` les noms des trois attributs sont `Cle`, `Nom` et `Prix`.
Les différentes valeurs prises par un même attribut appartiennent toutes à un même ensemble, appelé le **domaine de l'attribut**. Le schéma relationnel d'une table est la donnée de ses attributs et de leurs domaines.

Suivant son domaine, chaque attribut est représenté par un type. Les principaux types rencontrés sont les entiers (`INT`), les flottants (`REAL`) et les chaînes de caractères (`CHAR` ou `VARCHAR(20)`, l'entier précisant la longueur maximale). Celui de la table `commandes` est le suivant (on a fait figuré les types plutôt que les domaines) :

```
(Cle : INT ;  Client : INT ; Pizza : INT  ; Date : DATE ;
Livre : INT ; Paye : INT  )
```

et celui de la table `pizzas` :

```
(Cle : INT ; Nom : CHAR ; Prix : FLOAT).
```

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 et qui permet d'identifier l'attribut de façon univoque (deux attributs distincts d'une même table ne peuvent pas avoir le même nom).

1.3 Clé primaire et clé étrangère

Dans la relation, l'ordre des lignes n'est pas important. En revanche, chaque tuple doit être unique. Pour les identifier, on définit une **clé primaire** qui peut être constituée d'un ou plusieurs attributs. En pratique la clé primaire est souvent limitée à un seul attribut qui prend des valeurs distinctes pour chaque entrée de la table.

Exemples : Dans la table `pizzas`, l'attribut `Cle` joue le rôle de clé primaire.

Quand il existe plusieurs clés possibles (par exemple dans la table `pizzas` , l'attribut `Nom` pourrait aussi l'être), on parlera de clé primaire (ou clé principale) pour la clé choisie et de clé secondaire pour les autres clés possibles. Pour des raisons d'efficacité lors de l'interrogation d'une table, il est nécessaire de disposer d'une clé la plus simple possible.
Une colonne (ou un ensemble de colonne) dont le rôle est de référencer une colonne d'une autre table est dénommé **clé étrangère**.

Exemples : Dans la table `commandes`, l'attribut `Pizza` est une clé étrangère qui référence la clé primaire `Cle` de la table `pizzas`. Ces clés sont essentielles lorsque l'on souhaite obtenir des informations qui nécessitent la lecture de plusieurs tables. Elles sont représentées par des flêches entre les tables (à faire apparaître sur le schéma). La donnée des tables d'une base de donnée et des clés étrangères qui les relient les unes aux autres constituent le schéma relationnel global de la base de données.

1.4 En pratique

Au lycée nous utiliserons le logiciel `SQLite browser`. Ce logiciel est disponible à l'adresse `https://sqlitebrowser.org/`. On peut également utiliser `SQLite` en ligne à l'adresse `https://sqliteonline.com/`. Il faut alors charger la base de données que l'on souhaite manipuler.

2 Quelques principes de construction d'une base de données

Le but de cette partie est de présenter les contraintes auxquelles permet de répondre la structure d'une base de données et de donner quelques rudiments sur les outils permettant de construire une telle base de données.
Pour cela revenons en arrière au sujet de l'exemple de la partie précédente et partons des besoins de l'utilisateur (la pizzeria en l'occurence ici).
On pourrait imaginer que les informations concernant les clients, les commandes et les pizzas soient stockées dans un tableau à deux dimension (type tableau Excel) :

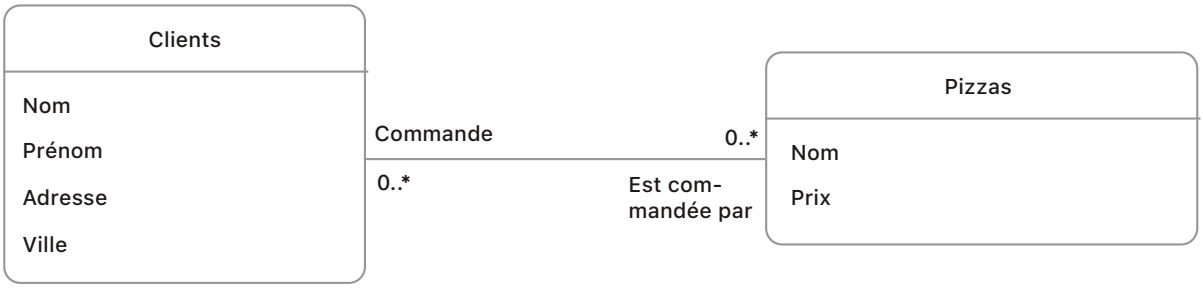
pizza	prix	nom du client	prénom	adresse	ville	date	livraison	paiement
Margarita	12,5	Dubois	Pierre	18 rue des Roses	Dijon	23/06/2015	non	oui
Calzone	14	Dubois	Amélie	18 rue des Roses	Dijon	04/06/2015	oui	oui
Margarita	12,5	Albert	Alberic	25 rue Devosges	Dijon	03/05/2015	oui	oui
Regina	20	Albert	Alberic	25 rue Devosges	Dijon	03/05/2015	oui	oui
...	...	...	...	...	...	...	...	...

Cette façon de faire présente plusieurs inconvénients :

- Certaines informations sont présentes de manière redondante : l'adresse de chaque client par exemple, qui apparait à chaque commande de ce client. Cela nécessite des ressources plus importantes en espace mémoire.

- Conséquence du précédent : si un client change d'adresse, il faudra la changer dans toutes ses commandes actives.
- Si momentanément un client n'a plus de commande active (et si assez naturellement un mécanisme supprime les commandes anciennes), les informations au sujet de ce client sont perdues.

Une première idée pour éviter certains de ces inconvénients est de créer deux tableaux : un pour les clients et un pour les pizzas. Il faut aussi préciser les relations entre ces tableaux.
Afin de représenter ces informations, la méthode utilisée est le modèle entité/association (les entités sont ici les tables, et les associations les liens entre des éléments de ces tables). Nous adoptons pour cela le formalisme UML (Unified Modeling Language).
Sur notre exemple cela donnerait :



Sur ce schéma, les traits entre les entités sont affectés d'une précision sur la nature de l'association et d'une cardinalité : a..b signifie que a est le nombre minimal associé à la relation et b le nombre maximal (* signifiant que ce nombre n'est pas majoré).
A ce stade nous sommes confrontés au problème suivant. Les conventions sur les SGBD font qu'une base de données doit être constituée uniquement de tables. Comment faire apparaître les informations sur les commandes de pizzas par les clients ? On pourrait rajouter dans la table `clients` la liste des pizzas que chacun a commandé, mais cela nécessiterait de nombreuses colonnes en plus, certaines pouvant être vides ; ou bien rajouter une ligne client pour chaque pizza qu'il commande, mais l'ont retombe dans le problème de redondance de l'information.
La bonne solution dans ce contexte est de créer une nouvelle table qui permet de "casser" l' association de cardinalité `0..*/*..0`. Pour cela, on crée une table `commandes`, contenant toutes les commandes (une ligne pour chaque pizza, on pourrait discuter ce choix) :

On se retrouve avec des associations de cardinalité `0..*/1..1`, qui peuvent être représentées par une clef étrangère. Plus précisément, c'est la cardinalité `1..1` qui permet ce lien (la cardinalité `0..1` le permettrait aussi).
On tombe sur le schéma relationnel présenté dans la première partie.

3 Interrogation d'une base de données en langage SQL

Dans cette, nous allons détailler l'écriture de requêtes permettant d'interroger une BDD dans le langage SQL.

3.1 La projection et la sélection

La projection consiste à sélectionner certaines colonnes d'une table. Sélectionner les attributs A_i et A_j de la table R_1 s'écrit en SQL : `SELECT` A_i, A_j `FROM` R_1 ;
Exemple : Requête donnant le nom et le prénom de chaque client de la table `clients`.

```
SELECT Nom , Prenom FROM clients;
```

Remarques :

- La commande pour sélectionner la totalité des attributs et donc visualiser la totalité de la table R_1 s'écrit : `SELECT` * `FROM` R_1 ;
- On utilise pour éviter les doublons la commande : `SELECT DISTINCT`.
  Exemple : pour obtenir la liste des villes où habitent des clients à partir de la table `clients` :

  ```
  SELECT DISTINCT Ville FROM clients ;
  ```

La sélection consiste à sélectionner certaines lignes (sous-ensemble de tuples) qui vérifient une condition. Le schéma de la table résultat est identique à celui de la table sur laquelle porte la sélection.
Pour exprimer la condition, on pourra utiliser les opérateurs suivants : =,<> (différent), >, <, <=, >=, AND, OR, NOT.
Sélectionner les tuples tel que l'attribut A de la relation R_1 vérifie la condition c sécrit : `SELECT` * `FROM` R_1 `WHERE` $A = c$;
La plupart du temps, on veut faire à la fois une sélection et une projection.
Exemple : Requête retournant le nom des clients habitant Dijon.

```
SELECT Nom FROM clients WHERE Ville='Dijon';
```

3.2 Tris des résultats d'une requête

La commande `ORDER BY` permet de trier les résultats d'une requête selon un critère.
Exemple : Requête permettant d'obtenir un tableau contenant la liste des pizzas par ordre de prix.

```
SELECT * FROM pizzas ORDER BY Prix ASC;
```

Ou pour avoir par ordre de prix decroissant :

```
SELECT * FROM pizzas ORDER BY Prix DESC;
```

Par défaut l'ordre est ascendant.
On peut en plus n'obtenir qu'une partie des tuples renvoyés par la commande précédente.
Exemple : Requête permettant d'obtenir la liste des trois pizzas les moins chères par ordre de prix croissant.

```
SELECT *
FROM pizzas
ORDER BY Prix ASC LIMIT 3
```

On peut compléter par une dernière instruction permettant d'obtenir un décalage dans cette sélection : la commande : `LIMIT nbValeurs OFFSET decalage` permet de sélectionner suite à la commande précédente un nombre de résultats `nbValeurs` décalé du nombre `decalage`.
Exemple : Requête permettant d'obtenir la deuxième et la troisième pizza les moins chères par ordre de prix croissant :

```
SELECT *
FROM pizzas
ORDER BY Prix ASC LIMIT 2 OFFSET 1
```

3.3 Les opérateurs ensemblistes

3.3.1 L'union

L'union est un opérateur dédié à des relations de même schéma relationnel. L'union de deux tables R_1 et R_2 donne une nouvelle table contenant les tuples de R_1 et les tuples de R_2 en éliminant les doublons s'il y en a. L'union de deux relations de même schéma s'écrit en SQL : R_1 `UNION` R_2

3.3.2 L'intersection

L'intersection est un opérateur dédié à des relations de même schéma relationnel. L'intersection de deux tables R_1 et R_2 donne une table contenant les tuples appartenant à la fois à R_1 et à R_2. Cela s'écrit : en SQL : R_1 `INTERSECT` R_2

3.3.3 La différence

La différence est un opérateur dédié à des relations de même schéma relationnel. La différence entre les tables R_1 et R_2 donne une table contenant les tuples de R_1 n'appartenant pas à R_2. Cela s'écrit en SQL : R_1 `EXCEPT` R_2

3.4 Produit cartésien et jointure

3.4.1 Le produit cartésien

Le résultat du produit cartésien entre deux tables R_1 et R_2 est une relation contenant l'ensemble des possibilités d'association entre une valeur de R_1 et une valeur de R_2.
Cela s'écrit en SQL : R_1 `CROSS JOIN` R_2 ou plus simplement R_1, R_2.
Exemple : La commande

```
SELECT * FROM  pizzas,commandes
```

renvoie tous les couples formés d'une pizza et d'une commande. Il n'y a pas forcément de lien entre les deux éléments du couple ; ainsi l'information obtenue n'est pas très pertinente.

3.4.2 Renommage

Un attribut "A" peut être renommé en "B". Le schéma du résultat est identique à celui sur lequel porte le renommage. Ce renommage n'est valable que pour la requête dans lequel il est défini, son but est principalement de faciliter la lecture et l'écriture des requêtes un peu complexes.
Un renommage s'écrit : `SELECT` A `AS` B `FROM` R_1 ;
On verra des exemples dans les paragraphes suivants.

3.4.3 Jointure

Le produit cartésien crée de nouveaux tuples, mais sans que ceux-ci aient un sens utile. Une jointure consiste également à combiner une paire de tuples de deux relations en un seul tuple, mais en regroupant les tuples qui ont un point en commun (cela peut-être considéré comme la composition d'un produit cartésien et d'une sélection).
On ne s'intéresse qu'aux jointures symétriques simples qui permettent de recoller deux relations qui ont un attribut en commun (en général, il s'agit d'une clé étrangère). L'écriture d'une jointure où l'attribut A_i de la relation R_1 est égal à l'attribut A_j de la relation R_2 est :
`SELECT` * `FROM` R_1 `JOIN` R_2 `ON` $R_1.A_i = R_2.A_j$;
Exemple : requête permettant d'obtenir un tableau contenant la liste des commandes (leur numéro) et le nom de la pizza commandée.

```
SELECT  commandes.Cle,pizzas.Nom
FROM commandes  JOIN pizzas
               ON commandes.pizza=pizzas.Cle;
```

ou, en utilisant le renommage :

```
SELECT  com.Cle,piz.Nom
FROM commandes  AS com JOIN pizzas  AS piz
               ON com.pizza=piz.Cle;
```

3.5 Les fonctions d'agrégation

Les fonctions d'agrégation dans le langage SQL permettent d'effectuer des opérations statistiques sur un ensemble de tuples. Les principales sont :

1. `AVG()` pour calculer la moyenne sur un ensemble de tuples
2. `COUNT()` pour compter le nombre de tuples sur une table ou une colonne distincte
3. `MAX()` pour récupérer la valeur maximum d'un attribut sur un ensemble de lignes. Cela s'applique à la fois pour des données numériques ou alphanumériques
4. `MIN()` pour récupérer la valeur minimum de la même manière que `MAX()`
5. `SUM()` pour calculer la somme sur un ensemble de tuples

Exemples : requête qui permet d'établir le prix moyen d'une pizza.

```
SELECT  AVG(Prix) FROM pizzas   ;
```

Requête qui permet d'obtenir le nombre de villes dont proviennent les clients.

```
SELECT  COUNT  (DISTINCT Ville) FROM clients   ;
```

On peut également utiliser les opérations numériques usuelles (+,-,*,/).
Exemples : requête qui permet d'obtenir l'amplitude de prix des pizzas.

```
SELECT  Max(Prix)-MIN(Prix) FROM pizzas   ;
```

3.6 Requêtes imbriquées

Exemples : requête qui permet d'obtenir les pizzas dont le prix est supérieur au prix de la margharita.

```
SELECT  nom
FROM Pizzas
WHERE prix > (SELECT prix FROM pizzas WHERE Nom = 'Margharita')
```

3.7 Structure complète d'une requête `SELECT`

De manière générale, la strucure d'une requête " `SELECT`" s'établit comme suit :

1. `SELECT <liste d'attributs>`
2. `FROM <liste de tables>`
3. `WHERE <conditions>`
4. `GROUP BY <liste d'attributs>`
5. `HAVING <conditions>`
6. `ORDER BY <liste d'attributs>`

On a déjà vu les rôles des trois premières lignes :

1. La clause `SELECT` spécifie le schéma de sortie (projection).
2. La clause `FROM` précise les tables impliquées et leurs liens (produit cartésien et jointures).
3. La clause `WHERE` fixe les conditions que doivent remplir les n-uplets résultats (sélection).

Il reste à comprendre ce à quoi servent les trois dernières lignes.

1. La clause `GROUP BY` permet de regrouper des n-uplets résultats en groupes.
   Exemple : Donner le nombre de clients dans chaque ville.

```
SELECT  Ville,COUNT(*)
FROM clients
GROUP BY Ville ;
```

   Remarque : seul un attribut figurant dans le `GROUP BY` peut être projeté.

2. La clause `HAVING` impose une condition sur les groupes (i.e. permet d'éliminer l'intégralité d'un groupe, en se basant sur la valeur d'un agrégat calculé sur ce groupe).
   Exemple : Donner le nombre de clients dans chaque ville en ne gardant que les villes qui ont au moins 5 clients parmi leurs habitants.

```
SELECT  Ville, COUNT(*)
FROM clients
GROUP BY Ville
HAVING COUNT(*)   > 4;
```

   ou bien, en effectuant un renommage :

```
SELECT  Ville, COUNT(*) AS nbClients
FROM clients
GROUP BY Ville
HAVING nbClients > 4;
```

3. Enfin `ORDER BY` définit les critères de tris des résultats (déjà vu).

4 Exercice

On reprend la base de données ayant servi d'exemple dans le cours.
Écrire les requêtes qui permettent de répondre aux questions suivantes (et les tester) :

1. (a) Quel est le prix de la pizza Margarita ?
   (b) Donner le nombre de commandes effectuées pour le 12/07/2015 et qui ont été payées.
   (c) Quel est le prix et le nom des trois pizzas les plus chères ?
   (d) Quel est le prix et le nom de la deuxième pizza la plus chère ?
2. (a) Quelles sont les pizzas dont le prix est compris entre 15 et 18 euros ?
   (b) Quel est le prix moyen d'une pizza ?
   (c) Quelles sont les pizzas dont le prix est supérieur au prix moyen (on utilisera une requête composée) ?
   (d) Quel est le prix moyen des trois pizzas les moins chères ?
3. (a) Quels sont les noms et prénoms des clients ayant une commande impayée ?
   (b) Quels sont les noms et prénoms des clients ayant commandé une Margarita ?
   (c) Quels sont les noms et prénoms des clients n'ayant aucune commande impayée ?
   (d) Quels sont les noms et prénoms des clients ayant commandé une Margarita et à qui il reste une commande impayée ?
4. (a) Déterminer le nombre de pizzas commandées pour chaque client (en désignant chaque client par sa clef).
   (b) Déterminer le nombre de pizzas commandées pour chaque client ayant plus de deux pizzas commandées.