Document d'accompagnement pour la partie «Base de données»

4 mai 2019

Le déroulement du cours

1. Les élèves regardent chez eux les vidéos «Pythonnerie n°23 et n°24».
2. Je donne le document de cours fourni par l'IG avec l'exemple de la bibliothèque. On voit le vocabulaire et on dessine les schémas relationnels. On visualise les données avec une interface graphique (SQLlite manager ou SQLliteBrowser) On commence à écrire des requêtes :
   - Donner les noms et prénoms des auteurs.
   - Donner les noms des emprunteurs (attention aux doublons)
   - Donner les noms, prénoms et salaire des employés du site 5 dont le salaire dépasse 3000 euros
   - Donner le titre et le nom de l'auteur des ouvrages de type roman (attention jointure)
   - Donner le titre des ouvrages empruntés par Mr Biraud. (encore jointure)
   - Donner le salaire moyen des employés (fonction AVG)
   - Donner le salaire moyen, maximum et minimum.
   - Donner le salaire moyen par poste.
   - Donner les salaires moyens par site au dessus de 3000 euros.
3. En deuxième heure de cours, on peut faire un deuxième exemple (population, ou films, ou vins...)
4. En devoir maison, requêtes sur la base triathlon.
5. On fait un premier TP avec des premières requêtes (pas de regroupement)
6. On fait un deuxième TP avec des regroupements et on voit comment transformer un fichier CSV en base de données.

1 Vovabulaire en vrac

1. Trois approches pour conserver des données : mémoire volatile (RAM), fichiers et les SGBD

2. SGBD : système de gestion de bases de données (en anglais, DBMS Data Base Management System). C'est un gros logiciel qui permet de stocker et gérer des données. Quelques caractéritiques en vrac :

   - une capacité énorme de stockage
   - on peut interroger les données de façon efficace (on parle de requête) avec un langage déclaratif (SQL).
   - on peut les alimenter, les modifier, les détruire de façon massive.
   - plusieurs utilisateurs simultanés (des dizaines de milliers) peuvent accéder aux données, le SGBD gère les files d'attente...
   - protection des informations avec des contraintes d'intégrité, des droits d'accès et de modification différents selon les utilisateurs.

3. Le SQL (Structure Querry Language). C'est le langage pour effectuer des requêtes (querry en anglais) avec un SGBD. Il est dit déclaratif (à opposer à procédural).

   Quelques noms :

   - MYSQL (Oracle), DB2 (IBM), SQL Server (Microsoft), Sybase (SAP), PostgreSQL (de Michael Stonebreaker, logiciel libre), SQLLite (2000, libre).

   Ce langage est la concrétisation d'un modèle théorique pour gérer les BD développé par Codd, appelé **modèle relationnel**.

4. Les personnages à retenir :

   - Charles Bachman (langage COBOL, Architecture ANSI-PARC, premier SGBD, prix Alan Turing en 1973)
   - Edgard Franck Codd (article fondateur du modèle relationnel, publié dans CACM en 1970, prix Alan Turing en 1981)
   - Michael StoneBreaker (créateur de PsotgreSQL, médaille John von Neumann 2005, prix Turing 2014)

5. Architecture ANSI/SPARC (1965)

   On sépare les données de leur traitement.

   Architecture client-serveur (2 niveaux) (morale : « un client dans un café, le serveur est à son écoute, le client émet des requêtes, c'est le serveur qui exécute...»

   Architecture à n niveaux : par exemple trois tiers : exemple le web dynamique (cf schéma mooc) client, serveur d'application, serveur de données

2 Le modèle relationnel

2.1 Le vocabulaire

1. Une base de données est constituée de plusieurs tables (on parle de relation en algèbre relationnelle). Les colonnes d'une table sont appelés attributs, ils ont un type (int, char, numeric, date NULL...) (on parle de domaine en algèbre relationnelle). Les lignes d'une table s'appellent des n-uplets ou des tuples.

   Une relation R (ou une table) est donc un ensemble de tuples, formellement c'est donc une partie d'un produit cartésien.

2. Notion de clé primaire (primary key) d'une relation : c'est un ou plusieurs attributs d'une même table qui permettent de déterminer de manière unique un tuple de cette table. Nous parlerons aussi de clé étrangère (foreign key) mais cette notion est hors-programme.

3. Notion de schéma relationnel

   C'est un moyen de représenter une base de données en faisant figurer les différentes tables, leurs attributs avec leur types mais aussi leurs relations.

   On souligne le ou les attribut(s) qui constitue(nt) une clé primaire de la table.

   Quelques exemples :

   (a) La base du triathlon était composée d'une seule table nommée resultats :

   `resultats( rang: entier, dossard: entier, nom: texte, prenom: texte, ....)`

   (b) La base sur les populations de commune était composée de 3 tables communes, departements et regions

   `communes(id: entier, dep: texte, nom: texte, pop: entier)`

   `departements(id: entier, reg: entier, nom: texte)`

   `regions(id: entier, nom: texte)`

   Les contraintes d'inétgrité sont assurés par le fait que :

   - le numéro de département d'une commune doit être parmi les identifiants des départements, on note :

   $$communes[dep] \subset departements[id]$$

   - le numéro de région d'un département doit être parmi les identifiants des régions, on note :

   $$departements[reg] \subset regions[id].$$

2.2 Les opérations

Morale : tout résultat d'une opération sur les relations est une relation

Il y en a 8 :

1. opérations unaires :
   - la sélection (restriction) (sous-ensemble de la relation) notée $T \leftarrow \sigma_{\text{condition}}(R)$ (on ne garde que les tuples de la relation qui vérifient la condition).
   - la projection (on ne garde que certains attributs de la relation) notée $T \leftarrow \Pi_{\text{attribut}}(R)$
2. opérations binaires :
   - l'intersection, la réunion et la différence (les deux tables doivent avoir les mêmes attributs)
   - le produit cartésien et la jointure. Exemple à la main : voici une base constituée des deux tables suivantes etudiants et villes

nom	naissance	Ville
Alice	1997	Strasbourg
Bob	1998	Paris
Eve	1995	Strasbourg

nomV	dpt
Strasbourg	67
Paris	75

Le produit cartésien des deux tables est la table suivante :

nom	naissance	Ville	nomV	dpt
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

Il s'obtient avec la requête SQL suivante :

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

On voit que le nombre de tuples est $t1 \times t2$ et le nombre d'attributs $a_1 + a_2$

Le produit cartésien est une opération coûteuse.

Il y a une relation entre les deux tables, les villes de la table `etudiants` doivent être parmi les noms de ville de la table `villes`. Nous allons donc plutôt demander le produit cartésien avec la condition que `Ville = nomV`. La relation obtenue s'appelle la jointure. La requête est

```
SELECT *
FROM etudiants, villes
WHERE Ville = nomV
```

ou

```
SELECT *
FROM etudiants JOIN villes ON Ville = nomV
```

nom	naissance	Ville	nomV	dpt
Alice	1997	Strasbourg	Strasbourg	67
Bob	1998	Paris	Paris	75
Eve	1995	Strasbourg	Strasbourg	67

- la division (hors-programme, pas à connaître)

Les trois principales opérations sont la sélection, la projection et la jointure.

3 La concrétisation SQL

SQL = Algèbre relationnelle + Fonctions d'agrégation + Tri

Nous utiliserons

3.1 Opérations

1. Sélection et projection

   - 3 clauses :

     ```
     SELECT attribut1, attribut2
     FROM table
     WHERE condition1 AND condition2...
     ```

   - On termine une requête par un point virgule.
   - Les conditions de sélection peuvent être combinées avec les opérateurs logiques AND, OR, NOT (il existe aussi BETWEEN mais ce n'est pas nécessaire)
   - On remplit les clauses dans cette ordre : FROM, WHERE, SELECT
   - Le SELECT DISTINCT pour supprimer les éventuels doublons lorsque la clé primaire ne fait pas partie des attributs que l'on sélectionne.

     Exemple : si l'on veut les différents prénoms des triathlètes, la requête `SELECT prenom FROM resultats` renverrait tous les prénoms mais avec des répétitions. La bonne requête serait :

     ```
     SELECT DISTINCT prenom
     FROM resultats
     ```

   - On peut trier les lignes de la table selon un attribut avec la clause ORDER BY attribut DESC (par ordre décroissant de l'attribut). On utilise ASC pour croissant.

     Exemples :

     – la liste des communes triés par ordre décroissant de population

     ```
     SELECT *
     FROM communes
     ORDER BY pop DESC
     ```

– la liste des différents prénoms des triathlètes triés par ordre alphabétique

```
SELECT DISTINCT prenom
FROM resulats
ORDER BY prenom
```

- Le caractère $*$ dans un SELECT permet de sélectionner tous les attributs `SELECT * FROM ma_table`.
- Le caratère joker % combinée avec La fonction LIKE permet de comparer des chaînes de caractère.

  Exemple : la requête suivante sélectionne les communes dont le nom commence par la lettre P.

```
SELECT *
FROM commune
WHERE nom LIKE 'P\%'
```

2. La jointure

   Il y a deux syntaxes possibles pour joindre deux tables A et B.

```
SELECT attribut1, attribut2,...
FROM tableA , tableB
WHERE conditions de jointure
```

   ou avec `JOIN` et `ON` pour différencier les conditions de jointure des conditions de sélection

```
SELECT attribut1, attribut2,...
FROM tableA  JOIN tableB ON conditions de jointure
```

   Penser à utiliser des alias parfois pour une écriture plus légère.

   Exemple : les communes de l'Herault de plus de 20000 habitants.

```
SELECT C.nom, C.pop
FROM communes C JOIN departements D ON C.dep = D.id
WHERE C.pop > 20000 AND D.nom = 'Hérault';
```

   Remarques :

   - on peut joindre autant de tables que souhaitées

```
SELECT attribut1, attribut2,...
FROM tableA  JOIN tableB JOIN tableC ON conditions de jointure
```

   - on peut fait faire le produit cartésien d'une table par elle même mais il faut créer des alias.

     Exemple : au triathlon, la liste des binômes (homme, femme) avec leur temps cumulé.

```
SELECT R1.nom,R1.prenom, R2.nom, R2.prenom, R1.temps + R2.temps AS temps_cumule
FROM resultats R1, resultats R2
WHERE R1.Sx_cat LIKE 'M%' AND R2.Sx_Cat LIKE 'F%'
```

3. opérateurs UNION, INTERSECT, EXCEPT

   Par exemple, la requête suivante

```
select * from communes where dep = 34
UNION
select * from communes where dep = 11
```

   est équivalent à celle-ci

```
select *
from communes
where dep = 34 or dep = 11
```

4. les sous-requêtes

   Il n'y pas d'affectation en SQL (en ce qui nous concerne...). On pourra donc être emmené à imbriquer une requête à l'intérieur d'une requête. Attention, il faudra alors mettre entre parenthèse la sous-requête.

   Exemple : la commune la plus peuplée de Seine-Saint Denis (département 93)

```
SELECT nom,pop
FROM communes
WHERE dep = 93
      AND pop = (SELECT MAX(pop) FROM communes WHERE dep = 93)
```

5. les NULL

   La requête suivante permet de sélectionner les triathlètes qui n'ont pas de temps en vélo. Cela peut arriver par exemple, si l'on a cassé son dérailleur et que l'on abandonne. `SELECT * FROM triathlon WHERE temps_velo IS NULL`

3.2 Le partitionnement en groupes et fonctions d'agrégats

1. **cinq fonctions d'agrégats : COUNT (nombre de tuples), SUM (somme), AVG (average), MAX, MIN.**

   - Elles s'appliquent à l'ensemble des valeurs d'un attribut d'une table. Attention, elles renvoient un seul nombre (et pas un tuple).
   - Pour une requête sans partitionnement, uniquement dans le SELECT (et sans aucun attribut supplémentaire) , jamais dans le WHERE.

   Exemples :

   - Le nombre de tuples d'une table
     `SELECT COUNT(*) FROM table`
   - La population totale de l'Hérault

```
SELECT D.nom, SUM(pop) AS pop_totale
FROM communes C JOIN departements D ON C.dep = D.id
WHERE  D.nom = 'Hérault';
```

- Le nombre de prénoms différents des participants du triathlon.
  Attention, la requête suivante (qui renvoie 449) n'est pas la bonne car certains participants ont les mêmes prénoms.

  ```
  SELECT COUNT(prenom)
  FROM resultats
  ```

  On peut utiliser une requête imbriquée avec

  ```
  SELECT COUNT(*)
  FROM (SELECT DISTINCT prenom FROM resultats)
  ```

  Il y a une syntaxe équivalente plus courte :

  ```
  SELECT COUNT(DISTINCT(prenom))
  FROM resultats
  ```

2. **pour faire des stats par paquets au sein d'une relation.**

   On partitionne horizontalement la relation, selon les valeurs d'un attribut (ou de plusieurs attributs) spécifié dans la clause **GROUP BY**.

   Attention : l'attribut de groupage doit toujours être dans la clause **SELECT**.

   On peut ensuite sélectionner certains tuples de cette nouvelle table à l'aide de la clause **HAVING**.

   Exemple : les départements de plus de 20000 habitants.

   On pourrait être tenté d'écrire une requête de la forme :

   ```
   SELECT nom_dept, sum(population) AS population_dept
   FROM commune
   WHERE population_dept > 20000
   GROUP BY nom_dept
   ```

   Ca ne marche pas car la clause `WHERE` est exécutée avant l'agrégation. La clause `HAVING` permet d'indiquer au SQL d'effectuer une nouvelle sélection à la fin du calcul sur les résultats du regroupement.

   On écrira donc :

   ```
   SELECT nom_dept, sum(population) AS population_dept
   FROM commune
   GROUP BY nom_dept
   HAVING population_dept > 20000
   ```