Sujet d'entraînement

Introduction

Une requête sur une base de données est décrite au moyen d'un langage déclaratif. Le langage SQL est le plus connu. Pour évaluer une requête, un système de gestion de base de données (SGBD) établit un plan d'exécution combinant les opérateurs de l'algèbre relationnelle. L'objectif de ce sujet est l'étude de ces opérateurs.

Nous étudierons en partie I l'implémentation en Python de ces opérateurs. Nous appliquerons ensuite en partie II ces résultats à des requêtes SQL. Nous verrons en partie III comment il est possible de tirer parti des propriétés des données pour améliorer les performances.

Les parties peuvent être traitées indépendamment. Néanmoins, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes.

Bases de données, tables, attributs, enregistrements

Nous détaillons ici la représentation des données dans le modèle relationnel. Une *base de données* est un ensemble de *tables*. Chaque table porte un nom et est associée à un vecteur d'attributs de longueur au moins 1. Le nombre d'attributs d'une table est appelé *l'arité* de la table. Le vecteur des attributs $< a_0,a_1,...,a_{k-1} >$ d'une table T d'arité k est noté *attributs(T)* et la table est notée $T[[a_0,a_1,..., a_{k-1}]]$.

Une table $T[[a_0,a_1,...,a_{k-1}]]$ est constituée *d'enregistrements*. La *taille* d'une table est le nombre des enregistrements qu'elle contient. Dans ce sujet, nous considérerons qu'un enregistrement est un vecteur $< v_0,v_1,...,v_{k-1} >$ de longueur l'arité k de la table. Chaque élément de ce vecteur est la valeur de cet enregistrement par rapport à l'attribut correspondant de la table. La valeur v_i à l'indice i de l'enregistrement est la valeur associée à l'attribut a_i à l'indice i du vecteur d'attributs de cette table. On pourra donc identifier un attribut et son indice et parler de *la valeur d'un enregistrement* associée à un indice. La valeur d'un enregistrement e associée à un indice i est notée $e[i]$.

> Nous considérons dans ce sujet que toutes les valeurs d'attributs sont des chaînes de caractères et que la comparaison entre deux valeurs d'un attribut a un coût unitaire quelles que soient ces valeurs.

Deux enregistrements représentés par des vecteurs contenant les mêmes valeurs aux mêmes indices sont *égaux*.

> Une table peut contenir des enregistrements égaux. L'élimination des enregistrements égaux est une opération complexe qui est l'objet de l'opérateur SQL appelé `DISTINCT` que nous étudierons plus loin.

Exemple (Tables et enregistrements). Considérons une agence de voyages qui vend des trajets et des chambres d'hôtel. La table

`Vehicule[[IdVehicule,Type,Compagnie]]`

contient les données relatives aux divers véhicules disponibles. Pour chaque enregistrement e représentant un véhicule dans la table `Vehicule`, la valeur de e associée à l'attribut `IdVehicule` est l'identifiant du véhicule ; la valeur de e associée à l'attribut `Type` est le type de véhicule ; la valeur

associée à l'attribut `Compagnie` est le nom de la compagnie qui gère ce véhicule.

Cette table contient trois *enregistrements* qui décrivent des véhicules : un bus de la compagnie *IBUS*, un train de la compagnie *SNCF* et un avion de la compagnie *Hop !*.

```
<98300, Bus, IBUS>
<1562,TGV, SNCF>
<30990, A320, Hop !>
```

Considérons d'autre part la table

```
Trajet[[IdTrajet,VilleD,VilleA,DateD,HeureD,IdVehicule]]
```

Cette table contient les trajets élémentaires possibles avec les valeurs des attributs associés : l'identifiant du trajet, la ville de départ, la ville d'arrivée, la date du départ de ce trajet, l'heure de départ du trajet, l'identifiant du véhicule utilisé pour le trajet. On rappelle que toutes ces valeurs sont des chaînes de caractères.

Cette table contient 3 trajets possibles le *5 octobre 2016* pour aller de Lille à Rennes. Ils partent respectivement à *9h00*, à *10h00* et à *14h00*.

```
<Trajet1, Lille, Rennes, 5 oct. 2016, 09h00, 30990>
<Trajet2, Lille, Rennes, 5 oct. 2016, 10h00, 98300>
<Trajet3, Lille, Rennes, 5 oct. 2016, 14h00, 1562>
```

Représentation des tables et des enregistrements en Python

Dans ce sujet, nous représentons un enregistrement d'une table d'arité k par une *liste Python* de longueur k. L'élément d'indice i de cette liste représente la valeur de l'enregistrement pour l'attribut d'indice i de la table.

Nous représentons une table d'arité k par une liste d'enregistrements. Une table vide est représentée par une liste vide.

Voici par exemple une représentation en Python de la table `Vehicule` d'arité 3.

```
> > > Vehicule
[['98300', 'Bus', 'IBUS'], ['1562', 'TGV', 'SNCF'], ['30990', 'A320', 'Hop!']]
```

Notez qu'une table peut être représentée par plusieurs listes différentes. Voici une autre représentation possible de cette table.

```
> > > Vehicule
[['1562', 'TGV', 'SNCF'], ['98300', 'Bus', 'IBUS'], ['30990', 'A320', 'Hop!']]
```

1 Implémentation des opérateurs SQL en Python

! Dans tout le sujet, on demande de ne pas modifier les entrées si ce n'est pas explicitement demandé !

Condition de n-uplet avec test d'égalité à une constante

❏ **Q1** – Implémentez la fonction `ConditionConstante(table,indice,constante)` qui prend une `table` en entrée, un `indice` représentant la position d'un attribut et une valeur `constante`, et renvoie une nouvelle table `t` tel que `t` contient seulement les enregistrements `v` de `table`, `v[indice]` est égale à `constante`.
On implémente une requête SQL de type **SELECT * FROM table WHERE attindice = constante** avec attindice l'attribut correspondant au numéro d'indice donnée.

Condition avec test d'égalité entre deux attributs

❏ **Q2** – Implémentez la fonction `ConditionEgalite(table, indice1, indice2)` qui prend une `table` en entrée, deux indices `indice1, indice2` représentant la position i,j d'un attribut. Elle renvoie une nouvelle table `t` contenant les seuls enregistrements `v` de `table` tel que $v[i]$ et $v[j]$ sont égaux.

❏ **Q3** – Donnez la complexité de votre implémentation de la fonction `ConditionEgalite` par rapport à la taille de la table `table`. Justifiez votre réponse en vous appuyant sur la structure du programme.

Sélection sur des indices

Le mot-clef SELECT permet de sélectionner des attributs qui seront présents dans la table renvoyée. La fonction à construire va prendre en entrée une table $T[[a_0,a_1,...,a_{k-1}]]$ d'arité k et une liste $L = [l_0,...,l_{k'-1}]$ telle que $0 < k' \leqslant k$ identifiant des attributs de la table T et va construire la table contenant les mêmes n-uplets que T mais ne contenant que les attributs présents dans L. On se restreindra au cas où la liste L est donnée dans l'ordre croissant sans répétition.

❏ **Q4** – Considérons une table d'enregistrements $T[[a_0,a_1,...,a_{k-1}]]$ d'arité k. Implémentez la fonction `SelectionEnregistrement(enregistrement,listeIndices)` qui prend en arguments un enregistrement `enregistrement` de cette table et une liste `listeIndices =` $[l_0,...,l_{k'-1}]$ telle que $0 < k' \leqslant k$ d'indices identifiant des attributs de cette table. Elle renvoie une nouvelle liste représentant l'enregistrement $< a_{l_0},...,a_{l_{k-1}} >$.

❏ **Q5** – Implémentez la fonction `Selection(table,listeIndices)` qui prend en arguments une table table d'enregistrements d'arité k et une liste `listeIndices =` $[l_0,...,l_{k'-1}]$. telle que $0 < k' \leqslant k$ d'indices identifiant des attributs de cette table dans *attributs(table)*. Cette fonction renvoie une nouvelle liste représentant la table `table` où seules les attributs représentés par la `listeIndices` sont présents.

Produit cartésien

Le but de cette partie est de réussir à produire un produit cartésien entre deux tables. Le produit cartésien entre `Trajet` et `Véhicule` donne :

```
<98300, Bus, IBUS,Trajet1, Lille, Rennes, 5 oct. 2016, 09h00, 30990>
<98300, Bus, IBUS,Trajet2, Lille, Rennes, 5 oct. 2016, 10h00, 98300>
<98300, Bus, IBUS,Trajet3, Lille, Rennes, 5 oct. 2016, 14h00, 1562>
<1562,TGV, SNCF,Trajet1, Lille, Rennes, 5 oct. 2016, 09h00, 30990>
<1562,TGV, SNCF,Trajet2, Lille, Rennes, 5 oct. 2016, 10h00, 98300>
<1562,TGV, SNCF,Trajet3, Lille, Rennes, 5 oct. 2016, 14h00, 1562>
<30990, A320, Hop !,Trajet1, Lille, Rennes, 5 oct. 2016, 09h00, 30990>
<30990, A320, Hop !,Trajet2, Lille, Rennes, 5 oct. 2016, 10h00, 98300>
<30990, A320, Hop !,Trajet3, Lille, Rennes, 5 oct. 2016, 14h00, 1562>
```

❑ **Q6** – Implémentez la fonction `ProduitCartesien(table1, table2)` qui prend en arguments deux tables `table1` et `table2` d'enregistrements. Elle renvoie une nouvelle liste représentant la table du produit cartésien entre `table1` et `table2`

❑ **Q7** – Donnez la complexité de votre implémentation de la fonction `ProduitCartesien(table1, table2)` par rapport à la taille des tables ainsi que leurs arités respectives. Justifiez votre réponse en vous appuyant sur la structure du programme.

Jointure

Le but de cette partie est de réaliser la jointure entre deux tables. La jointure «JOIN, ON» prend quatre arguments, deux tables `table1` et `table2` ainsi que les deux attributs représentés par deux indices `k1,k2` indiquant la condition d'égalité, clef primaire/clef étrangère.

La jointure entre les tables `Vehicule,Trajet` avec pour condition d'égalité `0 et 5` donne :

```
<98300, Bus, IBUS,Trajet2, Lille, Rennes, 5 oct. 2016, 10h00, 98300>
 <1562,TGV, SNCF,Trajet3, Lille, Rennes, 5 oct. 2016, 14h00, 1562>
<30990, A320, Hop !,Trajet1, Lille, Rennes, 5 oct. 2016, 09h00, 30990>
```

❑ **Q8** – En n'utilisant aucune boucle for, ni déclaration de liste vide, ni boucle while mais en utilisant les fonctions précédentes, écrire la fonction `Jointure(table1, table2, indice1, indice2)` qui prend en argument deux tables `table1`, `table2` et deux indices `indice1,indice2` et renvoie une nouvelle table représentant la jointure entre les deux tables avec condition d'égalité entre les deux indices.

❑ **Q9** – Donnez la complexité de votre implémentation de la fonction `Jointure(table1, table2, indice1, indice2)` par rapport à la taille des tables ainsi que leurs arités respectives. Justifiez votre réponse en vous appuyant sur la structure du programme.

Distinct

Cet opérateur permet de supprimer les répétitions d'enregistrements égales dans une table `T`. Il renvoie une table `T'` qui est associée aux mêmes attributs que `T`. Cette table contient exactement un représentant pour chaque classe d'enregistrements égaux de `T`.

❑ **Q10** – Implémentez la fonction `SupprimerDoublons(table)` qui prend en entrée une table `table` et renvoie une nouvelle table qui contient une seule fois chaque ligne de la table `table`. On n'utilisera pas l'opérateur == pour comparer directement les listes, seulement pour comparer les attributs entre eux.

❑ **Q11** – Donnez la complexité de votre implémentation de la fonction `SupprimerDoublons(table)` par rapport à la taille et l'arité de la table `table`. Justifiez votre réponse en vous appuyant sur la structure du programme.

❑ **Q12** – Pour améliorer les performances de `SupprimerDoublons(table)`, il est proposer de remplacer les implémentations des *enregistrements* par des `tuples (ou n-uplets)` à la place des *listes* dans le but de pouvoir utiliser les *enregistrements* comme clefs d'un dictionnaire. Expliquer en quoi ce changement aurait permis d'améliorer la complexité de la fonction précédente.

2 Requêtes SQL

Les données de notre agence de voyage sont enregistrées par les tables suivantes :

- **Vehicule(IdVehicule, Type, Compagnie)** : enregistre les véhicules disponibles — l'identifiant du véhicule, son type et sa compagnie.
- **Trajet(IdTrajet, VilleD, VilleA, IdVehicule)** : enregistre les trajets élémentaires possibles — l'identifiant du trajet, la ville de départ, la ville d'arrivée ainsi que le véhicule utilisé
- **Ticket(IdTicket, IdTrajet, Place, Date, Heure, Prix)** : enregistre les tickets disponibles — l'identifiant du ticket, l'identifiant du trajet auquel ce ticket donne accès, le numéro de la place, la date, l'horaire et le prix en euros.
- **Hotel(IdHotel, Classe, Ville)** : enregistre les hôtels connus — l'identifiant de l'hôtel, sa classe et sa ville.
- **Chambre(IdReservation, IdHotel, Date, Prix)** : enregistre les chambres d'hôtel qui sont disponibles — l'identifiant de réservation à utiliser, l'identifiant de l'hôtel où se trouve la chambre, la date et le prix

A. Requêtes SQL

❑ **Q13** – Écrire une requête SQL permettant d'obtenir toutes les informations des tickets dont la date est la chaîne de caractère "2025-10-17".

❑ **Q14** – Écrire une requête SQL donnant le prix et le type de véhicule de tous les trajets partant de "Lille" et arrivant à "Strasbourg".

❑ **Q15** – Écrire une requête comptant le nombre de Véhicule différents que gère l'agence de voyage

❑ **Q16** – Écrire une requête donnant les hôtels qui ont une moyenne de prix inférieur à 50 euros sur leurs chambres disponible.

❑ **Q17** – Écrire une requête donnant les identifiants de véhicules ne passant pas par Zurich.

❑ **Q18** – Donner les id de tous les hôtels de Bordeaux proposant une chambre au prix le plus bas pouvant être trouvé à Bordeaux.

❑ **Q19** – On souhaite venir deux fois à Paris sur deux dates différentes. Écrire une requête SQL donnant tous les couples d'hôtel dans lequel je pourrais passer la nuit. Enlever les couples où les deux hôtels sont de même classe.

3 Implémentation de requêtes SQL en Python

Le but de cette partie est d'implémenter des requêtes SQL en s'aidant des fonctions qui ont été construites dans la **partie 1**.

Par convention, la liste Python représentant une table aura le même nom que cette table. En Python, on n'oubliera pas qu'un attribut est représenté par l'entier donnant sa position. Par exemple, l'attribut `IdTrajet` de la table `Trajet` est représenté par l'entier 0.

Dans chaque cas, le résultat de la requête sera affecté à une variable nommée `resultat`. Par exemple, la requête SQL :

```
SELECT Vehicule.Compagnie FROM Vehicule
```

pourra être implémentée par

```
resultat = Selection(Vehicule,[2])
```

en supposant que la variable Python `Vehicule` représente la table `Vehicule`. Dans des cas plus complexes, on pourra simplifier l'expression en utilisant des variables auxiliaires pour stocker la valeur de certaines sous-expressions, comme dans l'exemple suivant.

```
r1 = Vehicule
resultat = Selection(r1, [2])
```

!! Il est attendu que les candidats rédigent leurs réponses en combinant uniquement les implémentations des requêtes SQL présentées dans la partie I, à l'exclusion de toute autre fonction ou structure de contrôle Python. !!

❑ **Q20** – Proposez une implémentation pour la requête suivante :

```
SELECT *
   FROM Trajet
   WHERE Trajet.VilleD = Rennes
```

❑ **Q21** – Proposez une implémentation pour la requête suivante :

```
SELECT *
   FROM Trajet, Vehicule
```

❑ **Q22** – Proposez une implémentation pour la requête suivante.

```
SELECT *
   FROM Trajet, Vehicule
   WHERE Trajet.IdVehicule = Vehicule.IdVehicule
```

❑ **Q23** – Proposez une implémentation pour la requête suivante.

```
SELECT Classe, Ville, Date, Prix
   FROM Hotel JOIN Chambre
   ON Hotel.IdHotel = Chambre.IdHotel
```

❑ **Q24** – Proposez une implémentation pour la requête suivante.

```
SELECT Hotel.IdHotel
   FROM Hotel, Trajet, Ticket
   WHERE Hotel.Ville = Trajet.VilleA
    AND Trajet.IdTrajet = Ticket.IdTrajet
    AND Ticket.Prix = '50'
```

❑ **Q25** – Proposez une implémentation pour la requête suivante. (Le `attribut IN table` ne sélectionne que les lignes où `attribut` est présent dans la table `table`)

```
SELECT *
   FROM Chambre
   WHERE Chambre.Prix = '100'
    AND Chambre.IdHotel IN
      (SELECT Hotel.IdHotel
         FROM Hotel, Trajet, Ticket
         WHERE Hotel.Ville = Trajet.VilleA
          AND Trajet.IdTrajet = Ticket.IdTrajet
          AND Ticket.Prix = '50')
```

4 Amélioration des performances

Il est possible dans certains cas d'améliorer l'implémentation d'une requête en tenant compte de propriétés particulières de la représentation des données ou en utilisant des structures de données supplémentaires.

Dans cette partie, nous allons montrer que l'on peut améliorer les performances en triant les données avant de les traiter ou en utilisant des tables associatives (dictionnaires) auxiliaires.

Tables triées par rapport à un indice

Une table d'arité k est représentée par une liste d'enregistrements, eux-mêmes représentés par des listes à k éléments. Supposons tout d'abord avoir à disposition une fonction

`TrieTableIndice(table,indice)`

qui trie par ordre croissant suivant l'ordre lexicographique les enregistrements de la liste `table` d'arité k par rapport à la valeur de l'attribut d'indice `indice` dans le vecteur des attributs de cette table. On suppose que la valeur `indice` est strictement inférieure à k.

Par exemple, la liste `Trajet` ci-dessous est triée par rapport à l'attribut d'indice 1 pour l'ordre lexicographique $<$ sur les chaînes de caractères de Python.

```
> > > Trajet
[['30990', 'A320', 'Hop!'], ['98300', 'Bus', 'IBUS'], ['1562', 'TGV', 'SNCF']]
```

❑ **Q26** – Implémentez la fonction `VerifieTrie(table, indice)` qui renvoie True si la table table est triée pour l'indice indice et False sinon.

❑ **Q27** – Considérez la fonction de la partie 1 `ConditionConstante(table, indice, constante)`, on suppose que la table `table` est triée. Proposez une implémentation de cette fonction basé sur la dichotomie pour améliorer les performances nommée `ConditionConstanteTrie(table, indice, constante)`.

❑ **Q28** – Expliquer les cas où cette implémentation n'est pas plus efficace ?

Utilisation de dictionnaires pour indexer les bases de données

Une autre idée pour explorer les tables efficacement est d'utiliser des dictionnaires.

Considérons une table T d'arité k et de taille n représentée par une liste Python d'enregistrements $[e_0,...,e_{n-1}]$. Soit L cette liste. Considérons un indice i d'attribut de T tel que $0 \leqslant i < k$.

On peut associer à T et i un dictionnaire Python $Dico_{T,i}$ de la manière suivante. Les clés de ce dictionnaire sont les valeurs possibles pour les valeurs v qui apparaissent pour l'attribut d'indice i dans la table T. L'image associée à une clé v est la liste des positions dans L des enregistrements e tels que $e[i] = v$.

Si l'attribut d'indice i de T ne prend la valeur v pour aucun enregistrement, cette clé n'est pas enregistrée dans le dictionnaire. L'image d'une clé est donc une liste non vide.

Exemple : (Dictionnaire associé à une table). Considérons la table

`Vehicule[[IdVehicule,Type, Compagnie]]`

avec les enregistrements suivants.

```
<98300, Bus, IBUS>
 <1562,TGV, SNCF>
<30990, A320,Hop !>
 <1789,TGV, SNCF>
```

Soit `dico` le dictionnaire $Dico_{Vehicule,1}$ associé à l'attribut `Type` de position 1.

Nous avons

```
> > > for c in dico : print(str(c)+ " → " + str(dico[c]))
Bus → [0]
A320 → [2]
TGV → [1, 3]
> > > dico['Ariane6']
KeyError: 'Ariane6'
```

Application à la condition

❏ **Q29** – Implémentez la fonction `CreerDictionnaire(table,indice)` qui prend en arguments une table `table` et un indice `indice` d'attribut de table. Elle renvoie un dictionnaire de la table `table` pour l'attribut d'indice `indice`.

❏ **Q30** – Implémentez la fonction `ConditionConstanteDictionnaire(table,indice,constante,dico)` qui a la même fonctionnalité que `ConditionConstante`, mais qui prend en plus en argument un dictionnaire `dico` de la table `table` pour l'indice `indice`.

❏ **Q31** – Comparer les complexités de `ConditionConstanteDictionnaire` et de `ConditionConstante`. Décrivez des cas pour lesquels cette nouvelle approche est plus performante. Y a-t-il des cas où elle n'est pas plus performante ?

Application à la jointure

❏ **Q32** – Implémentez la fonction `JointureDictionnaire(table1,table2,indice1,indice2,dico2)` qui a la même fonctionnalité que `Jointure`, mais qui prend en plus en argument un dictionnaire `dico2` pour la table `table2` par rapport à l'indice `indice2`.

❏ **Q33** – Donnez la complexité de votre implémentation par rapport aux tailles et arités respectives des tables `table1` et `table2` ainsi que par rapport à la longueur maximale d'une liste renvoyée par le dictionnaire `dico2`, qui sera notée k_2. Justifiez votre réponse en vous appuyant sur la structure du programme

❏ **Q34** – L'opérateur de jointure prend en arguments deux tables, l'une jouant le rôle de clef primaire, l'autre de clef étrangère. Il serait donc possible d'utiliser un dictionnaire pour `table1` au lieu d'un dictionnaire pour `table2`. Laquelle des deux tables choisir pour avoir des meilleurs performances ?