Rapport
Base de données I

Vandekerckhove Adrien

Année Académique 2020-2021
BAB2 Sciences Informatiques

Faculté des Sciences, Université de Mons

Contents

1 Introduction

Dans ce rapport, je discute de mes réflexions lors de l'implémentation des fonctionnalités demandées pour ce projet. Pour une description détaillée des classes et fonctions, la documentation est fournie dans les fichiers python. Un guide d'utilisation est aussi fourni sous la forme d'un fichier markdown nommé README.md

2 Design et implémentation

Tout d'abord, il convient de réfléchir au fonctionnement de l'algèbre SPJRUD pour mieux le modéliser sous python. Chaque opérateur algébrique s'éxécute avec une ou deux relations et retournent toujours une unique relation. Autrement dit, nous devons prendre en paramètre une ou deux relations pour chaque opérateur et ce dernier doit retourner une nouvelle relation.

La première étape est donc de définir une relation en python. Pour ce faire, j'ai utilisé une classe nommée Rel. Celle-ci possède :

- Un nom
- Un dictionnaire python avec en clé le nom d'une colonne et en valeur le type de cette colonne.
- Les tuples de données que cette relation possède

Pour pouvoir implémenter toutes les fonctionnalités. J'ai choisi d'utiliser une classe par opérateur. L'idée est que pour chaque instance d'opérateur, on a la relation associée à l'opération et la requête SQL qui sont en mémoires sous forme d'attributs. Cette méthode permet de construire récursivement la nouvelle relation et la requête SQL composée.

Pour pouvoir modifier les informations d'une relation, j'ai écris les méthodes appropriées dans la classe Rel qui retournent chacune une nouvelle relation. C'est aussi dans ces méthodes que l'on vérifie les erreurs d'arguments avant de procéder à l'exécution de l'opérateur. Nous pouvons distinguer deux cas d'exécution lorsqu'un opérateur est appelé :

- **Cas de base**
  L'opérateur interagit directement avec une relation. Dans ce cas là, la méthode appropriée est appelée par la classe Rel
- **Cas de récursion**
  L'opérateur de base interagit avec un autre opérateur. Dans ce cas, l'opérateur de base va appeler la méthode d'éxécution de l'opérateur en paramètre. Ce dernier se retrouve alors à nouveau dans l'un des deux cas et s'exécute de manière appropriée.

Pour illustrer ce fonctionnement, voici un exemple avec Select :

```python
class Select(Operator):

    def __init__(self, relation, column_name, target):

        self.rel = relation
        self.column_name = column_name
        self.target = target
        self.execute()

    def execute(self):

        if self.is_atomic(self.rel):
            self.sql = f"""SELECT DISTINCT * FROM {self.rel.name} WHERE {self.column_name} = '{self.target}'"""
            self.result = self.rel.select(self.column_name, self.target)
        else:
            self.sql = f"""SELECT DISTINCT * FROM ({self.rel.sql}) WHERE {self.column_name} = '{self.target}'"""
            self.result = self.rel.result.select(self.column_name, self.target)
```

Figure 1: Exemple pour l'exécution de Select

Le **cas de base** est déterminé à l'aide de *is_atomic()* qui retourne *True* si la relation n'est pas un opérateur. *is_atomic()* est déclaré dans la superclasse *Operator* qui contient les fonctionnalités communes utiles aux opérateurs.

2.1 Avantages du dictionnaire python

Pour garder en mémoire le type de données de chaque colonnes, il est nécessaire d'utiliser une structure de données appropriée. J'ai choisi pour cela d'utiliser les dictionnaires en python. Ceux-ci permettent de vérifier la présence d'une clé avec une complexité algorithmique en O(1) (Explication du hash pour une clé en python)

```
>>> print(students.dtypes)
{'id': 'integer', 'name': 'text', 'age': 'int', 'studies': 'text'}
>>> print("id" in students.dtypes)
True
>>> print("surname" in students.dtypes)
False
```

Figure 2: Vérification de la présence d'une clé dans un dictionnaire

Si la clé est présente dans le dictionnaire, nous pouvons aussi récupérer sa valeur en O(1) :

```
>>> print(students.dtypes["id"])
integer
```

Figure 3: Récupération de la valeur d'une clé.

Ces deux propriétés du dictionnaire en python permettent d'écrire du code simple et optimisé. C'est pour cela que j'ai choisi de m'en servir pour la vérification des types de données de chaque colonnes. Plusieurs morceaux de code en font l'usage dans les méthodes de manipulations de données de la classe Rel.

2.2 La classe utils

Pour communiquer avec une base de données SQLite, il est nécessaire d'utiliser du boilerplate code, c'est à dire du code répétitif. Pour simplifier l'utilisation du programme. Il est possible de se connecter à une base de données sqlite3 à l'aide de la classe utils.Database. Une fois connecté à la base de données, récupérer une relation est trivial. Voici un exemple de comment se connecter à la base de données *testing.db* à partir du répertoire *./python/src* :

```
>>> from utils import Database
>>> db = Database("../resources/testing.db")

>>> students = db.get_relation("students")
>>> cities = db.get_relation("cities")
```

Figure 4: Exemple d'utilisation de la classe Utils

3 Tests

Des tests unitaires pour chaque opérateurs ont été écrits dans le fichier unit_testing.py. Ce fichier teste d'abord la bonne éxécution des opérateurs de manière individuelle puis vérifie le bon fonctionnement des compositions d'opérateurs. Le nombre total de combinaisons entre opérateurs étant très grand, il est possible que des bugs ne soient pas couverts.

4 Bugs

Il existe un bug avec la classe Rename. Etant donné la nature des dictionnaires en python, si l'on utilise Rename et que l'on nomme le nom d'une colonne par le nom d'une colonne existante dans la relation. Alors la nouvelle clé va écraser l'ancienne et le dictionnaire pour les data types va diminuer en taille. En pratique, il n'y a pas de raisons de renommer le nom d'une colonne par celui d'une déjà présente dans la relation donc la probabilité de tomber sur ce bug est relativement faible.

```
>>> students = db.get_relation("students")
>>> print(students.dtypes)
{'id': 'integer', 'name': 'text', 'age': 'int', 'studies': 'text'}
>>> bug = Rename(students, "name", "age")
>>> print(bug.result.dtypes)
{'id': 'integer', 'age': 'int', 'studies': 'text'}
```

Figure 5: Démonstration du bug dans la console python

5 Conclusion

Dans l'ensemble, la logique pour les opérateurs est suffisament consistante et claire à lire. Les manipulations de données sont effectuées en priorisant l'optimisation ce qui permet de travailler avec des bases de données suffisament grandes. Si il y a un point à améliorer, ce serait la gestion des erreurs. Celle-ci est assez générale mais pourrait être plus précises et renvoyer des messages d'erreurs spécifiques au contexte d'exécution.