Année académique 2022-2023

Rapport de projet : Base de donnée

Auteurs

Razanajao Aina
Foucart Axel

Professeur

Wijsen Jef
Buys Alain

Assistant

Bonte Sébastien

Table des matières

1 Introduction

Dans le cadre du cours de base de donnée I en BAC2 science informatique de l'Umons, nous avons reçu commme projet de créer un interpréteur capable de transformer des requêtes SPRJUD en requêtes SQL et ensuite les exécuter. Le language de programmation imposé est Python(version python 3.10.6). La bibliothèque SQLite3 est également imposé pour nous aider à vérifier que les requêtes SQL données fonctionnent sur une base de données.

Nous nous sommes donc répartie les tâches :

Axel : les requêtes "JOIN", "RENAME", "PROJECT" et la classe Relation.

Aina : les requêtes "SELECT", "UNION", "DIFFERENT".

Tâche Communes : La conception de l'arbre syntaxique, du rapport et des différents tests.

Voici un exemple d'interprétation d'une requête, il s'agit d'une projection des attributs "First" et "Second" dans une relation appellée "Exemple" :

Entrée : PROJECT[(First,Second),Exemple]

Sortie : SELECT First, Second FROM Exemple ;

2 Organisation des classes

2.1 Classe Relation

Se trouvant dans le fichier rel.py, elle est la première qui fut développée. Elle permet d'instancier, dans un fichier .bdd, une table qui agira comme une relation vu en théorie en spécifiant ses attributs(i.e. nom de colonne) et les types des attributs(INTEGER, REAL, TEXT, etc...). On pourra alors lui ajouter des tuples s'ils sont compatibles avec la relation (i.e. il respecte le nombre et les types d'attributs spécifiés lors de la création de l'objet Relation).

```
1 args = {"Name":sType.TEXT, "Country": sType.TEXT, "Population" :
      sType.REAL}
2 Cities = Relation("relationdata.db","Cities",args)
3 Cities.addTuple(("Begen","Belgium",20.3))
4 Cities.addTuple(("Bergen","Norway",30.5))
5 Cities.addTuple(("Brussels","Belgium",370.6))
```

Listing 1 – Exemple de création d'un objet Relation

Chaque instance peut être supprimée, ce qui signifie supprimer la table de la base de données. Il est aussi possible de voir la relation après sa création dans un terminal en convertissant la relation en une chaîne de caractères.

```
Résultat de print(Cities):
Cities| Name      | Country  | Population  |
      |-----------------------------------
      | Begen     | Belgium  | 20.3        |
      | Bergen    | Norway   | 30.5        |
      | Brussels  | Belgium  | 370.6       |
```

Pour permettre de retrouver une relation à l'aide de son nom, le fichier rel.py possède une liste auquelle est ajouté chaque nouvelle relation créée. Ceci sera utile pour pouvoir sauvegarder les tables/relations créées pendant une requête. Elle possède aussi d'autre fonctions utiles comme getArgs() qui retournera les arguments de la relation ou encore getNbOfTuple() qui donne le nombre de tuples présent actuellement dans une relation.

2.2 Classes SPJRUD

Chaque instance de classe du fichier SPJRUD.py permet de créer un objet Expression qui va générer une requête SQL, une relation contenant les tuples du résultat de cette requête et un nouveau nom pour cette relation si aucun n'a été donné en paramètre.
Ci-dessous se trouve l'ensemble des classes SPJRUD :

- ◇ Expression : C'est la classe parent de toutes les autres Expressions SPJRUD, elle ne sera jamais instancié directement.

- ◇ Select : Select(attribut1,condition,attribut2/valeur,relation/expression,nom)
  Les trois premiers arguments sont simplements les mêmes arguments d'une selection théorique comme $\sigma_{a=b}R$.
  Ici, tous les paramètres sont des str, sauf la relation, car le constructeur s'occupera de vérifier si l'utilisateur veut comparer une colonne avec une autre ou avec une constante.
  Le premier argument doit être le nom d'un atttribut de la relation. Concernant le troisième argument, si un nom d'une autre colonne est donné, alors le type de celui-ci doit être compatible avec le type du premier argument.
  La condition, c'est à dire le deuxième argument, doit faire partie de : $!=, <>, =, >, <, >=, <=$. Le quatrième argument est la relation dans laquelle on effectue l'opération ou une expression. Si une de ces conditions n'est pas respectées, une erreur sera levée avant l'execution de la requête SQL.
  Exemple:
  ```
      U = Select("W","=","D2",STOCK,"U")
  print(U.newRel) donne:
      U| W   | Product  | Color   | Qty  |
       |---------------------------------
       | D2  | handle   | red     | 100  |
       | D2  | hinge    | yellow  | 200  |
       | D2  | lock     | blue    | 100  |
  ```

Pour avoir une requête équivalente en SQL, il suffit de rajouter "WHERE attribut1 condition attribut2" à la fin d'une requête.

Exemple : considérons la relation DEPARTEMENT, on veut les départements possédant un budget supérieur ou égale à 4000.

$$select[(budget, >=, 4000), DEPARTEMENT]\ ^{1}$$

$$\equiv$$

$$SELECT\ *\ FROM\ DEPARTEMENT\ WHERE\ budget < 4000;$$

◇ Project : Project[(attribut1,...,attributN),relation/expression]
Le dernier argument est la relation dans laquelle on effectue l'opération ou une expression. Les premiers arguments sont les mêmes qu'une projection théorique : $\pi_{a,b,c}R$.
Ces arguments sont des chaînes de caractères car on désire obtenir des colonnes. Si une de ces conditions n'est pas respectées, une erreur sera levée avant l'execution de la requête SQL.
Pour avoir une requête équivalente en SQL, il suffit de spécifier les colonnes(attributs) données dans la clause SELECT de la requête.

Exemple : considérons la relation EMPLOYES où on veut savoir depuis quand l'employée travaille dans le département.

$$project[(Emp, Depuis), EMPLOYES]$$

$$\equiv$$

$$SELECT\ Emp, Depuis\ FROM\ EMPLOYES;$$

◇ Join : join(relationA/expressionA,relationB/ExpressionB)
Les arguments sont les relations ou les expressions dont on veut faire la jointure. En théorie, cela donne $relA \bowtie relB$.
Une erreur sera levée avant l'exécution de la requête si une des conditions n'est pas respectées, i.e. si les deux paramètres ne possèdent pas les mêmes arguments.
Pour avoir une requête équivalente en SQL, il faut sélectionner la première table puis ajouter *NATURAL JOIN* avant d'écrire la deuxième table.
En SQL, il existe l'opération "NATURAL JOIN". Ceci nous permet d'effectuer une jointure entre deux requêtes si et seulement si le format demandé est respecté. En effet, pour effectuer cette opération, les colonnes des relations doivent avoir les mêmes noms et les mêmes types de valeurs.

Exemple : considérons la relation STOCK et la relation WAREHOUSES. On veut faire une jointure entre eux.

$$join[STOCK, WAREHOUSES]$$

$$\equiv$$

$$SELECT * FROM\ STOCK\ NATURAL\ JOIN\ WAREHOUSES;$$

1. Écrit avec la syntaxe de l'interpréteur

◇ Rename : Rename(oldName,newName,relation/expression)
Les premiers arguments sont des chaînes de caractères car elles designent la colonne à modifer. Le permier est le nom initiale de la colonne, celle dont on veut changer le nom. Le deuxième est le nouveau nom.
Le dernier argument est également une chaîne de caractère car s'il s'agit d'une relation ou bien, d'une expression. Une erreur est levée avant l'exécution de la requête SQL.
En SQL, l'oprération "AS" est utilisé pour faire un alias. C'est-à-dire, un renommage temporaire d'une colonne ou d'un tableau. Dans notre cas, le renommage est permanent car elle donne une nouvelle table.

Exemple : considérons la relation STOCK où l'on veut changer l'attribut "Qty" en "Amount".

$$rename[(Qty, Amount), STOCK]$$
$$\equiv$$
$$SELECT\ Qty\ AS\ Amount\ FROM\ STOCK;$$

◇ Union : Union(relationA/expresionA,relationB/expressionB)
Il ne peut avoir que deux arguments pour cette opération. Les deux doivent être une relation, donc une chaîne de caractère, ou alors une expression. Si ce n'est pas le cas, alors une erreur est levée avant l'exécution de la requête SQL.
L'équvalent SQL est l'opération "UNION" permet d'effectuer la concaténation de deux relations ou expressions sous certaines conditions : le nombre de colonnes, le type des valeurs et l'ordre des colonnes.

Exemple : considérons la relation DEPARTEMENTS et la relation EMPLOYES. On veut faire l'union de ces tables.

$$union[DEPARTEMENTS, EMPLOYES]$$
$$\equiv$$
$$SELECT * FROM\ DEPARTEMENTS\ UNION\ EMPLOYES;$$

◇ Different : different(relationA/expressionA,relationB/expressionB)
Cette opération ne prend que deux arguments. Une relation, dans ce cas les arguments sont des chaînes de caractères, ou une expression. En cas de non-respect, une erreur est levée avant l'exécution de la requête SQL.
En SQL, l'opération "EXCEPT" est utilisée entre deux tables ou deux expressions. Elle permet d'obtenir les valeurs du premier argument sans celle présente dans le deuxième. En d'autre mot, si une valeur se trouve dans les deux arguments, alors elle n'est pas retenu.
Exemple : considérons la relation EMPLOYE. Nous voulons faire la différence de cette relation avec lui même.

$$difference[EMPLOYE, EMPLOYE]$$
$$\equiv$$
$$SELECT * FROM\ EMPLOYE\ EXCEPT\ EMPLOYE;$$

3 L'arbre Syntaxique

Pour convertir une requête provenant du terminal, i.e. une chaîne de caractère, en une requête SQL, nous utilisons un arbre syntaxique qui fonctionne de manière récursive pour décomposer chaque expression d'une requête en créant pour chacune un objet Expression approprié. Voici ci-dessous un exemple :

Soit une requête :

```
r = "select[(W,=,D1),project[(W,Product),STOCK]]"
```

Un tokenizer [2] va décomposer **r** en une liste :

```
tokens = ['select', '[', '(', 'W', ',', '=', ',', 'D1', ')', ',', '
    project', '[', '(', 'W', ',', 'Product', ')', ',', 'STOCK', ']'
    , ']']
```

Si tokens[0] n'est pas reconnu comme une expression SPJRUD ou encore que le deuxième et dernier tokens ne sont pas des crochets ouvrant et fermant, une erreur sera levée. Sinon il faut trouver les différents arguments de l'expression, ce qui dépendra de la syntaxe de chaque expression.
Ici, dans le cas d'un select, on s'attend à avoir : *"(", "arg1", "condition", "arg2", ")", ",", "relation/expression"*. (Si c'est une expression, alors elle s'étendra sur plus d'un tuple comme dans cette exemple). Si la syntaxe n'est pas respectée, alors une *syntax error* sera affichée décrivant le problème.
Lors d'une requête imbriquée, un appel récursif sera utilisé pour décomposer les expressions. Dans cette exemple, on va renvoyer :

```
tokens = ['project', '[', '(', 'W', ',', 'Product', ')', ',', '
    STOCK', ']']
```

Et donc les étapes précédentes seront répétées pour cette expression. On peut alors voir le cas où le nom d'une relation est donnée comme le *cas de base* de cet algorithme. Aussi si le nom de la relation donnée n'existe pas, i.e. ce n'est pas une relation crée pendant l'exécution de ce programme, alors une *Argument Error* sera affichée.
Ici l'appel récursif précédent nous retournera un objet Project. On pourra donc créer un objet Select en passant en paramètre tous les arguments récupérés à partir des tokens.

```
rep = Select("W","=","D1",expProject)
return rep
```

Il suffira alors d'afficher dans le terminal la nouvelle relation créée et sa requête sql.

```
print(rep.newRel)
print(rep.querry)
```

2. Vu qu'il n'existe pas de Tokenizer en python, nous avons dû l'implémenter

4 Manuel D'utilisation

4.1 Configuration de l'interpréteur

Pour utiliser l'interpréteur il suffira d'importer le fichier AST.py dans le fichier executé et d'utiliser la fonction readUserQuery(database), avec database, le nom du fichier *.db* dans lequel les requêtes seront effectuées.

4.2 Utilisation dans le terminal

Une fois le programme lancé, plusieurs choix s'offrent à l'utilisateur.

Afficher la syntaxe des expressions : Il est possible d'avoir la syntaxe de toutes les expressions en entrant *help*.

Afficher une relation : Pour afficher une relation il suffit d'entrer le nom de cette dernière.

Effectuer une requête : Pour interpréter une requête SPJRUD en SQL, il faut entrer l'expression voulu dans le terminal. La relation résultante sera alors affichée avec l'expression SQL générée. Il n'y a pas besoin de spécifier si un argument est une constante ou non, l'interpréteur s'en occupera.

```
Ex:
    > select[(W,=,D1),STOCK]
    ...
```

Sauvegarder une requête avec un nom : Il est possible de donner un nom à une relation pour pouvoir la réutiliser avec d'autres requêtes du programme. Attention ces relations ne sont que temporaires et ne seront plus utilisable après la fin du programme. Il faut aussi retenir que le nom choisit ne doit pas être séparé ni contenir de caractères spéciaux.

```
Ex:
    > R = select[(W,=,D1),STOCK]
    ...
    > project[(W),R]
```

Quitter l'interpréteur : Pour quitter l'interpréteur, il suffit d'entrer *stop, close, exit* ou *quit*.

4.3 Les erreurs possibles

Plusieurs erreurs sont possibles lors de l'execution du programme. Elles décrivent le problème rencontré(dans la mesure du possible) pour aider l'utilisateur à le résoudre.

Syntax Error : Décrit une erreur liée au non-respect de la syntaxe d'une expression SPJRUD.

Argument Error : Arrive lorsqu'une relation donnée en argument n'existe pas.

Unknown Error : Peut parfois arriver lorsqu'une erreur de syntaxe n'est pas détectée et provoque une erreur d'index pendant l'exécution de l'arbre syntaxique.

Expression Error : Décrit une erreur lancée par une expression lors du non-respect de ses paramètres.

4.4 Les relations disponibles

Pour tester notre code, il est possible d'utiliser des relations(provenant des exercices du syllabus) déjà implémentées dans le code en ajoutant dans les imports :

```
import relationsForTesting
```

Elles seront alors utilisables par l'interpréteur dans la base de données "relationdata.db". Ci-dessous se trouvent l'ensemble des tables implémentées :

1. WAREHOUSES

W	Address	City
D1	6, Rue de l'Eglise	Mons
D2	18, Place du Parc	Mons
D3	18, Place du Parc	Chimay
D4	5, Avenue Louise	Enghien

2. STOCK

W	Product	Color	Qty
D1	hinge	yellow	200
D1	hinge	blue	150
D2	lock	blue	100
D2	hinge	yellow	200
D2	handle	red	100
D4	hinge	red	150
D4	lock	red	600

3. EMPLOYE

NrEmp	Dept	Pourcent
E1	Info	40
E1	Bio	40
E2	Eco	100
E3	Bio	50
E3	Eco	50
E4	Eco	100
E5	Eco	50
E5	Bio	25
E5	Info	25

4.

DEPARTEMENT	NomDept	Budget	Chef
	Info	5000	E1
	Bio	3500	E1
	Eco	4000	E5

5.

EMPLOYES	Emp	Dept	Depuis
	Ed	Jeux	8 jan 1982
	Ed	MIS	11 jan 1997
	An	Jeux	13 sep 2001
	Pierre	MIS	23 oct 1992
	Pierre	Jeux	11 nov 1995

6.

DEPARTEMENTS	Dept	Chef
	Jeux	An
	MIS	Pierre

7.

Cities	Name	Country	Population
	Begen	Belgium	20.3
	Bergen	Norway	30.5
	Brussels	Belgium	370.6

8.

CC	Name	Capital	Population	Currency
	Belgium	Brussels	10255.6	EUR
	Norway	Oslo	4463.2	NOK
	Japan	Tokyo	128888.0	YEN

5 Choix d'implémentation

5.1 Difficultés rencontrées

SQLite3 : Lors de la création du fichier *rel.py*, SQLite3 nous a causé beaucoup de problème vu qu'une base de données ne peut pas posséder plusieurs connexions à la fois. Nous avons été obligé de trouver une solution pour permettre à plusieurs d'objets d'intéragir sur une même base de données.

La solution utilisée a été de faire attention à bien fermer la connexion entre l'objet et la base de données après avoir effectué l'action, ce qui nous oblige à établir une nouvelle connexion avant chaque modification.

L'arbre Syntaxique : Avant de commencer son implémentation, nous avons chercher différentes façons de le réaliser. Nous avions le choix de l'implémenter de façon itérative, récursive ou encore en utilisant des RegEx, mais nous avons finit par choisir la version récursive.

Son implémentation n'a pas été facile, dû à la syntaxe des requêtes SPJRUD, qui varient avec chaque expression, ce qui nous a fait rencontrer beaucoup d'erreurs d'indices liées à la recherche d'arguments dans la liste des tokens. Mais nous avons quand même réussit à créer un algorithme stable et répondant aux exigences de ce projet.

5.2 Fonctionnalités supplémentaires

L'interpréteur entièrement utilisable en dehors du terminal à l'aide des différentes classes implémentées et la sauvegarde temporaire des requêtes avec un nom n'étaient pas demandé dans l'application de base.