UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Contributions fonctionnelles à JALF :

l'algèbre relationnelle en Java

Mémoire présenté par

Aimable HIRWA , Anne-France VAN SWIETEN

en vue de l'obtention du grade de Master

sciences informatiques

Option: software engineering and programming systems

Promoteur

Dr. Bernard LAMBEAU

Lecteurs

Dr. Kim MENS, Antoine CAILLIAU

Année académique 2015-2016

Remerciements

En premier lieu, nous tenons à remercier notre promoteur le Dr. Bernard LAMBEAU qui a été très disponible et impliqué dans l'avancement de ce projet. En tant que Directeur de mémoire, il nous a guidés dans la réalisation de notre travail et nous a aidés à trouver des solutions aux problèmes à résoudre.

Nous remercions ensuite Dr. Kim MENS ainsi que M. Antoine CAILLIAU qui ont accepté d'être nos lecteurs. Nous remercions également toutes les personnes qui nous ont aidés dans la réalisation du mémoire, que ce soit dans la correction orthographique et grammaticale ou pour la correction du contenu. Ainsi, nous remercions tout particulièrement Rosine INGABIRE, Nadine UMUTONI SENGEGERA, Stéphane ONDERBEKE, Thierry VAN SWIETEN et Pascale FRANSOLET.

Et enfin, nous remercions nos familles ainsi que nos amis qui nous ont apporté leur soutien moral durant la réalisation de ce mémoire.

Résumé

Dans ce mémoire, nous présenterons une librairie de connexion à une base de données JAlf qui s'appuie sur la théorie de l'algèbre relationnelle. Cette librairie, contrairement aux autres, offre une solution plus adaptée aux travaux des développeurs. Elle permet de faciliter la définition de requêtes complexes et la découpe de celles-ci en modules, ce qui est une fonctionnalité importante pour un développeur. Ce mémoire va débuter par la présentation des motivations qui ont conduites à la création de la librairie, en mettant en avant les problèmes liés aux autres librairies. Il va ensuite mentionner les différents outils utilisés lors du développement de la librairie et le modèle théorique sur lequel elle s'appuie. Et pour terminer le mémoire va expliquer le travail réalisé sur cette librairie.

Table des matières

Introduction 1

1 Motivations **3**

1 Alf 3

1.1 Définition 3

1.2 Limitation d'Alf 5

2 Relation as first-class citizen 5

2.1 Modularité 6

2.2 Du calcul relationnel à l'algèbre relationnelle 9

2.3 Problèmes avec Query Builders 10

2.4 Problèmes avec Object-Relational Mapping 11

2.5 Inspiration sur HaskellDB 11

3 Exemple concret d'utilisation d'Alf 12

3.1 Définition du problème 12

3.2 Solution au problème 12

4 Projet apparenté 15

4.1 Tutorial-D 15

4.1.1 Définition 15

4.1.2 Limitation de Tutorial-D 16

4.1.3 Rel 16

2 Outils et méthodes de travail **18**

1 Outils utilisés 18

1.1 Git 18

1.2 GitHub 19

1.3 Trello 20

1.4 Java 8 20

1.4.1 Streams 21

1.4.2 Les lambdas et les fonctions 22

2 Méthodes 22

2.1 Méthode agile 22

2.1.1 Définition 22

2.1.2 Implication d'agile dans le projet . . . 23
2.1.3 Trello pour agile . . . 23
2.1.4 Revue de code . . . 24
2.2 TDD - Test Driven Development . . . 25
2.2.1 Approche traditionnelle . . . 25
2.2.2 Approche Test Driven Development (TDD) . . . 25

3 Modèle théorique **27**
1 Algèbre relationnelle . . . 27
1.1 Introduction . . . 27
1.2 Attribut, Type, Tuple et Relation . . . 28
1.3 Opérateurs relationnels . . . 29
1.3.1 Projection (PROJECT) . . . 29
1.3.2 Sélection (SELECT) . . . 30
1.3.3 Jointure (JOIN) . . . 31
1.3.4 Renommage . . . 32
1.4 Opérateurs ensemblistes . . . 33
1.4.1 Intersection (INTERSECT) . . . 33
1.4.2 Union . . . 34
1.4.3 Différence (MINUS) . . . 34
2 Clés candidates . . . 35

4 Notre contribution au projet Jalf **37**
1 Notions spécifiques à JAlf . . . 37
1.1 La notion de relation . . . 37
1.1.1 L'interface principale Relation . . . 37
1.1.2 Relation en mémoire . . . 38
1.1.3 Relation résultante d'un opérateur . . . 38
1.2 La notion de Type . . . 42
1.3 La notion de compilation . . . 44
1.4 La notion d'optimisation . . . 47
1.4.1 Optimisation de l'opérateur project . . . 47
1.4.2 Optimisation de l'opérateur Restrict . . . 48
1.4.3 Optimisation de l'opérateur Union . . . 48
1.4.4 Optimisation de l'opérateur Join . . . 49
2 Construction de nos deux relations d'exemples . . . 52
3 Ajout de Union . . . 53
3.1 Utilisation de Union dans JAlf . . . 53
3.1.1 Petit rappel théorique . . . 53
3.1.2 Emploi de union dans la JAlf . . . 54

3.2 Comment avons-nous implémenté Union dans JAlf 55
3.2.1 Ajout d'un test 55
3.2.2 Ajout de Union dans la classe DSL 55
3.2.3 Ajout Union dans l'interface Relation et dans la classe AbstractRelation 56
3.2.4 Ajout de la classe Union dans le package jalf.relation.algebra . . 56
3.2.5 Compilation de l'opérateur Union 57
3.2.6 Optimisation de l'opérateur Union 60
4 Ajout de l'opérateur Intersect 64
4.1 Utilisation de Intersect dans JAlf 64
4.1.1 Petit rappel théorique 64
4.1.2 Emploi de Intersect dans JAlf 64
4.2 Comment avons-nous implémenté Intersect dans JAlf 64
5 Ajout de Minus 65
5.1 Utilisation de minus dans JAlf 65
5.1.1 Petit rappel théorique 65
5.1.2 Emploi de Minus dans la Dsl 65
5.2 Comment avons-nous implémenté Minus dans JAlf 66
6 Ajout de Summarize 67
6.1 Ajout summarize dans la classe DSL.java 68
6.2 Ajout summarize dans l'interface Relation.java et la classe AbstractRelation.java 68
6.3 Ajout des différents agrégats dans la classe DSL 69
6.4 L'interface Aggregator 69
6.4.1 La classe Count 70
6.4.2 La classe Max 71
6.4.3 La classe Avg 71
6.5 La classe Summarize 72
6.6 Optimisation de l'opérateur Summarize 75
7 Ajout des classes Key et Keys 76
7.1 Création du package jalf.constraint 76
7.2 Ajout des méthodes key et keys dans la classe DSL 76
7.3 Classes key et Keys 77
7.3.1 Calcul de Keys de l'opérateur Intersect 78
7.3.2 Calcul de Keys de l'opérateur Minus 79
7.3.3 Calcul de Keys de l'opérateur Union 79
7.3.4 Calcul de Keys de l'opérateur Project 80
7.3.5 Calcul de Keys de l'opérateur Restrict 80
7.3.6 Calcul de Keys de l'opérateur Rename 81

7.3.7 Calcul de Keys de l'opérateur Join . . . 81
7.3.8 Optimisation de l'opérateur Project grâce à la clé . . . 82
Conclusion . . . 83
Bibliographie . . . 84

Annexe **86**

A Architecture **i**
1 Les packages . . . i
1.1 Le package jalf . . . i
1.1.1 La classe DSL.java . . . i
1.1.2 Les classes AttrName et AttrList . . . ii
1.1.3 L'interface Type . . . ii
1.1.4 L'interface Relation . . . ii
1.1.5 L'interface Visitor . . . ii
1.1.6 La classe Tuple . . . iii
1.2 Le package jalf.compiler . . . iii
1.2.1 La classe AbstractRelation . . . iii
1.2.2 La classe Compiler . . . iii
1.2.3 La classe Cog . . . iii
1.2.4 La classe BaseCog . . . iii
1.3 Le package jalf.constraint . . . iii
1.3.1 L'interface Constraint . . . iv
1.3.2 La classe Key . . . iv
1.3.3 La classe Keys . . . iv
1.4 Le package jalf.optimizer . . . iv
1.4.1 La classe DefaultMapper . . . iv
1.4.2 La classe Optimizer . . . iv
1.4.3 La classe Optimized<R extends Relation> . . . v
1.4.4 La classe OptimizedMinus . . . v
1.5 Le package jalf.relation.algebra . . . v
1.5.1 La classe Operator . . . v
1.5.2 L'interface LeafOperand . . . v
1.5.3 La classe Project . . . v
1.5.4 La classe Union . . . vi
1.6 Le package jalf.relation.csv . . . vi
1.6.1 La classe CsvRelation . . . vi
1.7 Package jalf.relation.materialized . . . vii
1.7.1 La classe MemoryRelation . . . vii
1.7.2 La classe SetMemoryRelation . . . vii

1.8 Package jalf.type vii
1.8.1 La classe Heading vii
1.8.2 La classe HeadingBasedType vii
1.8.3 La classe RelationType vii
1.8.4 La classe TupleType viii
1.9 Le package jalf.aggregator viii
1.9.1 La classe Summarize<T> viii
1.9.2 L'interface Aggregator<T> viii
1.9.3 La classe Avg viii
1.9.4 La classe Count ix
1.9.5 La classe Max ix

Introduction

Dans le domaine des bases de données relationnelles, il est habituel d'utiliser le langage SQL pour écrire des requêtes. SQL est un langage plus basé sur le calcul relationnel que sur l'algèbre relationnelle. En effet, pour son inventeur, l'idée de base était qu'il serait plus facile pour un utilisateur de décrire ce qu'il voulait comme résultat plutôt que de décrire l'algorithme pour obtenir ce même résultat. Pour cela, il fallait donc que le langage soit déclaratif plutôt qu'impératif ou procédural. Par la suite, des applications ont permis aux utilisateurs de faire des requêtes sans devoir connaître le langage SQL, ce qui ne correspondait finalement plus au but premier du langage et c'est pour cette raison que SQL a été principalement utilisé par les programmeurs. Avec le temps, nous avons vu apparaître une multiplicité de librairies facilitant l'utilisation de SQL pour les programmeurs, à travers divers composants lors de développements de logiciels.

Développer un logiciel est toutefois bien différent que de faire des requêtes dans une base de données. En effet, une requête pose une seule exigence à résoudre alors qu'un logiciel en exige plusieurs. Bien que très pratique pour résoudre des requêtes de tous types, SQL n'est pas adapté quand il s'agit de combiner plusieurs concepts d'architectures. Ce besoin de pouvoir tout moduler est pour un développeur une exigence capitale. De plus, SQL ne facilite pas la définition de requêtes complexes. Des requêtes qui parfois sont dans leurs conceptions faciles à énoncer et peuvent devenir un cauchemar pour les écrire. C'est le cas avec les requêtes qui contiennent des sous-requêtes pouvant mener à une hiérarchie de requêtes donnant alors un plan d'exécution erroné pour le système de gestion de base de données. De ces points de vue, il apparaît comme une évidence que pour le développement de logiciel, SQL ne convient pas.

La plupart des librairies de connexion à une base de données sont fortement liées à SQL et sont confrontées aux soucis qu'apporte SQL. De plus, elles ne permettent que des connexions à des bases de données SQL. On peut remarquer de nos jours qu'il existe plusieurs possibilités de sources de données comme des fichiers .csv, .xml ou encore .txt, qui ne sont pas exploitées par ces librairies. Or nous pouvons très facilement trouver dans des entreprises des tas de classeurs .csv contenant des tableaux de données. Ne pourrions-nous pas les exploiter comme données de requêtes tout comme les tables d'une base de données SQL ? C'est une des problématiques que notre solution tente de résoudre.

Notre solution se base sur l'algèbre relationnelle contrairement à SQL. C'est donc une solution offrant une programmation procédurale qui est beaucoup mieux adaptée aux développeurs. Notre librairie propose une solution qui pallie aux problèmes énoncés et liés au langage SQL. Elle expose complètement l'algèbre relationnelle et les relations, au sens relationnel (ensemble de tuples), à l'utilisateur en lui donnant la possibilité de manipuler les relations comme des citoyens de première classe, ce qui n'est pas le cas des librairies actuelles qui elles, exposent SQL. Toutefois, comme l'algèbre relationnelle et le calcul relationnel ont la même puissance d'expressivité, notre

solution peut traduire les requêtes de SQL vers l'algèbre relationnelle.

Chapitre 1

Motivations

Dans ce chapitre nous allons expliquer les motivations du projet JAlf. Dans cette perspective, nous nous basons sur la motivation du projet Alf qui est la version Ruby de JAlf. En effet, les deux projets sont quasi identiques, si ce n'est qu'elles sont écrites dans des langages différents. Il s'agira également pour nous de faire une comparaison avec d'autres projets existants pour démontrer l'intérêt de contribution au projet JAlf.

1 Alf

1.1 Définition

JAlf est une librairie de connexion à une base de données offrant un langage de requêtes basé sur l'algèbre relationnelle. JAlf est la version Java de Alf, une librairie de connexion à une base de données offrant un langage moderne de requête et développé par le docteur Dr. Bernard LAMBEAU de l'Université Catholique de Louvain. Elle est basé sur la théorie de l'algèbre relationnelle inspirée par le langage Tutorial-D et l'ouvrage « The Third Manifesto » de Hugh DARWEN et C.J.Date. Alf est implémentée comme un *Domain Specific langage* en Ruby et permet de faire des requêtes en Ruby. Elle offre aussi un outil de commande en ligne « Alf in Shell » pour faire des requêtes à partir du Shell.
Alf permet d'effectuer des requêtes sur des bases de données multiples et variées sans se restreindre aux bases de données SQL. Elle dispose d'un compilateur intelligent qui lui permet de déléguer à des systèmes sous-jacents tels que MySQL, PostgreSQL, ce qui peut l'être. Il est également possible de travailler directement avec des fichiers .csv, .json, .yaml, .txt au lieu d'une base de données. Alf permet également de convertir un format de fichier en un autre format. Les données lues d'une source de données sont transformées dans un format unique appelé relation. Alf nous permet d'exécuter des requêtes sur des données pouvant provenir de différentes bases de données comme si ces données se situaient dans une seule et même base de données SQL.
Étant donné qu' Alf se base sur l'algèbre relationnelle, elle fournit un langage procédural, visant

à décrire la manière dont doit être construite la solution, contrairement à SQL, qui est un langage déclaratif visant à décrire le résultat attendu.

Pour mieux comprendre le fonctionnement d' Alf, voici quelques exemples d'opérateurs. Par exemple, une projection en Alf aura la syntaxe suivante :

Listing 1.1 – projection en Alf

```
project(operande: Relation, attributs: AttrList) -> Relation
```

dans laquelle l'`operande` est la relation sur laquelle se porte la projection et dont `attributs` représente les attributs concernés dans la relation résultat. La même requête en SQL donne :

```
SELECT attributs FROM operande
```

Autre exemple avec une jointure en Alf, on aura la syntaxe suivante :

Listing 1.2 – recherche les noms et prénoms des personnes qui sont employées en Alf

```
join(gauche: Relation, droite: Relation) -> Relation
```

La même requête SQL de la jointure donne :

```
SELECT * FROM gauche NATURAL JOIN droite
```

On peut également combiner l'utilisation de plusieurs opérations pour des requêtes plus complexes. Par exemple, si on définit la relation PERSONNES et la relation EMPLOYEES :

IdPersonne	Nom	Prenom
5	Durand	Caroline
1	Germaine	Stan
12	Dupont	Lisa
1	Germaine	Rose-Marie

Table 1.1 – PERSONNES

Nom	Prenom	Ville
Durand	Caroline	Mons
Germaine	Stan	Tubize
Zeuz	Rose	Louvain-la-Neuve

Table 1.2 – EMPLOYEES

et qu'on recherche les noms et prénoms des personnes qui sont employées, en Alf on aura la requête :

Listing 1.3 – recherche les noms et prénoms des personnes qui sont employées en Alf

```
project(join(PERSONNES, EMPLOYEES), [:Nom, :Prenom]) -> Relation
```

qui est l'équivalent SQL de :

```
SELECT Nom, Prenom FROM PERSONNES NATURAL JOIN EMPLOYEES
```

Nous obtenons après calcul la relation :

Nom	Prenom
Durand	Caroline
Germaine	Stan

TABLE 1.3 – JOIN de PERSONNES et EMPLOYEES

La requête Alf décrit qu'il faut d'abord effectuer une jointure des relations PERSONNES et EMPLOYEES, et ensuite faire la projection sur les attributs `Nom` et `Prenom`. Dans le chapitre 3 nous définirons les modèles théoriques utilisés.

1.2 Limitation d'Alf

L'inconvénient d'Alf réside dans le fait qu'il s'agisse d'une librairie écrite en Ruby. Ruby étant un langage peu utilisé, les entreprises sont donc moins enclines à utiliser Alf. Dans la mesure où il est notable que de nombreuses entreprises utilisent Java comme langage de développement, il serait donc plus judicieux pour ces entreprises d'utiliser un langage de requête en Java. Dès lors, il ne sera plus nécessaire de procéder à toute autre configuration afin d'effectuer la conversion des requêtes Java vers Ruby et inversement. L'objectif du projet JAlf est d'amener l'algèbre relationnelle et les relations dans le monde des entreprises, ce qu'Alf n'a pas encore réalisé.

2 Relation as first-class citizen

JAlf s'appuie sur un nouveau genre d'interopérabilité [1] de bases de données appelé **Relations as First-Class Citizens**. Ce paradigme fait que JAlf donne une approche différente des autres, car il permet d'exposer des relations fortement typées comme des citoyens de première classe au sein du langage, comme d'autres structures de données telles que les listes, les hash-tables, les tableaux, les arbres, etc. JAlf expose à l'utilisateur des relations au sens relationnel (un ensemble de tuples) et l'algèbre relationnelle. Contrairement aux autres approches, JAlf veut amener le concept de l'algèbre relationnelle en dehors du monde des bases de données. Le but étant de permettre au développeur de manipuler des objets relations provenant d'une source de données quelconque et non nécessairement d'une base de données SQL.

1. Capacité de matériels, de logiciels ou de protocoles différents à fonctionner ensemble et à partager des informations

2.1 Modularité

Le concept de modularité est un des objectifs du paradigme **Relation as first-class citizen**. C'est un concept-clé, car il permet d'une part, d'éviter un grand couplage entre la requête principale et ses sous-requêtes, et d'autre part de découper une requête en plusieurs requêtes réutilisables. Ce qui est important dans ce concept, c'est la volonté de pouvoir encapsuler des exigences dans des méthodes/fonctions pour les réutiliser quand c'est nécessaire. Par exemple : si on définit les relations FOURNISSEURS et VILLES.

:sid	:name	:status	:city
S1	Smith	20	Londres
S2	Jones	10	Paris
S3	Black	30	Paris
S4	Clark	20	Londres
S5	Adams	30	Athènes

TABLE 1.4 – FOURNISSEURS

:city	:country
Londres	Angleterre
Paris	France
Athènes	Grèce
Bruxelles	Belgique

TABLE 1.5 – VILLES

On définit ensuite une requête qui doit nous fournir pour une ville donnée une relation avec comme attributs `:sid`, `:name`, `:city` et `:country`.
On a comme exigences :

1. que le `:status` ne soit visible pour les utilisateurs ;
2. que l'utilisateur ne puisse voir que les fournisseurs de la même ville ;
3. que lui et le `:country` soient affichés en même temps que le `:city`.

Ces trois exigences sont des spécifications que devra respecter un logiciel permettant des requêtes dans cette base de données. Donc si on est l'utilisateur (fournisseur) $S3$ on pourra voir la relation :

:sid	:name	:city	:country
S1	Jones	Londres	Angleterre
S3	Black	Paris	France

TABLE 1.6 – résultat pour le fournisseur $S3$

Lors de la conception d'un logiciel, la modularité est importante pour éviter de réécrire les mêmes lignes de code tout au long de l'implémentation. Dans notre cas, nous souhaitons rendre

modulable notre requête afin de réutiliser certaines de ses parties. Si on revient à notre requête, on peut voir que ces exigences impliquent une restriction (même `city`), une sélection (pas de `status`) et une jointure (relations VILLES et FOURNISSEURS). Ces exigences sont applicables à chaque fois et devraient donc être mises en module. Si on en revient à notre requête, on pensera naturellement à cette architecture dans le langage Sequel : [2]

Listing 1.4 – tentative de moduler la requête en Sequel

```
# exigences 1 et 2
def suppliers_in(city)
  DB[:suppliers]
    .select(:sid, :name, :city)
    .where(:city => city)
end

# exigence 3
def with_country(operand)
  operand.natural_join(:cities)
end

# composition des exigences
requester_city = ... # une ville provenant du contexte
with_country(suppliers_in(requester_city))
```

Or la réponse renvoyée par `with_country` ne donne pas le résultat souhaité. En effet, la requête SQL générée est :

```
SELECT sid, name, city
FROM suppliers NATURAL JOIN cities
WHERE (city = ...)
```

Cette requête perd l'attribut `country` qui est l'intérêt de la jointure. Le problème dans l'exemple ci-dessus est que l'opérateur de jointure n'est pas une jointure telle que définie par l'algèbre relationnelle. D'autres opérateurs présentent également le même problème que la jointure.

Dans les librairies actuelles de connexion à une base de données, la modularité a tendance à échouer étant donné que ces librairies se basent la plupart du temps sur le langage SQL. En SQL il n'est pas possible de formuler les requêtes dans le sens qu'on le désire, c'est-à-dire que deux requêtes qui en algèbre sont équivalentes peuvent donner des résultats différents. Par exemple, on pourra voir que les deux requêtes ci-dessous, dans le langage Sequel sont semblables mais ne donnent pas le même résultat :

2. C'est une librairie Ruby de la famille des librairies Query Builders.

Listing 1.5 – requêtes non équivalentes en Sequel

```
fournisseurs.natural_join (cities ).select (:sid, :name, :city, :country )
# <=!=>
fournisseurs.select (:sid, :name, :city ).natural_join (cities.select (:city,
    :country ) )
```

Cela démontre aussi pourquoi on ne peut pas construire le type d'architecture vu avec l'exemple du code Sequel. SQL n'offre pas de moyen simple et stable de découper une requête en plusieurs requêtes, et c'est pour cette raison que les librairies actuelles ne peuvent pas résoudre le problème de découpe en module.

Toutefois, SQL permet de résoudre ce problème, mais malheureusement difficilement. De plus, cela peut mener à des requêtes complexes avec un plan d'exécution erroné pour le système de gestion de base de données SQL. Par exemple, si on repart de notre requête avec nos trois exigences, on peut voir que la librairie Sequel permet de résoudre notre problème en utilisant à chaque fois `from_self` :

Listing 1.6 – requêtes résolue en Sequel

```
def suppliers_in(city)
  DB[:suppliers]
    .select(:sid, :name, :city)
    .where(:city => city)
    .from_self
end

def with_country(operand)
  operand
    .natural_join(:cities)
    .from_self
end

requester_city = ... # une ville provenant du contexte
with_country(suppliers_in(requester_city))

# SELECT * FROM (
#   SELECT * FROM (
#     SELECT sid, name, city FROM suppliers
#     WHERE (city = ...)
#   ) AS 't1'
#   NATURAL JOIN cities
# ) AS 't1'
```

JAlf a été construit pour faciliter la modularité grâce à son utilisation de l'algèbre relationnelle. Avec JAlf, l'exemple donné ci-dessus est facilement réalisable avec les deux requêtes suivantes :

Listing 1.7 – recherche les noms et prénoms des personnes qui sont employées en Alf

```
project (join (suppliers, cities),[:sid, :name, :city, :country])
# ==
join (project (suppliers, [:sid, :name, :city]), project (cities, [:city, :country]))
```

Ce qui nous donnent la requête SQL suivante :

```
SELECT t1.sid AS sid, t1.name AS name, t1.city AS city, t2.country AS country
FROM suppliers AS t1 INNER JOIN cities AS t2 ON (t1.city = t2.city)
```

Ceci est rendu possible car JAlf utilise des règles d'optimisation qui lui permettent de générer la même requête SQL pour les deux requêtes algébriques. Ces règles d'optimisation sont aussi une façon de réécrire la requête pour la rendre plus simple à exécuter par le système de gestion de base de données SQL. JAlf utilise également l'évaluation paresseuse, ce qui lui permet d'attendre avant d'exécuter les parties modulées de la requête et ainsi d'optimiser toute la requête. Cette évaluation paresseuse est importante, car elle permet avant tout de pouvoir assembler les parties modulées avant l'exécution de la requête. Si on repart de notre problème avec Alf, nous pourrons découper la requête comme suit :

Listing 1.8 – découpe de la requête en Alf

```
requester_city = 'Paris'
solution = suppliers

# exigence 1, le fournisseur voit seulement les informations concernant les
    fournisseurs d'une même ville
solution = restrict(solution, city: requester_city)

# exigence 2, le status du fournisseur n'est pas fourni dans la solution
solution = allbut(solution, [:status])

# exigence 3, le pays doit être affiché
solution = join(solution, cities)
```

Pour mieux comprendre l'optimisation, nous vous renvoyons au « blog »[13] qui fournit une console permettant de voir l'optimisation des requêtes.

2.2 Du calcul relationnel à l'algèbre relationnelle

Pour l'utilisateur lambda d'une base de données, il est plus facile de décrire la solution plutôt que de définir l'algorithme qui décrit le problème. SQL offre cette facilité de décrire la solution. Aujourd'hui, l'utilisation de SQL est abstraite par un programme, ce qui fait que l'utilisateur final n'a plus besoin de faire des requêtes SQL. Le programmeur ayant plus l'habitude de travailler

avec des langages procéduraux, il est plus difficile pour celui-ci de programmer dans un langage déclaratif. Or le langage SQL est déclaratif et permet très difficilement de moduler les requêtes. Il est alors évident que l'utilisation de l'algèbre relationnelle est plus adaptée. Exposer l'algèbre relationnelle apparaît alors plus naturel pour un développeur lorsqu'il implémente un logiciel.
Le paradigme **Relation as first-class citizen** s'appuie sur le fait qu'il est plus facile de mettre ensemble des relations plutôt que d'assembler des requêtes SQL (voir l'exemple avec Sequel). Avec l'algèbre relationnelle, une relation générée par une requête peut être un argument d'une autre requête.

Ce qui rend le paradigme **Relation as first-class citizen** plus intéressant pour des requêtes complexes. Prenons cette requête Sequel :

Listing 1.9 – requête complexe en Sequel

```
# montre les fournisseurs qui ne fournissent plus aucune part
DB[:suppliers___s].where(~DB[:shipments___sp].where(Sequel.qualify(:sp, :sid) =>
    (Sequel.qualify(:s, :sid))).exists)
```

Exprimer une requête `WHERE NOT EXISTS` s'avère habituellement compliqué pour les librairies existantes et bien souvent impossible à moduler alors que pour Alf la requête est simple :

Listing 1.10 – requête équivalente en Alf

```
# montre les fournisseurs qui ne fournissent plus aucune part
not_matching(suppliers, shipments)
```

La section 3, page 12 donne un exemple d'utilisation de Alf pour faciliter la génération d'une requête complexe.

2.3 Problèmes avec Query Builders

Les librairies Query Builders proposent une interface (API) qui est plus proche de l'algèbre relationnelle que du calcul relationnel. L'avantage de JAlf, c'est que celui-ci va plus loin en ajoutant une abstraction plus forte sur SQL en se basant plutôt sur le langage Tutorial-D. Bien que JAlf ressemble à une librairie Query Builders, elle reste toutefois différente dans l'abstraction proposée à l'utilisateur. JAlf expose la relation, dans le sens relationnel (un ensemble de tuples) alors que les Query Builders exposent un arbre syntaxique de SQL. Le fait d'exposer un arbre syntaxique de SQL indique que les librairies Query Builders sont fortement liées au langage SQL, ce qui nous ramène au problème de modularité expliqué précédemment (voir les exemples donnés précédemment avec Sequel).

Nous avons également constaté que les Query Builders ne permettent pas de traiter une source de données autre que les bases de données SQL. Il est en effet impossible pour une Query

Builders de traiter des relations se trouvant dans un fichier .csv ou encore dans un fichier .txt contrairement à JAlf.

2.4 Problèmes avec Object-Relational Mapping

Les librairies Object-Relational Mapping sont des librairies qui ne manipulent pas des requêtes SQL mais bien des objets et des collections d'objets. Ces librairies modélisent le domaine par des classes où une table d'une base de données est représentée par une classe. En réalité, une instance de classe représente un tuple et les attributs d'une table sont des champs typés de ces classes.

Les librairies Object-Relational Mapping posent les mêmes problèmes que les librairies Query Builders, à savoir qu'elles sont très couplées au langage SQL. En effet, elles exposent à l'utilisateur un arbre syntaxique SQL, ce qui nous ramène au problème de modularité. De plus, ces librairies ne permettent pas de traiter une source de données autre que les bases de données SQL.

2.5 Inspiration sur HaskellDB

HaskellDB est une librairie de connexion à une base de données qui s'appuie sur l'algèbre relationnelle. Celle-ci a inspiré la création de JAlf. HaskellDB utilise le langage Haskell comme langage de requête. Elle peut se connecter à des systèmes de gestion de base de données telles que SQL, SQLite, PostgreSQL, MYSQL. HaskellDB a le même comportement que JAlf. Elle permet facilement de rendre modulable les requêtes étant donné qu'elle s'appuie sur l'algèbre relationnelle d'une part et d'autre part elle optimise les requêtes pour créer une requête SQL plus facilement exécutable pour le système de gestion de base de données SQL.
Une des différences avec JAlf, c'est que HaskellDB ne propose pas la notion de relation qui est un concept-clé dans notre paradigme. En effet, le concept de relation est important dans le paradigme **Relation as first-class citizen** étant donné que la théorie de l'algèbre relationnelle traite des relations. Dans HaskellDB, les opérations algébriques sont faites directement sur les tuples contrairement à JAlf. Par exemple dans la requête HaskellDB suivante :

Listing 1.11 – exemple de requête Sequel

```
do p <- table person
   restrict (p ! name . == . constant "Gerrit")
   project (dob << p ! dob)
```

la table `person` est parcourue et les opérations de restriction (sur l'attribut `nom`) et de projection (sur l'attribut `dob`) se font sur chaque tuple. Voici la requête Alf correspondante :

Listing 1.12 – requête Alf correspondant à la requête HaskellDB

```
project(restrict(person, name:"Gerrit"),[:dob])
```

Une autre différence avec JAlf est que tout comme les autres systèmes de connexion à une base de données, HaskellDB ne permet pas de se connecter sur des sources de données autres que des bases de données SQL. De plus, HaskellDB est une librairie peu utilisée dans le monde de l'entreprise étant donné qu'elle utilise le langage Haskell qui est peu répandu.

3 Exemple concret d'utilisation d'Alf

JAlf étant la version Java d'Alf, nous allons vous présenter un exemple concret d'utilisation d'Alf. Notre exemple nous vient du mémoire d'un ancien étudiant de l'Université Catholique de Louvain David COLPAERT dont le titre est : « Un outil de gestion d'incohérences de données basé sur les vues intentionnelles et l'algèbre relationnelle appliquée à un cas de localisation de téléphones IP ».

Nous allons présenter la définition du problème et sa solution sans toutefois rentrer dans les détails. Nous allons nous contenter d'expliquer quelle a été la contribution d'Alf dans ce projet. Pour en savoir plus sur ce projet, nous vous invitons à consulter le mémoire.

3.1 Définition du problème

Lors d'une opération de migration, de normalisation ou de fusion dans une base de données, il arrive souvent que ces opérations apportent des incohérences dans les données provenant des différentes tables. Alors qu'on s'attend à avoir de la redondance dans les données, il s'avère en réalité que des données décrivant le même objet sont différentes. Il faut souvent parcourir manuellement la base de données pour trouver ses incohérences et utiliser des requêtes SQL bien souvent complexes pour trouver des cas particuliers qui causent problème. Pour vérifier les incohérences, il faut pouvoir vérifier des contraintes sur une base de données via des requêtes SQL fastidieuses et complexes. Ces contraintes peuvent être vérifiées manuellement lorsqu'elles proviennent d'une petite base de données, mais pour une grande base de données cela devient plus fastidieux ; d'où l'obligation de faire des requêtes SQL souvent très complexes pour l'utilisateur. Ces requêtes SQL peuvent également donner des résultats erronés si elles sont mal écrites. L'objectif de cet outil est de permettre à l'utilisateur final de pouvoir vérifier des contraintes dans une base de données sans avoir à connaître ni SQL, ni l'algèbre relationnelle. L'outil se veut intuitif tout en étant très expressif.

3.2 Solution au problème

L'outil propose une solution avec une interface graphique pour exprimer les contraintes en toute simplicité. A partir de cette interface graphique, l'utilisateur choisit via des menus déroulants les données des relations concernées auxquelles il souhaite appliquer des contraintes. Une fois les contraintes définies, une requête en algèbre relationnelle est générée automatiquement.

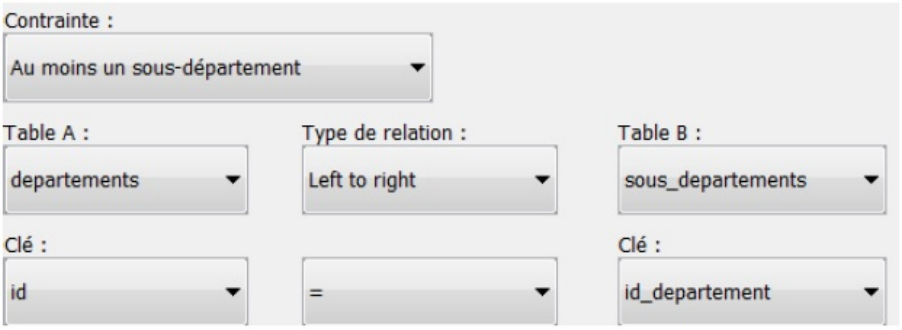


FIGURE 1.1 – Contrainte : Au moins un sous département par département.

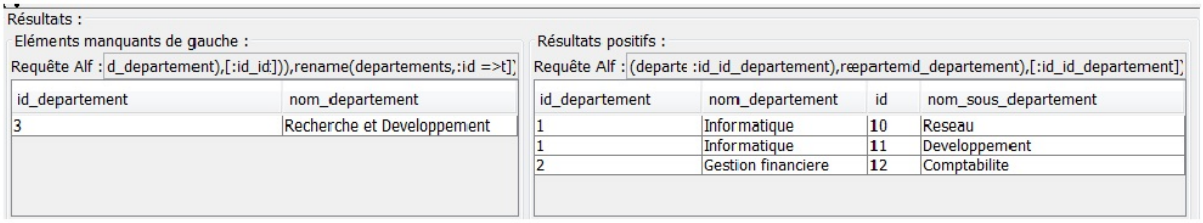


FIGURE 1.2 – Résultat : Au moins un sous département par département.

Cette requête est générée avec Alf, qui à son tour génère la requête SQL correspondante. La figure 1.1 montre un exemple de l'interface où l'utilisateur choisit les données de la requête et la figure 1.2 le résultat de la requête. Ces deux figures sont tirées du mémoire de David COLPAERT.

D'après l'auteur du projet, l'utilisation de l'algèbre relationnelle a été motivée par ses requêtes concises. En effet, contrairement à SQL, Alf permet de faire des requêtes complexes et concises. Par exemple, voici une requête Alf reprise du mémoire en question :

Listing 1.13 – requête d'exemple en Alf du mémoire de David COLPAERT

```
restrict(
   join_on(
      rename(join_on(locations_from_network,ucl_id_attributions,[:mac]),
         :building => :locations_from_network_building,
         :local => :locations_from_network_local),
      rename(join_on(locations_from_deployments,ucl_id_attributions,[:
            ucl_id]),
         :building => :locations_from_deployments_building,
         :local => :locations_from_deployments_local),
      [:mac]),
   eq(:locations_from_network_building,:locations_from_deployments_building) &
   eq(:locations_from_network_local,:locations_from_deployments_local)
```

```
)
```

Et son équivalente SQL est :

Listing 1.14 – requête équivalente SQL reprise du mémoire

```
WITH
   "t9" AS (
      SELECT
         "t1"."mac" AS "mac";
         "t1"."building" AS "locations_from_network_building";
         "t1"."local" AS "locations_from_network_local";
         [...]
      FROM "locations_from_network" AS "t1"
         INNER JOIN "ucl_id_attributions" AS "t2"
            ON ("t1"."mac" = "t2"."mac"));

   "t10" AS (
      SELECT DISTINCT
         "t7"."building" AS "locations_from_deployments_building";
         "t7"."local" AS "locations_from_deployments_local";
         [...]
         "t8"."mac" AS "mac"
         FROM "locations_from_deployments" AS "t7"
            INNER JOIN "ucl_id_attributions" AS "t8"
               ON ("t7"."ucl_id" = "t8"."ucl_id"))

SELECT "t9"."mac" AS "mac";
   "t9"."locations_from_network_building" AS
      "locations_from_network_building";
   "t9"."locations_from_network_local" AS
      "locations_from_network_local";
   [...]
   "t10"."locations_from_deployments_building" AS
      "locations_from_deployments_building";
   "t10"."locations_from_deployments_local" AS
      "locations_from_deployments_local";
   [...]
   FROM "t9" AS "t9"
      INNER JOIN "t10" AS "t10"
         ON ("t9"."mac" = "t10"."mac")
   WHERE (("t9"."locations_from_network_building" =
         "t10"."locations_from_deployments_building")
```

```
AND ("t9"."locations_from_network_local" =
   "t10"."locations_from_deployments_local"))
```

Dans cette requête SQL, certaines lignes inutiles (`[...]`) ont été supprimées afin de simplifier le code. Ces lignes étaient de simples renommages.

Une autre motivation pour l'auteur a été le concept de relation. L'algèbre relationnelle est fermée sur les relations, ce qui signifie qu'une relation résultante d'une requête peut être utilisée en argument d'une autre requête (voir dans la requête Alf ci-dessus). Cela implique qu'il est possible de composer des relations les unes avec les autres, ce qui selon l'auteur a grandement simplifié son approche. D'après l'auteur, SQL était moins bien adapté qu'Alf pour les requêtes imbriquées auxquelles ils ont dû faire face et qui étaient pourtant, en terme de conception, faciles à énoncer.

4 Projet apparenté

4.1 Tutorial-D

4.1.1 Définition

Tutorial-D est un langage d'interaction avec les bases de données relationnelles, conçu par Hugh DARWEN et Christopher J. DATE, et présenté dans leur document « The Third Manifesto » en 1994. Tutorial-D est un langage procédural dans la récupération des données mais qui dans l'ensemble est un langage impératif comme la plupart des langages. Tutorial-D se base sur des prescriptions et des proscriptions de D [3] décrites dans le document « The Third Manifesto ». Il s'agit d'un langage complet, ce qui implique qu'une application peut être codée dans ce langage.

Tutorial-D est un langage relationnel mais il peut également apparaître comme un langage orienté objet dans la mesure où il permet à l'utilisateur de définir ses propres types. Comme il n'y a pas de dépendance à l'égard d'un langage hôte, il n'y a aucune différence entre les types disponibles dans la base de données et ceux disponibles de l'extérieur. Il est ici question de base de données fournies par le serveur Rel qui est le système de gestion de base de données supportant Tutorial-D comme langage de requête. Plus d'explications sont données sur le serveur Rel dans la section suivante.

3. C'est une spécification de langage abstrait. Il ne décrit pas la syntaxe d'un langage mais plutôt les caractéristiques désirables et indésirables d'un système de gestion de base de données relationnelle selon les croyances de ces auteurs.

Pour mieux comprendre Tutorial D, voici la syntaxe des opérateurs relationnels et ensemblistes de base :

- la jointure de la relation R_1 et de la relation R_2 : $R_1\ JOIN\ R_2$ ou alors $JOIN\ \{R_1, R_2\}$. Étant donné que la jointure est une opération n-adic, elle peut avoir également la syntaxe $JOIN\ \{R_1, R_2, .., R_n, \}$ si on veut joindre plusieurs relations ;
- la projection sur la relation R_1 : $R_1\,\{A_1, A_2, ...\}$ ou alors $R_1\,\{ALLBUTA_1, ...\}$ si on veut omettre de faire une projection sur les attributs qui suivent $ALL\ BUT$;
- la sélection sur la relation R : R pour sélectionner tous les tuples de la relation ou alors R *WHERE* A_1 *opdecomparaison value* ;
- l'unification a la même syntaxe que la jointure, il suffit donc de remplacer $JOIN$ par $UNION$;
- l'intersection a également la même syntaxe que la jointure. Il faut remplacer $JOIN$ par $INTERSECT$;
- la différence entre la relation R_1 et la relation R_2 : $R_1\ MINUS\ R_2$. L'opérateur de différence est représenté par le mot clé $MINUS$ comme en SQL.

4.1.2 Limitation de Tutorial-D

Tutorial-D est un langage de requête tout comme le langage SQL. Le problème de Tutorial-D par rapport à JAlf est qu'il n'offre pas la possibilité de travailler avec les bases de données SQL alors que ce sont les plus répandues. C'est un langage de programmation complet qui toutefois n'est pas assez puissant pour la production industrielle. Il a été conçu pour servir en tant que langage d'apprentissage. Les fonctionnalités que doit avoir un langage dans le monde industriel ont intentionnellement été omises comme dit précédemment. Il ne supporte pas les fonctionnalités liées à la sécurité (sessions, connexions et transactions), aux traitements des exceptions, à la maintenance de la base de données, etc.

4.1.3 Rel

Définition

Rel est un système de gestion de base de données libre, open-source, et implémenté par Dave VOORHIS. Il utilise Tutorial-D comme langage de requête. Il est implémenté comme une application Java et doit tourner sur n'importe quel hôte qui supporte une plate-forme J2SE 7.x. Il a aussi été testé sous Linux, Mac OS X et d'autres versions de Microsoft Windows.

Le serveur Rel est construit à partir de plusieurs composants majeurs dont une machine virtuelle basée sur les piles, implémentée en Java, qui supporte la définition des opérateurs

et leur exécution, l'évaluation d'expressions, le contrôle de flux, les types, la vérification des types statiques et des variables. Il est composé d'un moteur de stockage Java Oracle. Le cœur de Rel, qui implémente l'algèbre relationnelle, interagit avec le moteur de stockage, compile, et exécute Tutorial-D sur la machine virtuelle. Rel a un interpréteur de langage qui reconnaît la syntaxe de Tutorial-D. Celui-ci interagit avec le cœur de Rel pour exécuter le code Tutorial-D et générer le résultat. Le parseur du serveur a été construit en utilisant le parseur JavaCC. Rel a également un composant qui permet la gestion de connexions et de sessions. Rel est un serveur multi-utilisateurs c'est-à-dire qu'il peut servir plusieurs clients en local ou en réseau.

Rel est fourni avec deux applications Java : un serveur de base de données Rel et un client interactif appelé DBrowser. Le client DBrowser envoie le code Tutorial-D entré par l'utilisateur au serveur Rel, qui à son tour parse et exécute le code. Un résultat est envoyé au client DBrowser qui l'affiche.

Limitation de Rel

Le problème de Rel, est qu'il n'est pas utilisable en dehors du monde des bases de données, contrairement à JAlf qui offre la possibilité de travailler avec différentes sources de données (fichiers .yaml, .csv, .txt, .json). Il est donc impossible dans ce cas d'utiliser Tutorial-D pour manipuler des relations provenant d'autres sources de données. Rel est également un système qui n'est pas utilisable en production étant donné qu'il utilise Tutorial-D. En effet, Tutorial-D ne supporte pas les fonctionnalités que doit avoir un langage dans le monde industriel.

Chapitre 2

Outils et méthodes de travail

Étant donné que nous avons travaillé en équipe pour la réalisation du projet, nous avons dû utiliser des outils permettant de faciliter notre travail. Pour ce faire, plusieurs outils de collaboration nous ont permis de gérer le projet sans empiéter sur le travail des uns et des autres. Cette section va aborder succinctement la question des logiciels et méthodes utilisés dès lors qu'il ne s'agit pas du sujet de ce mémoire mais bien de simples outils au service du projet.

1 Outils utilisés

1.1 Git

L'outil principal pour la gestion de notre collaboration était Git, un système de gestion de version distribué qu'on peut voir à figure 2.1, page 19. Un système de gestion de version distribué est un système qui enregistre l'évolution d'un fichier ou d'un ensemble de fichiers au cours du temps de sorte qu'on puisse récupérer une version antérieure d'un fichier quand bon nous semble. Git permet de ramener un projet à un état antérieur, ce qui permet de retourner à un état stable lorsque l'état actuel du projet est instable. Il permet également de connaître l'auteur d'une modification de code, qui découlerait d'une éventuelle erreur et ainsi faciliter la compréhension de l'erreur et sa correction.
Ce système, contrairement aux versions précédentes (systèmes de gestion de version locaux et systèmes de gestion de version centralisés) a l'avantage de permettre aux collaborateurs de récupérer le dépôt se trouvant sur un serveur distant, ce qui permet d'avoir un backup complet en cas de crash du serveur distant. En effet, chaque collaborateur du projet dispose d'une version du dépôt distant en local, ce qui permet à chacun de pouvoir faire les modifications ou création de fichier localement avant de les pousser sur le dépôt distant. Dans le cas de notre projet, nous avons fait une copie [1] du projet JAlf sur le serveur distant. Cette manière de faire a facilité la

1. C'est un fork en terme Git. C'est l'équivalent d'un clone qu'on ferait sur le serveur local sauf que le fork fait le clone sur le serveur distant

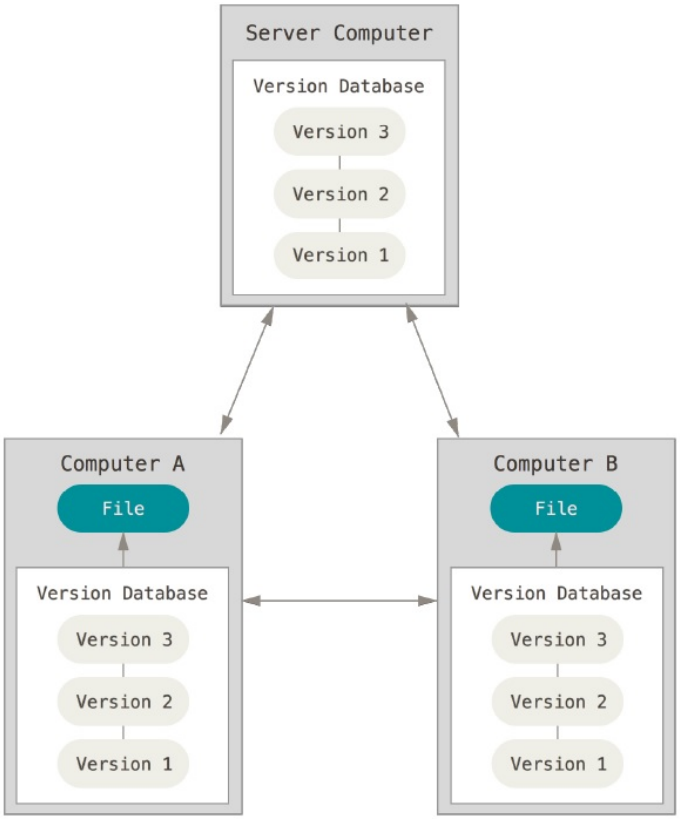


FIGURE 2.1 – Système gestion de version distribué.

collaboration tout en gardant cohérent le dépôt principal. La copie du dépôt a été notre copie de travail tout au long du projet. Une fois la version sur le dépôt de travail jugée cohérente, elle était corrigée, testée et acceptée par notre promoteur, elle était poussée par ses soins sur le dépôt principal.

1.2 GitHub

Nous avons utilisé GitHub [2] qui est un service offert en complémentarité de Git. Il est une plate-forme web pour le contrôle de version et de collaboration. Github fournit un répertoire distant accessible par tous les collaborateurs en vue des diverses modifications possibles via ce service. Il permet la gestion de projet utilisant le logiciel de gestion de versions Git. GitHub facilite l'utilisation de Git, car il offre une interface qui fournit des fonctionnalités qui ont la même utilité que celles offertes par Git, comme par exemple de créer une branche, créer un répertoire distant ou encore créer ou modifier un fichier.
Cette plate-forme, outre son rôle de gestion de projet, est un formidable outil de collaboration. Celle-ci fournit un réseau de communication particulièrement directe entre collaborateurs via un système de commentaires et de chat. Ce système nous a grandement facilité l'échange d'idées. Il

2. Lien vers JAlf : https ://github.com/aimablehirwa/jalf

permet non seulement d'envoyer des messages entre collaborateurs, mais aussi de commenter des lignes de code avec précision.

1.3 Trello

Trello est un gestionnaire de projet qui aide à l'organisation des différentes tâches inhérentes au projet. Cet outil de planification de tâches est hébergé sur une plate-forme internet, ce qui permet d'organiser le travail entre collaborateurs. Il permet de créer un tableau de bord sur lequel nous classons les tâches suivant plusieurs critères. Classiquement, la planification d'un projet fait appel à au moins trois critères :

- à faire ;
- en cours ;
- terminé.

Trello offre la possibilité à l'utilisateur d'ajouter, supprimer ou modifier les différents critères selon ses besoins spécifiques. Dans le cadre de notre projet nous avons déterminé six critères :

- à faire ;
- en cours ;
- terminé ;
- validé ;
- réunion ;
- documentation.

Les trois premiers, ainsi que les cinquième et sixième critères sont assez explicites. Quant au quatrième critère, son utilité était que chaque tâche terminée devait être validée par l'administrateur du projet, c'est à dire Dr. Bernard LAMBEAU , après une revue de code réalisée par ses soins.

1.4 Java 8

Pour l'implémentation de JAlf nous avons utilisé la version 8 du langage Java sortie en mars 2014. Cette version introduit de nouveaux concepts qui permettent d'écrire du code plus rapidement, comme par exemple trier une collection en une seule ligne de code. Java a décidé de tirer profit d'une avancée technologique, c'est-à-dire les processeurs à cœurs multiples qui sont présents à l'heure actuelle sur la plupart des ordinateurs personnels.
Java offrait déjà la possibilité de travailler avec des threads multiples travaillant en concurrence.

Mais la mise en œuvre de la concurrence était assez complexe et menait bien souvent à de nombreuses erreurs. Pour pallier à cette complexité, Java 8 propose une nouvelle API appelée « Stream ». Celle-ci offre la possibilité de pouvoir appliquer plusieurs traitements à des données en parallèle, et le tout avec simplicité. Une autre possibilité qu'offre Java 8, est de pouvoir passer du code à une méthode ou fonction rendant ainsi les fonctions ou méthodes comme des citoyens de première classe. Imaginez que nous ayons deux méthodes quasi identiques à une ligne près de code et qu'à la place d'avoir ces deux méthodes, nous avons dorénavant la possibilité d'avoir une et une seule méthode et de passer en paramètre la ligne de code qui diverge. Cela rend le code plus lisible. Et si par la suite une modification est nécessaire, nous devrons la faire à un seul endroit et non à deux, ce qui limite les erreurs et les oublis. Cette technique qui permet de passer du code et des méthodes en paramètres, fait référence à ce qu'on appelle la « programmation fonctionnelle ».

1.4.1 Streams

Une stream est une suite d'éléments de données, qui est par principe construite en un élément à la fois. Une stream se construit à partir d'une source de données (collection, tableau,...) et elle est caractérisée par quelques propriétés :

- une stream ne stocke pas de données, elle se contente de transférer les données de la source de données vers des opérations ;
- elle ne modifie pas les données de la source. Elle construit une nouvelle stream lorsqu'elle effectue une opération de filtrage ;
- le chargement des données s'opère de manière paresseuse (« lazy »), ce qui permet d'optimiser l'utilisation de la mémoire et ainsi optimiser les performances des applications ;
- une stream n'est pas réutilisable ce qui implique de devoir créer une nouvelle stream à partir de la source de données lorsqu'on veut réutiliser les données de la première stream.

Dans une stream d'entrée, on peut lire un élément à la fois, le traiter si nécessaire et ensuite produire une stream de sortie dans laquelle on écrit un élément à la fois. La stream de sortie d'un programme peut être réutilisée par la suite. Pour travailler avec l'API Stream en Java, il est nécessaire de l'importer avec l'instruction `java.util.stream`. Comme une stream traite des éléments de même type, le type des éléments est indiqué par le type générique T donnant ainsi la syntaxe suivante pour une stream : `Stream<T>`. Une stream offre une liste d'opérations d'agrégation qui sont applicables aux éléments contenus dans une stream. Il est possible d'appliquer des filtres suivant des prédicats, d'appliquer un traitement à chacun des éléments, ou encore de regrouper les éléments suivant certains critères. Ces diverses opérations peuvent s'enchaîner les unes après les autres pour fournir une autre stream, ou encore une nouvelle collection Java. Les streams ont également permis à Java d'améliorer le traitement des **big data**

qui sont des grosses bases de données qui demandent un traitement par la parallélisation. Du fait que les streams ne modifient pas les éléments, cela permet de garder de la cohérence dans les données lors de la parallélisation du traitement.

1.4.2 Les lambdas et les fonctions

Java 8 permet de créer des méthodes utilisables comme des valeurs. Une expression lambda est un bloc de codes qui peut être exécuté plus tard, une ou plusieurs fois si nécessaire. Une expression lambda est de la forme : `( )-> expression ;`. Java 8 permet également de définir des fonctions sous forme de classes. Il offre une série d'interfaces qui sont des interfaces fonctionnelles [3], c'est-à-dire des interfaces qui ne possèdent qu'une seule méthode abstraite, donc avec une seule signature. Une expression lambda représente l'implémentation de l'unique méthode d'une interface fonctionnelle tout en respectant sa signature.
Voici un exemple d'interface fonctionnelle qui décrit une fonction qui prend deux paramètres, un premier de type générique T et un second de type générique U, et renvoie un résultat de type générique R :

```
BiFunction<T, U, R>
```

Et voici un exemple de lambda expression qui implémente cette interface :

Listing 2.1 – exemple de code qui implémente l'interface BiFunction

```
BiFunction<String, String, Integer> func =
  (premier, second)-> Integer.compare (premier.length (),second.length());
```

ou `premier` représente le type T et `second` représente le type U.

2 Méthodes

2.1 Méthode agile

2.1.1 Définition

Pour réaliser le projet, nous avons utilisé la méthode agile avec Scrum comme schéma d'organisation. C'est une méthode de gestion de projet qui a pour intérêt d'encourager l'implication de toutes les parties prenantes dans la réalisation du projet. La méthode agile donne plus de visibilité au client en l'impliquant dans le projet du début jusqu'à la fin.
La méthode considère que les besoins ne sont pas figés dans le temps et qu'il convient donc de

3. L'API qui définit les interfaces fonctionnelles de base est `java.util.function`.

s'adapter aux changements. C'est la raison pour laquelle cette méthode est axée sur une approche itérative et incrémentale. Chaque itération se compose de travaux de conception, d'analyse fonctionnelle et conceptuelle, de développement et de test. A la fin de l'itération, un premier produit partiel mais utilisable est présenté au client en vue d'un compte rendu de ce dernier.
Dans un projet de développement de logiciel, le client élabore sa vision du logiciel et fait la liste des exigences ou fonctionnalités du projet. A partir de cette liste, l'équipe de développement planifie le projet. L'idée est de fixer un premier objectif à court terme appelé itération. A ce moment on va définir un ensemble de points pour le bon déroulement de l'itération, comme par exemple :

- discuter des tâches afin qu'elles soient comprises de chacun ;
- définir le niveau de difficulté ;
- assigner les tâches aux membres ;
- ...

Une fois que ce premier objectif est atteint, on définit l'objectif suivant en prenant en compte la situation du moment. Et ainsi de suite jusqu'à la fin du projet.

2.1.2 Implication d'agile dans le projet

Dans le cadre de notre projet, la méthode agile a été le moteur de notre avancement. Nous avons procédé à une planification en itération allant de deux à quatre semaines. Nous avons établi des itérations de quatre semaines dans un premier temps puis, nous sommes passés à des itérations de deux semaines. Nous avions en effet souhaité une interaction plus efficace avec notre promoteur Dr. Bernard LAMBEAU . Ce dernier est considéré comme le client du projet. Les fonctionnalités implémentées étaient déterminées en début de chaque itération au cours d'une réunion avec notre promoteur. Outre la définition des fonctionnalités de l'itération suivante, notre promoteur nous donnait un compte-rendu sur l'état d'avancement du projet et sur la dernière itération terminée.

2.1.3 Trello pour agile

Pour mettre en œuvre la méthode agile nous avons choisi d'utiliser l'outil Trello qui est expliqué dans la section 1.3, page 20). L'intérêt de Trello est de faciliter la mise en œuvre de la méthode agile par son système de critères pour lister les tâches suivant leur état d'avancement. Trello permet également : d'éditer (voir figure 2.2, page 24) une tâche pour lui fixer une durée de vie dans le cadre d'une itération ; de définir un niveau de difficulté à une tâche ; d'assigner des membres à une tâche, ...

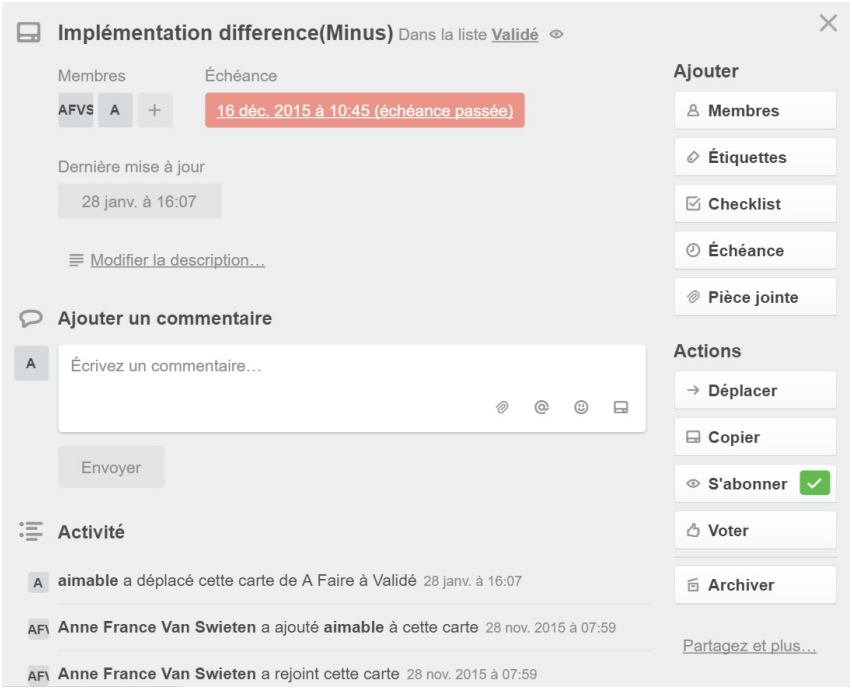


FIGURE 2.2 – Edition d'une tâche dans Trello.

2.1.4 Revue de code

Dans une équipe de développeurs, nous pouvons avoir des profils variés conférant à l'équipe des compétences diverses. Le développement de ces compétences s'opère grâce à la revue de code. Celle-ci permet aux développeurs d'apprendre des techniques de programmation qui leur permettront de faire évoluer leur compétences personnelles.
La revue de code consiste pour un développeur à vérifier le code d'un autre collaborateur une fois que ce dernier a terminé sa tâche. Dans le cas d'un projet agile, la revue de code va s'intégrer dans une itération une fois que l'ensemble du code a été écrit et que les tests ont confirmé le bon fonctionnement du code. La revue du code se fait avant de pousser du code sur le dépôt principal (voir section 1.1, page 20) en vue de vérifier les aspects manqués par les tests et éviter les mauvaises décisions de programmation qui pourraient polluer le dépôt principal. La revue de code est donc une solution pour améliorer du code, mais également un moyen de partager des connaissances entre les membres d'une équipe.
Durant le développement de notre projet nous avons donc utilisé la revue de code. D'une part pour les raisons décrites dans le paragraphe précédent mais aussi pour être toujours en cohérence avec chaque membre de l'équipe, car même si une analyse est faite au préalable, il est toujours possible que les membres n'aient pas exactement la même vision de l'architecture ou des méthodes supplémentaires à ajouter. La revue de code finale a été réalisée à chaque itération par notre promoteur.

2.2 TDD - Test Driven Development

2.2.1 Approche traditionnelle

Dans l'approche traditionnelle de développement, les tests sont écrits après le code source pour tester celui-ci en vue de vérifier si les méthodes ont le comportement souhaité. Mais cette approche a quelques désavantages. Il se peut par exemple que les méthodes testées contiennent des erreurs de conception, menant ainsi à un comportement qui ne répond pas exactement aux exigences souhaitées. Il peut également arriver que le développeur n'ait pas pensé à tous les détails de la méthode, l'obligeant constamment à reconsidérer son analyse. Le manque de tests pour couvrir toutes les subtilités auxquelles une méthode doit répondre peut aussi conduire à de graves erreurs de conception d'un programme.

2.2.2 Approche Test Driven Development (TDD)

TDD est une approche qui pallie certains désavantages de l'approche traditionnelle. Elle est une approche de développement qui consiste à écrire les tests avant d'écrire le code source. C'est un cycle rapide de développement et de refactorisation de code. Le but de TDD est de se concentrer sur les spécifications et non sur la manière de les implémenter. En d'autres termes, il s'agit d'une manière de définir l'architecture et les exigences des fonctionnalités avant de se lancer dans le code. Le fait de connaître en détail les exigences auxquelles doit répondre une méthode permet d'éviter les erreurs souvent rencontrées dans l'approche traditionnelle de développement. L'idée principale est d'arriver à ce que les méthodes réussissent les tests écrits au préalable. L'intérêt est de faire directement un code source impeccable évitant de devoir refactoriser. En effet, si on connaît exactement tous les détails d'une méthode, on peut espérer l'implémenter avec le meilleur code. Dans l'approche traditionnelle, on doit faire la refactorisation plus souvent.

La pratique du TDD est une partie essentielle de la méthode Agile guidant un projet de développement logiciel. Cette méthode exige une livraison fréquente et rapide de composants fonctionnels. Par son procédé de création de tests qui font figure de spécifications, TDD permet aux développeurs d'améliorer grandement la vitesse de production de code. En effet, la multiplicité des tests diminue les erreurs liées aux spécifications des méthodes.

L'approche TDD se fait en plusieurs étapes comme on peut voir sur la figure 2.3, page 26. La première étape de TDD consiste à écrire un test qui normalement doit échouer. La seconde étape, consiste à écrire du code jusqu'à ce qu'il passe le test. Dès que le code a réussi le test, il faut retourner à la première étape et poursuivre ainsi de suite jusqu'à avoir toutes les fonctionnalités souhaitées.

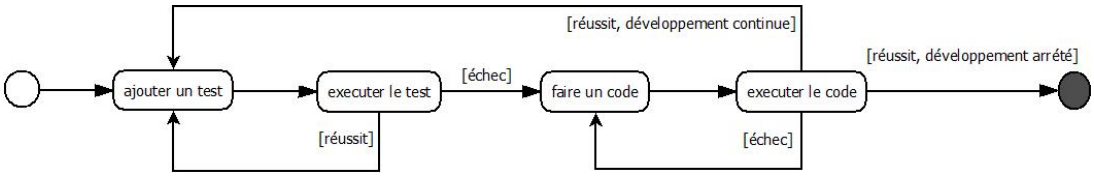


FIGURE 2.3 – Les étapes de TDD.

Chapitre 3

Modèle théorique

Dans cette section, nous allons aborder le modèle théorique sur lequel nous nous sommes basés tout au long du projet. Nous allons proposer une vue exhaustive du modèle sans pour autant rentrer dans des détails inutiles. Le but ici est donc de permettre au lecteur de comprendre les fondements théoriques qui sous-tendent notre travail et ceci, sans l'obligation de recourir à des sources ou des ouvrages extérieurs.

1 Algèbre relationnelle

1.1 Introduction

L'algèbre relationnelle, décrit en premier par Edgar F. CODD en 1970, est le modèle mathématique utilisé pour modéliser les données enregistrées dans une base de données relationnelles. Codd avait proposé cette algèbre comme fondement pour les langages de requêtes de bases de données. L'algèbre relationnelle propose un ensemble d'opérations élémentaires formelles sur les relations en vue d'en créer de nouvelles. Des opérations peuvent être appliquées sur ces relations créées. Ces opérations permettent de représenter des requêtes sur la base de données dont le résultat s'exprime sous la forme d'une relation (une table). Nous pouvons relever deux grandes familles d'opérateurs :

- **les opérateurs relationnels spécifiques au modèle relationnel** : PROJECT, SELECT, JOIN ;
- **les opérateurs ensemblistes provenant de la théorie des ensembles** : UNION, INTERSECT, DIFFERENCE.

Ces opérateurs sont appelés « opérations de base ». De ces opérateurs de base, plusieurs autres opérateurs sont dérivés, ce sont les « opérateurs dérivés ». Ces opérateurs sont des raccourcis d'écriture qui n'apportent pas de nouvelles fonctionnalités, mais sont pratiques pour l'utilisateur. Les opérations peuvent encore être classées en différentes catégories :

- **les opérations unaires** (PROJECT, SELECT), qui à partir d'une table produisent une autre ;
- **les opérateurs binaires ensemblistes** (UNION, INTERSECT, DIFFERENCE), qui produisent une relation à partir de deux relations.
- **les opérateurs binaires ou n-aires** (JOIN), qui produisent une relation à partir de deux ou plusieurs relations.

1.2 Attribut, Type, Tuple et Relation

Avant d'introduire les opérateurs nous allons définir quelques notions utilisées pour expliquer la théorie de l'algèbre relationnelle.

Un attribut est un élément qui est représenté par un nom et une valeur. Celle-ci possède un certain type [1] décrivant le genre de valeur possible. Un type, dans le sens relationnelle, est un ensemble fini d'attributs non ordonnés et sans duplication. Un type peut par exemple être représenté sous la forme :

```
type := { Nom : String, Age : Integer }
```

Un tuple est un ensemble de paires $[nom, valeur]$ non ordonnés en telle sorte que deux paires n'ont pas le même nom. Toutes les paires d'un tuple correspondent chacune à un attribut d'un type défini pour ce tuple et la cardinalité du tuple doit être la même que celle du type. Par exemple, si on reprend le type défini au-dessus, on pourra avoir les tuples suivants : `{}`

```
tuple := { Nom : "Roland", Age : 28 }
tuple := { Nom : "Albert", Age : 23 }
tuple := { Nom : "Martine", Age : 40 }
```

mais on ne pourra pas avoir les tuples suivants :

`tuple := { Nom : "Martine", Age : 40.0 }` <– l'attribut `Age` ne peut pas être un double
`tuple := { Nom : 10, Age : 40 }` <– l'attribut `Nom` ne peut pas être un entier

Chacun des opérateurs de l'algèbre relationnelle porte sur au moins une relation. Une relation est un ensemble non ordonné de tuples qui répondent tous au même type. Le type représente la structure d'une relation, la manière dont les éléments d'un tuple sont agencés. Chaque relation correspond à une ou des formule(s) logique(s) appelée(s) prédicat(s). Une formule indique que la relation doit contenir des tuples pour lesquels la formule est vraie à tout moment. Donc un tuple qui rend la formule vraie pourra se trouver dans la relation alors, que inversement, un tuple qui rend la formule fausse ne pourra pas s'y retrouver. Si on définit par exemple la relation

IdPersonne	Nom	Prenom
5	Durand	Caroline
1	Germaine	Stan
12	Dupont	Lisa
2	Germaine	Rose-Marie

Table 3.1 – PERSONNES

PERSONNES suivante :

on peut définir le prédicat comme étant « La personne dont l'identifiant est `IdPersonne` a le nom `Nom` et le prénom `Prenom` ».

Les formules logiques font appel aux opérateurs logiques. C'est pourquoi les opérateurs de l'algèbre relationnelle sont liés aux opérateurs logiques AND, OR et NOT et aux quantificateurs existentiels. Lorsqu'un opérateur est exécuté sur une ou des relation(s), la relation résultante est une extension de(s) prédicat(s) des/de la relation(s) sur le(s)quelle(s) se fait l'opération. Par exemple, si on repart de la relation PERSONNES et qu'on fait une jointure avec la relation EMPLOYEES suivante :

Nom	Prenom	Ville
Durand	Caroline	Mons
Germaine	Stan	Tubize
Zeuz	Rose	Louvain-la-Neuve

Table 3.2 – EMPLOYEES

on peut alors définir le prédicat comme étant « La personne `IdPersonne` s'appelle `Nom` `Prenom` AND habite à `Ville` ». Ici les prédicats sont joints par l'opérateur logique AND. L'opérateur relationnel de projection est lié au quantificateur existentiel d'un attribut. Par exemple, si on reprend la relation PERSONNES, le résultat de la projection sur le champ `IdPersonne` peut définir le prédicat « La personne `IdPersonne` est dans le registre ». Autrement dit, « Existe-t-il une personne `IdPersonne` qui est dans le registre ». Le résultat sera une relation avec un attribut `IdPersonne` et dont les tuples contiennent `IdPersonne` des personnes.

1.3 Opérateurs relationnels

1.3.1 Projection (PROJECT)

Cet opérateur construit une relation résultat où n'apparaissent que certains attributs de la relation opérande et supprime les tuples dupliqués dans la nouvelles relation. C'est une opération unaire qui permet de générer une relation comportant les attributs spécifiées dans les attributs de la projection. Ceux-ci appartiennent bien sûr à la relation sur laquelle se fait la projection.

1. Le type de valeur d'une variable, à ne pas confondre avec le type d'une relation.

Définition : soit $R\,(A_1, A_2, ..., A_n)$ une relation, et soit $S\,(A_{i1}, A_{i2}, ..., A_{ij})$ un sous ensemble des attributs de R, la projection de R sur le sous-ensemble S notée :

$$\pi\,[A_{i1}, A_{i2}, ..., A_{ij}]\,R$$

crée une nouvelle relation, de type $(A_{i1}, A_{i2}, ..., A_{ij})$ dans laquelle on enlève les colonnes qui ne font pas partie du sous-ensemble et où l'on supprime également les tuples en doubles. Une requête SQL utilisera la clause `DISTINCT` pour supprimer les doublons.
Repartons de notre relation PERSONNES, en appliquant la projection $\pi\,[IdPersonne, Nom]\,(PERSONNES)$ qui est l'équivalent en SQL de :

```
SELECT DISTINCT IdPersonne, Nom FROM PERSONNES
```

nous obtenons la relation :

IdPersonne	Nom
5	Durand
1	Germaine
12	Dupont

TABLE 3.3 – $\pi\,[IdPersonne, Nom]\,(PERSONNES)$

1.3.2 Sélection (SELECT)

L'opération de sélection est un opérateur unaire qui permet d'extraire les tuples d'une relation qui répondent à une liste de conditions appelés prédicats. L'opérateur construit une relation résultat où n'apparaissent que certains tuples de la relation opérande. La relation résultat ne contient que les tuples dont toutes les conditions sont vraies.

Définition : soit $R\,(A_1, A_2, ..., A_n)$ une relation, la sélection de R suivant un prédicat p notée :

$$\sigma\,[p]\,R$$

génère une nouvelle relation, de même type que celui de R et composée des tuples de R pour lesquels le prédicat p est vrai. L'équivalent SQL d'une sélection est la clause `WHERE condition`. Le prédicat d'une sélection permet de comparer la valeur des attributs de R à d'autres attributs de R ou à des constantes. Au sein d'une condition, on peut utiliser des opérateurs de comparaison et des opérateurs logiques pour former un prédicat. Un prédicat peut être composé de plusieurs prédicats. Une sélection aura la forme suivante :

<p> : :=<condition> | <p><opérateur logique><p> | ¬ < p >
<opérateur logique> : := $\wedge | \vee | \neg$

<condition> : := nom-attribut <opérateur comparaison> valeur | nom-attribut <opérateur comparaison> nom-attribut
<opérateur comparaison> : := $\neq \mid = \mid \leq \mid < \mid > \mid \geq$

Par exemple, si on reprend la relation PERSONNES vue précédemment, en appliquant la sélection $\sigma\,[Nom = Germaine]\,R$, qui est l'équivalent SQL de :

```
SELECT * FROM PERSONNES WHERE Nom = Germaine
```

nous obtenons la relation :

IdPersonne	Nom	Prenom
1	Germaine	Stan
2	Germaine	Rose-Marie

TABLE 3.4 – $\sigma\,[Nom = Germaine]\,PERSONNES$

1.3.3 Jointure (JOIN)

L'opération de jointure est une opération portant sur deux ou plusieurs relations et qui génère une relation résultante qui contient des tuples qui sont la combinaison des tuples des différentes relations jointes sur leurs attributs communs. En d'autres termes, elle permet de lier des relations en fusionnant les tuples sur les attributs communs aux deux relations. Pour ce faire, il faut que ces attributs en commun aient la même valeur.

Définition : étant donné deux relations R(X, Y) et S(Y, Z), où X, Y, Z un ensemble d'attributs, et où Y n'est pas vide, la jointure (naturelle) de R et S, notée :

$$R \bowtie S$$

crée une nouvelle relation de type (X, Y, Z) dont le contenu de R * S est l'ensemble des tuples <x, y, z> créés par composition d'un tuple <x, y> de R et d'un tuple <y, z> de S, tels que les deux tuples ont la même valeur pour Y. En SQL, la jointure peut être définie par la clause `NATURAL JOIN` mais également à l'aide de la clause `WHERE condition`.
Le contenu de la relation $R \bowtie S$ comporte n tuples, ou $n \in [0 : (card\,(R) + card\,(S))$ - 1].
Reprenons notre relation PERSONNES et définissons la relation ETUDIANTS suivante :

IdEtudiant	Année	Prenom
1	Sinf22	Caroline
2	Ecge22	Lisa

TABLE 3.5 – ETUDIANTS

si on applique la jointure $PERSONNES \bowtie ETUDIANTS$ dont le SQL :

```
SELECT * FROM PERSONNES NATURAL JOIN ETUDIANTS
```

nous obtenons la relation :

IdPersonne	Nom	Prenom	IdEtudiant	Année
5	Durand	Caroline	1	Sinf22
12	Dupont	Lisa	2	Ecge22

TABLE 3.6 – $PERSONNES \bowtie ETUDIANTS$

La nouvelle relation est générée par la combinaison des deux relations sur l'attribut en commun, en l'occurrence ici sur `Prenom`. Le premier tuple de la relation ETUDIANTS est combiné avec le premier tuple de la relation PERSONNES étant donné que l'attribut `Personne` a comme valeur `Caroline` dans les deux tuples. On fait le même raisonnement pour le tuple 3 de la relation PERSONNES avec le tuple 2 de la relation ETUDIANTS pour former un seul tuple. Pour les tuples où il n'y a pas de correspondance, ceux-ci ne sont pas repris dans la relation générée.

La jointure offre deux cas intéressants. Si les deux relations n'avaient aucun attribut en commun, l'opération de jointure aurait effectué un produit cartésien. Toutefois, dans l'hypothèse où les deux relations avaient les mêmes attributs, la jointure aurait donné l'intersection entre des deux relations. Et dans ce cas, l'opérateur INTERSECT(voir section 1.4.1, page 33) aurait pu être utilisé.

Remarque : l'opérateur de jointure utilisé est la jointure naturelle, c'est à dire la jointure sur les attributs communs. Une jointure naturelle offre trois propriétés intéressantes :

- la **commutativité**, $R1 \bowtie R2$ est équivalent à $R2 \bowtie R1$
- l'**associativité**, $(R1 \bowtie R2) \bowtie R3$ est équivalent à $R1 \bowtie (R2 \bowtie R3)$. Si nous pouvons joindre plus de deux relations, alors nous pouvons les joindre dans n'importe quel ordre
- l'**idempotence**, $R \bowtie R$ est équivalent R.

1.3.4 Renommage

L'opérateur de renommage est un opérateur important qui ne fait pas partie de la théorie de l'algèbre relationnelle, mais qui est très utile pour faire la correspondance entre les attributs des différentes tables. C'est un opérateur qui permet de renommer un ou des attributs d'une relation. En SQL le renommage se fait grâce au `AS`. L'opérateur de renommage génère une relation R dans laquelle certains attributs sont renommés. L'opération est notée α :

$$\alpha\,[A_1 : NouveauNomDeA_1, \dots]\,R$$

Par exemple, si on reprend notre relation PERSONNES et qu'on applique un renommage sur l'attribut $\alpha\,[IdPersonne : Matricule]\,PERSONNES$ dont la requête SQL est

```
SELECT IdPersonne AS Matricule, Nom, Prenom
FROM PERSONNES
```

nous obtenons la relation suivante :

Matricule	Nom	Prenom
5	Durand	Caroline
1	Germaine	Stan
12	Dupont	Lisa
2	Germaine	Rose-Marie

TABLE 3.7 – $\alpha\,[IdPersonne : Matricule]\,PERSONNES$

Remarque : l'opérateur de renommage est utile comme par exemple avant certaines opérations de jointure lorsqu'il se pose un problème d'homonymie ou de synonymie, ou encore avant les opérations ensemblistes UNION, INTERSECT et DIFFERENCE qui exigent que les attributs correspondants dans les relations à opérer aient le même nom. La pré-condition au renommage est que le nouveau nom n'existe pas dans la relation.

1.4 Opérateurs ensemblistes

Les opérateurs ensemblistes proviennent de la théorie des ensembles. Les relations sur lesquels les opérations sont faites doivent avoir le même type, c'est-à-dire les mêmes attributs.

1.4.1 Intersection (INTERSECT)

L'opération d'intersection est une opération sur deux relations R_1 et R_2 ayant le même type et générant une relation résultat dont les n-tuplets sont constitués de ceux appartenant aux deux relations. Le résultat de l'intersection donne une relation qui a les mêmes attributs que les relations R_1 et R_2. C'est une opération binaire qui a la propriété d'être commutative et se note :

$$R_1 \cap R_2$$

Par exemple, si on reprend notre relation PERSONNES et qu'on fait l'intersection avec la relation PERSONNES2 suivante :

IdPersonne	Nom	Prenom
5	Durand	Caroline
2	Zeuz	Albert
12	Dupont	Lisa
10	Opra	Rose

TABLE 3.8 – PERSONNES2

nous obtenons la relation :

dont le code SQL est :

```
SELECT * FROM PERSONNES
INTERSECT
```

IdPersonne	Nom	Prenom
5	Durand	Caroline
12	Dupont	Lisa

TABLE 3.9 – $PERSONNES \cap PERSONNES2$

```
SELECT * FROM PERSONNES2;
```

Remarque : Si une des deux relations est vide ou que toutes les deux sont vides, le résultat de l'opération donne une relation d'intersection vide.

1.4.2 Union

L'opération d'union est une opération sur deux relations R_1 et R_2 ayant le même type et générant une relation constituée des n-tuplets appartenant à l'une ou l'autre des deux relations R_1 et R_2 sans tuples dupliqués. Le résultat de l'union est une nouvelle relation qui a les mêmes attributs que R_1 et R_2 et contient les tuples des deux relations. Il s'agit d'une opération binaire ensembliste qui se note :

$$R_1 \cup R_2$$

Par exemple, si on repart de la relation PERSONNES et qu'on l'unit à la relation PERSONNES2, nous obtenons la relation :

IdPersonne	Nom	Prenom
5	Durand	Caroline
2	Zeuz	Albert
1	Germaine	Stan
12	Dupont	Lisa
10	Opra	Rose
2	Germaine	Rose-Marie

TABLE 3.10 – $PERSONNES \cup PERSONNES2$

dont le code SQL est :

```
SELECT * FROM PERSONNES
UNION
SELECT * FROM PERSONNES2;
```

Remarque : Si les deux relations sont vides, la relation qui résulte de l'union est vide. Si R_1 ou R_2 est vide, la relation qui résulte de l'union est identique à R_1 (respectivement R_2). L'union a également les mêmes propriétés que la jointure à savoir la commutativité, l'associativité et l'idempotence.

1.4.3 Différence (MINUS)

L'opération de différence, aussi appelée opération Minus, est une opération sur deux relations R_1 et R_2 qui ont le même type. L'opération génère une relation dont les n-tuplets sont constitués

des tuples qui ne se trouvent que dans la relation R_1 et qui n'apparaissent pas dans la relation R_2. Le résultat de la différence est une relation qui a les mêmes attributs que R_1 et R_2 étant donné qu'ils doivent avoir le même type pour que l'opération puisse être possible.
Il s'agit d'une opération binaire ensembliste notée :

$$R_1 \text{ - } R_2$$

Par exemple, si nous reprenons notre relation PERSONNES et que nous calculons la différence avec la relation PERSONNES2, nous obtenons la relation :

IdPersonne	Nom	Prenom
1	Germaine	Stan
2	Germaine	Rose-Marie

TABLE 3.11 – *PERSONNES − EMPLOYEES*

dont le code SQL est

```
SELECT * FROM PERSONNES
MINUS
SELECT * FROM PERSONNES2;
```

Remarque : Si R_1 est vide, la relation qui est le résultat de la différence est alors vide, et par contre si R_2 est vide, la relation qui résulte de la différence est identique à R_1.

2 Clés candidates

Dans une relation, nous nous intéressons à un type de clé, les clés candidates. Une clé est définie par un ensemble d'attributs de la relation. Dans le cas de notre définition d'une relation, une relation doit avoir obligatoirement au moins une clé candidate définie lors de la création de la dite relation. Deux remarques intéressantes sont à retenir :

- si une clé n'est pas définie par l'utilisateur, la clé de la relation est définie comme étant tous les attributs de la relation ;
- la clé peut être un ensemble vide. Dans ce cas, la relation ne peut pas contenir de tuples.

Une clé candidate doit être unique, c'est-à-dire que deux tuples d'une même relation ne peuvent pas être égaux dans les valeurs d'attributs. Formellement, si la clé K est un sous-ensemble de la relation R, alors K est une clé de la relation R. Si à tout moment deux paires de tuples *t1* et *t2* apparaissent dans le contenu de la relation R, la projection $\pi\,[K]\,t1$ est différente de la projection $\pi\,[K]\,t2$. Et si plusieurs clés sont définies pour la relation alors aucune clé ne peut être sous-ensemble d'une autre clé.

L'intérêt pour nous d'avoir plusieurs clés est de pouvoir optimiser le système. Nous considérons que les contraintes de clés apportent des informations sur les relations. Ces informations sont

considérés lors du calcul d'une requête pour optimiser le code des opérateurs utilisés. Par exemple, considérons la relation suivante :

A	B	C
...	...	...
...	...	...

TABLE 3.12 – ...

pour laquelle on définit l'attribut A comme première clé et la combinaison des attributs B et C comme seconde clé. Si on exécute une projection sur une des clés de la relation, disons la première clé, c'est-à-dire A, alors on sait que dans l'implémention de la projection, on ne doit pas chercher à éliminer les doublons une fois la projection faite.

Chapitre 4

Notre contribution au projet Jalf

Dans ce chapitre, nous allons décrire notre contribution au projet JAlf. Nous avons ajouter quatre opérateurs dont l'opérateur **UNION**, l'opérateur **INTERSECT**, l'opérateur **MINUS** et l'opérateur **SUMMARIZE**. Nous avons de plus apporté l'inférence des clés. Au travers de notre contribution, nous expliquerons en partie l'architecture de JAlf, ainsi que quelques nouvelles notions qu'offre Java 8. C'est une architecture imaginée et créée par Dr. Bernard LAMBEAU . Pour notre mémoire, nous avons dû comprendre et nous adapter à cette architecture, afin de pouvoir ajouter notre contribution au projet JAlf de manière efficace et compréhensible. Pour faciliter la compréhension de ce chapitre nous commencerons par quelques notions spécifiques à l'architecture et à l'implémentation de JAlf.

1 Notions spécifiques à JAlf

1.1 La notion de relation

Dans cette partie, nous allons détailler la manière dont JAlf représente une `Relation`. Deux types de `Relation` se détachent, les relations dites en mémoire et les relations résultantes d'un opérateur. Les relations en mémoire sont celles que nous construisons via une commande ou encore en lisant un fichier .csv. Les opérateurs prennent en paramètre une ou deux relations, et renvoie une relation. C'est ce qu'on appelle « closure » des opérateurs.

1.1.1 L'interface principale Relation

C'est l'interface principale de tous les types de relation de JAlf. Ces relations de type différent peuvent être utilisées ensemble et être les opérandes des opérateurs relationnelles de JAlf. C'est la classe `AbstractRelation` qui implémente cette interface. Cette classe abstraite est la classe parente de tous types de relation. C'est dans cette classe que nous trouverons les méthodes qui appellent les constructeurs des différents opérateurs relationnels. Voici ci-dessous un exemple de

la méthode minus :

Listing 4.1 – une méthode d' AbstractRelation

```
@Override
public Relation minus(Relation right) {
    return new Minus(this, right);
}
```

1.1.2 Relation en mémoire

Sur la figure 4.1, nous avons représenté le diagramme d'une relation en mémoire. Une relation en mémoire est une instance de la classe `SetMemortRelation`. Cette classe hérite de `MemoryRelation` classe parente de tous les types de relation en mémoire. La construction de cette instance se fait via une méthode statique `relation` implémentée dans la classe `DSL`. Cette classe possède deux champs : `tuple` et `type`. Le champ `tuple` est un ensemble (Set) de `Tuple`, tandis que `type` représente le type de la relation. La méthode `tuples` est appelée via la DSL. Elle vérifie que tous les tuples reçus en paramètre sont cohérents, c'est-à-dire que pour un attribut donné toutes les valeurs doivent être du même type. Si ce n'est pas le cas la méthode lève une exception.

1.1.3 Relation résultante d'un opérateur

Nous partons comme précédemment de l'interface `Relation` qui est implémentée par la classe abstraite `AbstractRelation`. La classe abstraite `Operator` hérite de la classe `AbstractRelation` et est la classe parente de toutes les relations de type d'opérateur.

Nous avons deux types d'opérateurs : des opérateurs unaires et binaires représentés respectivement par la classe abstraite `UnaryOperator` et la classe abstraite `BinaryOperator`. Une relation résultante d'un opérateur est définie par sa classe concrète portant le nom du dit opérateur. Sur le diagramme de classe de la figure 4.2, nous avons représenté deux opérateurs comme exemples. La classe `Project` représente notre opérateur project et hérite de la classe `UnaryOperator`. Notre classe `Project` possède trois champs : le champ `operand`, le champ `atrributes` et le champ `type`. Le champ `attributes` est une liste d'attributs appartenant à la relation sur laquelle porte la projection. Ce sont les attributs que nous garderons dans le résultat de la projection. Le champ `type` définit le type de la relation, nous en parlerons dans la section qui suit consacrée aux différents types rencontrés dans JAlf. Quant au champ `operand`, il est de type Relation et représente une relation en mémoire qui est la seule opérande de l'opérateur unaire Project. La classe `Union` hérite de la classe abstraite `BinaryOperator`. Cette classe possède trois champs : le champ `left`, le champ `right` et le champ `type`. Les deux opérandes de l'opérateur Union sont représentés par les champs `left` et `right`, tous deux sont des instances de la classe `SetMemoryRelation`. Nous pouvons noter dans chacune des deux classes `Union` et `Project` une méthode `accept`.

Cette méthode prend en paramètre un `Visitor` qui peut-être de type `DefautMapper` ou encore `Compiler`. Nous mentionnerons ce paramètre dans les section 1.3, page 44 et la section 1.4, page 47.

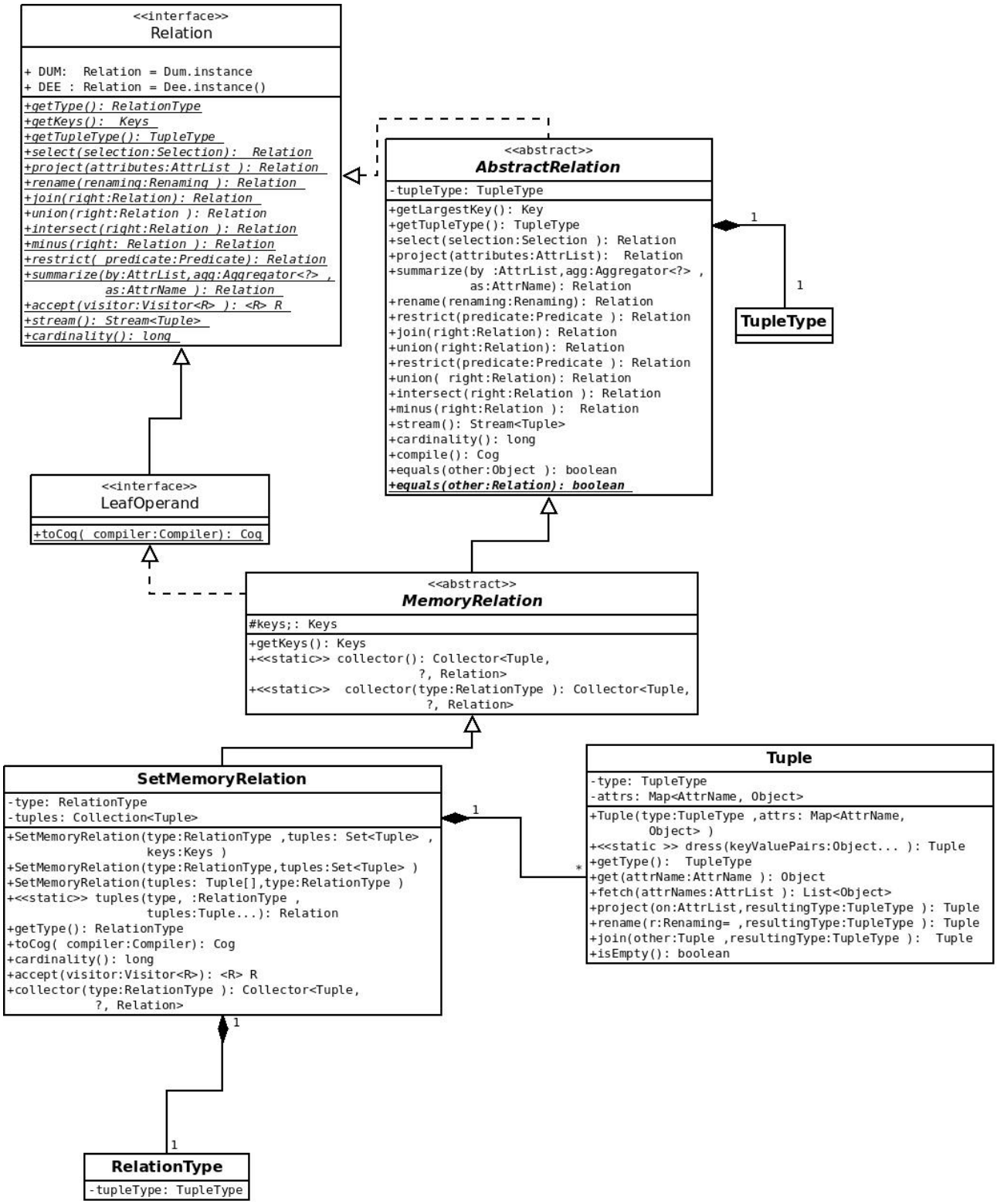


FIGURE 4.1 – Relation en mémoire.

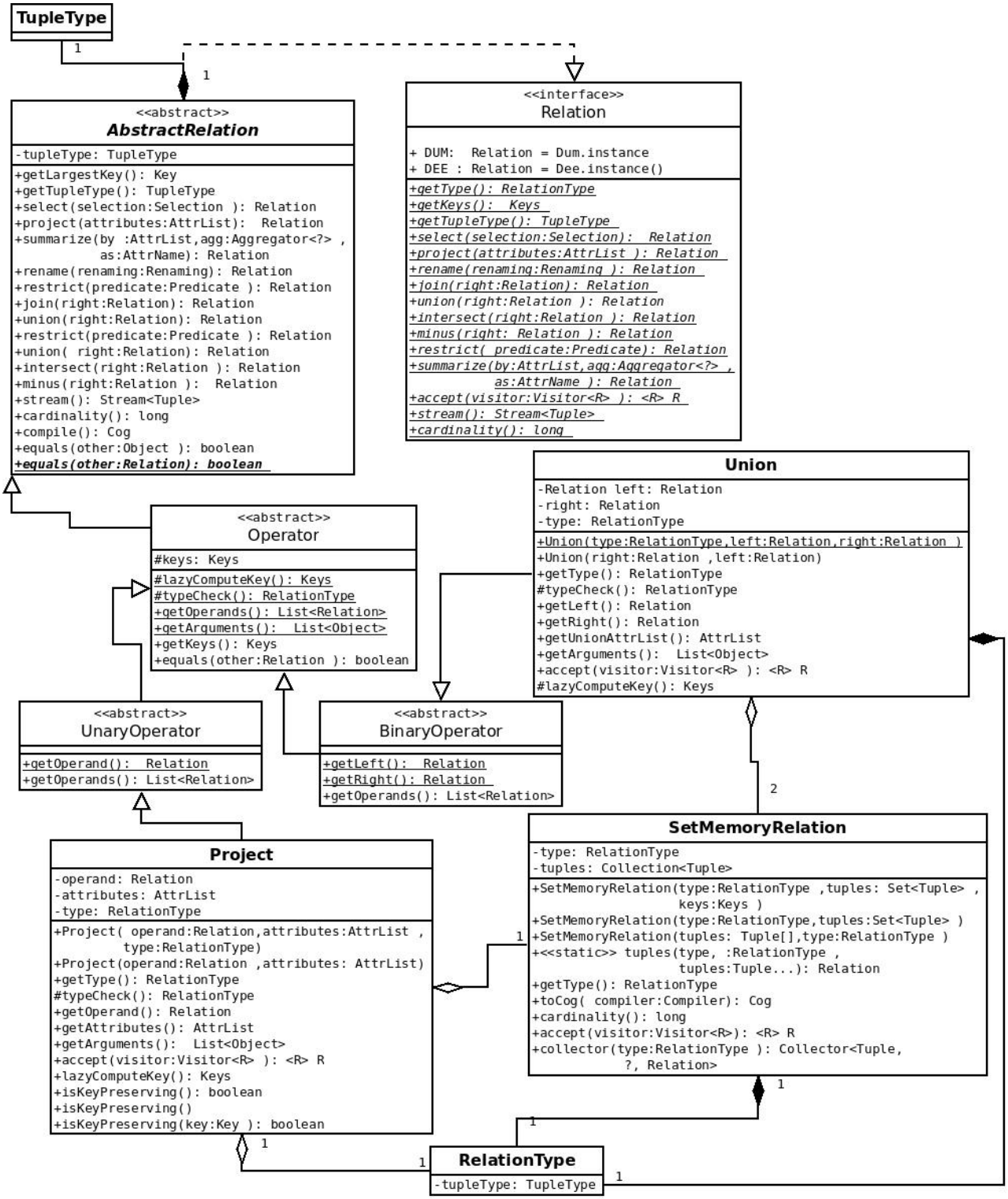


FIGURE 4.2 – Relation Project et Union.

1.2 La notion de Type

Une relation est définie par un en-tête. La classe `Heading` de JAlf représente cet en-tête, elle contient un champ `attributes`, c'est-à-dire une liste immuable de paires. Chaque paire est composée d'un nom d'attribut et de son type. Le nom d'un attribut pour JAlf a le type `AttrName` et son type doit être un type qui implémente l'interface `Type`. Une relation qu'elle soit dite en mémoire, ou résultante d'un opérateur a toujours un champ `type` de type `RelationType`. De plus, une la relation a un champ de type `TupleType` représentant le type des tuples contenus dans une relation. Tous les tuples contenus dans une relation devront respecter ces types ainsi définis. La classe `TupleType` a un champ `heading` de type `Heading`. Pour construire une relation dite en mémoire, nous appelons une des méthodes statiques `relation` de notre DSL , celles-ci prennent toutes au moins en paramètre un liste de `Tuple`. Une des méthodes `relation` prend un paramètre un `Heading` défini par l'utilisateur. Grâce à ce `Heading` une méthode de `RelationType` peut construire le type de la relation. Si l'utilisateur ne précise pas de `Heading`, c'est une méthode `relation` sans `Heading` en paramètre qui sera appelée. Celle-ci, grâce à la liste contenant des objets de type `Tuple` reçue en paramètre, appelle alors une méthode de la classe `RelationType` qui construira alors le type de la relation d'après cette liste de tuples. Pour les relations résultantes d'opérateur, elles appellent toutes une méthode `typeCheck` dans leur constructeur. Cette méthode appellera une méthode spécifique à leur opérateur dans la classe `RelationType` qui lui retournera le type de la Relation.

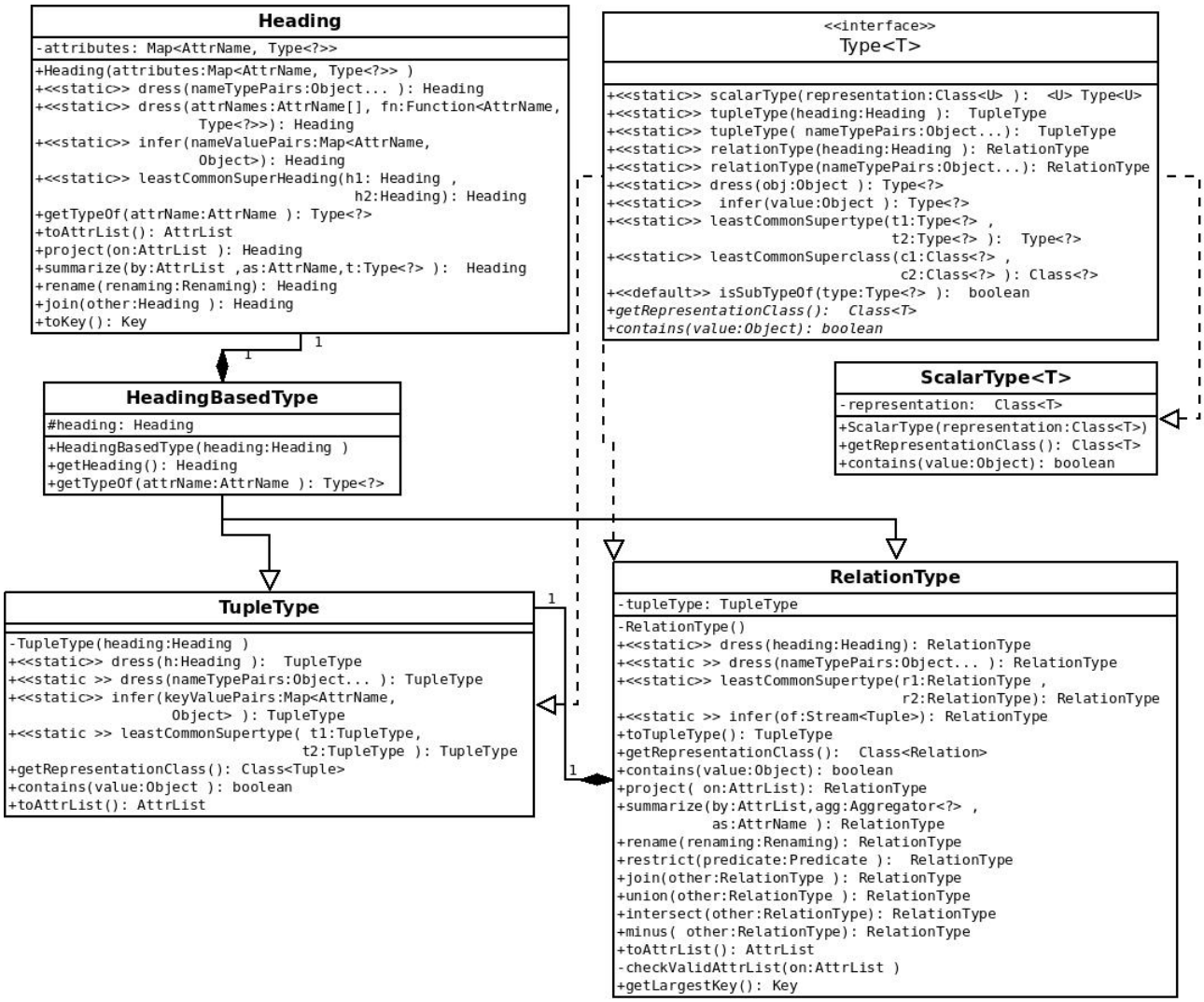


FIGURE 4.3 – Type en jalf.

1.3 La notion de compilation

JAlf implémente le design pattern « visitor »pour lancer la compilation sur des opérateurs. Nous allons prendre comme exemple la compilation de l'opérateur project. Ce design pattern est classé comme design pattern de comportement. Il permet d'ajouter un comportement à un objet sans changer le code de ses propres méthodes ni ajouter du nouveau code. Ce comportement est confié à une autre classe qui implémente une interface `Visitor`. Une interface `Visitor` possède une méthode abstraite `visit` qui doit prendre en argument l'objet visité. La classe qui implémente l'interface `Visitor` définit la méthode `visit`. Cette méthode reçoit en argument l'objet visité à qui elle pourra appliquer un comportement particulier. Elle pourra également récupérer les champs de l'objet si nécessaire. L'objet visité possède une méthode `accept` qui prend en argument un objet `Visitor`. La méthode `accept` d'un objet visité appelle la méthode `visit` du `Visitor` qu'elle a accepté, tout en lui donnant sa propre référence en argument. Regardons ce design pattern pour la compilation de `Project`. La classe `Compiler` implémente `Visitor<Cog>` qui est un visiteur de type `Cog`. La méthode `visit` est définie pour pouvoir prendre en paramètre une relation `Project`. Cette méthode `visit` exécute la compilation c'est-à-dire qu'elle compile la relation de type `Project` reçue en argument. Voici ci-dessous la méthode `visit` :

Listing 4.2 – méthode visit pour une projection dans Compiler

```
public Cog visit(Project relation) {
    Cog compiled = relation.getOperand().accept(this);
    return compiled.project(relation);
}
```

Ci-dessous, le code du client qui est un objet `AbstractRelation`. Il appelle la méthode `accept` :

Listing 4.3 – le client AbstractRelation

```
private Cog compile() {
    return accept(new Compiler());
}
```

Ci-dessous, le code de l'objet `Project` qui via sa méthode `accept`, appelle la méthode `visit` de la classe `Compiler` tout en lui passant sa propre référence en argument.

Listing 4.4 – la méthode accept de Project

```
@Override
public <R> R accept(Visitor<R> visitor) {
    return visitor.visit(this);
}
```

Le diagramme de séquence 4.4 montre l'appel de la compilation de l'opération de projection et le diagramme de classe 4.5 montre les différentes classes liées à la compilation d'un opérateur.

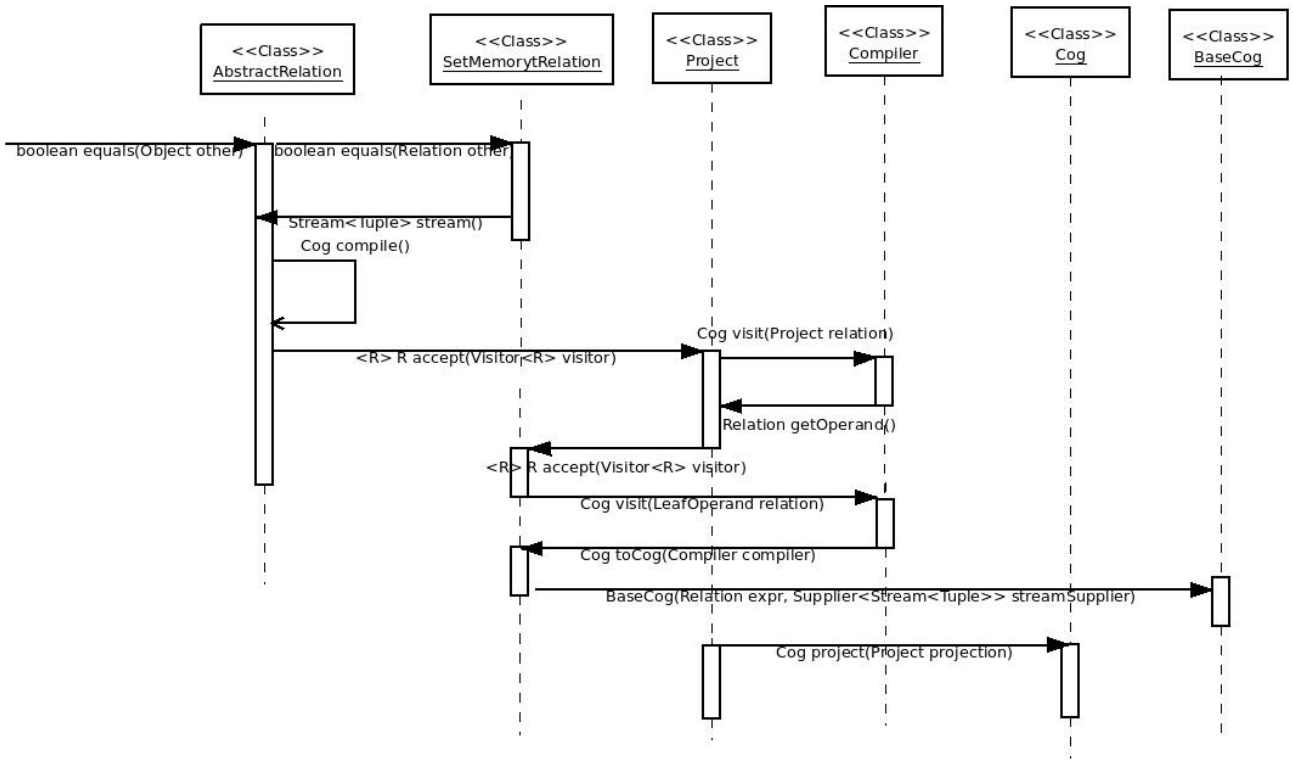
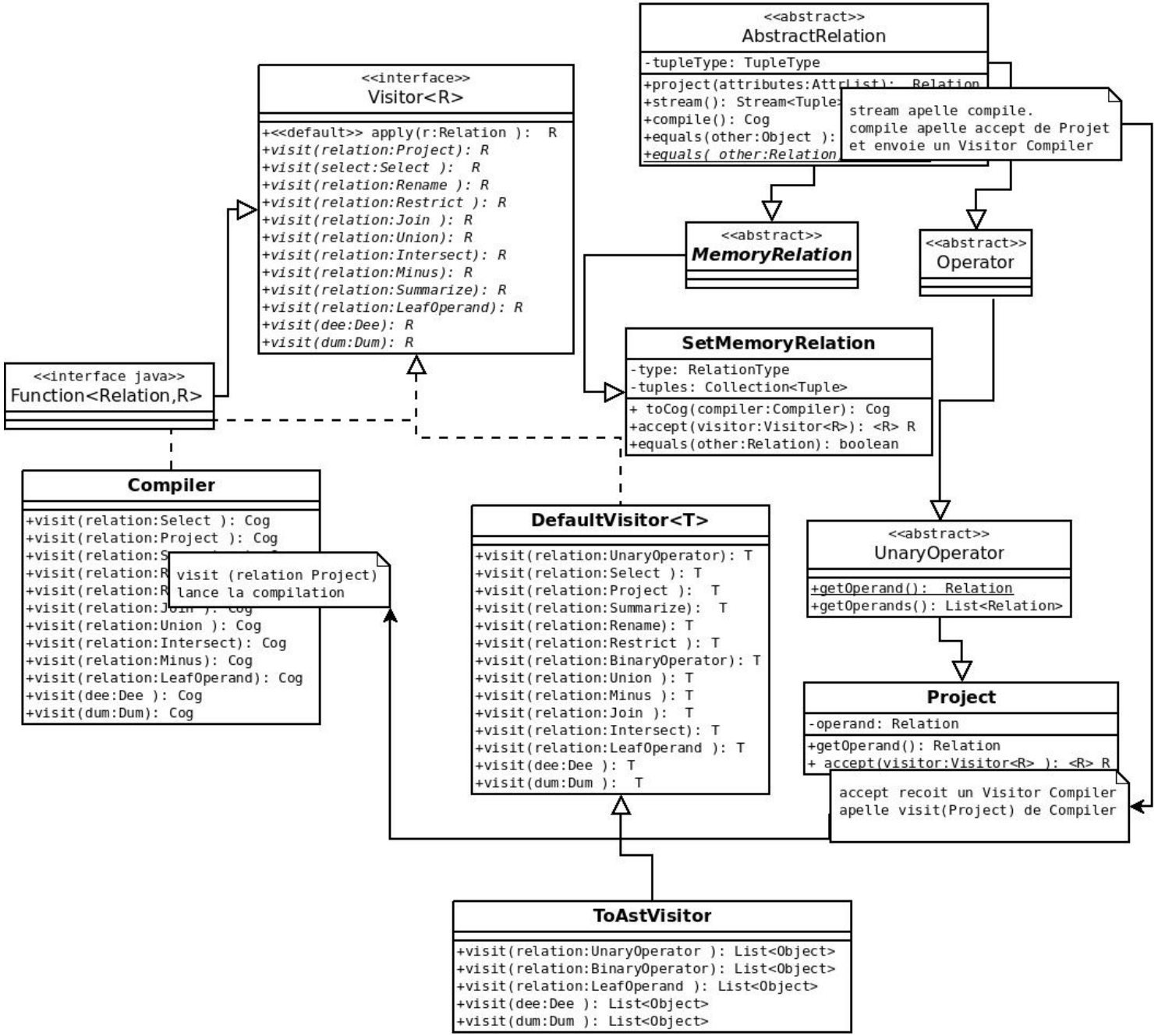


FIGURE 4.4 – Appel de la compilation de Project


<<abstract>>
AbstractRelation
-tupleType: TupleType
+project(attributes:AttrList): Relation
+stream(): Stream<Tuple>
+compile(): Cog
+equals(other:Object): boolean
+equals(other:Relation): boolean
stream apelle compile.
compile apelle accept de Projet
et envoie un Visitor Compiler
<<interface>>
Visitor<R>
+<<default>> apply(r:Relation): R
+visit(relation:Project): R
+visit(select:Select): R
+visit(relation:Rename): R
+visit(relation:Restrict): R
+visit(relation:Join): R
+visit(relation:Union): R
+visit(relation:Intersect): R
+visit(relation:Minus): R
+visit(relation:Summarize): R
+visit(relation:LeafOperand): R
+visit(dee:Dee): R
+visit(dum:Dum): R
<<abstract>>
MemoryRelation
<<abstract>>
Operator
<<interface java>>
Function<Relation,R>
SetMemoryRelation
-type: RelationType
-tuples: Collection<Tuple>
+ toCog(compiler:Compiler): Cog
+accept(visitor:Visitor<R>): <R> R
+equals(other:Relation): boolean
Compiler
+visit(relation:Select): Cog
+visit(relation:Project): Cog
+visit(relation:Union): Cog
+visit(relation:Intersect): Cog
+visit(relation:Minus): Cog
+visit(relation:LeafOperand): Cog
+visit(dee:Dee): Cog
+visit(dum:Dum): Cog
visit (relation Project)
lance la compilation
DefaultVisitor<T>
+visit(relation:UnaryOperator): T
+visit(relation:Select): T
+visit(relation:Project): T
+visit(relation:Summarize): T
+visit(relation:Rename): T
+visit(relation:Restrict): T
+visit(relation:BinaryOperator): T
+visit(relation:Union): T
+visit(relation:Minus): T
+visit(relation:Join): T
+visit(relation:Intersect): T
+visit(relation:LeafOperand): T
+visit(dee:Dee): T
+visit(dum:Dum): T
<<abstract>>
UnaryOperator
+getOperand(): Relation
+getOperands(): List<Relation>
Project
-operand: Relation
+getOperand(): Relation
+ accept(visitor:Visitor<R>): <R> R
accept recoit un Visitor Compiler
apelle visit(Project) de Compiler
ToAstVisitor
+visit(relation:UnaryOperator): List<Object>
+visit(relation:BinaryOperator): List<Object>
+visit(relation:LeafOperand): List<Object>
+visit(dee:Dee): List<Object>
+visit(dum:Dum): List<Object>

1.4 La notion d'optimisation

L'optimisation est implémentée pour chaque opérateur. Le but de l'optimisation est de fournir plus rapidement des résultats équivalents lors d'une requête. Grâce aux différentes propriétés des opérateurs, nous pouvons modifier l'ordre d'exécution des différents opérateurs contenus dans une requête. De telle sorte que l'exécution de la requête sera plus rapide ou/et utilisera moins d'espace disque.
La règle la plus simple consiste à pousser au plus profond de l'arbre syntaxique l'opérateur unaire Project. Si ce dernier porte par exemple sur un opérateur binaire tel que Union. Ce sont les opérateurs les plus profonds dans l'arbre qui sont bien sûrs exécutés en premier lieu. Une autre règle est de pousser aussi au plus profond de l'arbre l'opérateur Restrict. L'opérateur Project permet d'avoir moins d'attributs dans une relation, tandis que Restrict permet d'obtenir moins de tuples dans une relation. Il est donc intéressant d'appliquer ces opérations en premier lieu.

1.4.1 Optimisation de l'opérateur project

Quand nous avons une suite d'opérations project représenté par la caractère π, nous pouvons toutes les ignorées sauf la dernière.
$\pi_{List_1}(\pi_{List_2}(...(\pi_{List_n}(R))...)) \equiv \pi_{List_1}(R)$

JAlf propose cette optimisation pour l'opérateur Project. Voici ci-dessous le code de la méthode d'optimisation sur project :

Listing 4.5 – Optimisation de project

```
public Relation project(AttrList attributes) {
    Relation r = operator.getOperand();
    r = optimized(r).project(attributes);
    return r;
}
```

Permutation entre Restrict noté σ et Project π, si la condition de restriction implique seulement les attributs $A_1, ..., A_n$ dans la liste des attributs sur lesquels se fait la projection. Et si la condition est respectée alors les deux opérations sont commutatives.

$\pi_{A_1,A_2,...,A_n}(\sigma_c(R)) \equiv \sigma_c(\pi_{A_1,A_2,...,A_n}(R))$
JAlf utilise cette propriété de commutativité pour d'abord exécuter une restriction et puis une projection sur la relation. Voici ci-dessous le code de la méthode d'optimisation sur restrict :

Listing 4.6 – Optimisation de project avec restrict

```
public Relation restrict(Predicate pred) {
    Relation r = operator.getOperand();
    r = optimized(r).restrict(pred);
```

```
    r = optimized(r).project(operator.getAttributes());
    return r;
}
```

1.4.2 Optimisation de l'opérateur Restrict

Une suite de conjonctions de prédicats (conditions sur lesquelles se portent une restriction) peut être casée en une de séquence d'opérations individuelles de restriction et inversement.
$\sigma_{c_1 and c_2 ... and ... c_n}(R) \equiv \sigma_{c_1}(\sigma_{c_2}(..\sigma_{c_n}(R))...))$
JAlf regroupe les prédicats des différentes opérations restrict en une conjonction de prédicats et une seule opération de restriction. Voici ci-dessous le code de la méthode d'optimisation sur restrict :

Listing 4.7 – Optimisation de restrict

```
public Relation restrict(Predicate outer) {
    Predicate inner = operator.getPredicate();
    Relation r = operator.getOperand();
    r = optimized(r).restrict(inner.and(outer));
    return r;
}
```

1.4.3 Optimisation de l'opérateur Union

L'opérateur Project π est commutatif avec l'opérateur Union $\cup$.
$\pi_L(R \cup S) \equiv (\pi_L(R)) \cup (\pi_L(S))$

Voici ci-dessous le code de la méthode d'optimisation sur project :

Listing 4.8 – Optimisation de union avec project

```
public Relation project(AttrList attributes){
    // get the first operand
    Relation left = operator.getLeft();
    left = optimized(left).project(attributes);

    // get the second operand
    Relation right = operator.getRight();
    right = optimized(right).project(attributes);
    Relation r = left.union(right);
    return r;
}
```

Les opérateurs Union $\cup$, Intersect $\cap$ et Minus $\setminus$ peuvent être commutés avec Restrict σ. Posons θ qui peut remplacer les trois opérateurs précités, nous pouvons alors écrire :
$\sigma_c(R\ \theta\ S) \equiv (\sigma_c(R))\ \theta\ (\sigma_c(S))$

Grâce à cette règle, nous pouvons dans un premier temps appliquer l'opérateur Restrict aux deux opérandes de ces opérateurs, et par la suite exécuter Union, Intersect ou encore Minus.

1.4.4 Optimisation de l'opérateur Join

Commuter l'opérateur Project π avec l'opérateur Join $\bowtie$:

Supposons que la liste de projection $L = [A_1, \ldots, A_n, B_1, \ldots, B_m]$, où $A_1, \ldots, A_n$ représentent les attributs de la relation R et $B_1, \ldots, B_m$ représentent les attributs de la relation S. Si la condition $_c$ de l'opérateur Join implique seulement les attributs compris dans L, les deux opérations peuvent être permutées de la sorte.
$\pi_L(R\ \bowtie_c\ S) \equiv (\pi_{A_1,\ldots,A_n}(R))\ \bowtie_c\ (\pi_{B_1,\ldots,B_m}(S))$
Si la condition $_c$ de l'opérateur Join contient des attributs qui ne se retrouvent pas dans L, ces attributs doivent être ajoutés à L (la liste de la projection). Si $A_{n+1}, \ldots, A_{n+k}$ de R et $B_{m+1}, \ldots, B_{m+p}$ de S, sont dans la condition $_c$ de l'opérateur Join et ne se retrouvent pas dans L (la liste de la projection) alors les opérations peuvent être inter-changées comme suit :
$\pi_L(R\ \bowtie_c\ S) \equiv \pi_L((\pi_{A_1,\ldots,A_n,A_n+1\ldots,A_n+k}(R))\ \bowtie_c\ (\pi_{B_1,\ldots,B_m,B_m+1\ldots,A_m+p}(S)))$

Commuter l'opérateur Restrict σ avec l'opérateur Join $\bowtie$:

Si tous les attributs de la condition $_c$ de l'opérateur Restrict impliquent seulement les attributs d'une des relations de l'opérateur Join, dans notre cas la relation R, alors les deux opérations peuvent être permutées comme ceci :
$\sigma_c(R\ \bowtie_c\ S) \equiv (\sigma_c(R))\ \bowtie\ S$
De plus, une condition $_c$ de l'opérateur Restrict peut s'écrire $(_{c_1\ and\ c_2})$, si $_{c_1}$ s'applique seulement à R et que $_{c_2}$ s'applique seulement à S, alors les opérations peuvent être commutées de la sorte :
$\sigma_c(R\ \bowtie_c\ S) \equiv (\sigma_{c_1}(R))\ \bowtie\ (\sigma_{c_2}(S))$

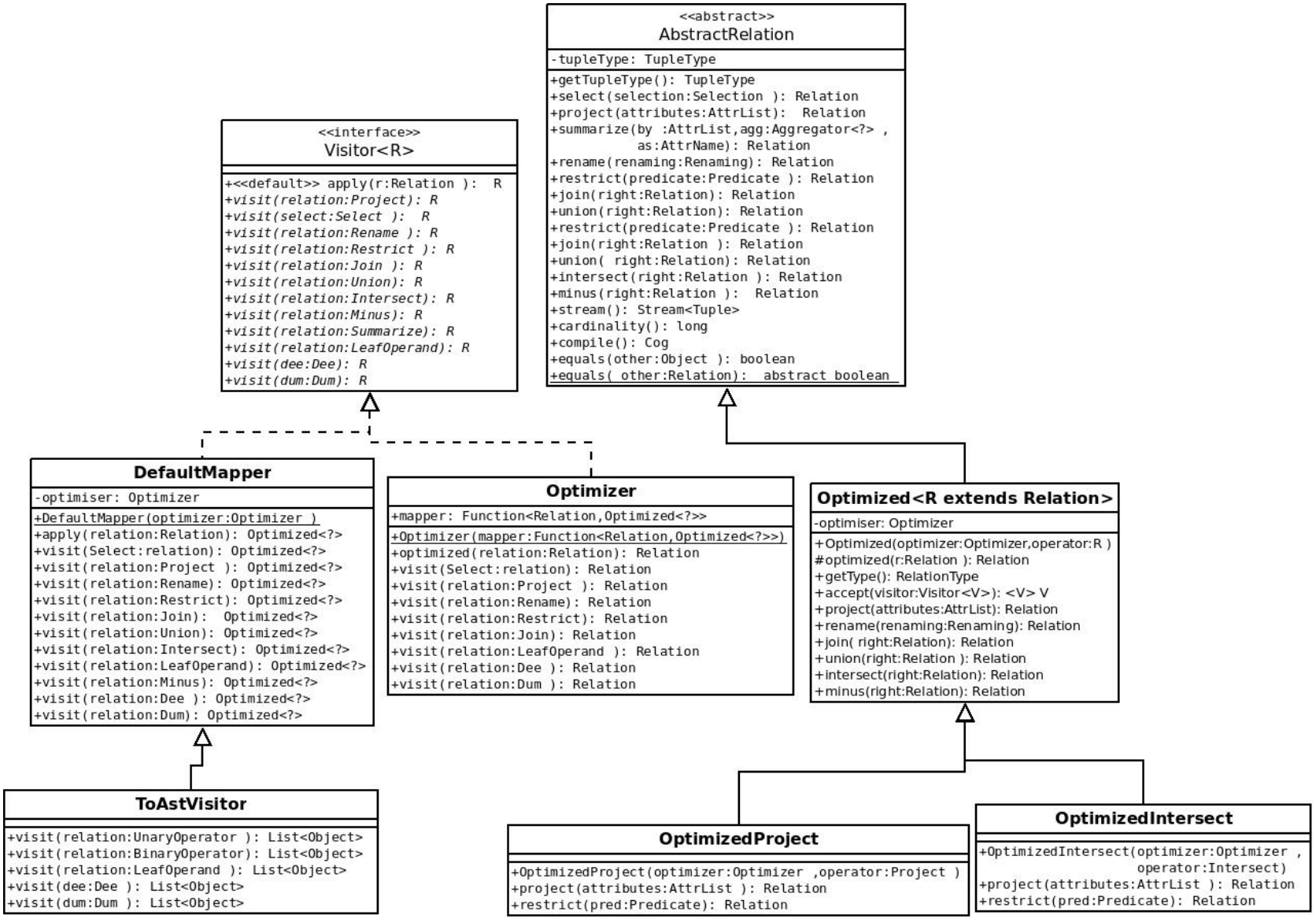


FIGURE 4.6 – Diagramme de classe pour l'optimisation en JAlf.

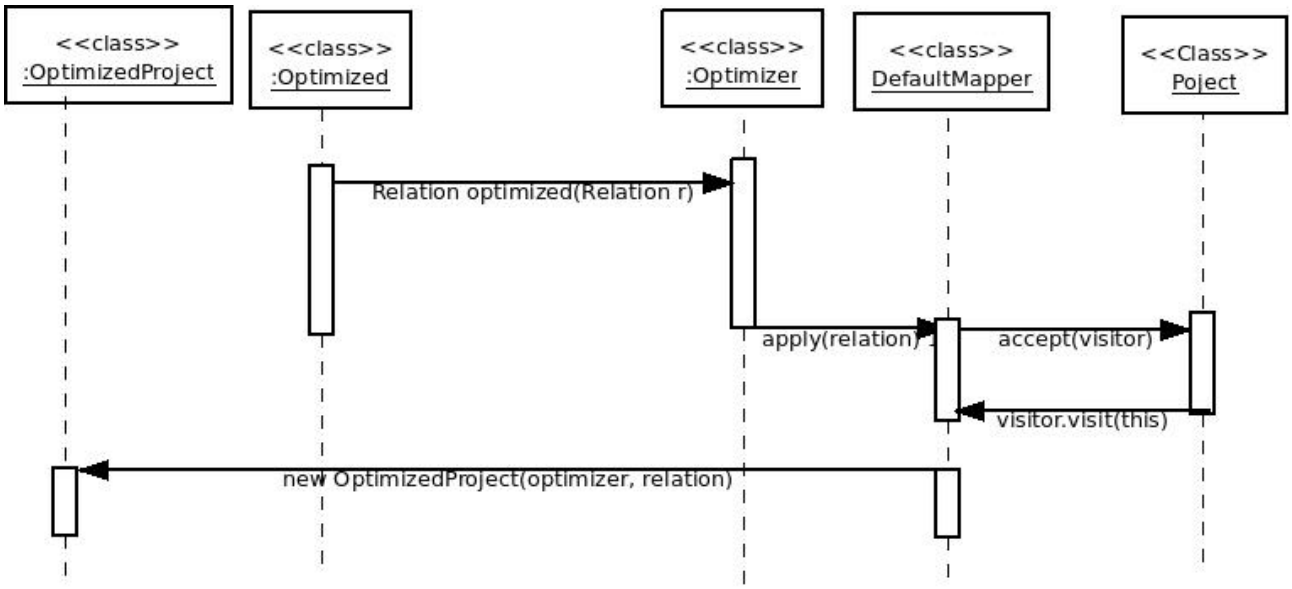


FIGURE 4.7 – Appels des méthodes pour l'optimisation en JAlf.

2 Construction de nos deux relations d'exemples

Dans nos explications des différentes fonctionnalités que nous avons ajoutées à JAlf, nous nous servirons de deux relations d'exemples appelées respectivement R_1 et R_2.

id_personnage	nom	prenom	statut	ville
P1	Haddock	Archibald	20	Louvain-la-Neuve
P2	Rastapopoulos	Roberto	10	Anvers
P3	Castafiore	Bianca	30	Tubize
P4	Lampion	Seraphin	30	Louvain-la-Neuve
P5	Halambique	Nestor	50	Huy

TABLE 4.1 – R_1

id_personnage	nom	prenom	statut	ville
P6	Halambique	Alfred	30	Huy
P7	Sanzot	Raoul	30	Bruxelles
P8	Tournesol	Tryphon	30	Louvain-la-Neuve
P9	Bergamotte	Hippolyte	30	Louvain-la-Neuve
P5	Halambique	Nestor	50	Huy

TABLE 4.2 – R_2

Dans un premier temps, nous créons les noms d'attributs, grâce à la méthode statique `attr` de la classe DSL.java et qui nous renvoie une instance de `AttrName`. Ensuite, nous devons créer nos deux relations dites « en mémoire ». Ceci se fait grâce à la méthode statique `relation` de la classe DSL.java. Cette méthode prend en paramètre un en-tête (`Heading`), une liste de clés (`Keys`) et une liste de tuples (`Tuple...`). Nos deux relations représentent les opérandes de nos divers opérateurs.

Listing 4.9 – construction des noms d'attributs

```
public static final AttrName IDP = attr("id_personnage");
public static final AttrName NOM = attr("nom");
public static final AttrName PRENOM = attr("prenom");
public static final AttrName VILLE = attr("ville");
public static final AttrName STATUT = attr("statut");
```

Listing 4.10 – construction de nos deux relations d'exemples

```
public static Relation R_1() {
    return relation(
            heading(IDP, String.class, NOM, String.class, PRENOM, String.class,
                STATUT, Integer.class, VILLE, String.class),
```

```
            keys(key(IDP)),
            tuple(IDP, "P1", NOM, "Haddock", PRENOM, "Archibald", STATUT, 20, VILLE,
                "Louvain-la-Neuve"),
            tuple(IDP, "P2", NOM, "Rastapopoulos", PRENOM, "Roberto", STATUT, 10,
                VILLE, "Anvers"),
            tuple(IDP, "P3", NOM, "Castafiore", PRENOM, "Bianca", STATUT, 30, VILLE,
                "Tubize"),
            tuple(IDP, "P4", NOM, "Lampion", PRENOM, "Seraphin", STATUT, 30, VILLE,
                "Louvain-la-Neuve"),
            tuple(IDP, "P5", NOM, "Halambique", PRENOM, "Nestor", STATUT, 50, VILLE,
                "Huy")

            );
}

public static Relation R_2() {
    return relation(
            heading(IDP, String.class, NOM, String.class, STATUT, Integer.class,
                VILLE, String.class),
            keys(key(IDP)),
            tuple(IDP, "P6", NOM, "Halambique", PRENOM, "Alfred", STATUT, 30, VILLE,
                "Huy"),
            tuple(IDP, "P7", NOM, "Sanzot", PRENOM, "Raoul", STATUT, 30, VILLE,
                "Bruxelles"),
            tuple(IDP, "P8", NOM, "Tournesol", PRENOM, "Tryphon", STATUT, 30, VILLE,
                "Louvain-la-Neuve"),
            tuple(IDP, "P9", NOM, "Bergamotte", PRENOM, "Hippolyte", STATUT, 30,
                VILLE, "Louvain-la-Neuve"),
            tuple(IDP, "P5", NOM, "Halambique", PRENOM, "Nestor", STATUT, 50, VILLE,
                "Huy")

            );
}
```

3 Ajout de Union

3.1 Utilisation de Union dans JAlf

3.1.1 Petit rappel théorique

UNION : le résultat de cet opérateur noté $R_1 \cup R_2$ nous renvoie une relation qui inclut tous les tuples qui sont soit dans R_1 ou dans R_2 et les tuples qui sont dans les deux relations R_1 et

R_2. Le résultat renvoyé est sans doublon. R_1 et R_2 doivent être union compatible, autrement dit, elles doivent avoir le même en-tête (`Heading`).

id_personnage	nom	prenom	statut	ville
P1	Haddock	Archibald	20	Louvain-la-Neuve
P2	Rastapopoulos	Roberto	10	Anvers
P3	Castafiore	Bianca	30	Tubize
P4	Lampion	Seraphin	30	Louvain-la-Neuve
P5	Halambique	Nestor	50	Huy
P6	Halambique	Alfred	30	Huy
P7	Sanzot	Raoul	30	Bruxelles
P8	Tournesol	Tryphon	30	Louvain-la-Neuve
P9	Bergamotte	Hippolyte	30	Louvain-la-Neuve

TABLE 4.3 – $R_1 \cup R_2$.

3.1.2 Emploi de union dans la JAlf

Nous construisons notre opérateur union en appelant la méthode statique `union` de la classe `DSL` en lui passant en paramètre nos deux relations d'exemples, ce qui nous renvoie comme résultat une relation à qui nous donnons comme nom de variable : `actual`. Nous construisons également une relation à qui nous donnons comme nom de variable `expected`. Cette dernière contient les tuples attendus par l'exécution de notre opérateur Union. Ce qui lance réellement notre compilation, comme nous le découvrirons par la suite, c'est la méthode `assertEquals` de JUnit. Celle-ci appelle la méthode `equals` d'une classe qui implémente l'interface `Relation`. Cette méthode `equals` compare nos deux relations `actual` et `expected`.

Listing 4.11 – methode `union` de la classe DSL.java et lancement de la compilation

```
Relation actual = union(R_1(), (R_2());
 assertEquals(expected, actual);
```

Listing 4.12 – création de la relation attendue (`expected`) en appliquant union

```
Relation expected = relation(
   heading(IDP, String.class, NOM, String.class, PRENOM, String.class, STATUT,
       Integer.class, VILLE, String.class),
              keys(key(IDP)),
              tuple(IDP, "P1", NOM, "Haddock", PRENOM, "Archibald", STATUT, 20,
                  VILLE, "Louvain-la-Neuve"),
              tuple(IDP, "P2", NOM, "Rastapopoulos", PRENOM, "Roberto", STATUT, 10,
                  VILLE, "Anvers"),
              tuple(IDP, "P3", NOM, "Castafiore", PRENOM, "Bianca", STATUT, 30,
                  VILLE, "Tubize"),
              tuple(IDP, "P4", NOM, "Lampion", PRENOM, "Seraphin", STATUT, 30, VILLE,
                  "Louvain-la-Neuve"),
```

```
tuple(IDP, "P5", NOM, "Halambique", PRENOM, "Nestor", STATUT, 50,
    VILLE, "Huy"),
        tuple(IDP, "P6", NOM, "Halambique", PRENOM, "Alfred", STATUT,
            30, VILLE, "Huy"),
tuple(IDP, "P7", NOM, "Sanzot", PRENOM, "Raoul", STATUT, 30, VILLE,
    "Bruxelles"),
tuple(IDP, "P8", NOM, "Tournesol", PRENOM, "Tryphon", STATUT, 30,
    VILLE, "Louvain-la-Neuve"),
tuple(IDP, "P9", NOM, "Bergamotte", PRENOM, "Hippolyte", STATUT, 30,
    VILLE, "Louvain-la-Neuve")
);
```

3.2 Comment avons-nous implémenté Union dans JAlf

C'est le premier opérateur que nous avons ajouté à JAlf. Pour ajouter cette opérateur, nous avons suivi la méthode TDD que nous avons expliqué à la (section 2.2, page 25). Méthode qui consiste à ajouter un test avant d'implémenter le code source qui doit réussir le test. Nous avons suivi plusieurs étapes que nous allons décrire dans les sections suivantes. Pour l'ajout des autres opérateurs nous avons suivi les mêmes étapes avec quelques variantes pour certains opérateurs.

3.2.1 Ajout d'un test

En premier lieu, nous avons ajouté un test définissant les spécificités de l'opérateur Union. Comme nous l'avons expliqué précédemment dans la méthode TDD, notre but est de réussir le test Junit.

3.2.2 Ajout de Union dans la classe DSL

Cette étape consistait à ajouter une méthode statique `union` dans la classe `DSL`. Cette méthode prend deux relations en paramètre et retourne une `Relation` qui correspond à l'union des deux relations reçues en paramètre. Voici ci-dessous cette méthode :

Listing 4.13 – methode union dans DSL.java

```
public static Relation union(Relation left, Relation right) {
    return left.union(right);
}
```

Remarque : La classe `DSL` se situe dans le package `jalf`. Celui-ci est le package principal de l'application. C'est dans celui-ci que se situent les classes et interfaces principales. Dans le même package, nous trouvons l'interface `Relation` qui est l'interface principale de tous type de relation de JAlf, aussi bien les relations dites « en mémoire » que les relations résultantes d'opérateur. Nous avons parlé des différentes relations plus en détail lorsque nous avons évoqué

la notion de relation (voir section 1.1, page 37).

3.2.3 Ajout Union dans l'interface Relation et dans la classe AbstractRelation

Dans cette étape nous ajoutons l'opérateur Union dans l'interface `Relation` et dans la classe `AbstractRelation`. Ici notre opérateur union retourne une relation, puis nous avons ajouté la signature de la méthode `union` dans notre interface `Relation`. Le code ci-dessous est l'ajout de `union` dans l'interface `Relation` :

Listing 4.14 – méthode Union dans Relation.java

```
        Relation union(Relation right) ;
```

Et voici le code de l'ajout de Union dans la classe `AbstractRelation` :

Listing 4.15 – méthode union dans AbstractRelation

```
@Override
public Relation union(Relation right) {
    return new Union(this, right);
}
```

Cette méthode appelle le constructeur de l'opérateur concerné, ici en l'occurrence `Union`, qui prend deux relations en paramètre.
La classe `AbstractRelation` implémente l'interface `Relation`. Cette classe est appelée par des méthodes statiques de la classe `DSL` afin de créer des objets opérateurs, des clés, des relations et des tuples. Cette classe se trouve dans le package `jalf.compiler`. Ce package concerne la compilation des différents opérateurs. Nous avons mentionné ce package plus en détail dans la partie « notion de compilation »(section 1.3, page 44).

3.2.4 Ajout de la classe Union dans le package jalf.relation.algebra

A cette étape, nous allons construire notre opérateur Union. Pour ce faire, nous allons nous intéresser au package `jalf.relation.algebra`. Ce package contient tous les opérateurs : les opérateurs unaires tels que Project ou les opérateurs binaires tels que Join ou encore Union. Chaque opérateur a sa propre classe et nous renvoie toujours après compilation une relation résultante. Dans ce package, la classe `Operator` est la classe parente de toutes les classes définissant des opérateurs. Cette classe hérite de la classe `AbstractRelation` et est hérité des classes `UnaryOperateur` et BinaryOperator. La classe `UnaryOperateur` prend une relation en argument et retourne une relation. Tandis que la classe `BinaryOperator` prend deux relations en argument et retourne une seule relation.

Notre classe `Union` hérite de la classe `BinaryOperator` qui représente donc les opérateurs binaires. La classe Union devra contenir trois champs :

- left : de type Relation ;
- right : de type Relation ;
- type : de type RelationType.

Les champs `left` et `right` représentent les deux relations sur lesquelles porte l'opérateur Union. L'attribut `type` représente le type de la relation résultante de l'opération Union. Voici le code du constructeur appelé notamment par la méthode `union` de notre classe `AbstractRelation`.

Listing 4.16 – Constructeur de Union

```
public Union(Relation left, Relation right){
    this.left = left;
    this.right = right;
    this.type = typeCheck();
}
```

Les deux champs de type `Relation` peuvent être des relations résultantes d'un opérateur ou de relations que nous appellerons « relation en mémoire », c'est-a-dire des relations que l'utilisateur aura construites, encodées ou encore provenant d'un fichier de type csv. Ces deux sortes de relation sont expliquées dans la section 1.1, page 37.

La méthode `typeCheck` nous retourne le type de la relation après une vérification du type `RelationType`. Ici, nous devons vérifier ce que l'on appelle « union compatible », c'est-à-dire que les attributs des deux relations ont des noms identiques et sont du même type. La méthode `typeCheck` appelle une méthode `union` qui se trouve dans le fichier `RelationType` et renvoie le nouveau `RelationType` pour l'opérateur Union. Le type d'une Relation est basée sur l'en-tête (`Heading`) de la relation nous l'avons expliqué plus en détail dans la section 1.2, page 42. La figure 4.8 montre le diagramme de séquence de tous les appels de méthodes que nous venons d'expliquer jusqu'ici.

3.2.5 Compilation de l'opérateur Union

Jusqu'ici nous avons une nouvelle instance de la classe Union, qui possède une relation $left$ et une relation *right* et un type. Mais nous n'avons pas encore la relation résultante de notre opérateur. Ce résultat est obtenu grâce a un processus que nous appellerons compilation. Ce processus consiste à parcourir en profondeur l'arbre syntaxique (AST). La compilation elle-même se fait dans le fichier`Cog.java`. Voici la méthode `union` qui nous fourni le résultat :

Listing 4.17 – Calcul du resultat de Union

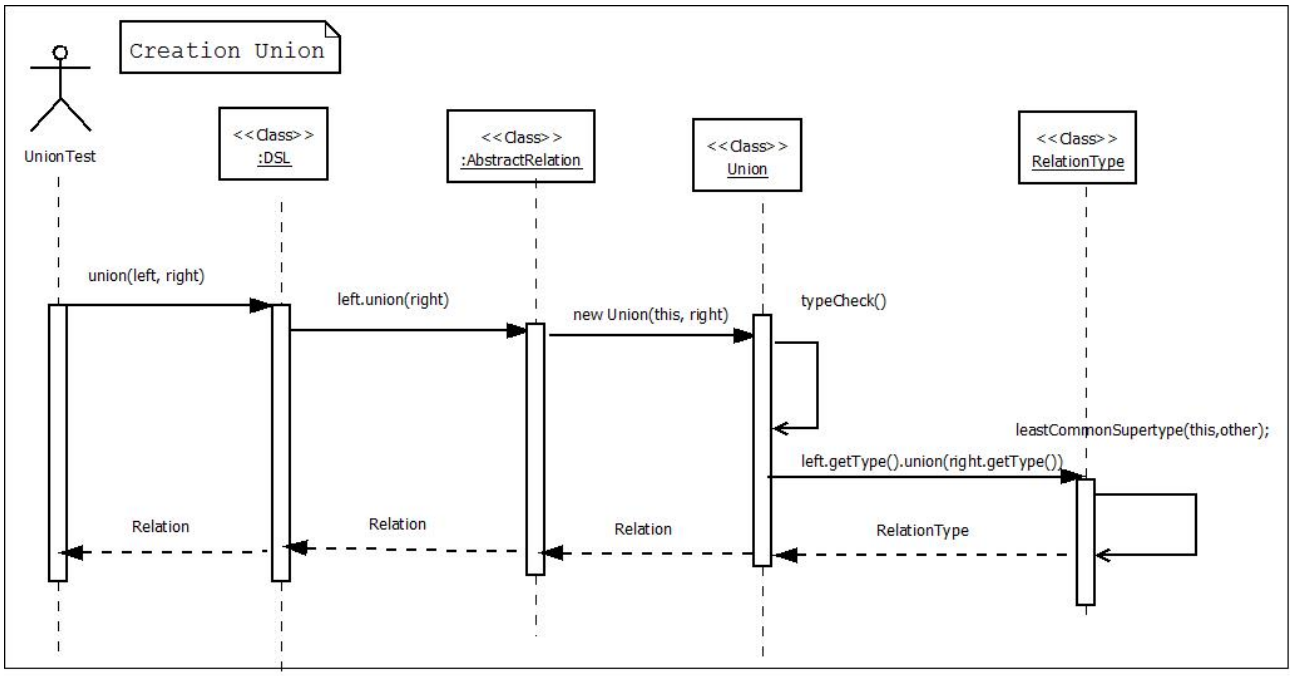


FIGURE 4.8 – Création de Union.

```
public Cog union(Union union, Cog right) {
    // stream compilation: concat + distinct
    Supplier<Stream<Tuple>> supplier = () ->{
           Stream<Tuple> leftStream = this.stream();
            Stream<Tuple> rightStream = right.stream();
            return Stream.of(leftStream, rightStream)
                    .reduce(Stream::concat)
                    .orElse(Stream.empty())
                    .distinct();
        };
    return new BaseCog(union, supplier);
}
```

Nous utilisons les streams de Java 8, dont nous avons évoqués certains avantages dans la section 1.4, page 20. Dans cette méthode, nous concaténons les deux streams de type `Tuple` contenant chacune les tuples des deux relations, pour former un `Supplier` de stream de `Tuple`. La méthode distinct de Java 8 fait le travail à notre place, ne retenant que les tuples différents. Pour ce faire la méthode `distinct` utilise les méthodes `hashCode` et `equals` des objets contenus dans la stream. Nous avons donc du vérifier si ces deux méthodes étaient correctement implémentées dans la classe Tuple, ce qui était évidemment le cas. L'opération `distinct` est ce qu'on appelle en Java 8, une opération intermédiaire d'une stream au même titre que l'opération `filter`. Quant à l'opération `reduce`, c'est une opération terminale. Comme nous l'avons vu, les opérations sur les streams sont paresseuses (« lazy »), et une opération terminale est nécessaire pour déclencher

l'opération. Quant à la méthode `supplier()` de Java 8, elle retourne un `Supplier<T>` qui est un conteneur pour des résultats de type T. Le `Supplier` n'a qu'une seule méthode `get` qui retourne les objets de type T qu'il contient. Comme nous venons de le voir, Java 8 nous permet de coder le résultat de l'opérateur union en quelques lignes.

Mais comment cette méthode `union` se trouvant dans le fichier `Cog.java` est appelée ? C'est ce que nous allons expliquer. La compilation est lancée lors de l'appel de la méthode `equals(Object other)` située dans la classe `AbstractRelation`. Cette méthode `equals` appelle la méthode `equals(Relation other)` dans la classe `SetMemoryRelation`.

La méthode `equals` ci-dessous appelle la méthode `stream` dans `AbstractRelation` qui appelle sa méthode `accept`.

Listing 4.18 – méthode equals dans SetMemoryRelation

```
public boolean equals(Relation other) {
    if (!(other instanceof SetMemoryRelation))
        other = other.stream().collect(SetMemoryRelation.collector(other.getType()));
    return equals((SetMemoryRelation)other);
}
```

La méthode `equals` appelle la méthode `stream` dans `AbstractRelation` qui appelle sa méthode `compile`. La méthode `compile` ci-dessous crée un nouvel objet `Compile` et appelle la méthode `accept` dans la classe `Union`.

Listing 4.19 – méthode compile dans AbstractRelation

```
private Cog compile() {
    return accept(new Compiler());
}
```

La méthode `accept` ci-dessous de la classe `Union` reçoit un `Visitor<Cog>` en paramètre et appelle la méthode `visit` du `Visitor` reçu en paramètre. Ici le `Visitor` est une instance de la classe Compiler. Remarquons que nous passons la référence de l'objet visité à la méthode `visit` de l'objet `visiteur`. Ce principe d'objet visiteur et d'objet visité correspond au design pattern « visitor ». Le principe est simple, nous avons un client qui envoie un visiteur à un objet à visiter. Ce dernier va via sa méthode `accept` accueille le visiteur et place sa référence dans l'appel de la méthode `visit` du visiteur. Le visiteur ayant obtenu la référence de l'objet visité, reprends la main grâce à sa méthode `visit` et peut appliquer un traitement à l'objet visité. Ceci permet d'ajouter des méthodes à un objet visité sans modifier son code, et de plus ses traitements seront délégués au visiteur. Mais encore, cela permet à un objet visité d'accepter différents types de visiteurs qui lui appliquerons différents traitements.

Listing 4.20 – méthode accept dans Union

```
public <R> R accept(Visitor<R> visitor) {
```

```
    return visitor.visit(this);
}
```

Voici donc la méthode `visit` ci-dessous du visiteur `Compiler`. Pour chacune des relations *left* et *right*, on appelle leur méthode `accept`. Si ces relations sont des relations en mémoire (c'est-à-dire des feuilles dans notre arbre syntaxique), c'est la méthode `accept` de la classe `SetMemoryRelation` qui est appelée. Si par contre elles sont des opérateurs, c'est la méthode `accept` de l'opérateur concerné qui est appelée. La dernière ligne appelle notre méthode `union` dans `Cog.java`.
A ce stade notre opérateur Union est ajouté avec succès et notre test le concernant passe.

Listing 4.21 – méthode visit dans Compiler

```
public Cog visit(Union relation) {
    Cog left = relation.getLeft().accept(this);
    Cog right = relation.getRight().accept(this);
    return left.union(relation, right);
}
```

3.2.6 Optimisation de l'opérateur Union

Le but de l'optimisation est de fournir plus rapidement des résultats équivalents, lors d'une requête ou une séquence d'opérateurs dans JAlf. La règle la plus simple consiste à pousser au plus profond de l'arbre syntaxique l'opérateur unaire Projet, si celui-ci porte par exemple sur un opérateur binaire tel que Union. Ce sont les opérateurs les plus profonds dans l'arbre qui sont bien sûr exécutés en premier lieu. La même règle s'applique à l'opérateur `Restrict`. L'opérateur `Projet` diminue le nombre d'attributs d'une relation, quant à l'opérateur `Restrict` il diminue le nombre de tuples d'une relation. Il est bien sûr avantageux de les appliquer en premier lieu aux deux opérandes de l'opérateur union et ensuite faire l'union des deux opérandes ainsi réduites. Nous pouvons trouver ces différentes règles d'optimisation et d'autres dans la section 1.4, page 47. Comme précédemment, notre but consistait à réussir le test contenu ici dans le fichier `TestOptimizedUnion.java`. Comme pour les autres opérateurs nous avons du créer un fichier spécifique à l'optimisation de notre opérateur Union, notre fichier est `OptimizedUnion.java`. La plupart des fichiers concernant l'optimisation se trouvent dans le package `jalf.optimiser`. La classe `OptimizedUnion` hérite de la classe `Optimized<R extends Relation>` dont R est un type générique qui est `Union` dans ce cas. Le test de l'optimisation de `Union` suivi de `Project` invoque une méthode `optimized` se trouvant dans le fichier `OptimizerTest.java`. Voici cette méthode ci-dessous, elle lance l'optimisation de l'opérateur `Union` :

Listing 4.22 – méthode optimized dans OptimizerTest

```
protected Relation optimized(Relation operator) {
```

```
    return optimizer.optimized(operator);
}
```

Elle appelle la méthode `optimized` dans `Optimizer`, le paramètre `relation` est de type `Union`.

Listing 4.23 – méthode optimized dans Optimizer

```
protected Relation optimized(Relation relation) {
    return mapper.apply(relation);
}
```

C'est la méthode ci-dessous la méthode `apply` de `DefaultMapper` qui est ensuite appelée.

Listing 4.24 – méthode apply dans DefaultMapper

```
@Override
public Optimized<?> apply(Relation relation) {
    return relation.accept(this);
}
```

Comme le paramètre `relation` est de type `Union`, c'est donc la méthode `accept` de la classe `Union` qui est donc appelée. Comme nous l'avons expliqué précédemment lors de la compilation de Union, nous utilisons ici encore le design pattern « visitor ». Une instance de `Union` reçoit un visiteur `DefaultMapper(optimiser)` et devient donc un objet visité. Cet objet visité appelle la méthode `visit` de son visiteur et lui passe bien sûr sa référence.

Listing 4.25 – méthode accept dans Union

```
@Override
public <R> R accept(Visitor<R> visitor) {
    return visitor.visit(this);
}
```

La méthode `visit` reçoit donc en paramètre une relation de type `Union` qui est ensuite passée en paramètre du constructeur de la classe `OptimizedUnion`. Dans cette méthode, on peut voir notre objet `Optimized<OptimizedUnion>` créé qui invoquera par la suite sa méthode `project`. Ci-dessous la méthode `visit` :

Listing 4.26 – méthode visit dans DefaultMapper

```
public Optimized<?> visit(Union relation) {
    return new OptimizedUnion(optimizer, relation);
}
```

Regardons de plus près notre méthode `project`, nous appliquons à nos deux opérandes $left$ et $right$ notre projection, mais de plus ses deux opérandes sont optimisées.

Listing 4.27 – méthode project dans OptimizedUnion

```
public Relation project(AttrList attributes){
    // get the first operand
    Relation left = operator.getLeft();
    left = optimized(left).project(attributes);

    // get the second operand
    Relation right = operator.getRight();
    right = optimized(right).project(attributes);

    Relation r = left.union(right);

    return r;
}
```

Comme ici nos deux opérandes ne sont pas des opérateurs, mais bien des relations dites en mémoires `SetMemoryRelation`, la méthode `optimized` appellera la méthode `visit` de `DefaultMapper` avec comme paramètre une relation `LeafOperand` (feuille dans l'arbre syntaxique). A noter que les deux opérandes pourraient être des opérateurs. En appelant l'optimisation de la relation *left* et de la relation *right*, nous parcourons récursivement notre arbre en profondeur jusqu'à rencontrer des relations dites en mémoire représentant les feuilles de notre arbre (« AST »). Ci-dessous la méthode `visit`.

Listing 4.28 – méthode visit avec paramètre LeafOperand dans DefaultMapper

```
@Override
public Optimized<?> visit(LeafOperand relation) {
       return new Optimized<Relation>(optimizer, relation);
}
```

Ici la méthode `visit` invoque la méthode `project` dans la classe `Optimized` et non comme précédemment celle de de `OptimizedUnion`. Comme nous pouvons le constater, l'optimisation de l'opérateur Project consiste à vérifier si tous les attributs de la relation sont égaux aux attributs portant sur la projection, et si tel est le cas nous omettons la projection. Ci-dessous la méthode `project`.

Listing 4.29 – méthode project dans Optimized

```
public Relation project(AttrList attributes) {
    AttrList opAttrs = operator.getType().toAttrList();
    if (opAttrs.equals(attributes)) {
        return operator;
    }
    return operator.project(attributes);
}
```

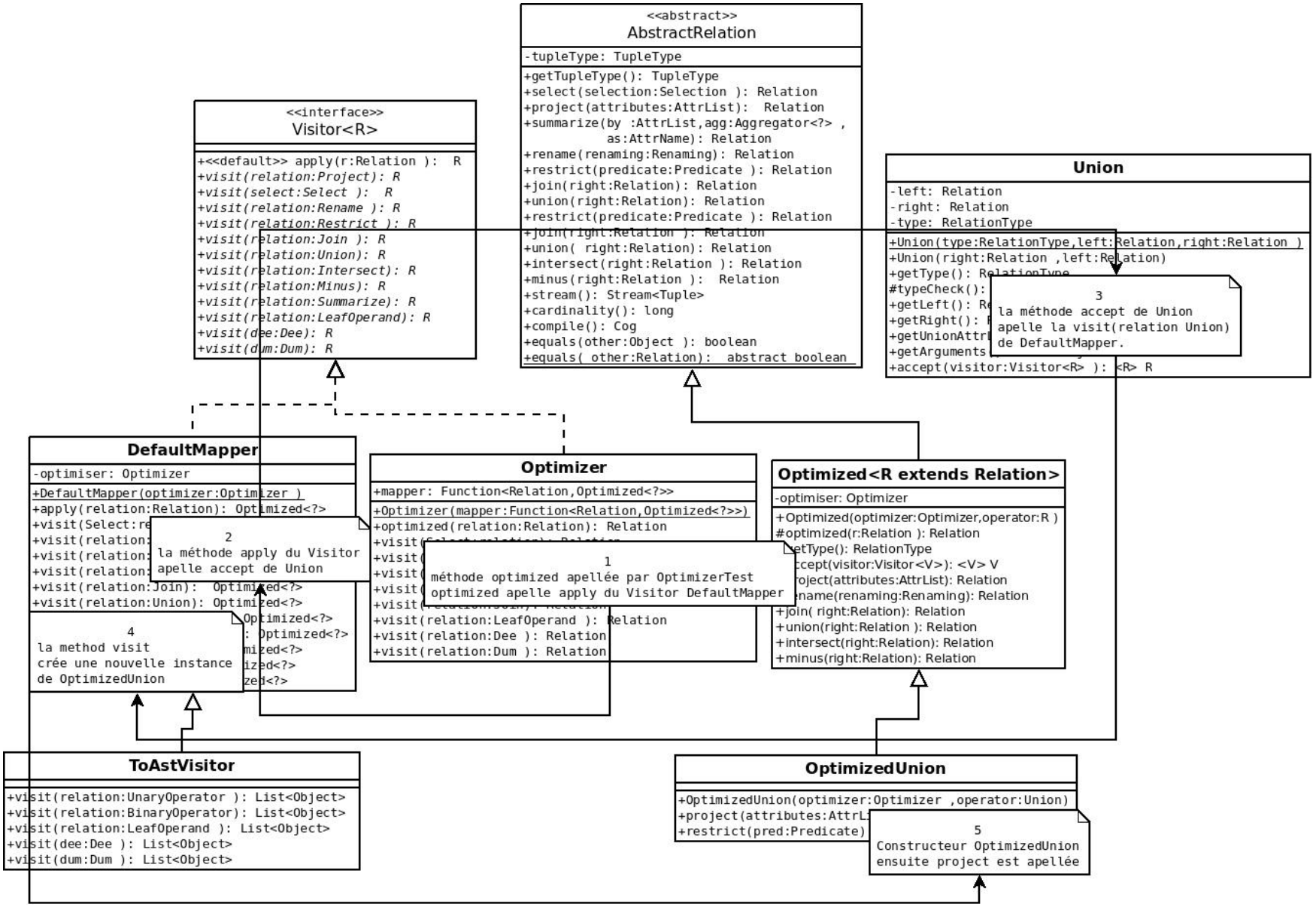


FIGURE 4.9 – Optimisation de Union.

4 Ajout de l'opérateur Intersect

4.1 Utilisation de Intersect dans JAlf

4.1.1 Petit rappel théorique

INTERSECT : le résultat de cet opérateur noté $R_1 \cap R_2$, nous renvoie une relation qui inclut tous les tuples qui sont à la fois dans R_1 et R_2. Le résultat renvoyé est sans doublons. R_1 et R_2 doivent être « union compatible ».

id_personnage	nom	prenom	statut	ville
P5	Halambique	Nestor	50	Huy

TABLE 4.4 – $R_1 \cap R_2$.

4.1.2 Emploi de Intersect dans JAlf

Nous construisons notre opérateur Intersect en appelant la méthode statique `intersect` de la classe `DSL` en lui passant en paramètre nos deux relations d'exemples. Elle nous renvoie comme résultat une relation à qui nous donnons comme nom de variable `actual`. Nous construisons également une relation à qui nous donnons comme nom de variable `expected`. Celle-ci contient les tuples attendus par l'exécution de notre opérateur Intersect. Ci-dessous un exemple d'utilisation de la méthode `intersect`.

Listing 4.30 – méthode intersect et lancement de la compilation

```
Relation actual = intersect(R_1(), (R_2());
assertEquals(expected, actual);
```

4.2 Comment avons-nous implémenté Intersect dans JAlf

L'ajout de cet opérateur est quasi similaire à la procédure suivie pour l'ajout de l'opérateur union. Nous avons dû bien sûr ajouter le fichier `Intersect.java`. La classe `Intersect` hérite comme la classe `Union` de la classe `BinaryOperator`. Les deux opérandes de cet opérateur doivent être également « union compatible ». L'optimisation de l'opérateur Intersect suit exactement les mêmes règles que l'opérateur Union. Voici notre méthode qui nous calcule la relation résultante. Nous utilisons une opération dite terminale de Java 8 : `collect`. Celle-ci nous crée un ensemble (Set) de tuples contenant les tuples de la relation $left$ qui est l'une des opérandes de intersect.

Ensuite nous appliquons à une `Stream<Tuple>`, représentant les tuples contenus dans l'autre opérande d'Intersect, c'est-à-dire l'opérande *right*, une opération intermédiaire Java 8 `filter`. Celle-ci ne retient que les tuples contenus dans notre `Set<Tuple>`.

Listing 4.31 – méthode intersect dans Cog

```
/** Default compilation of ‘intersect‘. */
public Cog intersect(Intersect intersect, Cog right) {

    Supplier<Stream<Tuple>> supplier = () ->{
        Stream<Tuple> leftStream = this.stream();
        Stream<Tuple> rightStream = right.stream();
        Set<Tuple> leftHashSet =
            leftStream.collect(Collectors.toCollection(HashSet::new));
        Stream<Tuple> intersectStream = rightStream
                    .filter(x -> leftHashSet.contains(x));
            return intersectStream;
        };
    return new BaseCog(intersect, supplier);
}
```

5 Ajout de Minus

5.1 Utilisation de minus dans JAlf

5.1.1 Petit rappel théorique

MINUS : le résultat de cet opérateur noté $R_1 \setminus R_2$, nous renvoie une relation qui inclut tous les tuples qui sont dans R_1 mais pas dans R_2. R_1 et R_2 doivent être « union compatible ».

id_personnage	nom	prenom	statut	ville
P1	Haddock	Archibald	20	Louvain-la-Neuve
P2	Rastapopoulos	Roberto	10	Anvers
P3	Castafiore	Bianca	30	Tubize
P4	Lampion	Seraphin	30	Louvain-la-Neuve

TABLE 4.5 – $R_1 \setminus R_2$.

5.1.2 Emploi de Minus dans la Dsl

Nous construisons notre opérateur Minus en appelant la méthode statique `minus` de notre DSL en lui passant en paramètre nos deux relations d'exemples, ce qui nous renvoie comme résultat une relation à qui nous donnons comme nom de variable `actual`. Nous construisons

également une relation à qui nous donnons comme nom de variable `expected`, celle-ci contient les tuples attendus par l'exécution de notre opérateur Minus. Ci-dessous un exemple d'utilisation de la méthode `minus` de la classe DSL.

Listing 4.32 – méthode minus et lancement de la compilation

```
Relation actual = minus(R_1(), (R_2());
assertEquals(expected, actual);
```

5.2 Comment avons-nous implémenté Minus dans JAlf

L'ajout de cet opérateur est quasi similaire à l'opérateur Intersect. Nous avons dû bien évidemment ajouter le fichier `Minus.java`. Les deux opérandes de cet opérateur doivent aussi être « union compatible ». L'optimisation de l'opérateur Minus suit exactement les mêmes règles que l'opérateur Union. Voici la méthode `minus` qui nous donne la relation résultante.

Listing 4.33 – méthode minus dans Cog.java

```
/** Default compilation of `minus`. */
public Cog minus(Minus minus, Cog right) {
    Supplier<Stream<Tuple>> supplier = () -> {
            Stream<Tuple> leftStream = this.stream();
            Stream<Tuple> rightStream = right.stream();

            Set<Tuple> rightHashSet = rightStream
                    .collect(Collectors.toCollection(HashSet::new));
            Stream<Tuple> minusStream = leftStream
                    .filter(x -> !rightHashSet.contains(x));
            return minusStream;
        };
        return new BaseCog(minus, supplier);
}
```

Nous utilisons une opération terminale de Java 8 `collect`, qui nous crée un ensemble (Set) de tuples contenant les tuples de la relation *right*. Attention que l'opérateur Minus n'est pas commutatif. La relation *right* représente la deuxième opérande de Minus. Ensuite, nous appliquons à une `Stream<Tuple>`, représentant les tuples contenus dans l'autre opérande nommée $left$ (la première opérande de Minus), une opération intermédiaire Java 8 `filter`. Celle-ci ne retient que les tuples qui ne sont pas contenus dans notre `Set<Tuple>`.

6 Ajout de Summarize

Summarize est un opérateur n'appartenant pas aux opérateurs de base de l'algèbre relationnelle. Il permet de regrouper des tuples d'une relation selon les valeurs de certains de ses attributs et d'y appliquer une fonction d'agrégation à chaque groupes ainsi constitués.
Les fonctions d'agrégats les plus communes qui peuvent s'appliquer à une collection de valeurs numériques sont SUM(somme), AVG(moyenne), MAX(maximum) et MIN (minimum). La fonction COUNT() quant à elle, est utilisée pour compter le nombre de tuples. A noter que dans JALF, nous avons aussi laissé la possibilité que MIN et MAX puissent également s'appliquer à des collections de String. Pour insérer Summarize dans JAlf, nous avons créé un package `jalf.aggregator` pour y implémenter nos fonctions d'agrégations. L'opérateur Summarize a nécessité la création de la classe `Summarize` se situant comme il se doit dans le package `jalf.relation.algebra`. Cette classe hérite de `UnaryOperator`, comme toute classe représentant un opérateur unaire. En plus de ses champs de classe inhérents aux opérateurs unaires, elle dispose de champs spécifiques. Le champ `aggregator` de type `Aggregator <?>` représente l'agrégat à appliquer à la relation via l'opérateur Summarize. Le champ de classe `by` de type `AttrList` représente quand à lui la liste d'attributs sur lesquels se portent le regroupement. Le champ `as` de type `AttrName` définit le nom du nouvel attribut résultant de l'opération de Summarize.

Listing 4.34 – les attributs de Summarize.java

```
private final Relation operand
private final RelationType type;
private final AttrList by;
private final Aggregator <?>aggregator;
private final AttrName as;
```

Notre opérateur Summarize devra définir un nouvel en-tête pour sa relation résultante, cet en-tête devra être composé de la liste des attributs reprise dans notre champ `by` augmentée de l'attribut stocké dans notre champ `as`. Notre opérateur définira pour cette relation un valeur pour son champ `type` de type `RelationType`. Dans son constructeur, notre opérateur appel une méthode `typeCheck`, qui elle appellera une méthode `summarize` dans la classe `RelationType` et renverra une nouvelle instance `RelationType` qui alimentera notre attribut `type`. Lors de la création d'une nouvelle instance de `RelationType`, le constructeur définit la valeur de son champ `tupleType` de type `TupleType` (une map composée de paires dont la clé est un nom d'attribut et la valeur le type de l'attribut) grâce à son attribut `heading` de type `Heading`. Voici la méthode `summarize` qui nous donne une instance de `RelationType`.

Listing 4.35 – méthode summarize dans RelationType.java

```
public RelationType summarize(AttrList by, Aggregator<?> agg, AttrName as) {
    AttrList l = by;
    if (agg.getAggregatedField() != null) {
```

```
        l = by.union(AttrList.attrs(agg.getAggregatedField()));
    }
     // check if the by+aggregated is valid
    checkValidAttrList(l);
    // check if by+aggregated doesn't contain `as`
    if (l.contains(as))
        throw new TypeException("Attribute `" + as + "` introduced by summarization
            already exists");

    return new RelationType(heading.summarize(by, as, agg.getResultingType(this)));
   }
```

6.1 Ajout summarize dans la classe DSL.java

Ici encore, comme pour les autres opérateurs, nous avons du écrire une méthode statique `summarize` dans la classe `DSL`. Voici ci-dessous cette méthode

Listing 4.36 – méthode summarize dans DSL.java

```
public static Relation summarize(Relation relation, AttrList by, Aggregator<?> agg,
    AttrName as ){
    return relation.summarize(by,agg,as);
}
```

Nous pouvons voir que les paramètres de cette méthode `summarize` correspondent aux champs de notre classe `summarize` : le paramètre `relation` correspond au champ `operand` ; le paramètre `by` au champ `by` ; le paramètre `agg` au champ `aggregator` et enfin le paramètre `as` au champ `as`.

6.2 Ajout summarize dans l'interface Relation.java et la classe AbstractRelation.java

Nous avons ajouté la signature de la méthode `summarize` dans notre interface `Relation` comme ci-dessous.

Listing 4.37 – signature de méthode summarize dans Relation.java

```
Relation summarize(AttrList by,Aggregator<?> agg,AttrName as);
```

Ensuite nous avons implémenté la méthode dans la classe `AbstractRelation`.

Listing 4.38 – méthode summarize dans AbstractRelation

```
@Override
public Relation summarize(AttrList by,Aggregator<?> agg, AttrName as) {
```

```
    return new Summarize(this, by,agg, as);
}
```

Comme nous avons déjà expliqué ce principe pour l'opérateur Union, c'est cette méthode qui invoque le constructeur de notre classe `Summarize`.

6.3 Ajout des différents agrégats dans la classe DSL

La méthode ci-dessous montre les différentes méthodes statiques qui appellent le constructeur de chaque agrégats. Les fonctions Max et Avg nécessitent un attribut sur lequel chaque fonction fera son calcul spécifique. La fonction Avg nécessite que l'attribut de son paramètre soit de type numérique. En effet on ne peut calculer une moyenne que sur des valeurs numériques. La fonction Count est particulière, dans le sens qu'elle ne prend pas d'attribut en paramètre. Elle nous renvoie le nombre de tuples.

Listing 4.39 – méthode static des agrégats dans DSL.java

```
public static Count count(){
    return Count.count();
}

public static Max max(AttrName attr) {
    return Max.max(attr);
}

public static Avg avg(AttrName attr) {
    return Avg.avg(attr);
}
```

.

6.4 L'interface Aggregator

Cette interface offre une série de méthodes implémentées par tous nos agrégats. La méthode `init` initialise les attributs d'une classe d'agrégat. Ces attributs sont nécessaires pour le calcul de la fonction. La méthode `acumulate` reçoit un tuple en paramètre, et suivant la valeur d'un attribut de ce tuple, la méthode fait un traitement spécifique aux attributs de l'agrégat.

La méthode `finish` applique un traitement final si nécessaire, et retourne la valeur attendue pour l'agrégat. La méthode `duplicate` crée une nouvelle instance de l'agrégat car à chaque collection de tuples retournée par `Summarize`, un nouvel agrégat devra être construit et initialisé.

La méthode `getResultingType` retourne le type du résultat attendu par l'agrégat et vérifie si l'attribut où se porte l'agrégat peut être calculé. En d'autres termes, le type de l'attribut correspond à un type qui peut être calculé par la fonction d'agrégation en question.

Listing 4.40 – interface Aggregator.java

```
public void init();
public void accumulate(Tuple t);
public T finish();
public AttrName getAggregatedField();
public Aggregator<T> duplicate();
public Type<?> getResultingType(RelationType type);
```

.

6.4.1 La classe Count

Utilisation de Summarize et Count dans JAlf

Pour vous expliquer l'utilisation de Summarise avec l'agrégat Count, commençons par définir la relation d'exemple R_1 :

ville	nbrepers_ville
Louvain-la-Neuve	2
Anvers	1
Tubize	1
Huy	1

TABLE 4.6 – nombre de personnage par ville dans R_1

Nous regroupons notre relation d'exemple R_1 par une liste d'attributs contenant notre attribut `VILLE`. A notre regroupement, nous lui appliquons l'agrégat Count et nous attribuons `nbrepers_ville` comme nom pour l'attribut contenant les résultats de notre fonction d'agrégation.

Listing 4.41 – summarize et count

```
Relation actual = summarize(R_1(), attrs(VILLE), count(), attr("nbrepers\_ville"));
```

.

Comment avons-nous implémenté Count dans JAlf

La classe `Count` implémente l'interface `Aggregator<Double>`. L'agrégat Count compte simplement le nombre de tuples pour chaque groupement de tuples. Il est le seul agrégat qui ne prend pas d'attribut `AttrName` en paramètre dans son constructeur.

La classe `Count` ne définit qu'un seul attribut, l'attribut `state` de type `Long`. Cet attribut est initialisé à zéro dans la méthode `init`, et est incrémenté de un par la méthode `accumulate`. La méthode `finish` nous retourne un `Long` qui est la valeur de l'attribut `state`.

6.4.2 La classe Max

Utilisation de Summarize et Max dans JAlf

Pour vous expliquer l'utilisation de Summarise avec l'agrégat Max, commençons par définir la relation d'exemple R_1 :

ville	max_statut
Louvain-la-Neuve	30
Anvers	10
Tubize	30
Huy	50

TABLE 4.7 – le statut maximal par ville dans R_1

Nous regroupons notre relation d'exemple R_1 par une liste d'attributs contenant notre attribut `VILLE`. A ce regroupement, nous appliquons l'agrégat Max à qui nous passons l'attribut `STATUT`, et nous attribuons `max_statut` comme nom pour l'attribut contenant les résultats de notre fonction d'agrégation.

Listing 4.42 – summarize et Max

```
Relation actual = summarize(R_1(),attrs(VILLE), max(STATUT),attr("max_Statut"));;
```

.

Comment avons-nous implémenté Max dans JAlf

La classe `Max` implémente l'interface `Aggregator<Comparable<?»`. Elle calcule la valeur maximale d'un attribut passé en paramètre `AttrName` de son constructeur. Cette valeur maximale est renvoyée pour chaque groupe de tuples agrégé par l'opérateur Summarize. La classe `Max` possède deux attributs : l'attribut `aggregatedField` de type `AttrName` représentant le nom de l'attribut dont nous allons retenir la valeur maximale ; et l'attribut `state` de type `Comparable<?>` qui renvoie la valeur maximale. L'attribut `state` est initialisé à `null` dans la méthode `init`, et prend par la suite la valeur maximal de l'attribut sur lequel se porte notre `Aggregator` Max . Cet attribut `state` est comparé à la valeur de l'attribut `aggregatedField` dans le tuple reçu en paramètre, dans la méthode `accumulate` pour définir sa nouvelle valeur. La méthode `finish` nous retourne la valeur de l'attribut `state` qui est de même type que `aggregatedField`.

6.4.3 La classe Avg

Utilisation de Summarize et Avg dans JAlf

Pour vous expliquer l'utilisation de Summarise avec l'agrégat Avg, commençons par définir la relation d'exemple R_1 :

ville	avg_statut
Louvain-la-Neuve	25
Anvers	10
Tubize	30
Huy	50

TABLE 4.8 – moyenne du statut par ville dans R_1

Nous regroupons notre relation d'exemple R_1 par une liste d'attributs contenant notre attribut `VILLE`. A notre regroupement, nous lui appliquons l'agrégat Avg à qui nous passons l'attribut `STATUS`, nous attribuons *avg_statut* comme nom pour l'attribut contenant les résultats de notre fonction d'agrégation.

Listing 4.43 – Summarize et Max

```
Relation actual = summarize(R_1(),attrs(VILLE), avg(STATUT),attr("avg_Statut"));;
```

.

Comment avons-nous implémenté Avg dans JAlf

La classe Avg implémente l'interface `Aggregator<Double>`. Elle représente la moyenne calculée sur un attribut de type `AttrName` passé en paramètre de son constructeur. Cette moyenne est calculée pour chaque groupe de tuples renvoyé par l'opérateur Summarize. La classe Avg possède trois attributs : l'attribut `aggregatedField` de type `AttrName` représentant le nom de l'attribut sur lequel on va calculer la moyenne ; l'attribut `state` de type `Double` qui va nous renvoyer la valeur moyenne ; et `counttuple` de Type `Integer` qui va compter le nombre de tuples.

L'attribut `state` sera initialisé à zéro ainsi que l'attribut `counttuple` par la méthode `init`. L'attribut `state` sera incrémenté par la valeur de l'attribut `aggregatedField` du tuple reçu en paramètre par la méthode `accumulate`. Dans cette dernière méthode, l'attribut `counttuple` sera incrémenté de un. La méthode `finish` retourne la valeur de l'attribut `state` de type `Double`, qui est au préalable divisé par l'attribut `counttuple`.

6.5 La classe Summarize

La méthode `apply` de notre classe `Summarize` est appelée par la méthode de `summarize` dans la classe `Cog.java`. Cette méthode `apply` utilise certaines nouvelles fonctionnalités de Java 8. La méthode est implémentée en utilisant une stream de tuples. L'Api Stream possède une méthode `collect`, méthode terminale, qui comme nous l'avons déjà expliqué, déclenche un traitement à appliquer à la stream. Cette méthode `collect` retourne un type R et utilise l'interface fonctionnelle `Collector<T, A, R>`. Elle transforme une stream dans une autre structure de données, comme par exemple une liste.
Pour mieux comprendre son fonctionnement, voici l'interface fonctionnelle Collector et ses méthodes à implémenter.

Listing 4.44 – interface Collector en java 8

```
public interface Collector<T, A, R> {
   Supplier<A> supplier();
   BiConsumer<A, T> accumulator();
   Function<A, R> finisher();
   BinaryOperator<A> combiner();
   Set<Characteristics> characteristics();
}
```

.

Dans cette interface, `T` est le type générique des éléments contenus dans la stream qui seront collectés. `A` est le type de l'accumulateur qui représente un objet dans lequel les données du résultat partiel seront accumulées pendant le traitement. `R` représente le type générique du résultat que la méthode `collect` nous fournira. La plupart du temps, le type `R` représente une collection, mais cela peut être un simple et unique entier (Integer) dans certains cas.

Les méthodes de l'interface Collector :

- La méthode `supplier` retourne un `Supplier`, c'est-à-dire une structure vide ou un objet qui nous servira d'accumulateur. Voici ci-dessous, le `Supplier` dans la méthode `apply` :

  Listing 4.45 – notre Supplier

  ```
  Supplier<Aggregator<?>> s = () -> ((Aggregator<?>) this.aggregator.duplicate();
  ```

  .

  Le `Supplier` est une nouvelle instance d'un `Aggregaror` (agrégat Max, Avg ou encore Count).

- La méthode `accumulator` retourne une fonction qui calcule la méthode de réduction. La fonction est de type interface fonctionnelle `BiConsumeur`, prend toujours deux objets en paramètre, et ne retourne rien (void). Sa seule méthode à implémenter est `accept`. Cette fonction change l'état de notre accumulateur. Voici ci-dessous notre `BiConsumeur`. Il prend en paramètre l'accumulateur qui est une instance d'agrégat et un tuple. La fonction appelle la méthode `accumulate` que nous avons implémentée au préalable pour chaque agrégat, et change donc l'état de notre accumulateur `agg`.

  Listing 4.46 – notre BiConsumer

  ```
  BiConsumer<Aggregator<?>, Tuple> b = new BiConsumer<Aggregator<?>,Tuple>(){
      @Override
      public void accept(Aggregator<?> agg, Tuple t) {
          agg.accumulate(t);
  ```

```
    }
};
```

- La méthode `finisher` retourne une fonction qui est invoquée à la fin du processus d'accumulation. Notre méthode `finish`, que nous avons implémentée dans chaque agrégat, représente cette fonction attendue. Par exemple pour l'opérateur Avg, après avoir additionné les différentes valeurs d'un attribut de la relation et compté le nombre de tuples d'un regroupement, la méthode `finish` calcule notre moyenne et nous la retourne.

- La méthode `combiner` retourne aussi une fonction qui combine les résultats de deux accumulateurs quand la stream est traitée en parallèle. La fonction est de type `BinaryOperateur` et possède deux paramètres de même type. Elle retourne un objet du type de ses paramètres. Sa méthode `apply` doit être obligatoirement implémentée. Dans notre cas, ce type est `Aggregator<?>`. Voici ci-dessous la génération de la fonction `BinaryOperateur`.

Listing 4.47 – notre BinaryOperateur

```
java.util.function.BinaryOperator<Aggregator<?>> f = new
    java.util.function.BinaryOperator<Aggregator<?>>(){
    @Override
    public Aggregator<?> apply(Aggregator<?> t, Aggregator<?> u) {
        return t;
    }
};
```

Une fois nos divers éléments `Supplier`, `BiConsumer` et le `BinaryOperator` implémentés, nous pouvons construire le `Collector`. Voici ci-dessous le code qui génère le `Collector` :

Listing 4.48 – Notre Collecteur

```
Collector<Tuple, Aggregator<?>, Aggregator<?>> coll = Collector.of(s, b, f);
```

Comme nous l'avons dit précédemment, nous devons passer un objet `Collector` en paramètre à la méthode `collect` de la stream. Pour faire les regroupements, nous avons utiliser la méthode `groupingBy` de `Collector` qui retourne un objet `Collector`. Cette méthode prend en paramètre une fonction de classification et un `Collector`. Voici ci-dessous la fonction de regroupement :

Listing 4.49 – fonction de regroupement

```
Function<Tuple,List<Object>> grouper = t -> t.fetch(this.by);
```

Elle prend un tuple en paramètre et retourne une liste des valeurs de ce tuple pour la liste d'attributs contenue dans `by`. Le `Collector` passé en paramètre est `coll`, celui que nous venons de construire. Il sert de méthode de réduction à appliquer aux valeurs de chacun des sous-groupes. La méthode `collect` nous retourne une `Map`, dont la clé est une liste de valeur des attributs

(résultat du regroupement obtenu par `groupingBy`), et dont la valeur est un `Aggregator`. Voici ci-dessous le code qui génère la Map.

Listing 4.50 – appel de groupingBy

```
Map<List<Object>,? extends Aggregator<?>> map=null;
 map = tuples.collect(Collectors.groupingBy(grouper, coll));
```

Une fois la Map créée, il reste à la parcourir afin de construire la relation résultante. Il faut exécuter la méthode `finish Aggragator` pour chaque valeur de la Map.

6.6 Optimisation de l'opérateur Summarize

Avec l'opérateur `Project`, nous pouvons optimiser `Summarize`. Il faut d'abord appliquer la projection, si et seulement si la liste des attributs de la projection ne contient pas l'attribut `as` de `Summarize` qui est le nom de l'attribut résultant de l'agrégat. Voici ci-dessous le code la méthode `project` pour l'optimisation de l'opérateur `Summarize`.

Listing 4.51 – optimisation de Summarize avec Project

```
public Relation project(AttrList attributes) {
    Relation r = operator.getOperand();
    if (!attributes.contains(operator.getAs())){
        r = optimized(r).project(attributes);
    } else{
        r = optimized(r).summarize(operator.getBy(), operator.getAggregator(),
            operator.getAs());
        r = r.project(attributes);
    }
    return r;
}
```

Avec l'opérateur Restrict, nous pouvons optimiser l'opérateur Summarize. Il faut d'abord appliquer la restriction, si et seulement si la liste des attributs contenue dans les différents prédicats de `Restrict`, ne contient pas l'attribut `as` de la classe`Summarize` qui est le nom de l'attribut résultant de l'agrégat. Voici ci-dessous le code la méthode `restrict` pour l'optimisation de l'opérateur Summarize.

Listing 4.52 – optimisation de l'opérateur Summarize avec l'opérateur Project

```
public Relation restrict(Predicate pred) {
    Relation r = operator.getOperand();
    AttrList predattrs = pred.getReferencedAttributes();
    if (!predattrs.contains(operator.getAs())){
        r = optimized(r).restrict(pred);
```

```
        r = r.summarize(operator.getBy(), operator.getAggregator(), operator.getAs());
    } else{
        r = optimized(r).summarize(operator.getBy(), operator.getAggregator(),
            operator.getAs());
        r = r.restrict(pred);
    }
    return r;
}
```

7 Ajout des classes Key et Keys

7.1 Création du package jalf.constraint

Pour implémenter l'inférence des clés, nous avons crée le package `jalf.constraint`. Ce package implémente les clés et pourra par la suite implémenter d'autres types de contraintes. Une autre contrainte pourrait être par exemple que les valeurs d'un attribut doivent être plus grandes qu'une valeur désignée. Dans JAlf, une relation peut posséder une ou plusieurs clés. Une clé désigne un attribut ou un groupe d'attributs dont les valeurs sont uniques pour chaque tuple de la relation. Une clé est représentée par la classe `Key`, et une collection de clé de type `Key` est représenté par `Keys`. Dans JAlf, une relation possède donc une collection `Keys`, qui contient au moins une `key`. Dans ce package, nous avons crée une interface `Constraint` qui n'a qu'une seule méthode : `check(Relation r)`. La méthode `check` vérifie si la contrainte est respectée pour une relation donnée.

7.2 Ajout des méthodes key et keys dans la classe DSL

Comme nous l'avons déjà fait lors de la création de nouveaux opérateurs, il faut ajouter des méthodes statiques dans la classe `DSL`. Pour permettre à l'utilisateur de définir une clé, nous avons donc ajouté les méthodes `key` et `keys`. Voici ci-dessous ces deux méthodes.

Listing 4.53 – méthode key et keys dans DSL.java

```
public static Key key(AttrName... attrkey) {
    return Key.candidate(attrs(attrkey));
}

public static Keys keys(Key... keys) {
    return Keys.keys(keys);
}
```

7.3 Classes key et Keys

La classe `Key` n'a qu'un seul champ, le champ `attrsKey` de type `AttrList` qui défini une liste d'attributs composant la clé. Nous trouvons aussi notre méthode statique `candidate` qui appelle le constructeur. Cette méthode est appelée par notre méthode `key` depuis notre DSL. La classe `Key` implémente notre interface `Constraint`, et doit de ce fait implémenter la méthode `check`. Cette méthode `check` doit vérifier une contrainte sur une relation et retourner un `boolean`. Dans notre cas, pour une clé, la méthode procédera à deux vérifications. La première est que les attributs composant notre clé doivent appartenir à la relation, et la seconde que les attributs de la clé désignent des valeurs uniques pour chaque tuple de la relation. Pour savoir si ces attributs désignent des valeurs uniques, nous procédons à une projection de la relation sur les attributs composant la clé et nous comparons la cardinalité (nombre de tuples) du résultat de cette projection avec la cardinalité de la relation. Si les cardinalités sont égales alors la clé est correct. Voici ci-dessous le code de la méthode `check` :

Listing 4.54 – méthode check de Key

```
public boolean check(Relation r) {
    AttrList attrsrelation= r.getType().getHeading().toAttrList();
    AttrList intersect= attrsrelation.intersect(this.attrsKey);
    if (intersect.equals(this.attrsKey)){
        return (r.project(this.attrsKey).cardinality() == r.cardinality());
    }
    else {
        return false;
    }
}
```

Comme dit précédemment, la classe `Keys` représente une collection de `Key` et de ce fait elle implémente l'interface `Iterable<Key>` et possède également un champ `keys` de type `Set<Key>`. Elle implémente la méthode `check` qui vérifie l'ensemble des clés contenues dans son set. Cette méthode `check` est appelée pour chaque clé pour vérifier qu'une clé n'est pas une super clé d'une autre, c'est-à-dire que la liste des attributs d'une clé n'est pas un sur-ensemble ($\not\supseteq$) de la liste des attributs d'une autre clé. Voici ci-dessous le code de la méthode `check` :

Listing 4.55 – méthode check de Keys

```
public boolean check(Relation r){
    return this.stream().allMatch(k -> k.check(r)) && this.areKeysValid();
}


public boolean areKeysValid(){
    Stream<Key> comp = this.stream();
    Set<Key> comp1 = this.keys;
```

```
    return comp.anyMatch(k -> {
        Predicate<Key> p = (k1) -> (!k1.equals(k)) && (k1.isSubKey(k) ||
            k1.isSuperKey(k));
        return (comp1.stream().anyMatch(p))? false:true;
    });
}
```

Mais où est stockée notre instance de `Keys` ? Quand et où sa méthode `check` est-elle exécutée ?

Nous retrouvons un champ `keys` de type `Keys` dans la classe `MemoryRelation` qui est la parente de la classe `SetMemoryRelation`, ainsi que dans la classe `Operator` (classe parente de tous les opérateurs). Toutes relations dites en mémoire possèdent une collection de clés, ainsi que toutes relations résultantes d'opérateurs possèdent aussi une collection de clés recalculée par rapport à son opérateur. Pour une relation dite en mémoire, soit l'utilisateur définit sa ou ses clés qui sont dès lors vérifiées dans le constructeur de `SetMemoryRelation` via la méthode `check` de la classe Keys, soit l'utilisateur ne précise pas de clés et à ce moment une clé est générée automatiquement. Cette dernière a comme attribut tous les attributs de la relation, ce qui signifie qu'une vérification sera inutile.

Dans le cas de relations résultantes d'opérateurs, l'instance de `Keys` est recalculée de manière paresseuse (lazy). C'est-à-dire que la valeur de cette collection de `Key` est calculée lors de l'appel à la méthode `getKeys` implémentée dans la classe `Operator`. Cette méthode appelle elle-même la méthode `lazyComputeKey` spécifique à chaque opérateur. Cette dernière méthode retourne la collection de `Key`.

Listing 4.56 – méthode getKeys de Operator

```
public Keys getKeys() {
    if (this.keys == null) {
        this.keys = this.lazyComputeKey();
    }
    return this.keys;
}
```

.

7.3.1 Calcul de Keys de l'opérateur Intersect

La collection de `Key` de la relation résultante de l'opérateur Intersect, est l'union des deux collections de ses deux opérandes. La collection est calculée comme suit : aux attributs de chaque clé de la première opérande sont ajoutés les attributs des différentes clés de la deuxième opérande pour former une nouvelle clé. Voici le code du calcul de `Keys` :

Listing 4.57 – méthode lazyComputeKey de Intersect

```
    protected Keys lazyComputeKey() {
```

```
        return left.getKeys().simpleUnion(right.getKeys());
    }
    public Keys simpleUnion(Keys other){
        Set<Key> keyunion = new HashSet<Key>();
        keyunion.addAll(this.keys);
        keyunion.addAll(other.keys);
        return new Keys(keyunion);
    }
```

Voici un exemple appliqué sur R_1 et R_2, nos deux relations d'exemples :
Keys de R_1 : keys(key(NOM,PRENOM))
Keys de R_2 :keys(key(NOM,STATUT))
Keys de $R_1 \cap R_2$: keys(key(NOM,PRENOM,STATUT))

7.3.2 Calcul de Keys de l'opérateur Minus

La collection de `Key` de la relation résultante de l'opérateur Minus, est la collection de sa première opérande. Voici le code du calcul de `Keys` :

Listing 4.58 – méthode lazyComputeKey de Minus

```
    protected Keys lazyComputeKey() {
        return this.left.getKeys();
    }
```

Voici un exemple appliqué sur R_1 et R_2, nos deux relations d'exemples :
Keys de R_1 : keys(key(NOM,PRENOM))
Keys de R_2 : keys(key(STATUT))
Keys de $R_1 \setminus R_2$: keys(key(NOM,PRENOM))

7.3.3 Calcul de Keys de l'opérateur Union

La collection de `Key` de la relation résultante de l'opérateur Union, est une nouvelle collection contenant la clé la plus grande possible. Celle-ci est constituée de tous les attributs de la relation résultante. Voici le code du calcul de `Keys` :

Listing 4.59 – méthode lazyComputeKey de Union

```
    protected Keys lazyComputeKey() {
        return new Keys(this.getLargestKey());
    }
```

Voici un exemple appliqué sur R_1 et R_2, nos deux relations d'exemples :
Keys de R_1 : keys(key(NOM,PRENOM))
Keys de R_2 : keys(key(NOM,STATUT))
Keys de $R_1 \cup R_2$: keys(key(IDP,NOM,PRENOM,VILLE,STATUT))

7.3.4 Calcul de Keys de l'opérateur Project

La collection de `Key` de la relation résultante de l'opérateur Project, est une nouvelle collection contenant les clés dont la liste des attributs est un sous-ensemble de la liste des attributs sur laquelle se porte la projection. Dans le cas où aucune clé n'a pu être contenue dans la nouvelle collection, il faut alors ajouter une nouvelle clé constituée de tous les attributs de la relation résultante. Voici le code du calcul de `Keys` :

Listing 4.60 – méthode lazyComputeKey de Project

```
public Keys lazyComputeKey(){
    Keys keys = operand.getKeys();
    Predicate<Key> p = (k)-> isKeyPreserving(k);
    Set<Key> kept = keys.stream().filter(p).collect(Collectors.toSet());
    if (kept.isEmpty()) {
        return new Keys(type.getLargestKey());
    } else{
        return new Keys(kept);
    }
}
```

Voici deux exemples appliqués sur R_1,une des relations d'exemples :
premier cas : clé préservée par la projection
Keys de R_1 : keys(key(NOM))
Keys de $\pi_{\{NOM,PRENOM\}}(R_1)$: keys(key(NOM))

deuxième cas : aucune clé préservée par la projection
Keys de R_1 :keys(key(IDP)))
Keys de $\pi_{\{NOM,PRENOM\}}(R_1)$: keys(key(IDP,NOM,PRENOM,VILLE,STATUT))

7.3.5 Calcul de Keys de l'opérateur Restrict

La collection de `Key` de la relation résultante de l'opérateur Restrict, est la même collection que son opérande. Voici le code du calcul de `Keys` :

Listing 4.61 – méthode lazyComputeKey de Restrict

```
    protected Keys lazyComputeKey() {
        return this.operand.getKeys();
    }
```

Voici un exemple appliqué sur R_1, une des relations d'exemples :
Keys de R_1 : keys(key(NOM,PRENOM))
Keys de $\sigma_{condition}(R_1)$: keys(key(NOM,PRENOM))

7.3.6 Calcul de Keys de l'opérateur Rename

La collection de `Key` de la relation résultante de l'opérateur Rename, est la même collection que son opérande à ceci près que ses attributs sont renommés si nécessaire. Voici, le code du calcul de `Keys` :

Listing 4.62 – méthode lazyComputeKey de Rename

```
  protected Keys lazyComputeKey() {
    return operand.getKeys().rename(renaming);
}
```

Voici un exemple appliqué sur R_1, une des relations d'exemples :
Keys de R_1 : keys(key(IDP))
Keys de $\alpha[\{IDP\}AS\{attr(pers\_identificateur)\}](R_1)$: keys(key(attr(pers_identificateur))

7.3.7 Calcul de Keys de l'opérateur Join

La collection de `Key` de la relation résultante de l'opérateur Join, est constituée de toutes les clés de la première opérande. La collection est calculée comme suit :

chaque clé de la première opérande est unie avec chaque clé de la seconde opérande. Voici le code du calcul de `Keys` :

des deux collections de ses deux opérandes.

Listing 4.63 – méthode lazyComputeKey de Join

```
protected Keys lazyComputeKey() {
    return left.getKeys().complexUnion(right.getKeys());
}

public Keys complexUnion(Keys other) {
    Set<Key> keyunion = new HashSet<Key>();
    this.stream().forEach(
            (k) -> {
                Set<Key> newset = other.keys.stream().map(o -> o.union(k))
                        .collect(Collectors.toSet());
                keyunion.addAll(newset);
                ;
            });
    return new Keys(keyunion);
}
```

Voici un exemple appliqué sur R_1 et R_2, nos deux relations d'exemples :
Keys de R_1 : keys(key(NOM,STATUT),key(VILLE))
Keys de R_2 : keys(key(IDP))
Keys de $R_1 \bowtie R_2$: keys(key(NOM, STATUT,IDP),key(VILLE, IDP))

7.3.8 Optimisation de l'opérateur Project grâce à la clé

Grâce à cette nouvelle information qu'apporte les clés, nous avons pu optimiser la compilation de notre opérateur Project. Si les attributs sur lesquels porte la projection sont un sous-ensemble d'une des clés de l'opérande de Project, nous pouvons omettre la vérification des doublons dans les tuples. Bien d'autres optimisations pourront être intégrées dans JAlf grâce à l'inférence des clés.

Listing 4.64 – méthode compilation de Project

```
/** Default compilation of ‘project‘. */
public Cog project(Project projection) {
    AttrList on = projection.getAttributes();
    TupleType tt = projection.getTupleType();
    Supplier<Stream<Tuple>> supplier;
    // avoid duplicate
    if (projection.isKeyPreserving()){
        supplier = () -> this.stream().map(t -> t.project(on, tt));
    } else {
        // stream compilation: map projection + distinct
        supplier = () -> this.stream().map(t -> t.project(on, tt))
                    .distinct();
    }
    return new BaseCog(projection, supplier);
}
```

Conclusion

Durant ce mémoire, nous avons étudié une nouvelle façon de définir des requêtes pour un développeur. Nous avons dû comprendre, dans un premier temps, les difficultés du langage SQL et celles des librairies actuelles de connexion à une base de données. Pour cela, nous nous sommes beaucoup documentés sur les inconvénients de ces outils que nous voulions éliminer. Nous nous sommes énormément référés à la documentation du blogLambTry qui nous a permis de comprendre les motivations de JAlf. Dans un second temps, nous nous sommes plongés dans l'architecture existante de JAlf afin d'en comprendre les spécificités et de pouvoir ajouter notre contribution à ce projet. Cette seconde étape fut un grand défi pour nous car il n'est pas aisé de s'imprégner d'une architecture dont nous n'avons aucune connaissance. En parcourant l'architecture, nous avons établis les différents liens avec les concepts théoriques utilisés et les différentes notions liées à une relation.

Une fois que nous avions maîtrisé l'architecture de JAlf, nous avons ajouté des fonctionnalités. Nous avons débuté par l'ajout des opérateurs Union, Intersect et Minus. Par la suite, nous avons ajouté l'opérateur Summarize qui nous a posé plus de difficultés que les trois premiers opérateurs. En effet, la grande difficulté était qu'il fallait également introduire les agrégats ; nous avons donc dû définir une architecture pour les agrégats et l'opérateur Summarize qui devait parfaitement s'adapter à l'architecture existante et être en cohérence avec celle-ci. Pour finir, nous avons ajouté le concept de clé pour une relation. L'ajout des clés a été également un grand défi à relever. La grande difficulté résidait, tout comme Summarize, dans la conception d'une architecture bien adaptée à l'architecture existante. Nous nous sommes intéressés de près aux concepts théoriques liés aux clés pour l'optimisation des opérations.

Dans ce mémoire, nous avons contribué à l'avancement de JAlf qui, pour nous, s'est révélé être une belle alternative aux librairies actuelles. Actuellement, elle est encore incomplète, certaines fonctionnalités reste à implémenter. Malgré tout cela, il est toujours possible d'utiliser Alf qui, comme nous l'avons déjà mentionné à plusieurs reprises, est la version Ruby de JAlf.

Pour nous, ce projet a été une belle expérience riche en connaissances et nous espérons, dans un futur proche, pouvoir utiliser JAlf dans des cas concrets de développements d'applications.

Bibliographie

[1] David Colpaert. Un outil de gestion d'incohérences de données basé sur les vues intentionnelles et l'algèbre relationnelle appliquée à un cas de localisation de téléphones IP, promoteur du mémoire Dr. Kim Mens, année 2013-2014.

[2] Hugh Darwen. An Introduction to Relational Database Theory, 4ème édition, 2014.

[3] Scott Chacon and Ben Straub.ProGit, page 17-21. Second edition, 2014.

[4] https ://guides.github.com/activities/hello-world/. Consulté le 29-03-2016.

[5] https ://www.abdn.ac.uk/toolkit/documents/uploads/trello.pdf. Consulté le 30-03-2016.

[6] http ://www.agiliste.fr/introduction-methodes-agiles/. Consulté le 30-03-2016.

[7] https ://fr.atlassian.com/agile/code-reviews. Consulté le 30-03-2016.

[8] http ://www-igm.univ-mlv.fr/ dr/XPOSE2009/TDD/pagesHTML/PresentationTDD.html. Consulté le 31-03-2016.

[9] http ://www.agiledata.org/essays/tdd.html. Consulté le 31-03-2016.

[10] http ://www.ai.univ-paris8.fr/ lysop/bd/seance5-ModeleRel-suite.pdf . Consulté le 05-04-2016.

[11] http ://www.hec.unil.ch/bda/Documents/Poly/Poly0040-0Langages0de.pdf. Consulté le 05-04-2016.

[12] Bernard Lambeau. Lingi2172 - databases - relational algebra. Slides, 2015.

[13] Bernard Lambeau. http ://www.try-alf.org/. Consulté le 09-04-2016.

[14] Raoul-Gabriel Urma , Mario Fusco , Alan Mycroft .Java 8 in action. édition Manning.

[15] Cay S. Horstmann. Java SE 8 for the really impatient édition Pearson.

[16] Claude Delannoy. Programmer en Java 9 ème édition, édition Eyrolles 2014.

[17] http ://blog.ippon.fr/2014/03/17/api-stream-une-nouvelle-facon-de-gerer-les-collections-en-java-8/.Consul le 10-05-2015.

[18] Benoît Gantaume. Junit mise en œuvre pour automatiser les tests en Java édition Eni 2010.

[19] Ramez Elmasari, Shamkant Navathe. Fundamentals of Database Systems Sixth Edition, édition Pearson 2014.

[20] : Benoît Charroux, Aomar Osmani, Yann Thierry-Mieg.UML Pratique de la modélisation 3ème édition, édition Pearson 2010.

[21] Hugh Darwen et Christopher J. Date. The Third Manifesto. 3ème édition, 2014.

[22] https ://fr.wikipedia.org/wiki/Tutorial_D . Consulté le 11-04-2016.

[23] http ://reldb.org/. Consulté le 19-04-2016.

[24] Gerrit van den Geest et Huib van den Brink.HaskellDB A short introduction and tutorial .

[25] https ://en.wikipedia.org/wiki/(D_data_language_specification). Consulté le 20-05-16.

Annexe

Annexe A

Architecture

Dans cette annexe, nous allons tenter d'expliquer le plus simplement possible l'architecture de l'application JAlf, les différents packages, ainsi que les classes qui s'y trouvent.

1 Les packages

Dans cette section, nous allons lister les différents packages de JAlf et en définir leurs objectifs principaux. Certains d'entre eux comme les packages `aggregator` et `constraint` ont été implémentés par nos soins. Notre contribution dans les packages déjà implémentés, consistait à l'ajout d'opérateurs non encore implémentés. Nous rentrerons dans les détails pour certains packages, pour d'autres les explications seront plus succinctes. Mais nous avons déjà fournis les explications de certains d'entre eux lors de l'évocation des différentes notions définies précédemment dans le chapitre 4, page 37.

.

1.1 Le package jalf

C'est le package principal de l'application, là où se situe les classes principales, interfaces et les autres packages. Regardons de plus près ce package et ses différentes classes.

1.1.1 La classe DSL.java

Cette classe contient toutes les méthodes statiques dont nous aurons besoin pour l'exécution de JAlf. Différentes méthodes statiques de la classe `DSL` appellent le constructeur adéquat de la classe concernée. Voici une exemple de la création d'un attribut :

Listing A.1 – methode type dsl

```
public static AttrName attr(String attr) {
    return AttrName.attr(attr);
```

```
}
//constructeur

AttrName(String name) {
    validateNotNull("Parameter 'name' must be non-null.", name);
        this.name = name;
}

public static AttrName attr(String name) {
    return new AttrName(name);
}
```

Un attribut ne possède qu'un nom et est défini par la classe `AttrName`.

La méthode `attr` de la classe `DSL` appelle la méthode statique `attr` se situant dans la classe `AttrName`, qui elle même appelle le constructeur de la classe `AttrName`. Toutes les méthodes situées dans notre classe `DSL` suivent la même démarche à quelques détails près.

1.1.2 Les classes AttrName et AttrList

La classe `AttrList` est une liste contenant des objets de type `AttrName`. C'est donc une liste de noms d'attributs. Ces deux classes concrètes servent dans d'autres classes telles que la classe `Tuple` ou encore `Heading`. La classe `Tuple` se situe dans le package `jalf`. Un tuple est une liste de paires composées d'un attribut et d'une valeur. Un attribut dans JAlf est une instance de la classe `AttrName`.

1.1.3 L'interface Type

L'interface des types de JAlf est implémentée par les classes `RelationType`, `TupleType` et `ScalarType`. Nous avons décris les différents types dans la section 1.2, page 42.

1.1.4 L'interface Relation

Cette interface est l'interface principale de tous types de relation de JAlf, aussi bien les relations dites en mémoire que les relations résultantes d'opérateur. Nous avons mentionné les différentes relations plus en détail dans la section 1.1, page 37.

1.1.5 L'interface Visitor

Cette interface est implémentée par la classe `Compiler`, nous l'avons expliqué dans la section 1.3, page 44 dans JAlf. Notre interface est implémentée par la classe `Optimizer` et `DefaultMapper`. Nous avons vu cela plus en détail dans la section 1.4, page 47.

1.1.6 La classe Tuple

La classe `Tuple` possède un champ `TypeTuple` qui est un type basé sur le `Heading` (entête) de la relation. Le `Heading` de la relation est représenté par un ensemble de paires composées d'un nom d'un attribut et de son type. Le champ `attrs` de la classe `Tuple` est une Map contenant les paires composées d'un nom `AttrName` et de sa valeur.

1.2 Le package jalf.compiler

Ce package comme son nom nous l'indique concerne la compilation des différents opérateurs.

1.2.1 La classe AbstractRelation

Cette classe implémente l'interface `Relation`. La création des différents opérateurs, des clés, des relations et des tuples se fait à l'aide des méthodes statiques dans la classe `DSL`. Ces méthodes statiques appellent des méthodes de la classe `AbstractRelation` qui elles-mêmes appellent le constructeur de l'opérateur concerné ainsi que les constructeurs de `Key` et `Keys`.

1.2.2 La classe Compiler

Cette classe implémente un objet Visitor (`Visitor<Cog>`) qui effectue un parcours en profondeur « depth-first »de l'arbre syntaxique (« AST »). Les opérandes feuilles se compilent elles-mêmes à travers leur méthode `toCog`. Les différents opérateurs se compilent en appelant leur propre méthode `toCog`. Nous avons parlé précédemment du design pattern Visitor dans la section 1.3, page 44.

1.2.3 La classe Cog

La classe Cog est la classe qui compile les différents opérateurs qui ont chacun leurs propres méthodes adéquates. C'est cette classe qui contient les méthodes qui génèrent le résultat des opérations.

1.2.4 La classe BaseCog

Cette classe hérite de de la classe `Cog`, elle possède un champ `streamSupplier` de type `Supplier<Stream<Tuple»`.

1.3 Le package jalf.constraint

Ce package implémente les clés et devra par la suite implémenter d'autres types de contraintes. Une autre contrainte pourrait être par exemple que les valeurs d'un attribut doivent être plus grandes qu'une valeur désignée.

1.3.1 L'interface Constraint

Cette interface n'a qu'une méthode `check(Relation r)` qui doit absolument être redéfinie dans chaque contrainte. La méthode `check` vérifie si la contrainte est respectée pour une relation donnée.

1.3.2 La classe Key

Cette classe représente une clé d'une relation et implémente l'interface `Constraint`. Une clé à un champ `AttrList` qui représente une liste d'attributs. Deux vérifications sont faites lors de la création d'une clé. La première vérifie que ses attributs appartiennent à la relation, et la seconde vérifie que les attributs de la clé désignent des valeurs uniques pour chaque tuple de la relation. Si une relation n'a pas de clé définie par l'utilisateur, JAlf lui en désigne une par défaut. Cette clé par défaut possède comme liste d'attributs, tous les attributs du `Heading` (en-tête) de la relation.

1.3.3 La classe Keys

La classe `Keys` représente une collection de `Key` appartenant à la relation. Pour chaque opérateur, la collection `Keys` est redéfinie pour la relation résultante.

1.4 Le package jalf.optimizer

Ce package implémente l'optimisation des différents opérateurs. Chaque opérateur à son optimisation propre.

1.4.1 La classe DefaultMapper

Cette classe implémente l'interface `Visitor<Optimized<?>` et implémente donc la méthode `visit` pour chaque type de relation reçue en paramètre. Ci-dessous la méthode `visit` pour une relation `Project`. Cette méthode crée une instance de `OptimizedProject`.

Listing A.2 – methode visit

```
@Override
public Optimized<?> visit(Project relation) {
    return new OptimizedProject(optimizer, relation);
}
```

1.4.2 La classe Optimizer

Cette classe implémente `Visitor<Relation>`.

1.4.3 La classe Optimized<R extends Relation>

C'est la classe mère de tous les opérateurs à optimiser, elle définit les optimisations communes à tous les opérateurs.

1.4.4 La classe OptimizedMinus

Cette classe représente l'optimisation de l'opérateur Minus. Elle hérite de la classe `Optimized<Minus>`

1.5 Le package jalf.relation.algebra

Tous les opérateurs se trouvent dans ce package, les opérateurs unaires tels que `Project` ou les opérateurs binaires tels que `Union`. Chaque opérateur a sa propre classe et nous renvoie toujours après compilation une relation résultante.

1.5.1 La classe Operator

Cette classe hérite de `AbstractRelation`. Les Classes `UnaryOperateur` et BinaryOperator héritent de cette dernière. La classe `UnaryOperateur` prend une relation en argument et retourne une relation, tandis que la classe `BinaryOperator` prend deux relations en argument et retourne une seule relation.

1.5.2 L'interface LeafOperand

Cette interface hérite de l'interface `Relation`. Elle doit être implémentée par toutes les relations qui ne sont pas des opérateurs. Elle permet de compiler `Optimizer`. Elle est notamment implémentée par la classe `CsvRelation` et la classe `MemoryRelation`.

1.5.3 La classe Project

La classe `Project` hérite de la classe`UnaryOperator`. Cette classe prend donc en argument une relation et retourne une relation représentant le résultat de la projection. Comme expliqué dans la classe `DSL`, la méthode statique `project` nous renvoie un objet Relation. Elle prend en paramètre une Relation et une liste d'attributs `AttrList`. La méthode `projet` de la classe DSL appelle la méthode `project(AttrList attributes)` dans la classe `AbstractRelation`, qui elle-même appelle le constructeur de la classe `Project`. Le constructeur fait une vérification du type `RelationType`. Cette vérification se déroule en deux temps : premièrement il vérifie que les attributs sur lesquels porte la projection appartiennent bien à la relation, et deuxièmement il change le `Heading`, en-tête de la `RelationType`, en ne retenant que les attributs repris par la projection. Tous ces appels de vérifications et fonctions successives nous retournent un objet `Relation`. A noter que tous les opérateurs suivent le même chemin en créant une instance de

leur propre classe, et en faisant la vérification de `RelationType` adéquate correspondant à leurs propriétés spécifiques.

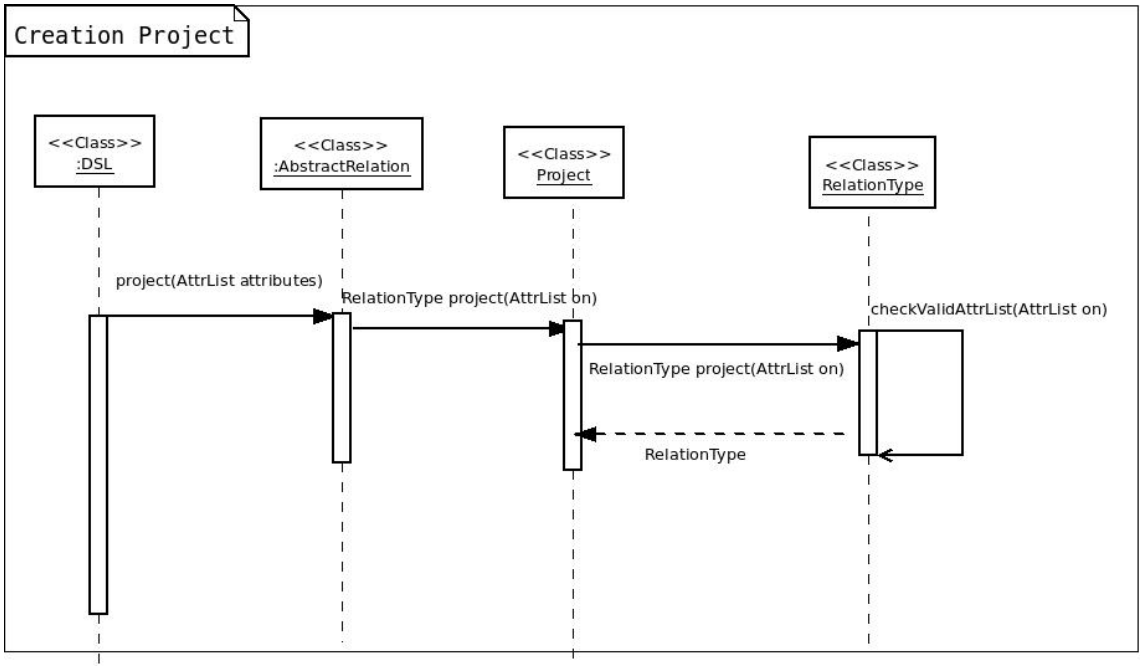


FIGURE A.1 – la création d'une Relation Project.

1.5.4 La classe Union

La classe `Union` hérite de la classe `BinaryOperateur`. Cette classe prend en argument deux relations et retourne une relation qui représente l'union des deux relations. Les deux relations reçues en argument sont appelées *left* et *right*. Le constructeur de la classe `Union` fait une vérification de `RelationType`. Cette vérification est bien sûre différente de celle de la classe `Project`. Ici, nous devons vérifier que les attributs ont des noms identiques et sont du même type, c'est-à-dire « union compatible ».

1.6 Le package jalf.relation.csv

Ce package permet de lire un fichier .cvs et de construire une relation à partir du contenu de ce fichier. Cette relation sera dite en mémoire.

1.6.1 La classe CsvRelation

Cette classe hérite de `AbstractRelation` et implémente l'interface `LeafOperand`. Comme nous avons pu le voir précédemment, l'interface `LeafOperend` permet de compiler des relations qui ne sont pas des relations de type opérateur, mais des relations dites en mémoire. Ici, cette relation en mémoire est construite en lisant un fichier .csv.

1.7 Package jalf.relation.materialized

Ce package permet de construire une `Relation` qui n'est pas un opérateur. Il permet de mettre une relation en mémoire et de la traiter par la suite en utilisant une stream Java 8.

1.7.1 La classe MemoryRelation

Cette classe abstraite hérite de `AbstractRelation` et implémente `LeafOperand`. Elle possède deux méthodes statiques qui appellent le constructeur de la classe `SetMemoryRelation`.

1.7.2 La classe SetMemoryRelation

Cette classe concrète hérite de `MemoryRelation`. Elle représente une relation en mémoire. Elle à un champ `tuples` qui est l'ensemble de tous les tuples (`Tuple`) de la relation. Elle possède un champ `type` qui est de type `RelationType`.

1.8 Package jalf.type

Ce package reprend les différents types rencontrés dans JAlf.

1.8.1 La classe Heading

La classe `Heading` n'est pas utilisée directement, c'est une aide pour la classe `RelationType` et la classe `TupleType`. Le champ `attributes` est une Map contenant des paires composées du nom d'un attribut et de son type. Deux attributs ne peuvent pas avoir le même nom. Les méthodes `project`, `summarize`, `rename` et `join` retournent un `Heading` spécifique à une relation résultante d'opérateur.

1.8.2 La classe HeadingBasedType

La classe `HeadingBasedType` est aussi une classe qui aide à la création d'un objet de type `RelationType` et `TupleType`. Elle contient un champ `heading` de type `Heading`, une méthode `getTypeOf` qui renvoie le type d'un `AttrName`.

1.8.3 La classe RelationType

La classe `RelationType` hérite de `HeadingBasedType` et implémente l'interface `Type<Relation>`. Elle possède un champ `tupletype` de type `TupleType`. Un de ses constructeurs `RelationType(Heading heading)` est appelé indirectement, notamment par la méthode de vérification `TypeCheck`. Comme nous l'avons vu précédemment lors de la création d'un objet de la classe `Project`, le constructeur de la classe `RelationType` est appelé par la méthode `project`, Nous pouvons le constater dans le code ci-dessous.

Listing A.3 – méthode project

```
public RelationType project(AttrList on) {
    checkValidAttrList(on);
    return new RelationType(heading.project(on));
}
```

La classe `RelationType` a un champ de type `Heading` représentant l'en-tête de la relation. Elle possède encore un champ de type `TupleType` représentant le type des tuples contenus dans une relation.

1.8.4 La classe TupleType

La classe `TupleType` hérite de `HeadingBasedType` et implémente l'interface `Type<Tuple>`. Cette classe représente le type d'un tuple. Un `TupleType` peut être défini directement à partir du `Heading`, si celui-ci est défini dans la relation. Par contre si la relation ne définit pas de heading, le type `TupleType` peut-être alors inféré à partir des valeurs contenues dans le tuple.

1.9 Le package jalf.aggregator

Ce package implémente les différentes fonctions d'agrégations utilisées par l'opérateur `Summarize`. Bien que cet opérateur se trouve dans le package `jalf.relation.algebra`, nous le mentionnons ici car il est étroitement lié au package `aggregator`. Ce package à été implémenté par nos soins et nous l'avons déjà mentionné dans le chapitre 4, page 67.

1.9.1 La classe Summarize<T>

Cette classe hérite de `UnaryOperator`, comme toutes classes représentant un opérateur unaire. En plus des champs inhérents aux opérateurs unaires, elle dispose de champs spécifiques. Le champ `aggregator` de type `Aggregator <?>` représente l'agrégat à appliquer à la relation via l'opérateur Summarize. Le champ `by` de type `AttrList` représente la liste d'attributs sur lesquels se portent le regroupement. Et le champ `as` de type `AttrName` définit le nom du nouvel attribut résultant de l'opération `Summarize` et de sa fonction d'agrégation.

1.9.2 L'interface Aggregator<T>

Cette interface doit être implémentée par toutes les fonctions d'agrégations.

1.9.3 La classe Avg

La classe `Avg` implémente l'interface `Aggregator<Double>`. Elle représente la moyenne calculée sur un attribut passé en paramètre `AttrName` de son constructeur. Cette moyenne est calculée pour chaque groupe de tuples renvoyé par `Summarize`.

1.9.4 La classe Count

La classe `Count` implémente l'interface `AggregatorAggregator<Comparable<?»`. La fonction d'agrégation Count compte le nombre de tuples pour chaque sous-groupe de tuples. C'est la seule fonction d'agrégation qui ne prend pas de paramètre `AttrName` dans son constructeur.

1.9.5 La classe Max

La classe `Max` implémente l'interface `Aggregator<Double>`. Elle représente la valeur maximale d'un attribut `AttrName` passé en paramètre de son constructeur. Cette valeur maximale est renvoyée pour chaque groupe de tuples renvoyé par l'opération `Summarize`.

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**