Contributions aux notions d’équivalence relationnelle dans un contexte d’essais

par

Mehri Alsadat Amiri

mémoire présenté au Département d'informatique
en vue de l'obtention du grade de maître ès sciences (M.Sc.)

FACULTÉ DES SCIENCES
UNIVERSITÉ DE SHERBROOKE

Sherbrooke, Québec, Canada, Février 2019

Le 21 février 2019

le jury a accepté le mémoire de Madame Mehri Alsadat Amiri
dans sa version finale.

Membres du jury

Professeur Luc Lavoie

Directeur de recherche

Département d'informatique

Professeur Hélène Pigot

Évaluatrice interne

Département d'informatique

Professeur Marie-Flavie Auclair-Fortier

Présidente rapporteuse

Département d'informatique

Sommaire

Plusieurs systèmes d'aide à l'enseignement ont la capacité d'évaluer et de comparer les travaux des étudiants. Dans un contexte d'enseignement des systèmes de gestion de bases de données relationnelles, une telle comparaison nécessite l'étude des équivalences entre les requêtes et leurs résultats. Plus précisément, les équivalences de requêtes peuvent être vérifiées selon deux modèles, à savoir le modèle SQL et le modèle relationnel. Bien que plusieurs travaux de recherche aient déjà été menés pour la comparaison de requêtes et de résultats basés sur des équivalences syntaxiques et sémantiques, la plupart d'entre eux ne traitent pas les deux modèles.

Dans ce mémoire, nous proposons quatre algorithmes : les deux premiers pour la vérification de l'équivalence sémantique (selon le modèle SQL et selon le modèle relationnel) et les deux autres algorithmes pour la vérification de l'équivalence des résultats (selon le modèle SQL et selon le modèle relationnel). Les deux premiers reposent principalement sur un algorithme proposé par Cohen [1]. L'algorithme appliqué au modèle SQL prend en compte les nuls ainsi que sept paramètres, à savoir le nom des attributs, le nombre d'attributs, le nombre de lignes, le type des attributs, la valeur des attributs, l'ordre des attributs et l'ordre des lignes. L'algorithme appliqué au modèle relationnel prend en compte cinq paramètres, à savoir le nom des attributs, le nombre d'attributs, le nombre de lignes, le type des attributs et la valeur des attributs. Ces algorithmes s'appliquent pour un sous-ensemble important des requêtes SELECT (comprenant notamment les clauses GROUP BY et HAVING). Ils ont été mis en œuvre dans l'outil Themis que nous avons développé. Themis permet en outre de vérifier partiellement la syntaxe des requêtes.

Remerciements

Je tiens à remercier mon directeur de recherche, Luc Lavoie, professeur à l'Université de Sherbrooke, pour ses nombreuses discussions perspicaces durant le développement des idées de ce mémoire et son soutien continu dans mes études et recherches. Sa patience, son enthousiasme, sa motivation et son immense savoir ont été d'une importance incommensurable. Il m'a toujours montré la bonne direction chaque fois qu'il pensait que j'en avais besoin. Ses conseils m'ont aidé tout au long de ma recherche et aussi durant l'écriture de ce mémoire. Je n'aurais pas pu imaginer avoir un meilleur conseiller et mentor pour mes études de maîtrise. Je remercie mes parents pour leur soutien continu et leurs encouragements.

Table des matières

Sommaire iii

Remerciements iv

Table des matières v

Liste des figures viii

Liste des abréviations x

Chapitre 1 Introduction 1

1.1 Contexte 1

1.2 Motivation et objectifs du travail 2

1.2.1 Correction semi-automatisée de travaux des étudiants 2

1.2.2 Tests unitaires de requêtes encapsulées 3

1.2.3 Objectifs 4

1.3 Méthodologie 5

1.3.1 Phase 1 (représentation des requêtes) 5

1.3.2 Phase 2 (équivalence sémantique des requêtes) 6

1.3.3 Phase 3 (équivalence opératoire des résultats) 6

1.3.4 Phase 4 (réalisation du logiciel Themis) 6

1.4 Organisation du mémoire 6

Chapitre 2 Fondements théoriques 8

2.1 Base de données (BD) et le modèle relationnel (MR) 8

2.2 Algèbre relationnelle 10

2.3 Le modèle SQL 12
2.3.1 Présentation 12
2.3.2 Principales différences entre le modèle SQL et le modèle relationnel 12
2.3.3 Les éléments du modèle SQL 13
2.4 Le langage SQL 13
2.4.1 L'instruction SELECT 14
Chapitre 3 Revue de littérature 17
3.1 Choix de l'approche 17
Chapitre 4 Themis 22
4.1 Algorithmes 23
4.1.1 Syntaxe abstraite des requêtes 25
4.1.2 Vérification d'équivalence 26
4.1.3 Comparaison avec d'autres algorithmes 36
4.2 Architecture 37
4.2.1 Interface (ThemisRunner) 37
4.2.2 Vérificateur d'équivalence (ThemisQueryComparison) 38
4.3 Principe de fonctionnement 39
4.3.1 Étape 1 : Configuration de bases de données et des chemins d'accès 39
4.3.2 Étape 2 : Validité syntaxique 40
4.3.3 Étape 3 : Équivalence sémantique 40
4.3.4 Étape 4 : Équivalence de résultats 40
4.3.5 Étape 5 : Résultat final 41
4.4 Analyse syntaxique 41
4.5 Implémentation 43
4.6 Outils 44
4.6.1 ANTLR 44
4.6.2 jOOQ 47
4.7 Comparaison 50

Chapitre 5 Proposition d'une méthodologie d'expérimentation 53
5.1 Environnement de test 53
5.1.1 Le choix du SGBD 54
5.1.2 Le choix du langage de programmation 54
5.1.3 Le choix de l'IDE 54
5.2 Plan de test 55
5.2.1 Exigences à tester 55
5.2.2 Stratégies de test 57
5.2.3 Structure générale d'un test simple 58
5.3 Le scénario des tests unitaires 59
5.4 Le scénario de la correction de travaux pratiques 62
Chapitre 6 Conclusion 64
6.1 Contributions 64
6.2 Travaux futurs 65
Annexe A Définition d'une grammaire Select en SQL avec ANTLR 66
Bibliographie 80

Liste des figures

Figure 1 : Concept de modèle relationnel 9
Figure 2 : Schéma des opérateurs (sélection représentative). 11
Figure 3 : Sous-ensemble considéré de l'instruction SELECT 15
Figure 4 : Architecture la vérification d'équivalences des requêtes 37
Figure 5 : Interaction de l'analyseur lexical avec le parseur 41
Figure 6 : Génération de analyseurs lexical et syntaxique 45
Figure 7 : Fichiers Java générés par ANTLR 45
Figure 8 : Example d'un contexte DSL 48
Figure 9 : Utiliser d'un contexte DSL dans Themis 48
Figure 10: Création de la base de données 59
Figure 11: Création de la table 59
Figure 12: Insertion dans la table 60
Figure 13: Script de procédure stockée en base de données 60
Figure 14: Appeler la procédure stockée 61
Figure 15: Test unitaire réussi 61
Figure 16: assertEquals en test unitaire 62

Liste des tableaux

Tableau 1 : Comparaison des opérateurs de base 15

Tableau 2 : Tableau comparatif des approches « Graph-based » appliquées à SQL 21

Tableau 3 : Comparaison de bibliothèques similaires à jOOQ..50

Tableau 4 : Grille d'interprétation des résultats de validation de requêtes (en présumant que la requête de l'enseignant est correcte)..63

Liste des abréviations

ANSI	***A**merican **N**ational **S**tandards **I**nstitute* – Institut national de normalisation américain.
AST	***A**bstract **S**yntax **T**ree* – Arbre syntaxique abstrait.
ANTLR	***AN**other **T**ool for **L**anguage **R**ecognition* – Framework libre de construction de compilateurs utilisant une analyse LL(*).
BD	Base de données.
BDR	Base de données relationnelle.
CFG	***C**ontext-**f**ree **g**rammars* – grammaire algébrique, aussi appelée hors contexte.
CTE	***C**ommon **T**able **E**xpressions* – Expressions de table communes.
DB2	Système de gestion de bases de données propriétaire d'IBM.
DSL	***D**omain **S**pecific **L**anguage* – Langage dédié.
IBM	***I**nternational **B**usiness **M**achines*.
IDE	***I**ntegrated **D**evelopment **E**nvironment* – Environnement de développement, aussi appelé EDI.
ITS	***I***ntelligent ***T***utoring ***S***ystems – Environnements informatiques pour l'apprentissage humain.
JDBC	***J**ava **D**ata **B**ase **C**onnectivity* – Interface de programmation d'application (API) pour le langage de programmation Java, qui définit comment un client peut accéder à une base de données.
JOOQ	***J**ava **O**bject **O**riented **Q**uerying* – Bibliothèque logicielle de correspondance entre deux objets de bases de données en Java qui implémente le modèle d'enregistrement actif.
LALR	***L**ook-**A**head **L**eft-to-**R**ight Rightmost* – Ensemble de séquences de symboles d'anticipation utiles pour le choix d'une action par un analyseur syntaxique LR.
LCD	***L**angage de **c**ontrôle de **d**onnées* – sous-ensemble de SQL permettant de contrôler l'accès aux données stockées dans une base de données.

LDD	***L**angage de **d**éfinition de **d**onnées* – Sous-ensemble de SQL permettant de définir les structures de données d'une base de données.
LL(*k*)	***L**eftmost derivation parser* – Classe d'analyseurs descendants pour un sous-ensemble de langages hors contexte. L'analyse procède en lisant l'entrée de gauche à droite, en effectuant la dérivation la plus à gauche de la phrase en anticipant sur au plus *k* symboles.
LL(*)	***L**eftmost derivation parser* – Classe d'analyseurs descendants pour un sous-ensemble de langages hors contexte. L'analyse procède en lisant l'entrée de gauche à droite, en effectuant la dérivation la plus à gauche de la phrase en anticipant sur un nombre de symboles non borné a priori (éventuellement en prenant tous les symboles restant jusqu'à la fin de la phrase).
LMD	***L**angage de **m**anipulation de **d**onnées* – Sous-ensemble de SQL permettant de manipuler les données d'une base de données.
LR(*k*)	***R**ightmost derivation parser* – Classe d'analyseurs ascendants pour un sous-ensemble de langages hors contexte. L'analyse procède en lisant l'entrée de gauche à droite, en effectuant l'inverse de la dérivation la plus à droite de la phrase en anticipant sur au plus *k* symboles.
ORM	***O**bject-**R**elational **M**apping* – Mapping objet-relationnel est un type de programme informatique qui se place en interface entre un programme applicatif et une base de données relationnelle pour simuler une collection d'objets.
PLI	***P**rogrammation **l**ogique **i**nductive* – Approche de l'apprentissage automatique qui utilise les techniques de la programmation logique.
POJO	***P**lain **O**ld **J**ava **O**bject* – Classe d'objets Java qui ne joue aucun rôle spécial. Entre autres, elle ne dépend pas d'un *framework* particulier.
PSM	***P**ersistent **S**tored **M**odules* – Sous-ensemble de SQL permettant de définir les procédures et fonctions persistantes.

RA	***R**elational **A**lgebra* – Notation mathématique proche de la notation issue de la théorie des ensembles qui définit des opérations qui peuvent être effectuées sur des relations.
SEQUEL	***S**tructured **E**nglish **QUE**ry **L**anguage.*
SGBD	***S**ystème de **g**estion de **b**ases de **d**onnées* – Logiciel qui prend en charge la structuration, la mise à jour, le stockage et la maintenance d'une base de données.
SGBDR	***S**ystème de **g**estion de **b**ases de **d**onnées **r**elationnelles* – Système servant à stocker, modifier et à partager des données en maintenant leur intégrité relativement à des contraintes.
SQL	***S**tructured **Q**uery **L**anguage* – Langage de requête standardisée permettant de demander des informations à une base de données.
TDD	***T**est **D**riven **D**evelopment* – Développement piloté par les tests.
XP	***Ex**treme **p**rogramming* – Méthode de développement logiciel agile qui vise à produire des logiciels de meilleure qualité en s'appuyant la vérification continue découlant de la réalisation des artéfacts en tandem (paires de développeu.rs.ses).

Chapitre 1

Introduction

1.1 Contexte

Les institutions scolaires utilisent des méthodes d'évaluation traditionnelles. Les étudiants qui sont évalués et les enseignants qui évaluent utilisent généralement un stylo et du papier. Les évaluateurs doivent lire les réponses, calculer les notes à partir des travaux et éventuellement fournir des commentaires écrits. Les étudiants reçoivent habituellement une note de l'évaluateur et ils sont invités à collecter les commentaires individuellement auprès de ce dernier. Cette approche traditionnelle d'évaluation, de correction, de notation et de rétroaction des examens possède des inconvénients en ce qui concerne l'efficacité de la correction et la qualité des commentaires. En outre, les correcteurs humains peuvent commettre plus d'erreurs, être moins constants et prendre plus de temps (principalement quand le nombre d'étudiants est élevé) que des correcteurs automatisés. Afin de pallier ces inconvénients, la coopération entre les chercheurs en éducation et en informatique a conduit à la conception d'un système automatique qui propose une nouvelle méthode d'évaluation et de correction automatisée. Cette méthode étend les limites de la méthode d'évaluation et de correction traditionnelle, qui est effectuée manuellement. Tyquement, un tuteur intelligent (ITS) fournit des instructions immédiates et personnalisées aux étudiants [1]. En général, l'ITS vise à reproduire les avantages démontrés du tutorat individualisé personnalisé. Dans ce contexte, les étudiants auraient donc l'avantage d'accéder à une aide semblable sous la forme d'instructions fournies par un tuteur informatique [2]. Les avantages d'un tel système automatique sont la réduction du temps requis pour la correction des copies et le calcul des

notes ainsi que la diminution marquée des erreurs humaines. Les inconvénients de cette méthode sont l'incapacité de ces systèmes à répondre aux problèmes complexes et la complexité des interactions entre les étudiants et ces systèmes.

La conception de ces systèmes automatiques est le plus souvent uniquement fondée sur la comparaison des résultats (ceux obtenus par l'exécution des requêtes proposées par les étudiants et ceux obtenus par l'exécution des requêtes proposées par le correcteur). La comparaison de la requête de l'étudiant à la requête de référence fournie par le correcteur n'est généralement pas automatisée. Dans le présent mémoire, nous proposons une adaptation d'algorithmes classiques d'évaluation d'équivalence entre requêtes relationnelles qui faciliterait le développement d'un ITS. En particulier, les nouveaux algorithmes sont capables de comparer deux requêtes et leurs résultats, selon le modèle SQL **et** selon le modèle relationnel. Nous proposons en sus une mise en oeuvre expérimentale de ces algorithmes sous la forme d'un logiciel nommé Themis.

Themis peut également être utilisé pour les tests unitaires. Il fournit un mécanisme semi-automatique dans lequel un utilisateur peut prendre une décision en vérifiant le résultat final exprimé sous la forme de Equal/NotEqual/Unknown.

1.2 Motivation et objectifs du travail

De manière générale, les objectifs de ce mémoire peuvent être considérés sous deux angles, à savoir : la validation des requêtes relationnelles (illustrée typiquement par la correction de requêtes dans le contexte d'un cours d'introduction à la programmation relationnelle) et la vérification de requêtes relationnelles (illustrée typiquement par les tests unitaires de requêtes dans un contexte de développement logiciel).

1.2.1 Correction semi-automatisée de travaux des étudiants

L'inconvénient des méthodes de correction et d'évaluation traditionnelles est qu'elles requièrent une intervention humaine ce qui peut engendrer des erreurs. De plus lorsque la quantité d'évaluations est grande, la correction devient plus difficile pour le correcteur.

Dans ce mémoire, nous avons conçu un logiciel qui pourrait être utilisé pour corriger et comparer de manière semi-automatique les requêtes selon deux modèles : le modèle relationnel et le modèle SQL. Les requêtes elles-mêmes sont exprimées à l'aide d'un sous-ensemble représentatif du langage SQL. L'application pourra être utilisée pour soutenir l'enseignement et l'apprentissage dans les cours d'introduction aux bases de données à l'Université de Sherbrooke. Les cours présentés dans les universités nécessitent beaucoup de temps pour corriger les devoirs, et le programme proposé peut être utilisé pour évaluer et noter les devoirs de manière semi-automatique. Pour ce faire, un enseignant ou un assistant saisit la solution du travail effectué et la réponse des étudiants dans le système. Pour que cela fonctionne, les solutions et les réponses correctes doivent être mises dans un fichier et les réponses des étudiants doivent être conservées dans un autre fichier. Lorsque les deux fichiers sont prêts, le programme commence à comparer tous les fichiers de réponses des étudiants avec le fichier clé et un retour immédiat est fourni. Ceci permet de noter les étudiants en très peu de temps. De plus, l'évaluation des réponses est plus précise et la probabilité d'erreur dans la procédure de notation est réduite. Ainsi, la satisfaction des étudiants est améliorée en minimisant les risques d'erreur.

1.2.2 Tests unitaires de requêtes encapsulées

Le deuxième scénario pris en compte par l'application est le test unitaire. Actuellement, les tests unitaires en base de données sont précaires. L'approche utilisée pour prendre en charge les tests unitaires est l'utilisation de requêtes encapsulées dans des procédures stockées sur les SGBD. Les procédures sont appelées selon les opérations à être exécutées. Par exemple, une insertion dans une table est testée en appelant une procédure de recherche qui reçoit comme paramètre le nom d'un attribut clé. La procédure renvoie la réponse « TRUE » avec un message confirmant que le tuple est bel et bien présent dans la table, et dans le cas contraire la réponse « FALSE » avec un message d'erreur. Pour cela, nous avons mis en place une série de procédures permettant de couvrir un nombre significatif d'opérations (INSERT, DELETE, etc.).

De nos jours, la plupart des processus de développement logiciel intègrent la rédaction des tests conjointement à celle des composants logiciels. Les méthodes agiles [3], comme Extreme programming (XP) [4], [5], ou encore le Test Driven Development (TDD) [5] ont remis les tests unitaires au centre de l'activité de programmation. Il existe un risque de régression lors de la refactorisation et de la modification d'une base de code existante, bien que ces processus soient facilités par les différents environnements de développement actuels. Ces risques sont en partie couverts par les tests unitaires réalisés sur les unités logicielles pour détecter des fautes, d'où l'importance de ces tests. Les fonctionnalités du programme sont généralement décomposées en comportements testables discrets qui peuvent être testés en unités individuelles.

Il existe de nombreux outils permettant de réaliser des tests unitaires dans un langage de programmation donné, par exemple : CPPUnit pour C++ [6], CUnit pour C, JSUnit et QUnit pour JavaScript [7], JUnit pour Java [8], Test : Unit pour Ruby [9], ScalaTest pour Scala [10], etc.

Par contre, très peu d'outils permettent de réaliser des tests unitaires dans les langages de bases de données. On peut cependant citer utPLSQL pour PLSQL [11] et tSQLt pour SQL Server [12]. Les outils existants sont limités et ne couvrent pas assez de notions contrairement aux outils des langages de programmation. Aussi, la plupart du temps, les tests unitaires sont réalisés manuellement. Cette activité manuelle est extrêmement lente, onéreuse, parfois difficile et sujette aux erreurs. De plus, les outils existants ne supportent pas les requêtes complexes ni les grammaires des requêtes SELECT.

1.2.3 Objectifs

Au meilleur de nos connaissances, il n'existe pas de solution qui intègre la comparaison automatisée des requêtes et de leurs résultats selon deux modèles (SQL et relationnel) dans les contextes de correction de travaux étudiants et de tests unitaires.

Les principaux objectifs de ce mémoire sont :

1. Développer les notions d'équivalence sémantique des requêtes et d'équivalence de résultats de façon à pouvoir être utilisée tant avec le modèle SQL qu'avec le modèle relationnel.
2. Proposer des algorithmes performants dans chacun des quatre cas.
3. Réaliser une étude expérimentale à l'aide d'un prototype (Themis) utilisable dans un contexte de correction de travaux étudiants et dans un contexte de tests unitaires.

Les requêtes ciblées par le mémoire se limitent aux requêtes de type SELECT. Par contre, nous ciblons non seulement les requêtes simples (incluant les clauses FROM, WHERE, GROUP BY et HAVING), mais aussi les requêtes complexes imbriquées.

Afin de vérifier l'atteinte de ces objectifs, un logiciel doit être développé.

1.3 Méthodologie

Il faut concevoir une méthode paramétrable selon le mode de comparaison (modèle SQL ou modèle relationnel) et la description de l'objet de comparaison (requête ou résultat). La comparaison des requêtes est fondée sur l'équivalence sémantique alors que la comparaison des résultats est fondée sur leur équivalence opératoire.

Les étapes nécessaires à la réalisation de l'objectif sont présentées ci-après.

1.3.1 Phase 1 (représentation des requêtes)

Dans cette phase, les étapes sont les suivantes :

- Définir la grammaire des requêtes et les exprimer à l'aide de ANTLR (voir Définition générale de ANTLR dans le 4.6.1.1).
- Construire un arbre syntaxique abstrait (AST) pour chaque requête à l'aide l'analyseur généré par ANTLR.
- Valider minimalement les requêtes sur la base de leur AST.

1.3.2 Phase 2 (équivalence sémantique des requêtes)

Les étapes pour la réalisation de cette phase sont les suivantes :

- Modifier la notation et les définitions formelles d'équivalence des requêtes définies dans [13] afin de rendre compte du modèle SQL et du modèle relationnel.
- Adapter l'algorithme proposé dans [13] en fonction des modifications de l'étape précédente.

1.3.3 Phase 3 (équivalence opératoire des résultats)

La réalisation de cette phase comprend les étapes suivantes :

- Proposer un algorithme pour comparer les résultats de deux requêtes en tenant compte de cinq paramètres, à savoir le nombre de colonnes, le nombre de lignes, la valeur, le nom de la colonne et le type de colonne.
- Paramétrer cet algorithme en fonction des deux modèles (SQL et relationnel).

1.3.4 Phase 4 (réalisation du logiciel Themis)

La réalisation de cette phase comprend les étapes suivantes :

- Développer le logiciel Themis.
- Développer un plan d'essai expérimental.
- Réaliser le plan d'essai.

1.4 Organisation du mémoire

Le reste du mémoire se présente de la manière suivante : le chapitre 2 présente les fondements théoriques, abordant les points tels que le modèle relationnel, l'algèbre relationnelle et le langage de requête SQL ; le chapitre 3 propose une revue de la littérature ; le chapitre 4 détaille le logiciel Themis, fournissant de plus amples informations sur son architecture, ses principes de fonctionnement, son implémentation, ainsi que l'analyse syntaxique ; le chapitre 5 propose une méthodologie d'essais pour la validation du logiciel

Themis ; et enfin, une synthèse des contributions et des prolongements futurs est proposée au chapitre 6.

Chapitre 2

Fondements théoriques

Le chapitre courant décrit certains concepts théoriques utilisés dans les chapitres subséquents. Dans un premier temps, le modèle relationnel est abordé, ensuite l'algèbre relationnelle et enfin le langage de requête SQL et son modèle spécifique. À noter que l'acronyme SQL désigne, selon le contexte, tantôt le langage lui-même, tantôt le modèle qui sous-tende ce langage. Le modèle SQL est dérivé du modèle relationnel et se voulait, au moment de sa définition, une adaptation pratique prenant en compte les limites technologiques de l'époque. De même, le langage SQL est dérivé de l'algèbre relationnelle et se voulait un vecteur d'expression pratique en regard des problèmes de gestion de données tels que perçus à l'époque de sa définition.

2.1 Base de données (BD) et le modèle relationnel (MR)

Le modèle relationnel a été introduit par E.F. Codd en 1970 à IBM San José [14] et ses concepts découlent de la théorie des ensembles [15]. Les idées de Codd ont été implémentées dans les systèmes de gestion des bases de données relationnelles (SGBDR) au cours des années 1972-1976 [16], [17]. Au fil des années, le langage de manipulation des données SQL[1] est devenu la norme et la grande majorité des produits actuels, qu'ils soient commerciaux (tels Oracle, DB2, SQL Server) ou *open source* (tels MariaDB [18], MySQL [19] et PostgreSQL [20]) en proposent une mise en œuvre plus ou moins complète.

[1] https://www.iso.org/standard/63555.html

Les notions les plus importantes associées au modèle relationnel sont définies dans les paragraphes qui suivent et illustrées dans la Figure 1. Il s'agit notamment des domaines, des relations (souvent appelées tables), attributs (souvent appelés colonnes ou champs), des tuples (ou n-uplet) et des clés [14], [21].

- Un domaine est un ensemble fini de valeurs [22].
- Un type est un sous-ensemble des valeurs d'un domaine, désigné par un nom et souvent soumis à des contraintes.

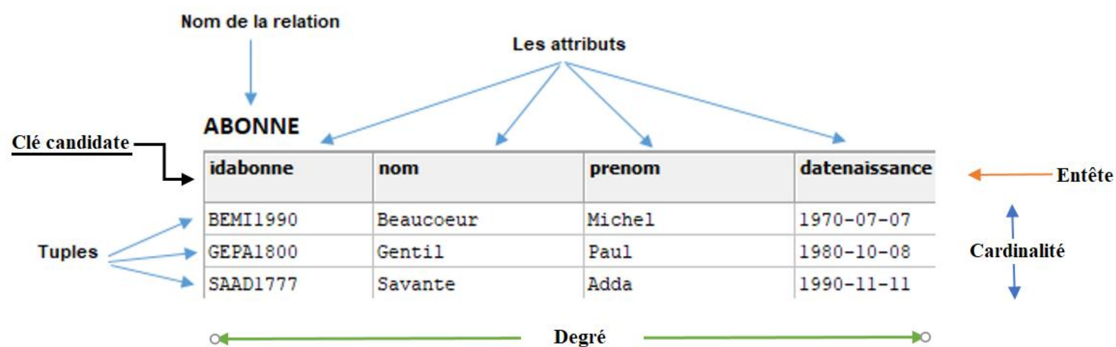

Figure 1 : Concept de modèle relationnel

- Un attribut est caractérisé par un nom et un type.
- Un entête est un ensemble d'attributs dont les noms sont tous distincts.
- Une relation est caractérisée par un entête et sa valeur est un ensemble de tuples de même entête.
- Un tuple est caractérisé par un entête et sa valeur est un ensemble de paires (nom, valeur) où chaque nom d'attribut de l'entête apparait exactement une fois et chaque paire est de type correspondant à l'attribut de même nom dans l'entête.
- Le degré est le nombre d'attributs d'un tuple ou d'une relation.
- La cardinalité est le nombre de tuples d'une relation.
- Deux tuples sont égaux ($=_{MR}$) si et seulement s'ils ont même entête et même valeur.

- Deux relations sont égales ($=_{MR}$) si et seulement si elles ont même entête et même valeur.
- Clés candidates [22].

Une clé candidate d'une relation est un ensemble minimal des attributs de la relation dont les valeurs identifient de façon unique chaque tuple de la relation. La valeur d'une clé candidate est donc distincte pour tous les tuples de la relation. La notion de clé candidate est essentielle dans le modèle relationnel. Toute relation a au moins une clé candidate et peut en avoir plusieurs. Ainsi, il ne peut jamais y avoir deux tuples identiques au sein d'une relation. Les clés candidates d'une relation n'ont pas forcément le même nombre d'attributs. Une clé candidate peut être formée d'un attribut arbitraire qui n'a d'autre objectif que de servir de clé.

Les relations d'une base de données relationnelle sont manipulées par des opérateurs de l'algèbre relationnelle. L'état cohérent de la base de données est défini par un ensemble de contraintes d'intégrité. L'algèbre relationnelle et ses opérateurs permettent de construire de nouvelles relations en combinant des relations préalablement définies. Les paragraphes qui suivent offrent quelques détails d'abord sur l'algèbre relationnelle puis sur SQL.

2.2 Algèbre relationnelle

L'algèbre relationnelle propose un ensemble d'opérations qui permettent de manipuler des *relations*, considérées comme des ensembles de tuples. Les expressions algébriques sont formées en appliquant des opérateurs à des opérandes. Les opérateurs mappent les valeurs tirées du domaine dans d'autres valeurs du domaine. Par conséquent, une expression impliquant des opérateurs et des arguments produit une valeur dans le domaine. Ici, le mot expression est synonyme de requête et la valeur produite est le résultat de la requête.

Il y a eu plusieurs propositions pour les opérateurs de base. Les principaux opérateurs usuels sont illustrés dans le Tableau 1.

L'algèbre relationnelle dispose d'une base mathématique solide, qui permet les développements de nombreux théorèmes intéressants. Parmi les avantages qu'offre l'algèbre relationnelle, on peut mentionner le fondement formel des opérations et le fait qu'elle soit utilisée comme base de l'implémentation et de l'optimisation des requêtes par plusieurs SGBD.

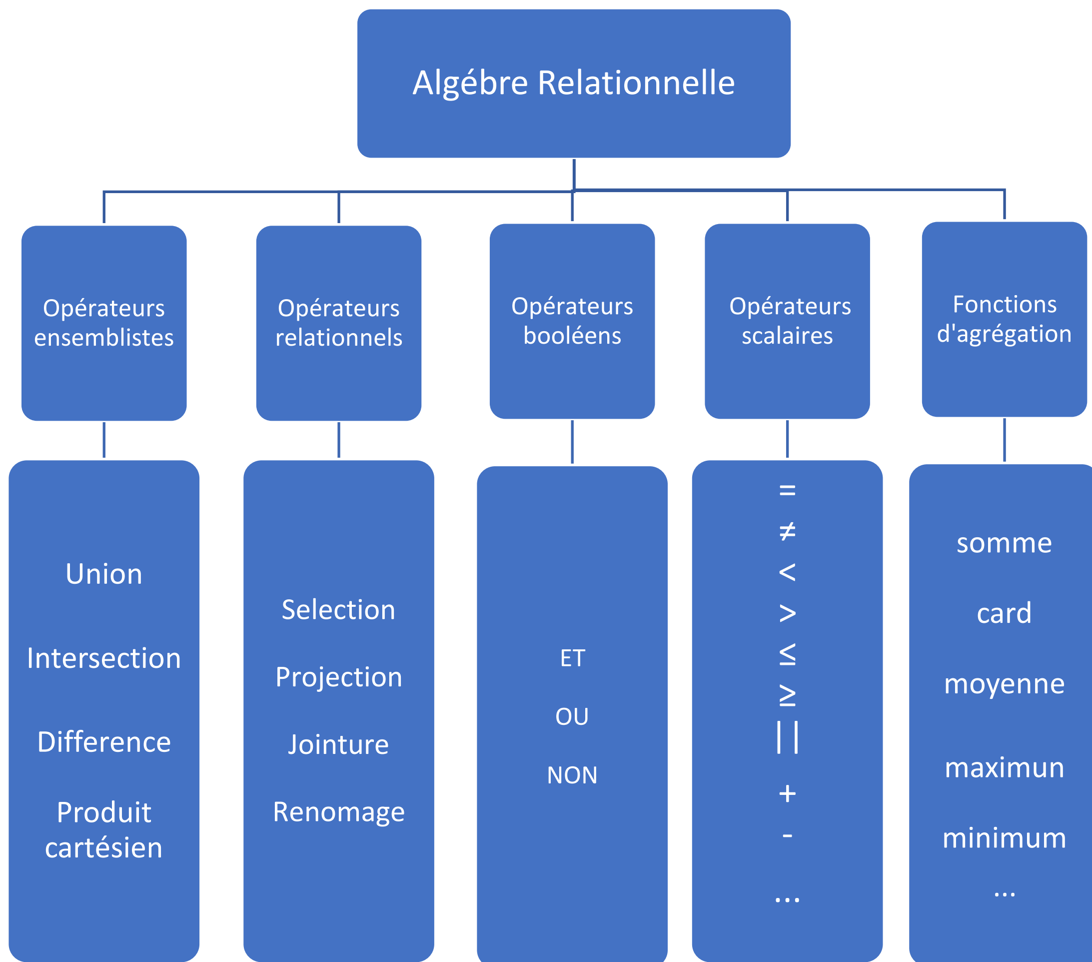

Figure 2 : Schéma des opérateurs (sélection représentative).

2.3 Le modèle SQL

2.3.1 Présentation

Le modèle SQL organise (représente) les données sous forme de relations. Il distingue deux types de relations :

- Les tables sont des relations stockées dans la base de données ; elles peuvent être modifiées à l'aide des instructions insert, delete et update.

- Les vues sont des relations temporaires définies par une expression relationnelle. Ces relations ne sont pas stockées, mais recalculées au besoin par le processeur SQL [23].

2.3.2 Principales différences entre le modèle SQL et le modèle relationnel

Les principales différences du modèle SQL avec le modèle relationnel sont :

- Les tables sont des collections de tuples, pas des ensembles de tuples comme pour les relations.

- Les entêtes de tables et de tuples sont formés par des listes (ordonnées) et non par des ensembles d'attributs comme dans le modèle relationnel.

- Les attributs peuvent être annulables (pas dans le modèle relationnel).

- Conséquemment, le modèle SQL utilise une logique tri-valuée (false, true, unknown) alors que le modèle relationnel utilise une logique bi-valuée (false, true).

En conséquence, la notion d'égalité entre tuples n'est pas là même dans le modèle SQL que dans le modèle relationnel.

La relation d’égalité entre deux tuples t1, t2 ($=_{SQL}$) est (a) true, si et seulement si tous les attributs sont non nuls et que, position par position, l’attribut de t1 a pour valeur la même que celle de t2 (b) unknown, si et seulement si au moins un attribut est nul et que, position par position, lorsque les attributs de t1 et de t2 sont non nuls, ils ont même valeur (c) false, autrement.

2.3.3 Les éléments du modèle SQL

Les principaux concepts du modèle SQL (attribut, domaine, tuple, relation, schéma de la relation, degré, cardinalité, clé candidate, clé primaire et clé étrangère) sont présentés ci-après [23].

2.4 Le langage SQL

À son origine, SQL [21] était appelé SEQUEL (Structured English QUEry Language) et a été implémenté comme l’interface du SYSTEM R, une base de données expérimentale du laboratoire de recherche de IBM. Les efforts conjoints de l’*American National Standards Institute* (ANSI) et de l’Organisation internationale de normalisation (ISO) ont conduit à la version standard de SQL (ANSI 1986), appelé SQL-86 ou SQL1. SQL a évolué au fil des années, incluant de nouvelles fonctionnalités (extensions) et la version la plus récente est SQL:2011.

SQL est généralement présente en trois sous-langages :

- Définition de données (LDD) qui comprend notamment les instructions de création (CREATE), de modification (ALTER) et de suppression (DROP) de tables, de vues, de domaines, de types, de procédures, etc.
- Manipulation de données (LMD) qui comprend notamment les instructions d’évaluation d’expressions relationnelles (SELECT) et d’affection sous diverses normes spécialisées : insertion (INSERT), modification (UDPATE) et suppression (DELETE).

- Contrôle d'accès aux données (LCD) qui comprend les instructions permettant d'accorder (GRANT) et de retirer (REVOKE) des droits d'accès ainsi que d'amorcer (BEGIN) et de mettre fin (COMMIT, ROLL BACK) à une transaction.

Chaque SGBD relationnel possède un compilateur LDD qui a comme fonction de traiter les instructions LDD pour identifier les descriptions des schémas construits et de sauvegarder les descriptions dans le catalogue de SGBD. Le SGBD comporte également un moteur d'exécution des instructions du LMD et du LCD.

Un catalogue d'une instance de base de données est constitué de métadonnées dans lesquelles sont stockées les définitions d'objets de base de données tels que les tables de base, les index, les utilisateurs et groupes d'utilisateurs, les noms et tailles des fichiers, les noms et types de données, les détails de stockage de chaque fichier, les informations de correspondance entre deux objets des schémas et les contraintes. En sus, le catalogue contient plusieurs autres informations qui sont utiles pour des modules de SGBD [24]. Par ailleurs, SQL est un langage de programmation en part entière permettant de définir des procédures et des fonctions au sein de PSM à l'aide d'un langage algorithmique de type procédural.

2.4.1 L'instruction SELECT

L'instruction SELECT, celle qui nous intéresse dans le cadre de ce mémoire, possède le même pouvoir d'expression que l'algèbre relationnelle. Le Tableau 2 illustre la comparaison des opérateurs de base.

En outre, l'instruction SELECT a plusieurs sous-composants (souvent appelées « clauses ») liés au regroupement, à l'ordonnancement, etc. Dans le cadre du mémoire, nous ne nous intéresserons cependant qu'aux seules clauses suivantes : WITH, SELECT, FROM, WHERE, GROUP BY, HAVING. En pratique, ce sous-ensemble permet donc d'exprimer les opérateurs relationnels de restriction, de projection, de jointure (incluant le produit cartésien et l'intersection), mais ne couvre pas l'union ni la différence. Il permet toutefois de traiter les regroupements.

Tableau 1 : Comparaison des opérateurs de base

Algèbre relationnelle	SQL
$(p)\sigma_{(d>e)\wedge(f=g)}$	SELECT * FROM p WHERE d > e AND f = g
$(p)\rho_{\ x1:a,x2:b}$	SELECT x_1 AS a, x_2 AS b FROM p
$\pi_{a,b}(p)$	SELECT a, b FROM p
p ⋈ q (jointure naturelle)	SELECT * FROM p NATURAL JOIN q
$p \times q$ = p ⋈ q (jointure sans attributs partagés)	SELECT * FROM p, q
p ∩ q = p ⋈ q (jointure dont tous les attributs partagés)	SELECT * FROM p INTERSECT SELECT * FROM q
p ∪ q	SELECT * FROM p UNION SELECT * FROM q
p – q	SELECT * FROM p EXCEPT SELECT * FROM q
∧ : symbole de conjonction logique ∨ : symbole de disjonction logique ρ : symbole de renommage σ : symbole de la restriction π : symbole de la projection ⋈ : symbole de la jointure × : symbole du produit cartésien ∩ : symbole de l'intersection ∪ : symbole de l'union – : symbole de la différence	

L'allure générale de l'instruction est donnée par la Figure 3.

WITH *liste de définition-de-sous-table*
SELECT *liste de* nom_de_colonne avec renommage
FROM *expression-de-jointure*
WHERE *condition*
GROUP BY *liste de* nom_de_colonne
HAVING *condition*
ORDER BY *liste de* nom_de_colonne

Figure 3 : Sous-ensemble considéré de l'instruction SELECT

Les six sous-composants que nous utiliserons sont décrits ici :

- WITH: fournit une façon d'écrire les sous-requêtes pour l'utilisation dans une requête SELECT plus étendue. Les sous-requêtes, qui sont souvent appelées des expressions communes de tables (CTE), peuvent être considérées comme la déclaration d'une table temporaire n'existant que pour la requête. Une utilisation de cette fonctionnalité est de découper des requêtes complexes en parties plus simples[2].
- SELECT : correspond à l'opération de projection de l'algèbre relationnelle. Il est utilisé pour lister les attributs du résultat d'une requête.
- FROM: correspond aux jointures (dont le produit cartésien) de l'algèbre relationnelle. Il répertorie les relations analysées dans l'évaluation d'une expression.
- WHERE: correspond au prédicat de sélection de l'algèbre relationnelle. Il s'agit d'un prédicat impliquant des attributs des relations qui apparaissent dans la clause FROM.
- GROUP BY: collecte les lignes restantes dans un groupe pour chaque valeur unique du groupe par expression. Les fonctions d'agrégation spécifiées dans la liste de sélection du SELECT calculent des valeurs de résumé pour chaque groupe.
- HAVING: exclut les lignes des résultats finaux qui ne remplissent pas ses conditions de recherche.

La syntaxe exacte du sous-ensemble traitée dans le cadre de ce mémoire est donnée à l'annexe A.

[2] https://docs.postgresql.fr/8.4/queries-with.html

Chapitre 3

Revue de littérature

3.1 Choix de l'approche

Une approche servant à identifier des fragments de code fonctionnellement équivalents a été proposée dans [25]. Dans cette recherche, l'équivalence fonctionnelle est un cas particulier d'équivalence sémantique qui concerne le comportement d'E/S d'un morceau de code. Dans la même voie, il existe un grand nombre de recherches connexes sur la détection de codes similaires, dont la plupart se concentrent sur la détection de codes syntaxiquement similaires. De telles approches, basées sur la détection de code similaire, peuvent être classées en trois catégories :

- Des approches basées sur des chaînes (*String-based*) qui utilisent des techniques comme les algorithmes de filtrage de chaînes « paramétrées » [26].

- Des approches basées sur une séquence de jetons (*Token-based*) qui est parcourue pour trouver des sous-séquences de jetons dupliquées [27].

- Des approches arborescentes (*Tree-based*) qui analysent un programme en arbres d'analyse ou des arbres syntaxiques abstraits qui sont analysés pour trouver sous-arbres similaires [28]–[30].

D'autres techniques de détection se fondent sur la sémantique des fragments de code et peuvent être classées en deux catégories :

- « *Graph-based* », des approches basées sur un graphe qui représente certaines informations sémantiques comme des graphes de dépendance de programme, qui sont ensuite analysées pour trouver des sous-graphes similaires [31].

- « *Birthmark-based* » des approches fondées sur des marques de naissance utilisées principalement pour détecter le code de vol illégal lorsqu'un programme est pris d'empreintes statiques ; ou pris d'empreintes dynamiques et vérifié par rapport aux empreintes digitales d'autres programmes [32].

L'approche de Cohen [13] utilise des requêtes conjonctives pour les deux sémantiques et définit les équivalences uniquement selon la sémantique de SQL. Notre approche est proche de cette méthode, mais prend en compte l'équivalence algébrique relationnelle.

Dans [33], un modèle logique solide et conceptuellement clair est présent. Il sert de base théorique pour le classement et les techniques basées sur la similarité dans les bases de données relationnelles.

Dans [34], un module de tutorat léger SQL appelé SQL-LMR a été développé. Ce module a mis l'accent sur deux approches : (1) il crée plus de modèles de transformation pour couvrir une catégorie plus large de requêtes et de constructions SQL que le système est ensuite capable de reconnaître et d'évaluer ; (2) il met en œuvre des stratégies sophistiquées de sélection de motifs dans le sélecteur modèle/moteur de décision pour guider plus efficacement le processus de transformation.

Le module SQL Tutor est un système indépendant, autonome, sans base de données en arrière-plan, qui détecte les erreurs syntaxiques et sémantiques et fournit un retour détaillé selon, un moteur de raisonnement basé sur les contraintes où la connaissance du domaine est représentée comme un ensemble de contraintes.

La sémantique des ensembles est étudiée pour les requêtes conjonctives [35]–[38], pour les requêtes conjonctives avec comparaisons [37], [38] et pour les requêtes conjonctives définies par Datalog [40], [41]. Étant donné l'union et la différence dans les requêtes conjonctives, une théorie des tableaux possédant toutes les expressions relationnelles par

rapport à la sélection, à la projection, à la jointure et à l'union a été étendue dans [42]. On présente l'union des tableaux et on montre chaque expression relationnelle sur les opérations mentionnées en les représentant comme une union de tableaux. Peut-être, la sémantique des poches et des collections est étudiée pour des problèmes d'équivalence [43]. L'équivalence des requêtes sous la sémantique de des collections peut être réduite au problème d'équivalence pour les requêtes avec le nombre de fonctions count. Ce problème est étudié pour diverses classes de requêtes [41].

Un langage de requête pour exprimer des requêtes d'agrégat basées sur le champ réel est fourni dans [45]. De plus, dans cette contribution, le problème de l'équivalence des requêtes et le problème subséquent de la convivialité des vues sont abordés. Dans [46], on fournit une caractérisation syntaxique des équivalences entre les requêtes agrégées, y compris les requêtes max, les compteurs-requêtes et les requêtes-sommaires.

Dans [35], la notion de requête conjonctive a été introduite et réduite à une requête appelée « pliage ». Dans cette recherche, chaque requête conjonctive a une requête équivalente minimale unique, qui peut être déterminée par le repliement. Dans [44], l'équivalence des requêtes agrégées pour la classe des requêtes conjonctives avec comparaisons est étudiée sur la base des opérateurs d'agrégation, y compris Min, Max, Count, Count-Distinct et Sum. Il convient de noter que cette classe est limitée aux requêtes SQL avec une clause WHERE consistant en une conjonction de comparaisons, sans clause HAVING et avec une seule fonction d'agrégation. De plus, une condition suffisante est fournie dans [46] pour l'équivalence des requêtes avec l'opérateur count-distinct en termes des requêtes conjonctives sous les deux sémantiques.

Dans [44], l'équivalence des requêtes agrégées pour la classe des requêtes conjonctives avec comparaisons et compte agrégé des opérateurs (Min, Max, Count, Count-Distinct, et Sum) est étudiée. Cette contribution interprète les comparaisons sur un domaine avec un ordre dense (comme les rationnels) ou avec un ordre discret (comme les entiers). De plus, l'équivalence sous sémantique des collections pour les questions conjonctives avec comparaisons, mais sans agrégation, en termes de combinaisons, est caractérisée. En outre, des caractéristiques syntaxiques pour les équivalences entre les requêtes agrégées sont

établies dans [46]. Notamment, la caractérisation de l'équivalence sous la sémantique de collection est un problème ouvert, puisque la sémantique de collection d'abord est considérée dans [43] et [47].

Dans [45], un langage de premier ordre avec des opérateurs d'arithmétique et d'agrégation polynomiaux réels (compte, somme itérative et multiplication) est proposé. Le langage du premier ordre convenait bien à la définition de requêtes agrégées impliquant des fonctions statistiques complexes. Dans cette contribution, une élimination efficace du quantificateur pour les formules avec agrégation est obtenue.

Les approches d'optimisation de requêtes imbriquées ont reçu une attention considérable depuis les années 1980 puisqu'il est observé que l'exécution de requêtes corrélées utilisant la méthode d'itération imbriquée traditionnelle peut être très inefficace. Un algorithme de transformation de requête est développé dans [48] pour réécrire les requêtes imbriquées en requêtes équivalentes et rapides qui peuvent être traitées de manière plus efficace. Les algorithmes de Kim sont étendus à une approche unifiée pour traiter les requêtes contenant des agrégats et des quantificateurs de sous-requêtes imbriquées dans [49]. Une proposition magique d'optimisation de requête imbriquée basée sur la transformation de réécriture des ensembles est faite et implémentée dans le système de base de données Starburst [50]. Dans [51], un traducteur est développé pour transformer les requêtes SQL en algèbre relationnelle étendue avec des fonctions agrégées. Dans [52], une méthode est proposée pour calculer de façon incrémentielle des requêtes imbriquées basées sur l'algorithme de transformation de l'algèbre relationnelle, ce qui rend possible l'utilisation intensive de l'opérateur ensemble de différence. Dans [51], une approche relationnelle imbriquée est proposée pour évaluer les requêtes imbriquées contenant des sous-requêtes non agrégées. Dans cette recherche, l'approche relationnelle imbriquée était basée sur l'algèbre relationnelle imbriquée et l'optimisation de l'algèbre des opérateurs utilisée dans l'approche relationnelle imbriquée était étudiée.

Tableau 2 : Tableau comparatif des approches « Graph-based » appliquées à SQL

Classification	Approche	Avantage	Désavantage
Requêtes conjonctives	• Requêtes sous la sémantique définie • Requêtes sous ensemble et collection sémantique	- Optimise requêtes conjonctives - Détermine les équivalences de requêtes par rapport aux contraintes d'intégrité générales - Détermine les équivalences de requêtes combinant ensemble et collection sémantique	- La combinaison d'un ensemble et d'un ensemble de poches n'a pas été considérée pour déterminer les questions d'équivalence.
Sous-requêtes agrégées	• Les différences structurelles entre les équivalences et les requêtes agrégées par opposition aux requêtes non agrégées • Conjonctif • Requêtes avec des comparaisons et des agrégations • Transformation de requête d'algorithme de Kim à partir de requêtes imbriquées en requêtes plates	- Optimise l'approche relationnelle imbriquée en proposant et en prouvant une série de règles de transformation algébrique	- Les équivalences entre les requêtes agrégées avec une clause HAVING n'ont pas été prises en compte. - Il considère uniquement les requêtes structurées linéairement avec des constructions SELECT et GROUP BY.

Chapitre 4

Themis

Dans la revue de la littérature, nous avons pu constater que les approches proposées pour détecter les codes équivalents sont limitées et ne sont pas très complètes comparativement aux outils traitant des langages de programmation classiques. Le projet Themis a été conçu pour combler en partie cette lacune.

Cet outil offre donc la possibilité de comparer des requêtes soit sur une base purement sémantique, soit sur une base de résultats (en fonction du jeu de données fourni par l'utilisateur). En outre, il peut le faire selon deux modèles : le modèle relationnel et le modèle SQL. Finalement, il couvre deux cas d'utilisation anticipés : la correction de travaux d'étudiants et les tests unitaires. Comme mentionné dans l'introduction, le traitement de deux fonctions d'équivalences, à savoir celle de SQL [14] et celle de la théorie relationnelle, n'est pas pris en charge par les outils existants. Themis possède le grand avantage de permettre la vérification de l'équivalence relationnelle.

Ce chapitre présente l'architecture du logiciel Themis proposé dans ce travail. Ses différentes composantes sont détaillées ainsi que son principe de fonctionnement. Une section du chapitre est aussi réservée pour les aspects pertinents liés avec l'implémentation de Themis.

4.1 Algorithmes

Nous définirons ici deux types d'équivalences entre des expressions relationnelles : l'équivalence de résultats (eq-r) relativement à une base de données (BD) et l'équivalence sémantique (eq-a) relativement à un schéma relationnel (SR). Celles-ci s'appuient sur l'équivalence de valeur (eq-v) entre deux valeurs de relation. La définition de ces équivalences varie selon le modèle relationnel considéré (SQL [relations construites sur la structure de collection [*bag*]] ou MR [relations sur la structure d'ensemble [*set*]]).

Soit SR le schéma relationnel de référence, BD, une base de données conforme à SR, R une relation de SR et v1 et v2 deux valeurs de cette relation,

- eq_{SQL}-v (v1, v2) : si et seulement si tout tuple de v1 est égal ($=_{SQL}$) à un tuple de v2 et présent le même nombre de fois dans v1 et dans v2, et réciproquement (ceci tient compte de l'ordre des attributs [$=_{SQL}$] et du fait que v1 et v2 soient des collections). Remarque importante : le cas unknown est assimilé à false conformément à l'interprétation de la clause WHERE en SQL.
- eq_{MR}-v (v1, v2) : si et seulement si v1 et v2 ont exactement les mêmes tuples (pas de duplication de tuples possible et l'ordre des attributs n'est pas pris en compte).

Soit e1 et e2 deux requêtes bien formées et r1 et r2 les résultats obtenus par l'évaluation de ces requêtes sur BD, respectivement.

- eq_{SQL}-r (e1, e2, BD) : si eq_{SQL}-v (r1, r2) tient.
- eq_{MR}-r (e1, e2, BD) : si eq_{MR}-v (r1, r2) tient.

Il est alors possible de définir l'équivalence sémantique par son corollaire sur toute BD conforme à SR :

- eq_{SQL}-a (e1, e2, SR) : si pour toute BD conforme à SR, eq_{SQL}-r (e1, e2, BD).
- eq_{MR}-a (e1, e2, SR) : si pour toute BD conforme à SR, eq_{MR}-r (e1, e2, BD).

Les équivalences entre des requêtes agrégées disjonctives comprenant la négation des sous-objectifs sont étudiées dans [44] et [46]. Cependant, les équivalences des requêtes

agrégées avec clause HAVING n'ont pas été fournies dans ces études. Contrairement à l'approche proposée dans [44] et [46], tous les types de requêtes sont pris en charge dans notre approche. Une approche similaire pour les équivalences de requêtes agrégées peut être trouvée dans [45], mais la clause HAVING n'a pas été prise en compte dans cette approche. De plus, les équivalences sémantiques entre les résultats n'étaient pas préoccupantes dans [45].

L'idée principale semble être de définir l'équivalence des requêtes en fonction de leurs prédicats associés. En d'autres termes, deux requêtes sont équivalentes si et seulement si les prédicats qui leur sont associés sont équivalents.

Dans cette section, tout d'abord, un algorithme est proposé pour vérifier les syntaxes grammaticales des requêtes. Du point de vue algébrique relationnel, deux algorithmes sont proposés dont l'un vérifie les équivalences sémantiques entre deux requêtes différentes et l'autre détermine si les résultats des requêtes sont sémantiquement équivalents ou non. Du point de vue SQL, un algorithme est introduit pour vérifier les deux équivalences entre deux requêtes et leurs résultats.

Soit $X = \{x_1 \cdots x_n\}$ l'ensemble des attributs sur les domaines finis de $V_1, \ldots, V_n$ tel que $val(x_i) \in V_i$ où $1 \leq i \leq n$. Une requête et un ensemble de requêtes sont désignés par $q(x_{j_1}, \ldots, x_{j_m})$ et Q, respectivement, où $1 \leq j_k \leq n, 1 \leq k \leq m$. Pour plus de commodité, une requête peut être notée par $q(\vec{x})$, où

$$\vec{x} = < x_{j_1}, \ldots, x_{j_m} >$$

Soit R un ensemble de relations. Une requête est définie comme suit :

$$q(\vec{x}) \text{ :- } R(\vec{y}),$$

où $R(\vec{y}) \in R$ et $\vec{y} = \{y_{j_1}, \ldots, y_{j_m}\}$ sont des tuples. Notons que les tuples $\vec{y}$ doivent correspondre aux arités de la relation correspondante. De plus, chaque variable $\vec{x} = < x_{j_1}, \ldots, x_{j_m} >$ est apparue une fois dans $\vec{y} = \{y_{j_1}, \ldots, y_{j_m}\}$.

4.1.1 Syntaxe abstraite des requêtes

Dans le but de réaliser un analyseur de grammaire de requêtes dans l'architecture de Themis, l'Algorithme 1 est proposé. Tout d'abord, pour chaque requête, un arbre de syntaxe abstraite (AST) est construit par ThemisParser. Un arbre syntaxique abstrait représente tous les éléments syntaxiques d'un langage de programmation tel que SQL. L'arbre se concentre sur les règles plutôt que sur des éléments tels que les points-virgules qui terminent les instructions dans le langage. L'arbre est hiérarchique, avec les éléments des instructions de programmation décomposés en parties. Enfin, sur la base des résultats de la comparaison, des requêtes valides ou invalides sont renvoyées à l'utilisateur sous la forme "Success" ou "Failed", respectivement.

Dans [34], le code source SQL est converti en formes de représentation XML de la requête pour analyser les équivalences sémantiques des requêtes. De la même façon, dans notre approche, le code source SQL est converti en ANTLR, qui est un outil java open source à la pointe de la technologie et qui n'est pas sujet aux erreurs.

Algorithme 1 : Vérifier la grammaire de la requête SELECT

Soit $q_1(\vec{x})$, $q_2(\vec{x}) \in Q$ sont deux requêtes
entrée : $q_1(\vec{x})$, $q_2(\vec{x}) \in Q$
sortie : état de la grammaire de la requête (*Success*, *Failed*)
procedure *Chk-Gram*($q_1(\vec{x})$, $q_2(\vec{x})$)
    créer AST($qi(\vec{x})$) // i= 1,2
    if AST ($q_i(\vec{x})$) est variable alors // i=1,2
        return *Success* **;**
    else
        return *Failed***;**
    end if
end procedure

Pour la réalisation de Themis, des algorithmes sont proposés. Ces algorithmes sont considérés pour vérifier la syntaxe des requêtes, en vérifiant les équivalences de requête et de résultat basées sur le modèle SQL, et les équivalences de requête et de résultat basées sur

l'algèbre relationnelle. Avant l'explication de l'algorithme, les notations suivantes doivent être introduites.

4.1.2 Vérification d'équivalence

Le calcul d'équivalence repose sur un cadre commun applicable aux deux modèles étudiés. Il est présenté à la section 4.1.2.1. Les éléments spécifiques au modèle relationnel (4.1.2.2) et au modèle SQL (4.1.2.3) suivent.

4.1.2.1 Cadre commun pour le calcul d'équivalence

Un terme, noté par « *t* », est un atome relationnel positif, un atome relationnel négatif ou une comparaison (entre deux attributs ou entre un attribut et une constante).

Définition 4.1.2.1.1 Une condition, notée L, est une conjonction de termes [13].

Exemple 4.1.2.1.1 On considère la requête suivante,

```
SELECT DISTINCT Province
FROM Suppliers S
JOIN Supplies U ON S.SName = U.SName
JOIN Parts D ON U.SNo = D.SNo
WHERE D.Loc = 'UdeS'
```

Il existe trois termes de comparaison :

t_1 : S.SName = U.SName ;

t_2 : U.SNo = D.SNo ;

t_3 : D.Loc = 'UdeS.

La condition de q_1 est donc $L = t_1 \wedge t_2 \wedge t_3$. ■

Les variables se distinguent selon qu'elles apparaissent dans l'entête de la requête ou non. Les premières sont appelées variables distinguées et les secondes variables non distinguées [13]. Les variables sont également classées en deux types : les variables associées à un attribut dont la table est un ensemble (variables d'ensemble) et celles associées à un attribut dont la table est une collection (variables de collection).

Lors de l'évaluation d'une requête, seules les variables de collection peuvent induire une multiplicité de tuples identiques dans les réponses renvoyées. Techniquement, afin de différencier les variables d'ensemble et les variables de collection, nous spécifions toujours l'ensemble des variables de collection dans chaque condition immédiatement à la droite de la condition.

Définition 4.1.2.1.2 [13] Une requête générale est une expression non récursive de la forme

$$q(\vec{x}) \leftarrow L_1, M_1 \vee ... \vee L_n, M_n,$$

où L_i est une condition et M_i est un ensemble de variables. Nous avons besoin de ce qui suit pour tout $i \leq n$:

- $q(\vec{x})$ est le tuple (ordonné) composé par les variables distinguées ;
- L_i représente les conditions (expressions booléennes) ;
- M_i est le sous-ensemble des variables de collection non distinguées de L_i.

Les variables dans le corps de L_i qui ne sont pas dans $\vec{x}$ ou dans M_i sont les variables d'ensemble de L_i. Nous utilisons $S_i(q)$ pour désigner les variables d'ensemble de L_i et $\bar{S}_i(q)$ pour désigner le reste des variables (c'est-à-dire les variables distinguées $\vec{x}$ et les variables de collection M_i).

Une requête comportant une agrégation est une requête de la forme :

$$q(\vec{x}) \leftarrow L_1, M_1 \vee ... \vee L_n, M_n, \text{ où}$$

- $q(\vec{x})$ est une ligne composée par les attributs distingués et les fonctions d'agrégation ;
- L_i représente les conditions booléennes ;
- M_i est le sous-ensemble des variables de collection non distinguées de L_i.

Les fonctions d'agrégation nous permettent de modifier la granularité des données. Par exemple, lorsqu'on souhaite connaître le nombre exact des fournisseurs, on peut utiliser la fonction COUNT pour synthétiser ce nombre.

Le calcul peut se présenter ainsi dans q_4, q_5 et q_6.

Exemple 4.1.2.1.2 Considérons les relations Suppliers(SName, Province) key {SName}, Supplies(SName, SNo) key {SName, SNo} et Parts(SNo, Loc) key {SNo} et les six requêtes SQL suivantes :

(q_1)

```
SELECT DISTINCT Province
FROM Suppliers S
JOIN Supplies U ON S.SName = U.SName
JOIN Parts D ON U.SNo = D.SNo
WHERE D.Loc = 'UdeS';
```

(q_2)

```
SELECT Province
FROM Suppliers S
JOIN Supplies U ON S.SName = U.SName
JOIN Parts D ON U.SNo = D.SNo
WHERE D.Loc = 'UdeS' or D.SNo= '25';
```

(q_3)

```
SELECT Province
FROM Suppliers S
WHERE exists (
        SELECT *
        FROM Supplies U JOIN Parts D ON U.SNo = D.SNo
        WHERE D.Loc = 'UdeS'
        );
```

(q_4)

```
SELECT Province, count(S.Sname)
FROM Suppliers S
JOIN Supplies U ON S.SName = U.SName
JOIN Parts D ON U.SNo = D.SNo
WHERE D.Loc = 'UdeS'
GROUP BY Province;
```

(q_5)

```
SELECT Province, count(U.Sno)
```

```sql
FROM Suppliers S
JOIN Supplies U ON S.SName = U.SName
JOIN Parts D ON U.SNo = D.SNo
WHERE D.Loc = 'UdeS'
GROUP BY Province;
```

(q_6)

```sql
SELECT Province, count(U.Sno)
FROM Suppliers S
JOIN Supplies U ON S.SName = U.SName
JOIN Parts D ON U.SNo = D.SNo
WHERE D.Loc = 'UdeS';
```

Ces requêtes sont exprimées sous forme de prédicats de la manière suivante :

- $q_1(x_1) \leftarrow$ Suppliers$(x_2, x_1) \wedge$ Supplies$(x_2, x_3) \wedge$ Parts$(x_3,$ 'UdeS'$)$, $\emptyset$.
- $q_2(x_1) \leftarrow$ Suppliers$(x_2, x_1) \wedge$ Supplies$(x_2, x_3) \wedge$ Parts$(x_3,$ 'UdeS'$)$, $\{ x_2, x_3 \}$.
  $\vee$ Suppliers$(x_2, x_1) \wedge$ Supplies$(x_2, x_3) \wedge$ Parts$(25, x_4)$, $\{ x_2, x_3 \}$.
- $q_3(x_1) \leftarrow$ Suppliers$(x_2, x_1) \wedge$ Supplies$(x_2, x_3) \wedge$ Parts$(x_3,$ 'UdeS'$)$, $\{ x_2 \}$.
- $q_4(x_1, x_4) \leftarrow$ Suppliers$(x_2, x_1) \wedge$ Supplies$(x_2, x_3) \wedge$ Parts$(x_3,$ 'UdeS'$)$, $\{ x_2, x_3 \}$.
- $q_5(x_1, x_4) \leftarrow$ Suppliers$(x_2, x_1) \wedge$ Supplies$(x_2, x_3) \wedge$ Parts$(x_3,$ 'UdeS'$)$, $\{ x_2, x_3 \}$.
- $q_6(x_1, x_4) \leftarrow$ Suppliers$(x_2, x_1) \wedge$ Supplies$(x_2, x_3) \wedge$ Parts$(x_3,$ 'UdeS'$)$, $\{ x_2, x_3 \}$.

Observons que q_1 n'a pas de variables de collection, q_2 et q_4 n'ont pas de variables d'ensemble et q_3 est à la fois une variable collection et une variable d'ensemble. Observez également que dans q_3, les variables correspondant aux attributs apparaissant uniquement dans la sous-requête de q_3 (c'est-à-dire les attributs non corrélés de la sous-requête) sont des variables d'ensemble. Le reste des variables non distinguées sont des variables collections.

Exemples i≠2

La forme générale est $q_i(\overrightarrow{x}) \leftarrow L(q_i)$, $M(qi)$ avec $M(q_1) = \emptyset$, $M(q_3) = \{ x_2 \}$, $M(q_4) = \{ x_2, x_3 \}$, $M(q_5) = \{ x_2, x_3 \}$ et $M(q_6) = \{ x_2, x_3 \}$. Il en découle que $S(q_1) = \{ x_2, x_3 \}$, $\bar{S}(q_1) = \{ x_1$

}, $S(q_3)$ = { x_3 }, $\bar{S}(q_3)$ = { x_1, x_2 }, $S(q_4) = \emptyset$, $\bar{S}(q_4)$ = { x_1, x_2, x_3, x_4}, $S(q_5) = \emptyset$, $\bar{S}(q_5)$ = { x_1, x_2, x_3, x_4 }, $S(q_6) = \emptyset$, $\bar{S}(q_6)$ = { x_1, x_2, x_3, x_4 }.

Exemple i=2

La forme générale est $q_2(\vec{x}) \leftarrow L_1(q_2), M(q_2) \vee L_2(q_2), M(q_2)$ avec $M(q_2)$ = { x_2, x_3 }, (q_2) = { x_1, x_2, x_3 }, il en découle $S(q_2)$ = { x_4 }, $\bar{S}(q_2)$ = { x_1, x_2, x_3 }.

La requête q_2 permet d'afficher la province (Province) des fournisseurs (Suppliers) des pièces (Parts) qui sont identifiées UdeS ou pièce numéro 25. ■

Soit D une base de données, $\Gamma_{\bar{S}_i}(q,D)$ est l'ensemble des affectations γ aux variables de de $\bar{S}_i(q)$ satisfaisant le prédicat L_i de q.

Définition 4.2.1.1.3 [13] Le résultat de l'application de q à D sous sémantique combinée, notée $\text{Res}_C(q, D)$, est défini comme suit :

$$\bigcup_{i=1}^{n} \{\{\, \gamma(\vec{x}) \mid \gamma \in \Gamma_{\bar{S}_i}(q, D) \,\}\}, (*)$$

Notez que $Res_C(q, D)$ est une collection de tuples et que même si q est conjonctif, $Res_C(q, D)$ peut contenir plusieurs occurrences du même tuple. Le traitement de ce problème pour les requêtes relationnelles est effectué dans la section **4.1.2.2**.

Exemple 4.1.2.1.3 À partir de l'exemple 4.1.2.1.2, soit D la base de données contenant les tables Parts, Suppliers et Supplies définies comme suit :

Parts := { (1, 'UdeS'), (2, 'UdeS'), (25, 'UdeM') },

Suppliers := { (Alice, 'Québec'), (Bob, 'Québec'), (Jean, 'Ontario') },

Supplies := { (Alice, 1), (Alice, 2), (Bob, 1), (Jean, 25) }.

L'ensemble des affectations satisfaisantes pour $q_1, \ldots, q_6$ est :

Γ := {{ x_2/Alice, x_1/Québec, x_3/1 }, {{ x_2/ Alice, x_1/Québec, x_3/2 }, { x_2/ Jean, x_1/Ontario, x_3/25 }},{ x_2/ Bob, x_1/Québec, x_3/1 }, {{ x_2/Alice, x_1/Québec }, { x_2/ Bob, x_1/Québec }}, {{ x_1/Québec, x_3/1 }, { x_1/Québec, x_3/ 2 }}, {{ x_1/Québec, x_3/1 }}}.

L'ensemble des affectations satisfaisant q_i (i = 1, …, 6) est le suivant:

Γ_{q_1} = {{ x_1/Québec }},

Γ_{q_2} = {{ x_2/Alice, x_1/Québec, x_3/1 }, { x_2/Alice, x_1/Québec, x_3/2 }, { x_2/ Bob, x_1/Québec, x_3/1 }, { x_2/ Jean, x_1/Ontario, x_3/25 }},

Γ_{q_3} = {{ x_2/Alice, x_1/Québec }, { x_2/ Bob, x_1/Québec }},

Γ_{q_4} = {{ x_2/Alice, x_1/Québec }, { x_2/ Bob, x_1/Québec }},

Γ_{q_5} = {{ x_1/Québec, x_3/1 }, { x_1/Québec, x_3/ 2 }},

Γ_{q_6} = {{ x_1/Québec, x_3/1 }}.

Donc,

$Res_C(q_1, D)$ = {{ (Québec) }},

$Res_C(q_2, D)$ = {{ (Québec), (Québec), (Québec), (Ontario) }},

$Res_C(q_3, D)$ = {{ (Québec), (Québec) }},

$Res_C(q_4, D)$ = {{ (Québec), (2) }},

$Res_C(q_5, D)$ = {{ (Québec), (2) }},

$Res_C(q_6, D)$ = {{ (Québec), (1) }}. ■

Nous utilisons $Res_S(q, D)$ et $Res_M(q, D)$ pour désigner le résultat de l'application de q à D sous les sémantiques ensemble et collection, respectivement [13].

- Si q est une requête-ensemble conjonctive, alors $Res_C(q,D) = Res_S(q,D)$.
- Si q est une requête collection, alors $Res_C(q,D) = Res_M(q, D)$.

Formellement, q est contenu dans q' sous une sémantique donnée si, pour toutes les bases de données, les collections de valeurs renvoyées par q sont des sous collections de valeurs renvoyées par q'. Nous écrivons $q \subseteq C\ q'$, $q \subseteq S\ q'$ and $q \subseteq M\ q'$ si q est contenu dans q' sous sémantique combinée, ensemble et collection, respectivement. Une requête combinée est une requête contenant les deux requêtes (ensemble et collection) dans le modèle SQL.

4.1.2.2 Équivalences fondées sur le modèle relationnel

Équivalence sémantique

Algorithme 2.1 – Équivalence sémantique fondée sur le modèle relationnel (eq_{MR}-a)

entrée : $q_1, q_2 \in Q$
sortie : résultats d'équivalence (*Not Equal, Equal*)
procedure $Eq_{MR\text{-}a}(q_1, q_2)$
    if $Res_C(q_1,D) \neq Res_C(q_2,D)$ **then**
        return *Not Equal*
    else
        return *Equal*
    end if
end procedure

Équivalence de résultats

Soit R un ensemble de relations. Une requête select est définie comme suit : $q(\vec{x})$:- $R(\vec{y})$ où $R(\vec{y}) \in R$ et $\overrightarrow{y} = \{y_1, \cdots, y_n\}$ sont des tuples. Notamment, les tuples $\vec{y}$ doivent correspondre aux arités de la relation correspondante. De plus, chaque variable $\vec{x} =< x_1, \cdots, x_m >$ est apparue une fois dans $\vec{y} = \{y_1, \cdots, y_m\}$. Une relation peut être définie par un ensemble contenant tout ses tuples ou un ensemble contenant toutes ses colonnes. Chaque enregistrement de $R(\vec{y})$ est noté par un ensemble $Rec^i(\vec{y})$, où $1 \leq i \leq l$ avec l le nombre d'enregistrements dans $R(\vec{y})$. L'ensemble des enregistrements de $R(\vec{y})$ est noté : $REC(\vec{y}) = \bigcup_{i=1}^{l} Rec^i(\vec{y})$. De plus, les valeurs de chaque colonne de $R(\vec{y})$ sont notées par un ensemble $Col(y_j)$, où $1 \leq j \leq m$ avec m le nombre d'attributs (colonnes). L'ensemble des valeurs des colonnes de $R(\vec{y})$ est noté $COL(\vec{y}) = \bigcup_{j=1}^{m} Col(y_j)$.

```
SELECT DISTINCT Province, U.SNo
FROM Suppliers S
JOIN Supplies U ON S.SName = U.SName
JOIN Parts D ON U.SNo = D.SNo
WHERE D.Loc = 'UdeS';
```

Province	SNo
Québec	1

```
SELECT Province, U.SNo
```

```
FROM Suppliers S
JOIN Supplies U ON S.SName = U.SName
JOIN Parts D ON U.SNo = D.SNo
WHERE D.Loc = 'UdeS';
```

Province	SNo
Québec	1
Québec	2
Québec	3

$Rec^{1}(q_1, D)$ = {Québec,1},

$Rec^{1}(q_2, D)$ = {Québec,1},

$Rec^{2}(q_2, D)$ = {Québec,2},

$Rec^{3}(q_2, D)$ = {Québec,3},

$Col\ (q_1, Province)$ = {Québec},

$Col\ (q_1, SNo)$ = {1},

$Col\ (q_2, Province)$ = {Québec},

$Col\ (q_2, SNo)$ = {1,2,3},

$REC(q_1, D)$ = {{Québec,1}},

$REC(q_2, D)$ = {{Québec,1},{Québec,2},{Québec,3}},

$COL(q_1, D)$ = {{Québec},{1}},

$COL(q_2, D)$ = {{Québec},{1,2,3}}. ■

Le type d'enregistrement est le même pour toutes les requêtes alors que les valeurs et le nombre de tuples ne sont pas les mêmes.

L'**Algorithme 2.2** compare deux requêtes et les résultats aux entrées le modèle relationnel, les algorithmes sont deux sous-ensembles de relations, à savoir $R_1(\vec{y}_1)$, $R_2(\vec{y}_2)$, considérés comme les résultats des requêtes. Si l'ensemble des valeurs de colonne dans $R_1(\vec{y}_1)$ et $R_2(\vec{y}_2)$ et l'ensemble des enregistrements dans $R_1(\vec{y}_1)$, $R_2(\vec{y}_2)$ sont égaux, l'algorithme retourne un message *Equal*. Sinon, il renvoie un message *Not Equal*.

Algorithme 2.2 – Équivalence de résultats fondée sur le modèle relationnel ($\mathbf{eq_{MR}}$-r)

soit $R_1(\vec{y}_1)$ et $R_2(\vec{y}_2)$, les résultats (relations) de q_1 et q_2, respectivement.
soit $REC(\vec{y}_1) = \bigcup_{i=1}^{l} Rec^i(\vec{y}_1)$ et $REC(\vec{y}_2) = \bigcup_{i=1}^{k} Rec^i(\vec{y}_2)$
//*n* et *m* sont les nombres de colonnes dans $R_1(\vec{y}_1)$ et $R_2(\vec{y}_2)$, respectivement.
entrée : $R_1(\vec{y}_1)$, $R_2(\vec{y}_2)$
sortie : résultats d'équivalence (*Not Equal, Equal*)
procedure $Eq_{MR\text{-}r}(R_1(\vec{y}_1), R_2(\vec{y}_2))$
  if $REC(\vec{y}_1) = REC(\vec{y}_2)$ **then**
    return *Equal*
  else
    return *Not Equal*
  end if
end procedure

4.1.2.3 Équivalences basées sur SQL

Équivalence sémantique

L'équivalence sémantique s'intéresse de son côté à ces structures en observant les techniques propres à la construction du sens.

Algorithme 2.3 – Équivalence sémantique fondée sur le modèle SQL ($\mathbf{eq_{SQL}}$-a)

entrée : $q_1, q_2 \in$ Q
sortie : résultat d'équivalence (*Not Equal, Equal*)
procedure $Eq_{SQL\text{-}a}(q_1, q_2)$
  if $Res_C(q_1, D) \neq Res_C(q_2, D)$ **then**
    return *Not Equal*
  else
    return *Equal*
  end if
end procedure

Équivalence de résultats

L'équivalence de résultats s'intéresse aux résultats des requêtes.

Algorithme 2.4 – Équivalence de résultats fondée sur le modèle SQL (eq$_{SQL}$-r)

entrée : $q_1, q_2 \in Q$
soit $R_1(\vec{y}_1)$ et $R_2(\vec{y}_2)$, les résultats (relations) de q_1 et q_2, respectivement.
soit $REC(\vec{y}_1) = \bigcup_{i=1}^{l} Rec^i(\vec{y}_1)$ et $REC(\vec{y}_2) = \bigcup_{i=1}^{k} Rec^i(\vec{y}_2)$
    //l et k sont les nombres d'enregistrements dans $R_1(\vec{y}_1)$ et $R_2(\vec{y}_2)$, respectivement.
soit $COL(\vec{y}_1) = \bigcup_{j=1}^{n} Col(y_j)$ et $COL(\vec{y}_2) = \bigcup_{j=1}^{m} Col(y_j)$
    //n et m sont les nombres de colonnes dans $R_1(\vec{y}_1)$ et $R_2(\vec{y}_2)$, respectivement.
entrée : $R_1(\vec{y}_1), R_2(\vec{y}_2)$
sortie : résultats d'équivalence (*Not Equal, Equal*)
procedure $Eq_{SQL\text{-}r}(R_1(\vec{y}_1), R_2(\vec{y}_2))$
    if $REC(\vec{y}_1) = REC(\vec{y}_2)$ *et* $COL(\vec{y}_1) = COL(\vec{y}_2)$ **then**
        return *Equal*
    else
        return *Not Equal*
    end if
end procedure

Équivalence de résultats des types

Soit $T_1(i,j)$ et $T_2(i,j)$ considérés comme deux matrices de résultats de deux requêtes $q_1(\vec{x})$ et $q_2(\vec{x})$ respectivement. L'hypothèse est que les types d'attributs sont désignés par $\overrightarrow{x.type} = <x_1.type_1, \ldots, x_m.type_m>$. L'**Algorithme 3** compare les requêtes et ses résultats basés sur SQL. Plus précisément, la comparaison implique le nombre d'attributs, les lignes, les valeurs des résultats, les ordres des résultats et le type d'attributs.

Algorithme 3 – Équivalences basées sur SQL

Soit n1 et n2 le nombre d'attributs (colonnes) dans $T_1(i,j)$ et $T_2(i,j)$, respectivement.
Soit m1 et m2 le nombre de lignes dans $T_1(i,j)$ et $T_2(i,j)$, respectivement.
entrée : $T_1(i,j)$, $T_2(i,j)$, $q_1(\vec{x})$, $q_2(\vec{x})$
sortie : equivalence results (*Not Equal*, *Equal*)
procedure *Eq-SQL*($T_1(i,j)$, $T_2(i,j)$, $q_1(\vec{x})$, $q_2(\vec{x})$)
    if n1 = n2 **and** m1= m2 **then**
        if $q_1(\overrightarrow{x.type}) = q_2(\overrightarrow{x.type})$ **then**
            if $T_1(i,j) = T_2(i,j)$ **then**
                return *Equal*
            else
                return *Not Equal*
            end if
        end if
    end if
end procedure

4.1.3 Comparaison avec d'autres algorithmes

L'étude des équivalences entre des requêtes agrégées disjonctives avec des sous-objectifs négatifs a été examinée dans [44] et [46]. Cependant, les équivalences des requêtes agrégées avec la clause HAVING n'étaient pas traitées. Contrairement à l'approche proposée dans [44] et [46], tous les types de requêtes sont pris en charge dans notre approche. Une approche similaire pour les équivalences de requêtes agrégées peut également être trouvée dans [45], mais la clause HAVING n'a pas été prise en compte dans cette approche. De plus, les équivalences sémantiques entre les résultats n'étaient pas présentes dans [45].

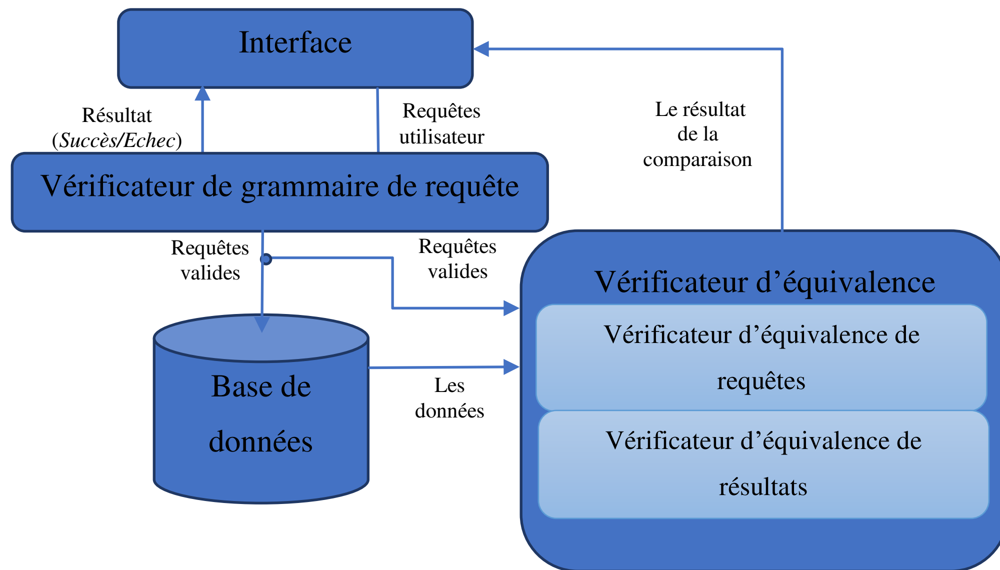

Figure 4 : Architecture la vérification d'équivalences des requêtes

4.2 Architecture

L'architecture de Themis, illustrée par la Figure 4, a trois grandes composantes principales qui sont : ThemisRunner, ThemisQueryComparison et CheckGrammarByANTLR. Chaque composante ainsi que ses différentes fonctions sont détaillées dans les paragraphes qui suivent.

4.2.1 Interface (ThemisRunner)

ThemisRunner est la composante de Themis responsable d'assurer l'interaction avec l'utilisateur. Il s'agit d'une interface entre l'utilisateur et le cœur du logiciel Themis. Tous les paramètres de configuration y sont fournis afin de rendre les comparaisons possibles. Le premier ensemble de paramètres que l'utilisateur du logiciel Themis spécifie est celui des paramètres de connexion aux bases de données auxquelles les requêtes seront connectées. Le deuxième ensemble de paramètres fournis par l'utilisateur est celui des chemins d'accès. Le premier chemin d'accès est celui du fichier où les résultats des comparaisons sont stockés, les

autres chemins d'accès sont ceux des deux fichiers contenant les deux requêtes. La composante ThemisRunner est liée au ThemisProvider, qui est la composante principale de l'interface ThemisQueryComparison.

4.2.2 Vérificateur d'équivalence (ThemisQueryComparison)

ThemisQueryComparison est l'interface entre ThemisRunner et ThemisProvider. Ses fonctions ne se limitent pas seulement à agir comme une simple interface appelant ThemisProvider. ThemisQueryComparison est responsable de l'affichage de tous les paramètres de configuration insérés par l'utilisateur dans le système. L'affichage permet à l'utilisateur de corriger les probables erreurs qui peuvent se glisser lors de la phase de configuration. Enfin, le ThemisQueryComparison assure que tous les détails des opérations sont enregistrés dans le fichier de résultat, spécifié lui aussi durant la phase de configuration.

ThemisProvider est la composante principale du logiciel Themis. Elle est à son tour composée de trois parties principales qui sont CheckGrammarByANTLR, ComparisonRelational et ComparisonSQL.

Ces sous-composantes assurent respectivement la validité syntaxique des requêtes soumises à Themis, la comparaison avec le modèle relationnel et enfin la comparaison avec le modèle SQL. La comparaison relationnelle garantit que les résultats des requêtes soumises possèdent des résultats équivalents. Un total de cinq facteurs sont comparés avant d'établir l'équivalence. Il s'agit du nombre de colonnes, du nombre de lignes, des noms des colonnes, de leurs types et des valeurs renvoyées des bases de données. Pour sa part, la comparaison SQL assure que les requêtes rationnellement équivalentes ont les colonnes et les lignes dans le même ordre. Donc, pour l'équivalence SQL, nous avons un ensemble de sept facteurs.

4.2.2.1 Vérificateur de grammaire de requêtes (CheckGrammarByANTLR)

CheckGrammarByANTLR est la première sous-composante de ThemisProvider. Cette sous-composante est d'une grande importance, car elle assure la validité syntaxique des deux

requêtes. Toute l'analyse syntaxique y est effectuée et aucune exécution des requêtes n'est faite. L'analyse syntaxique est réalisée par :

- ThemisLexer, qui produit les lexèmes à partir de chaînes de caractères des requêtes ;
- JarFileANTLR, à partir duquel les tokens sont produits ;
- ThemisParser, responsable de la production de l'arbre syntaxique (ANTLR).

Toutes les trois étapes ci-dessus sont liées et réalisées par le framework ANTLR. Les détails d'implémentation sont donnés dans la section d'implémentation.

4.2.2.2 Vérificateur d'équivalence de requêtes (ComparisonRelational)

Il s'agit de la sous-composante responsable de la comparaison des deux requêtes selon le modèle relationnel. Il compare les deux requêtes selon le modèle relationnel sémantique avec l'**Algorithme 2.1** et **Algorithme 2.2**.

4.2.2.3 Vérificateur d'équivalence de requêtes (ComparisonSQL)

Ce sous-composant sert à effectuer la comparaison des deux requêtes selon le modèle SQL. Il compare les deux requêtes selon le modèle SQL avec l'**Algorithme 2.3** mentionné.

4.3 Principe de fonctionnement

Le flux de fonctionnement de Themis est divisé en six étapes, tel qu'illustré dans la Figure 4. Chacune des étapes est expliquée dans les paragraphes suivants.

4.3.1 Étape 1 : Configuration de bases de données et des chemins d'accès

La première étape consiste à fournir au logiciel la configuration des bases de données sur lesquelles les requêtes seront exécutées. Pour commencer, l'utilisateur fournit un nom et un mot de passe pour la première base de données et après les mêmes informations sont utilisées pour la deuxième base de données. Ensuite, le chemin d'accès du fichier où les

résultats seront insérés est demandé à l'utilisateur. Enfin, les localisations de deux fichiers contenant les deux requêtes sont fournies (chemins d'accès Q1 et Q2).

4.3.2 Étape 2 : Validité syntaxique

Une fois l'étape précédente conclue, la composante ThemisQueryComparison actionne le ThemisProvider qui lance l'analyse syntaxique par le biais du CheckGrammarByANTLR. Cette sous-composante du ThemisProvider assure que les requêtes Q1 et Q2 sont valides syntaxiquement. Si la syntaxe d'une des requêtes n'est pas valide, Themis arrête l'exécution des comparaisons, et le fichier des résultats contient alors l'information d'échec des opérations (Failure). Le ComparisonRelational est actionné une fois que les requêtes sont syntaxiquement valides.

4.3.3 Étape 3 : Équivalence sémantique

Pour les équivalences sémantiques des requêtes, un algorithme dont les entrées sont deux requêtes a été proposé. Cet algorithme prend en charge tous les types de requêtes et de sorties (EQUAL ou NOT EQUAL) pour les cas d'équivalences ou de requêtes de non équivalences respectivement.

4.3.4 Étape 4 : Équivalence de résultats

Les équivalences de résultats du modèle relationnel et du modèle SQL sont réalisées à cette étape. En tout, dans le modèle relationnel, cinq facteurs sont comparés avant d'établir si les requêtes sont rationnellement équivalentes. Le résultat de l'équivalence est inséré dans le fichier du résultat.

Cette phase constitue la dernière comparaison. Les requêtes sont déclarées équivalentes si elles possèdent les mêmes valeurs pour les sept facteurs d'équivalence du modèle SQL établi. Le résultat de la comparaison est inséré dans le fichier de résultats comme pour la comparaison précédente.

4.3.5 Étape 5 : Résultat final

C'est la dernière étape. Le fichier de résultats est affiché avec tous les détails des comparaisons réalisées. En d'autres termes, le rapport complet des comparaisons est affiché.

4.4 Analyse syntaxique

L'analyse syntaxique consiste à mettre en évidence la structure d'un texte, généralement une phrase écrite dans une langue naturelle. Cette terminologie s'applique aussi pour l'analyse d'un programme informatique.

Lorsqu'une chaîne d'entrée (code source) est donnée à un compilateur, le compilateur la traite en plusieurs phases, en commençant par l'analyse lexicale pour générer le code cible. C'est la première phase d'un compilateur. C'est un processus de prise de chaîne de caractères d'entrée et de production de séquences de tokens, qui peuvent être manipulés plus facilement par l'analyseur. L'interaction de l'analyseur lexical avec le parseur est illustrée dans la Figure 5.

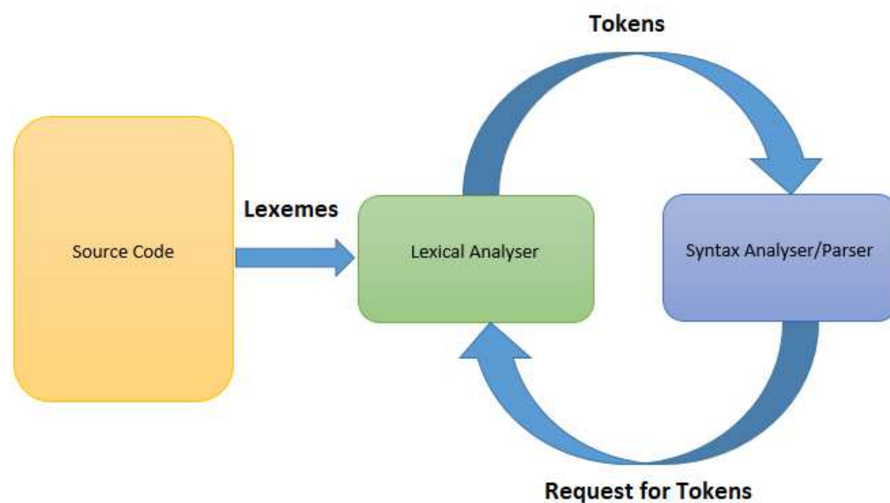

Figure 5 : Interaction de l'analyseur lexical avec le parseur

Les tokens sont des séquences de caractères ayant une signification collective. Il n'y a généralement qu'un petit nombre de tokens pour un langage de programmation : constantes (entier, double, caractère, chaîne, etc.), opérateurs (arithmétique, relationnel, logique), ponctuation et mots réservés. L'analyse lexicale est la première étape du traitement des chaînes de caractères : elle segmente les chaînes de caractères en une suite de mots, également appelés unités lexicales ou lexèmes. Un lexème est une séquence de caractères dans un programme source correspondant à un modèle de token (pattern).

La syntaxe peut être classée dans un document de trois catégories différentes qui sont :

- Une syntaxe qui est seulement pour *l'analyseur lexical* (lexeur).
- Une syntaxe qui est seulement pour *l'analyseur syntaxique* (parseur).
- Une syntaxe avec la capacité pour la combinaison de lexeur et parseur.

À cet effet, l'approche la plus courante est connue sous le nom de grammaire d'arbre dans un document (extension.g4).

La grammaire non contextuelle (ou hors contexte ou grammaire algébrique) est une notion intéressante en informatique formelle. C'est une grammaire formelle dans laquelle chaque règle de production est de la forme : $X \rightarrow \boldsymbol{\alpha}$, où X est un symbole non terminal et $\boldsymbol{\alpha}$ est une chaîne composée de terminaux et de non terminaux.

Les grammaires algébriques sont suffisamment simples pour permettre la création d'analyseurs syntaxiques efficaces. Un analyseur syntaxique ou parseur prend l'entrée d'un analyseur lexical sous la forme de flux de tokens. Le parseur analyse le code source (flux de tokens) par rapport aux règles de production afin de détecter toute erreur dans le code. La sortie de cette phase est un arbre syntaxique.

De cette manière, l'analyseur accomplit deux tâches, c'est-à-dire analyser le code pour rechercher des erreurs et générer un arbre syntaxique en tant que sortie de la phase. Les

analyseurs sont censés analyser le code entier même si certaines erreurs existent dans le programme. Généralement, les analyseurs utilisent des stratégies de récupération d'erreur.

L'écriture des parseurs à la main est fastidieuse et sujette aux erreurs. De nombreuses années des recherches ont été consacrées à étudier comment générer des parseurs efficaces à partir de grammaires de haut niveau. Malgré ces efforts, les générateurs d'analyseurs souffrent encore de problèmes d'expressivité et de facilité d'utilisation. Il était courant de forcer les programmeurs à contourner leurs grammaires pour s'adapter aux contraintes des générateurs d'analyseurs LALR (1) ou LL (1). En revanche, les ordinateurs modernes sont si rapides que l'efficacité du programmeur est maintenant plus importante. En réponse à ce développement, les chercheurs ont développé des stratégies d'analyse non déterministes plus puissantes, mais plus coûteuses, suivant à la fois l'approche « ascendante » (analyse de style LR) et l'approche « descendante » (analyse de style LL).

4.5 Implémentation

L'implémentation de la solution proposée par le projet Themis a été réalisée dans le langage JAVA. Toute l'implémentation se concentre sur la comparaison de requêtes relationnelles et SQL. La vérification de la syntaxe SQL est implémentée avec le framework ANTLR. Au meilleur de nos connaissances, c'est la première fois qu'une implémentation de la comparaison entre les requêtes est réalisée avec ANTLR.

jOOQ a été utilisé comme bibliothèque sous-jacente pour prendre en charge les différentes interactions entre l'application et les bases de données, une fois les connexions établies par l'interface de programmation JDBC (Java DataBase Connectivity)[3]. jOOQ fournit un langage dédié (Domain specific language - DSL) [54] pour construire des requêtes à partir de classes générées depuis un schéma de base de données.

[3] https://fr.wikipedia.org/wiki/Java_Database_Connectivity

Les détails des outils utilisés pour l'implémentation sont donnés dans la section suivante.

4.6 Outils

4.6.1 ANTLR

4.6.1.1 Définition générale

ANTLR [55] est un outil de génération automatique d'analyseur permettant de reconnaître les phrases d'un programme écrit dans un langage donné. L'analyseur est généré à partir d'un fichier grammaire contenant les règles définissant le langage. Pour ce faire, ANTLR définit un méta-langage utilisé pour écrire le fichier grammaire. ANTLR permet les analyses lexicale et syntaxique.

L'analyseur lexical (lexeur) est l'outil qui permet de découper un flux de caractères en un flux de mots du langage (tokens) [56] :

Pour sa part, l'analyse syntaxique consiste à la réception du flux de tokens, à son interprétation et à la production d'un arbre syntaxique abstrait (abstract syntax tree - AST) [57].

L'analyseur syntaxique (parseur) vérifie que l'ensemble des mots issus de l'analyse lexicale (tokens) forme une phrase syntaxiquement correcte. Il n'y a pas de garantie concernant la sémantique de cette phrase. L'entrée de ANTLR est une grammaire hors contexte augmentée de prédicats syntaxiques et d'actions intégrées.

Les prédicats syntaxiques sont donnés sous la forme d'un fragment de grammaire qui doit correspondre à l'entrée suivante. Les prédicats sémantiques sont donnés en tant que code de valeur booléenne arbitraire dans le langage hôte de l'analyseur.

ANTLR est connu comme un générateur puissant qui peut générer des lexeurs, des parseurs d'arborescence, des parseurs et aussi une combinaison de parseurs syntaxiques la Figure 6.

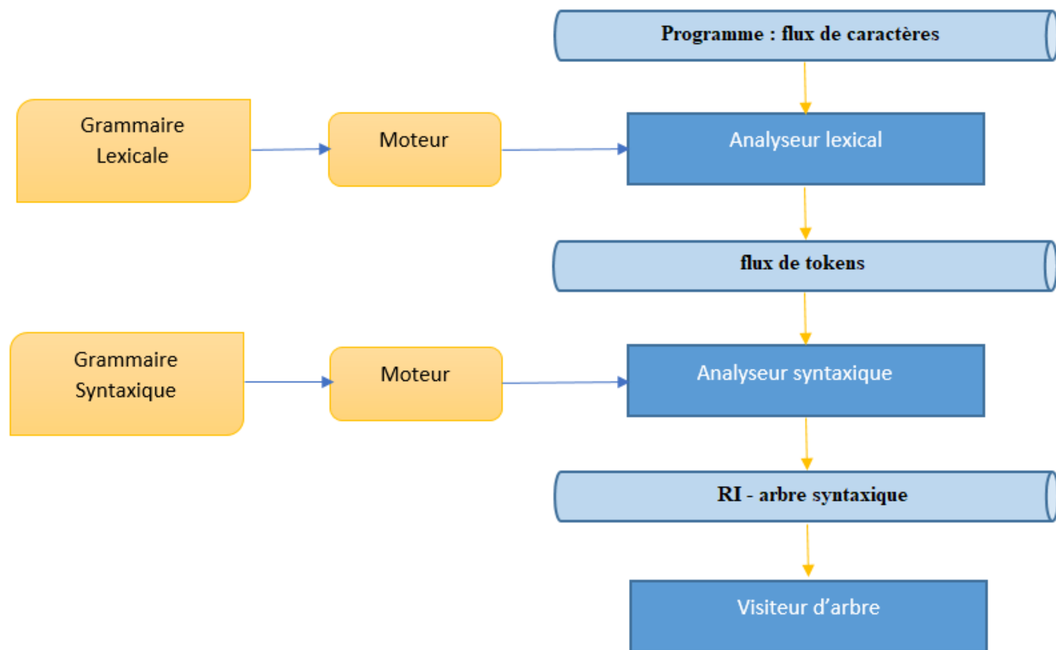

Figure 6 : Génération de analyseurs lexical et syntaxique

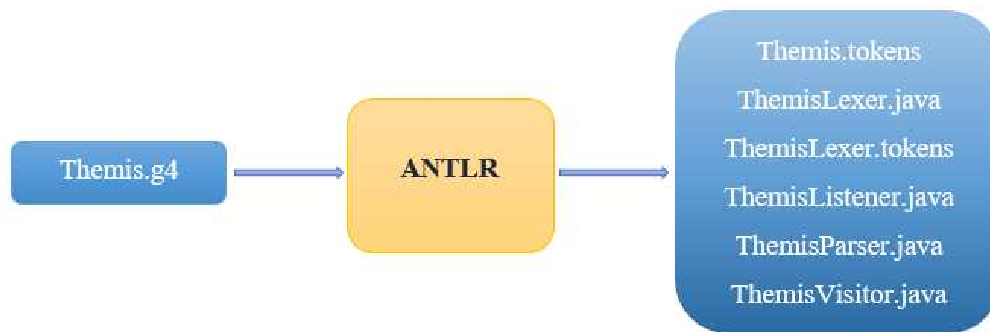

Figure 7 : Fichiers Java générés par ANTLR

ANTLR v4 simplifie l'écriture de règles de grammaire grâce à l'algorithme ALL(*). Il élimine les conflits principaux des grammaires LL et donc des parseurs descendants, tel que montré dans la figure 7.

L'utilisation de ANTLR présente plusieurs avantages, dont certains sont décrits ci-dessous [58].

- Les parseurs utilisant la technique de descente récursive (le type d'un analyseur top-down) sont faciles à analyser, modifier tout en étant très performants. ANTLR génère de tels parseurs en prenant soin de produire un code raisonnablement lisible, ce qui facilite le débogage.
- ANTLR est disponible en open source et a un nombre important d'utilisateurs, ce qui accroît les chances que les bogues soient identifiés et corrigés.
- ANTLR intègre la spécification de l'analyse lexicale et syntaxique. Une spécification lexicale séparée n'est pas nécessaire, car les expressions régulières lexicales (descriptions de tokens) peuvent être placées entre guillemets et utilisées comme références de token normales dans une grammaire ANTLR.
- ANTLR facilite la construction automatique d'arbre de syntaxe abstraite.
- ANTLR génère des analyseurs basés sur la descente récursive afin qu'il y ait une correspondance claire entre la spécification de grammaire et la sortie ANTLR. Par conséquent, il est relativement facile de concevoir et de déboguer une grammaire ANTLR.
- ANTLR dispose de mécanismes à la fois automatiques et manuelles pour la récupération et le signalement des erreurs. Le mécanisme automatique est simple et efficace pour de nombreuses situations d'analyse ; le mécanisme manuel, appelé « gestion des exceptions parseur », simplifie le développement de la gestion des erreurs en cascades.
- ANTLR permet à chaque règle de grammaire d'avoir des paramètres et des valeurs de retour, facilitant le passage des attributs pendant l'analyse. Comme ANTLR convertit chaque règle en une fonction Java dans un analyseur descendant récursif, un paramètre

de règle est simplement un paramètre de fonction. De plus, les règles ANTLR peuvent avoir plusieurs valeurs de retour [57].

ANTLR a de nombreuses autres caractéristiques qui en font un produit plutôt qu'un projet de recherche. ANTLR lui-même est écrit en Java ; sa sortie peut être déboguée avec des débogueurs existants au niveau source et est facilement intégrable dans des environnements de développement de programmes.

4.6.2 jOOQ

4.6.2.1 Définition générale

Java Object Oriented Querying (appelé jOOQ) [54], est un générateur de requêtes et une structure de force qui englobe SQL et qui donne la possibilité de créer une grande variété d'instructions spécifiques à une base de données au moyen d'une API Java. Il s'agit d'un framework de création de requêtes.

jOOQ permet d'accéder à une base de données à travers d'un DSL[4]. Les DSL sont des langages de spécification ou des langages de programmation de très haut niveau, généralement axés sur des domaines d'application spécifiques. Ils sont conçus pour faciliter l'utilisation d'applications complexes utilisées pour résoudre des problèmes spécifiques. Un DSL est différent des langages de programmation généraux, tels que Java et C, qui conviennent à presque tous les domaines. Comme peu d'autres outils similaires, le DSL [54] fourni par jOOQ supporte toutes les propriétés SQL, incluant les ***UNION***, « nested ***SELECT*** », tous les ***JOIN*** et « ***aliasing*** » (p.ex. pour réaliser un self-join). Y sont aussi incluses des fonctionnalités SQL non standard comme le support pour UDT et procédures stockées.

```
DSLContext sourceDB = DSL.using (connexion, SQLDialect.POSTGRES_9_4) ;
```

[4] https://fr.wikipedia.org/wiki/Langage_dédié

Figure 8 : Example d'un contexte DSL

Le contexte DSL est le point initial de génération d'une instruction SQL et l'étape d'entrée de la clé pour le jOOQ DSL. Il nécessite deux éléments de base :

- Une référence à une connexion JDBC
- Un dialecte de base de données

Avec le dialecte de base de données, il peut convertir l'illustration de la requête API Java en une requête SQL. À titre d'exemple, nous pouvons mentionner l'utilisation de PostgreSQL 9.4, pour lequel le DSLContext peut être généré comme le montre la Figure 9.

```
public Result<Record> fetchResult(Connection connection, String queryString) {
    DSLContext firstDSLContext = DSL.using(connection, SQLDialect.POSTGRES_9_4);
    return firstDSLContext.fetch(queryString);
}
```

Figure 9 : Utiliser d'un contexte DSL dans Themis

Dans le projet Themis, cela s'applique tel que montré dans la Figure 9.

4.6.2.2 Avantages d'utilisation

L'utilisation de jOOQ [60] présente de nombreux avantages. Les plus importants sont mentionnés ci-dessous.

- Le DSL fourni par jOOQ est fluide et fait la correspondance entre deux objets un à un avec SQL. Avec jOOQ, il est possible d'utiliser toutes les fonctionnalités d'une base de données.
- Le DSL permet à l'environnement de développement intégré (IDE) de s'autocompléter pendant que le programmeur écrit en SQL.
- jOOQ utilise le système de typage Java pour s'assurer que le SQL qu'il génère est valide.

- L'API jOOQ fait de son mieux pour s'assurer que le bon type de données Java est utilisé pour chaque colonne de la base de données. Par exemple, grâce à JDBC, il s'assure de l'utilisation d'un type de données suffisamment large pour contenir les données dans une colonne numérique.
- jOOQ fournit un convertisseur de type facile à utiliser pour les transformations de type de données courantes, telles que Timestamp ou SQL Date/Time à java.time.ZonedDateTime.
- Lorsque cela est possible, le langage DSL offre certaines extensions à SQL, telles que l'extension InsertSetStep de type instruction d'actualisation qui rend les insertions aussi lisibles que les mises à jour.
- jOOQ ne possède aucune incompatibilité d'impédance. Il existe une fonctionnalité de correspondance entre deux objets qui aide à apparier les ensembles de résultats sur les POJO [60] et fournit une implémentation du modèle d'enregistrement actif pour travailler de la même manière que le font les ORM, mais de manière bien plus claire [61].
- L'API utilise des listes pour les ensembles de résultats, ceci permet l'utilisation facile de stream de Java 8 pour transformer les ensembles de résultats.
- Si le schéma de base de données est modifié, des erreurs de compilation sont obtenues plutôt que des erreurs d'exécution.
- jOOQ fonctionne avec toutes les principales bases de données relationnelles : MS Access 2013, CUBRID, IBM DB2, Apache Derby, Firebird, H2, Hypersonic, Informix, Ingres, MariaDB, MySQL, Oracle, PostgreSQL, SQLite, SQL Server et Sybase[5] [61].
- jOOQ a une double licence : son utilisation est gratuite pour toutes les bases de données open source, mais Oracle, MS SQL Server, Informix, Ingres, DB2 et Sybase, en fournissant également un support.

---

[5] https://fr.wikipedia.org/wiki/Sybase

Tableau 3 : Comparaison de bibliothèques similaires à jOOQ[6]

	Hibernate	EclipseLink JPA	Spring Data	MyBatis	jOOQ
Persistance	Classes Annotation	Classes Annotation	Classes Annotation	Classes Annotation	Classes
Mise en correspondance	[M] : XML	[M] : XML	[M] : XML-Java	[A] [M]	[A] [M]:java
Type SGBD	SQL, NoSql	SQL, NoSql	SQL, NoSql	SQL	SQL
DSL	HEDL	Non	Non	XML,SQL,Java	Java-SQL
Typage fort	Non	Oui	Non	Oui	Oui
SQL injection	Oui	Non	Oui	Oui	Oui
Erreurs statiques	Non	Non	Non	Non	Oui
Procédure stockées	Oui	Oui	Oui	Non	Oui
SQL génération	Oui	Oui	Non	Oui	Oui
SQL dynamique	(annotation)	(Java)	Non	(XML+, Java)	(Java)
Transactions	Oui	Oui	Oui (Spring)	Oui	Non

4.7 Comparaison

Java Object Oriented Querying, communément appelé jOOQ, est une bibliothèque de programmes de correspondance entre deux objets de base de données légère en Java qui exécute le modèle d'enregistrement actif. Son avantage est d'être à la fois social et orienté objet en donnant un dialecte spécifique au domaine pour développer des questions à partir de classes créées à partir d'une construction de base de données.

[6] http://docs.jboss.org/hibernate/orm/5.2/userguide/html_single/Hibernate_User_Guide.html#domain-model
https://wiki.eclipse.org/EclipseLink/FAQ/JPA#What_are_the_Best_Practices_for_using_EclipseLink_JPA.3F
http://blog.mybatis.org
https://blog.jooq.org/tag/mybatis/
https://blog.jooq.org/tag/spring-data/
https://dzone.com/articles/jooq-vs-hibernate-when-choose
https://wiki.eclipse.org/EclipseLink/FAQ/Security
https://wiki.eclipse.org/EclipseLink/Examples/JPA/StoredProcedures
http://wiki.eclipse.org/EclipseLink/Examples/JPA/EMAPI#EntityManager_persist.28.29
http://www.eclipse.org/eclipselink/api/1.2/org/eclipse/persistence/jpa/JpaEntityManager.html
https://wiki.eclipse.org/EclipseLink/Examples/JPA/2.0/Criteria
https://www.petrikainulainen.net/programming/spring-framework/spring-data-jpa-tutorial-creating-database-queries-with-named-queries/

4.7.1.1 Quand choisir quoi ?

Les ORM (Hibernate par exemple) sont conçus pour résoudre des problèmes de persistance des objets en base de données relationnelles. jOOQ, quant à lui, incorpore SQL dans Java. Dans les deux cas, l'accès aux bases de données relationnelles est couvert, et il y a chevauchement dans les fonctionnalités. Hibernate prend également en charge l'interrogation simple, tandis que jOOQ prend également en charge la correspondance entre deux objets simples.

Hibernate est devenu un standard dans l'écosystème Java. De son côté, la force de jOOQ réside dans le fait qu'il prend en charge une grande variété d'instructions, de clauses et de fonctions SQL standards et propres au fournisseur, et qu'il est capable de les simuler dans des dialectes qui ne les prennent pas en charge. Cela fait de jOOQ un bon choix pour les applications qui font un usage intensif de SQL [24].

Avec jOOQ, les exécutions SQL sont le plus souvent réalisées à l'aide d'un arbre de structure de phrases théoriques Java non dynamiques et sécurisées, où les facteurs de rattachement font partie de cet arbre abstrait. Il n'est donc pas possible d'exposer les vulnérabilités d'injection SQL.

jOOQ propose des méthodes utiles pour présenter des chaînes SQL simples dans différents endroits de l'API jOOQ (qui sont commentées en utilisant org.jooq.PlainSQL depuis jOOQ 3.6). L'API de jOOQ permet d'indiquer des valeurs de liens pour une utilisation avec SQL, sans contraindre à le faire. jOOQ traite SQL comme un langage. Grâce aux techniques de conception d'API modernes et uniques, jOOQ intègre le langage SQL en tant que langage spécifique au domaine interne directement dans Java, ce qui permet aux développeurs d'écrire et de lire du code qui ressemble presque au SQL réel. En tant que langage spécifique au domaine interne, jOOQ peut tirer parti du puissant compilateur Java et

des génériques de Java pour les vérifications de type de colonne, les vérifications de type d'expression de valeur de ligne et les vérifications de syntaxe SQL[7].

[7] Pour plus d'informations sur JOOQ, visitez : https://blog.jooq.org/2015/03/24/jooq-vs-hibernate-when-to-choose-which/ et https://www.jooq.org/learn/

Chapitre 5

Proposition d’une méthodologie d’expérimentation

Ce chapitre présente une méthode à suivre pour faire l’expérimentation, la vérification et la validation de Themis en regard des exigences formulées pour les algorithmes d’équivalence proposés au chapitre 3. Pour réaliser l’expérimentation elle-même, il ne resterait donc qu’à construire les jeux de données requis et développer une procédure visant à faciliter la soumission des jeux de données et l’interprétation des résultats.

Le chapitre est divisé comme suit :

- Choix des composants principaux de l’environnement de test.
- Plan de test.
- Étapes principales pour un scénario de tests unitaires.
- Étapes principales pour un scénario de correction des travaux pratiques.

5.1 Environnement de test

Les différents tests présentés ici ont été conçus pour être exécutés sur des postes de travail contemporains. Par exemple :

- Un MacBook Air avec un système d’exploitation macOs High Sierra, un processeur Intel Core i7 cadencé à 2.2 GHz, une mémoire interne de 8 Go de type LPDDR3 cadencée à 1600 MHz DDR3 et un disque dur de 256 Go.

- Un poste HP avec un système d'exploitation Windows 10, un processeur Intel(R) Core i7-6700 cadencé à 3.40 GHz, une mémoire interne de 16.0 Go et un disque dur de 1.5 To.

5.1.1 Le choix du SGBD

PostgreSQL est un puissant système de base de données relationnelle objet open source qui utilise et étend le langage SQL associé à de nombreuses fonctionnalités qui stockent et adaptent en toute sécurité les charges de travail de données les plus complexes.[8]

5.1.2 Le choix du langage de programmation

Pour réaliser l'interface de notre application, on a besoin d'un langage de programmation pour développer. Pour faire notre choix, nos deux principaux critères étaient la puissance du langage de programmation et sa cohérence avec notre SGBD en termes de connectivité, d'où le choix du langage Java comme langage de programmation.

Java est un langage de programmation orienté objet et un environnement d'exécution développé par Sun Microsystems. Il fut présenté officiellement en 1995. Java était à la base un langage pour Internet, pour pouvoir rendre plus dynamiques les pages (tout comme le JavaScript aujourd'hui). Java a beaucoup évolué depuis et est devenu un langage de programmation très puissant à usage général. Java est aujourd'hui officiellement supporté par Oracle, mais plusieurs autres entreprises, comme IBM, l'utilisent.

5.1.3 Le choix de l'IDE

Un langage de programmation possède souvent son ou ses IDE propres. C'est le cas de Java pour lequel on peut utiliser de nombreux IDE, par exemple **intelliJ**, **NetBeans** et **Éclipse**.

Pour le développement de Themis, nous avons utilisé Éclipse (version Neon).

---

[8] https://www.postgresql.org/about/

5.2 Plan de test

Cette section décrit la stratégie de test ainsi que les types de tests et méthodes utilisés; plus spécifiquement, le plan s'intéresse aux aspects suivants: la vérification syntaxique, la vérification de l'équivalence sémantique (MR et SQL) et la vérification de l'équivalence de résultats (MR et SQL).

5.2.1 Exigences à tester

La liste suivante identifie les objets, les cas d'utilisation, les exigences fonctionnelles et les exigences non fonctionnelles qui ont été désignés comme cibles de test. En d'autres termes, cette liste représente ce qui sera testé:

- Connexion à la base de données.
- Vérification syntaxique des requêtes
- Vérification de l'équivalence sémantique des requêtes (MR et SQL).
- Vérification de l'équivalence de résultats des requêtes (MR et SQL.

La méthode `setUp()` sert à la mise en place des tests et elle est exécutée avant les méthodes de test. Elle initialise la connexion à la base de données.

```
@BeforeClass
public static void setUp() throws IOException {
	sqlQueryComparison = new SqlQueryComparison();
	compareProvider = new CompareProvider();
	firstFilePath = "jeux/scripts/T001-comma-d.sql";
	secondFilePath = "jeux/scripts/T002-star-d.sql";
	outPutFilePath = "test/results/allResult.sql";
	firstQueryString = FileUtils.getFileContent(firstFilePath);
	secondQueryString = FileUtils.getFileContent(secondFilePath);
	dbConfig = new DBConfig("postgres", "1234",
"jdbc:postgresql://localhost:5432/Themis");
}
```

La méthode `validateQueryByAntlr()` sert à la validation de la requête par ANTLR. Cette méthode prend en paramètre une requête de type chaine de caractère et vérifie la validation de la requête.

```
protected boolean validateQueryByAntlr(String queryString) {
        boolean isValid = false;
              try {
                     SqlQuery sqlQuery = new SqlQuery(queryString, true);
                     isValid = sqlQuery.isValid();
              } catch (QPCommonException e) {
                     fail(Constants.ANTLR_ERROR);
              }
              return isValid;
       }
```

La méthode `shouldValidateQueryByAntlr()` teste la méthode précédente et assure que le résultat est vrai.

```
@Test
public void shouldValidateQueryByAntlr() {
       assertTrue(validateQueryByAntlr(firstQueryString));
       assertTrue(validateQueryByAntlr(secondQueryString));
    }
```

La méthode `shouldCountResultRow()` permet de tester le nombre des requêtes et assure que le nombre des lignes est égal.

```
@Test
public void shouldCountResultRow() {
       assertEquals(107, getRowCount(firstQueryString));
       assertEquals(107, getRowCount(secondQueryString));
    }
```

5.2.2 Stratégies de test

Themis est utilisable à des fins de vérification (tests unitaires) et de validation (correction de travaux). Dans tous les cas, on suppose qu'au moins une base de données de référence est disponible, initialisée et qu'il est possible de la ramener à son statut initial, en utilisant par exemple des points de reprise (SAVEPOINT et COMMIT). Les stratégies suivantes peuvent être utilisées pour chaque base de données de référence et les résultats cumulés puis synthétisés.

Dans le premier cas (vérification), la stratégie la suivante (on suppose que les paramètres de la méthode à tester et le résultat attendu sont disponibles, par exemple dans un fichier associé au test) :

- Créer un objet de la classe à tester.
- Initialiser la classe.
- Pour chaque cas (déterminé par les paramètres de la méthode à tester) :
    - Exécuter la méthode et conserver le résultat.
    - Comparer le résultat obtenu au résultat prévu (équivalence de résultats).

Dans le deuxième cas (validation), la stratégie est la suivante (on suppose que la requête soumise par l'étudiant est disponible dans un premier fichier, celle de l'enseignant dans un deuxième et le résultat attendu dans un troisième) :

- Comparer sémantiquement la requête de l'étudiant à celle de l'enseignant
- Exécuter la requête de l'étudiant et en conserver le résultat.
- Exécuter la requête de l'étudiant et en conserver le résultat.
- Comparer les deux résultats (équivalence de résultats).

Les stratégies peuvent être utilisées tant avec le modèle RM que le modèle SQL, voir les deux, selon les exigences retenues.

5.2.3 Structure générale d'un test simple

Voici la méthode `compareColumnNames()` et son test :

```
protected boolean compareColumnNames() {
	Result<Record> firstQueryResult = getResult(firstQueryString);
	Result<Record> secondQueryResult = getResult(secondQueryString);
	List<NameType> firstQueryFields =
	compareProvider.findNameAndTypeFromResultFields(firstQueryResult);
	List<NameType> secondQueryFields =
	compareProvider.findNameAndTypeFromResultFields(secondQueryResult);
	return compareProvider.compareColumnsName(firstQueryFields,
secondQueryFields);
}
```

```
@Test
public void shouldCompareColumnNames() {
	assertTrue(compareColumnNames());
    }
```

Première requête	Deuxième requête

`SELECT id`
`FROM plant`
`WHERE NOT EXISTS`
`(`
`    SELECT id`
`    FROM obsetat`
`    WHERE plant.id = obsetat`
`  )`
`ORDER BY id`
`;`
	SELECT id, placette, parcelle, date, note **from** plant **WHERE** id **IN**
	(
	SELECT DISTINCT id
	FROM obsFloraison
	WHERE fleur = **FALSE AND** date >= '2017-06-30');

Remarque

Les évaluations ont été réalisées par l'entremise d'une méthodologie de tests automatisés qui vérifient que les résultats des requêtes sont corrects. Pour ce faire, un serveur PostgreSQL a été utilisé. La plateforme de test se connecte au serveur et les résultats des exécutions de Themis sont comparés avec ceux de la réponse attendue.

5.3 Le scénario des tests unitaires

Les tests unitaires sont destinés à tester une unité de logiciel. Voici un exemple simple avec la méthode suivante :

```
protected boolean validateQueryByAntlr(String queryString) {
	boolean isValid = false;
	try {
		SqlQuery sqlQuery = new SqlQuery(queryString, true);
		isValid = sqlQuery.isValid();
	} catch (QPCommonException e) {
		fail(Constants.ANTLR_ERROR);
	}
	return isValid;
	}
```

Pour tester si cette méthode valide la syntaxe de chaque requête entrée dans Themis ou non. Cependant, voici un exemple d'un test qui est utilisé pour simplifier cette tâche :

```
@Test
public void shouldValidateQueryByAntlr() {
	assertTrue(validateQueryByAntlr(firstQueryString));
	assertTrue(validateQueryByAntlr(secondQueryString));
    }
```

Pour procéder à un test unitaire sur une base de données, il faut disposer des fichiers suivants :

1. x.create.sql

   Son rôle est la création de la structure de la base de données.

```
CREATE DATABASE Themis;
```

Figure 10: Création de la base de données

```
create table test(arbre TEXT NOT NULL, description
TEXT NOT NULL);
```

Figure 11: Création de la table

```
insert into test(arbre, description) values
('Pin','Le pin est la désignation générique des
conifères appartenant au genre Pinus');
```

Figure 12: Insertion dans la table

2. x-init.sql

   Le but de ce script est l'initialisation de la base de données créée par le script x.create.sql. Il peut s'agir, par exemple, d'une alimentation de la base de données, voir Figure 11.

3. x-proc.sql

   Il contient l'ensemble des fonctions à tester. Il existe deux types de fonctions, celles qui retournent un scalaire et celles qui retournent une table. On peut distinguer trois types de paramètres: les paramètres d'entrée (**In**), les paramètres de sortie (**Out**) et les paramètres d'entrée/sortie (**In/Out**). Les paramètres **Out** sont très importants pour le logiciel Themis. Un paramètre de type **In** doit être une valeur (variable), qui ne peut pas être redéfinie ou assignée à un paramètre de type **Out.** Un paramètre de type **Out** doit être une variable assignable qui n'a pas besoin d'être initialisée. Enfin, un paramètre de type **In/Out** doit être une variable assignable.

```
CREATE OR REPLACE FUNCTION verification(arbreId text) RETURNS text AS $$
      DECLARE
            sp_result text;
    BEGIN
      Select description
      into sp_result
      from test as tb where tb.arbre = arbreId;
      RETURN sp_result;
    END;
$$ LANGUAGE plpgsql;
```

Figure 13: Script de procédure stockée en base de données

```
CallableStatement callableStatement = dbConnection.prepareCall(STORED_PROCEDURE)
```

Figure 14: Appeler la procédure stockée

Pour ce qui concerne le test de ces fonctions, on utilise JUnit pour les fonctions scalaires et pour les fonctions qui retournent une table. D'autres classes permettant d'effectuer ce genre de tests sont utilisées pour enrichir JUnit.

4. Script de tests unitaires

À cette étape, est écrit tous les tests insert, update, delete et autres et ont été vérifiés par des procédures encapsulées. Ici tous les tests doivent être vérifiés avec différentes assertions. Par exemple: assertEquals, assertFalse, assertNull et assertSame,…

Avant que l'utilisateur exécute le test, il doit mettre tous ses scripts comme create_db.sql et create-table.sql dans le dossier « SP » qui se trouve dans le chemin '/Themis/resources/'. Après l'exécution du test, Éclipse indique les tests réussis par la couleur verte et les tests qui ne sont pas réussis par la couleur rouge.

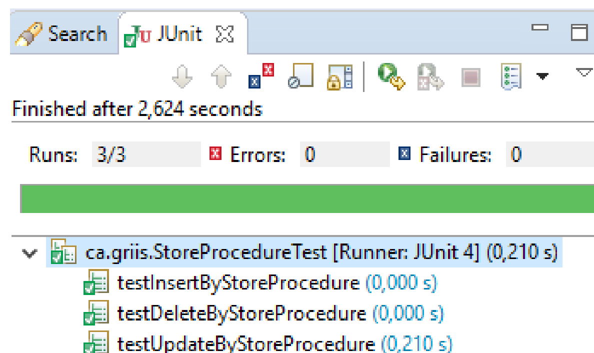

Figure 15: Test unitaire réussi

L'utilisateur doit mettre create.sql dans le bloc @BeforeClass, puis init.sql dans le bloc @Before. De cette façon, Themis va créer une base de données une seule fois, mais va l'initialiser chaque fois qu'on fait appel à un test défini par @Test. Ensuite, le rôle de @After est de créer du code pour fermer la connexion de la base de données. La notation

@AfterClass est exécutée une fois que tous les tests sont finis, il est donc préférable à ce moment de faire supprimer la base de données.

```
assertEquals(map.get(EXPECTED_DESC),callableStatement.getString(1));
```

Figure 16: assertEquals en test unitaire

5.4 Le scénario de la correction de travaux pratiques

Le deuxième scénario est l'utilisation de Themis comme outil de correction de travaux pratiques dans le cours d'introduction aux bases de données. Un enseignant ou un auxiliaire d'enseignement peut se servir de Themis pour effectuer la correction des travaux pratiques des étudiants. Themis offre une méthodologie semi-automatisée, qui permet que l'activité de correction de travaux soit plus rapide que la correction manuelle.

Themis permet de valider (partiellement) les réponses de l'étudiant en combinant l'équivalence sémantique et l'équivalence par résultats. Le correcteur a en sa possession les fichiers contenant le solutionnaire de tous les énoncés. Ces fichiers sont utilisés pour déterminer si les requêtes des étudiants sont sémantiquement équivalentes à celles du solutionnaire ou s'ils livrent les mêmes résultats (équivalence par résultats).

La méthode est la suivante : les solutions et les réponses correctes doivent être mises dans un fichier et les réponses des étudiants doivent être conservées dans un autre fichier. La stratégie de test est ensuite appliquée pour chaque parie de requête. Le tableau 4 résume les conclusions envisageables.

L'algorithme permet de comparer sept paramètres: nom de la colonne, le nombre de colonnes, la ligne de comptage, la valeur, le type, l'ordre des colonnes et l'ordre des lignes pour SQL. La méthode sémantique, quant à elle, porte sur les requêtes elles-mêmes. Dans ce dernier cas, un algorithme est proposé pour comparer les requêtes basées sur le modèle relationnel. Cet algorithme compare cinq paramètres : le nom de colonne, la colonne de comptage, la ligne de compte, la valeur et le type.

Les requêtes sont comparées sémantiquement et par résultat. Dans cette étape, la comparaison des résultats consiste à exécuter des requêtes dans les bases de données et à comparer les résultats ligne par ligne et colonne par colonne. Ils doivent être à 100% égaux. Enfin, les requêtes doivent être comparées de manière significative et la comparaison des résultats est à nouveau nécessaire, mais pas ligne par ligne et colonne par colonne, mais les données doivent être identiques par la suite.

Il faut procéder à ces étapes deux fois : l'une pour le modèle MR et l'autre pour le modèle SQL.

Tableau 4 : Grille d'interprétation des résultats de validation de requêtes (en présumant que la requête de l'enseignant est correcte).

#	Équivalence sémantique	Équivalence de résultats	Conclusion quant la solution soumise par l'étudiant
1	Oui	Oui	Solution correcte
2	Oui	Non	Erreur interne de Thémis
3	Indéterminé	Oui	Solution possiblement correcte
4	Indéterminé	Non	Solution incorrecte
5	Non	Oui	Solution incorrecte
6	Non	Non	Solution incorrecte

Une requête est considérée comme correcte lorsqu'elle est équivalente au solutionnaire et lorsque le résultat obtenu après son exécution est également équivalent à celui du solutionnaire. Dans ce premier cas, la solution soumise par l'étudiant est correcte. Le deuxième cas est indicatif d'une erreur interne de Themis. Le troisième cas nécessite l'examen de la solution soumise par l'étudiant par un correcteur, puisque l'équivalence sémantique n'a pu être déterminée et que la couverture de l'équivalence de résultats n'est pas totale en pratique. Dans les trois derniers cas, la solution soumise est incorrecte. On remarque dans le cas 5 que le jeu n'a pas permis d'illustrer la défaillance de la solution soumise par l'étudiant, c'est une invitation faite à l'enseignant de compléter son jeu de test.

Chapitre 6

Conclusion

Ce chapitre résume les contributions de ce mémoire, puis propose certains prolongements qui pourraient en découler.

6.1 Contributions

Ce mémoire a proposé un mécanisme de comparaison des requêtes relationnelles. Deux fonctions d'équivalences ont été définies : l'équivalence sémantique et l'équivalence par résultats. Ces fonctions ont ensuite été appliquées à deux modèles, le modèle SQL et le modèle relationnel proprement dit et les algorithmes correspondants en ont été dérivés. L'utilisation adéquate de ces algorithmes permet de développer une stratégie vérification des requêtes relationnelle ainsi qu'une stratégie de validation. Une méthodologie d'expérimentation a été proposée selon deux scénarios : les tests unitaires (pour la vérification) et la correction de travaux pratiques (pour la validation). Un premier banc d'essai a été développé sous la forme du logiciel Themis.

À terme, l'utilisation de Themis dans le cadre de la correction de travaux pratiques permettrait de noter les étudiants en moins de temps. De plus, l'évaluation est plus précise et la probabilité d'erreur de notation en serait réduite. Themis permettrait en outre à l'étudiant de voir la différence entre l'interprétation SQL et l'interprétation relationnelle de sa requête.

La logithèque développée dans le cadre Themis permet en outre d'opérationnaliser certains tests unitaires dans un contexte de vérification logicielle.

L'originalité de notre contribution par rapport aux algorithmes d'équivalence préexistants repose sur les éléments suivants :

- Application à deux modèles relationnels distincts (MR et SQL).
- Intégration de l'opérateur d'appartenance ensembliste dans les restrictions (WHERE).
- Intégration de la clause de regroupement (GROUP BY) dans l'instruction SELECT.

6.2 Travaux futurs

Ce mémoire ouvre des avenues pour des travaux futurs dans la comparaison de sémantique des requêtes et de leurs résultats. Des chercheurs s'intéressant à comprendre et à intervenir dans ce domaine pourraient étudier, grâce à Themis, la façon dont d'autres types de requêtes telles que la mise à jour, la suppression, l'intersection, etc. sont équivalentes entre les modèles SQL et relationnel.

Du point de vue pratique, un corpus de jeux d'essai devra être développé afin de mener à terme l'expérimentation. En plus de permettre la vérification empirique poussée des algorithmes, une telle expérimentation permettrait de dégager des cibles d'optimisation.

Pour réaliser l'expérimentation elle-même, il nous semble par ailleurs nécessaire de développer une procédure visant à faciliter la soumission des jeux de données et l'interprétation des résultats.

Themis pourrait être spécialisé et développé pour en faire un IST à part entière. Un premier prototype pourrait être réalisé en greffant une IPM graphique au logiciel actuel.

La logithèque développée pour Temis pourrait être complétée pour offrir un plus grand nombre de prédicats de base pour le développement des tests unitaires. Une logithèque connexe pourrait également être développée pour gérer la réinitialisation de la BD de référence afin de faciliter l'enchainement des tests.

Annexe A

Définition d’une grammaire Select en SQL avec ANTLR

```
// Define a grammar called Themis
grammar Themis;

@header { }

@members { }

/*
* Parser Rules
*/

/*
==========================================================================================
 SQL statement (Start Symbol)
==========================================================================================
*/
sql
: statement (SEMI_COLON)? EOF
;

statement: data_statement| data_change_statement| schema_statement| index_statement;
data_statement: query_expression;
data_change_statement: insert_statement;
schema_statement: create_table_statement;

index_statement
: CREATE (u=UNIQUE)? INDEX n=identifier ON t=table_name (m=method_specifier)?
LEFT_PAREN s=sort_specifier_list RIGHT_PAREN p=param_clause?;

create_table_statement
: CREATE EXTERNAL TABLE table_name table_elements USING file_type=identifier
(param_clause)? (table_partitioning_clauses)? (LOCATION path=Character_String_Literal)
| CREATE TABLE table_name table_elements (USING file_type=identifier)?
(param_clause)? (table_partitioning_clauses)? (AS query_expression)?
 | CREATE TABLE table_name (USING file_type=identifier)?
(param_clause)? (table_partitioning_clauses)? AS query_expression;

table_elements: LEFT_PAREN field_element (COMMA field_element)* RIGHT_PAREN;
field_element: name=identifier field_type;
field_type: data_type;
param_clause: WITH LEFT_PAREN param (COMMA param)* RIGHT_PAREN;
param:key=Character_String_Literal EQUAL value=numeric_value_expression;
method_specifier:USING m=identifier;
table_space_specifier: TABLESPACE table_space_name;
table_space_name: identifier;
```

```
/*
==========================================================================================
 <token and separator>
Specifying lexical units (tokens and separators) that participate in SQL language
==========================================================================================
*/

Identifier: Identifier | nonreserved_keywords;

nonreserved_keywords
: AVG | BETWEEN | BY | CENTURY | CHARACTER | COALESCE | COLLECT | COUNT | COLUMN | DAY | DEC | ECADE | DOW |
DOY | EPOCH | EVERY | EXISTS | EXTERNAL | EXTRACT | FILTER | FIRST | FORMAT | FUSION | ROUPING | HASH | INDEX |
INSERT | INTERSECTION | ISODOW | ISOYEAR | LAST | LESS | LIST | LOCATION | MAX | MAXVALUE | MICROSECONDS |
MILLENNIUM | MILLISECONDS | MIN | MINUTE | MONTH | NATIONAL | NULLIF | PRECISION | RANGE | REGEXP | QUARTER
| RLIKE | SECOND | SET | SIMILAR | STDDEV_POP | STDDEV_SAMP | SUM | TABLESPACE | THAN | TIMEZONE |
TIMEZONE_HOUR | TIMEZONE_MINUTE | TRIM | TO | UNKNOWN | VALUES | VAR_POP | VAR_SAMP | VARYING | WEEK |
YEAR | ZONE | BIGINT | BIT | BLOB | BOOL| BOOLEAN | BYTEA | CHAR | DATE | DECIMAL | DOUBLE | FLOAT | FLOAT4 |
FLOAT8 | INET4 | INT | INT1 | INT2| INT4| INT8 | INTEGER | NCHAR | NUMERIC | NVARCHAR | SMALLINT | REAL | TEXT |
TIME | TIMESTAMP | TIMESTAMPTZ | TIMETZ | TINYINT | VARBINARY | VARBIT | VARCHAR;

 /*
==========================================================================================
<literal>
==========================================================================================
*/
unsigned_literal: unsigned_numeric_literal | general_literal;
general_literal: Character_String_Literal | datetime_literal | boolean_literal;
datetime_literal: timestamp_literal | time_literal | date_literal;
time_literal: TIME time_string=Character_String_Literal;
timestamp_literal: TIMESTAMP timestamp_string=Character_String_Literal;
date_literal: DATE date_string=Character_String_Literal;
boolean_literal: TRUE | FALSE | UNKNOWN;

 /*
==========================================================================================
<data types>
==========================================================================================
*/

data_type: predefined_type;

predefined_type: character_string_type | national_character_string_type | binary_large_object_string_type | numeric_type | boolean_type |
datetime_type | bit_type | binary_type;

character_string_type: CHARACTER type_length? | CHAR type_length? | CHARACTER VARYING type_length? | CHAR VARYING
type_length? | VARCHAR type_length? | TEXT;

type_length: LEFT_PAREN NUMBER RIGHT_PAREN;

national_character_string_type: NATIONAL CHARACTER type_length?
 | NATIONAL CHAR type_length? | NCHAR type_length? | NATIONAL CHARACTER VARYING type_length? | NATIONAL CHAR
VARYING type_length? | NCHAR VARYING type_length? | NVARCHAR type_length?;

binary_large_object_string_type: BLOB type_length? | BYTEA type_length?
;numeric_type: exact_numeric_type | approximate_numeric_type;

exact_numeric_type: NUMERIC (precision_param)? | DECIMAL (precision_param)? | DEC (precision_param)? | INT1 | TINYINT | INT2
| SMALLINT | INT4 | INT | INTEGER | INT8 | BIGINT;

approximate_numeric_type: FLOAT (precision_param)? | FLOAT4 | REAL | FLOAT8 | DOUBLE | DOUBLE PRECISION;
```

```
precision_param: LEFT_PAREN precision=NUMBER RIGHT_PAREN | LEFT_PAREN precision=NUMBER COMMA scale=NUMBER
RIGHT_PAREN;

boolean_type: BOOLEAN | BOOL;
datetime_type: DATE | TIME | TIME WITH TIME ZONE | TIMETZ | TIMESTAMP | TIMESTAMP WITH TIME ZONE |
TIMESTAMPTZ;
bit_type: BIT type_length? | VARBIT type_length? | BIT VARYING type_length?;
binary_type: BINARY type_length? | BINARY VARYING type_length? | VARBINARY type_length?;

 /*
===============================================================================
<value_expression_primary>
===============================================================================
*/

value_expression_primary: parenthesized_value_expression | nonparenthesized_value_expression_primary;
parenthesized_value_expression: LEFT_PAREN value_expression RIGHT_PAREN;
nonparenthesized_value_expression_primary: unsigned_value_specification | column_reference | set_function_specification |
scalar_subquery | case_expression | cast_specification | routine_invocation;

 /*
===============================================================================
<unsigned value specification>
===============================================================================
*/
unsigned_value_specification: unsigned_literal;
unsigned_numeric_literal: NUMBER | REAL_NUMBER;
signed_numerical_literal: sign? unsigned_numeric_literal;

 /*
===============================================================================
<set function specification>
Invoke an SQL-invoked routine.
===============================================================================
 */
set_function_specification: aggregate_function;
aggregate_function: COUNT LEFT_PAREN MULTIPLY RIGHT_PAREN | general_set_function filter_clause?;
general_set_function: set_function_type LEFT_PAREN set_qualifier? value_expression RIGHT_PAREN;
set_function_type: AVG | MAX | MIN | SUM | EVERY | ANY | SOME | COUNT | STDDEV_POP | STDDEV_SAMP | VAR_SAMP |
VAR_POP | COLLECT | FUSION | INTERSECTION;
filter_clause: FILTER LEFT_PAREN WHERE search_condition RIGHT_PAREN;
grouping_operation: GROUPING LEFT_PAREN column_reference_list RIGHT_PAREN;

 /*
===============================================================================
<case expression>
===============================================================================
*/

case_expression: case_specification;

case_abbreviation
: NULLIF LEFT_PAREN numeric_value_expression COMMA boolean_value_expression RIGHT_PAREN | COALESCE LEFT_PAREN
numeric_value_expression ( COMMA boolean_value_expression )+ RIGHT_PAREN;
case_specification: simple_case | searched_case;
simple_case: CASE boolean_value_expression ( simple_when_clause )+ ( else_clause)? END;
searched_case: CASE (searched_when_clause)+ (else_clause)? END;
simple_when_clause: WHEN search_condition THEN result;
searched_when_clause: WHEN c=search_condition THEN r=result;
```

```
else_clause: ELSE r=result;
result: value_expression | NULL;

 /*
============================================================================================
<cast specification>
============================================================================================
 */

cast_specification: CAST LEFT_PAREN cast_operand AS cast_target RIGHT_PAREN;
cast_operand: value_expression;
cast_target: data_type;

 /*
============================================================================================
<value expression>
============================================================================================
*/

value_expression: common_value_expression | row_value_expression | boolean_value_expression;
common_value_expression: numeric_value_expression | string_value_expression | NULL;

 /*
============================================================================================
<numeric value expression>
Specify a comparison of two row values.
============================================================================================
*/

numeric_value_expression: left=term ((PLUS|MINUS) right=term)*;
term: left=factor ((MULTIPLY|DIVIDE|MODULAR) right=factor)*;
factor: (sign)? numeric_primary;
array: LEFT_PAREN numeric_value_expression (COMMA numeric_value_expression )* RIGHT_PAREN;
numeric_primary: value_expression_primary (CAST_EXPRESSION cast_target)* | numeric_value_function;
sign: PLUS | MINUS;

/*
============================================================================================
<numeric value function>
============================================================================================
 */
numeric_value_function: extract_expression;
extract_expression: EXTRACT LEFT_PAREN extract_field_string=extract_field FROM extract_source RIGHT_PAREN;
extract_field: primary_datetime_field | time_zone_field | extended_datetime_field;
time_zone_field: TIMEZONE | TIMEZONE_HOUR | TIMEZONE_MINUTE;
extract_source: column_reference | datetime_literal;

/*
============================================================================================
<string value expression>
============================================================================================
*/
string_value_expression: character_value_expression;
character_value_expression: character_factor (CONCATENATION_OPERATOR character_factor)*;
character_factor: character_primary;
character_primary: value_expression_primary | string_value_function;

/*
============================================================================================
<string value function>
============================================================================================
*/
```

```
string_value_function: trim_function;
trim_function: TRIM LEFT_PAREN trim_operands RIGHT_PAREN;
trim_operands: ((trim_specification)? (trim_character=character_value_expression)? FROM)? trim_source=character_value_expression |
trim_source=character_value_expression COMMA trim_character=character_value_expression;
 trim_specification: LEADING | TRAILING | BOTH;

 /*
==========================================================================================
<boolean value expression>
==========================================================================================
*/

boolean_value_expression: or_predicate;
or_predicate: and_predicate (OR or_predicate)*;
and_predicate: boolean_factor (AND and_predicate)*;
boolean_factor: boolean_test | NOT boolean_test;
boolean_test: boolean_primary is_clause?;
is_clause: IS NOT? t=truth_value;
truth_value: TRUE | FALSE | UNKNOWN;
boolean_primary: predicate | boolean_predicand;
boolean_predicand: parenthesized_boolean_value_expression | nonparenthesized_value_expression_primary;
parenthesized_boolean_value_expression: LEFT_PAREN boolean_value_expression RIGHT_PAREN;

/*
==========================================================================================
 <row value expression>
==========================================================================================
*/

row_value_expression: row_value_special_case | explicit_row_value_constructor;
row_value_special_case: nonparenthesized_value_expression_primary;
explicit_row_value_constructor: NULL;
row_value_predicand: row_value_special_case | row_value_constructor_predicand;
row_value_constructor_predicand: common_value_expression | boolean_predicand // | explicit_row_value_constructor;

/*
==========================================================================================
 <table expression>
==========================================================================================
*/

table_expression: from_clause | where_clause? | groupby_clause? | having_clause? | orderby_clause? | limit_clause? | offset_clause?;

/*
 ==========================================================================================
 <from clause>
 ==========================================================================================
*/

from_clause: FROM table_reference_list;
table_reference_list: table_reference (COMMA table_reference)*;

/*
==========================================================================================
 <table reference>
==========================================================================================
*/
table_reference: joined_table | table_primary;

/*
==========================================================================================
 <joined table>
```

```
==========================================================================================
*/
joined_table: table_primary joined_table_primary+;

joined_table_primary
: CROSS JOIN right=table_primary
| (t=join_type)? JOIN right=table_primary s=join_specification
| NATURAL (t=join_type)? JOIN right=table_primary
| UNION JOIN right=table_primary;
cross_join: CROSS JOIN r=table_primary;qualified_join: (t=join_type)? JOIN r=table_primary s=join_specification;
natural_join: NATURAL (t=join_type)? JOIN r=table_primary;
union_join: UNION JOIN r=table_primary;
join_type: INNER | t=outer_join_type;
outer_join_type: outer_join_type_part2 OUTER?;

outer_join_type_part2: LEFT | RIGHT | FULL;
join_specification: join_condition | named_columns_join;
join_condition: ON search_condition;
named_columns_join: USING LEFT_PAREN f=column_reference_list RIGHT_PAREN;
table_primary
: table_or_query_name ((AS)? alias=identifier)? (LEFT_PAREN column_name_list RIGHT_PAREN)?
| derived_table (AS)? name=identifier (LEFT_PAREN column_name_list RIGHT_PAREN)?;
column_name_list: identifier ( COMMA identifier )*;
derived_table: table_subquery;

/*
==========================================================================================
 <where clause>
==========================================================================================
*/

where_clause: WHERE search_condition;
search_condition: value_expression // instead of boolean_value_expression, we use value_expression for more flexibility.

/*
==========================================================================================
 <group by clause>
==========================================================================================
*/

groupby_clause: GROUP BY g=grouping_element_list;
grouping_element_list: grouping_element (COMMA grouping_element)*;
grouping_element: empty_grouping_set | ordinary_grouping_set;
ordinary_grouping_set: row_value_predicand | LEFT_PAREN row_value_predicand_list RIGHT_PAREN;
ordinary_grouping_set_list: ordinary_grouping_set (COMMA ordinary_grouping_set)*;
empty_grouping_set: LEFT_PAREN RIGHT_PAREN;
having_clause: HAVING boolean_value_expression;
row_value_predicand_list: row_value_predicand (COMMA row_value_predicand)*;

/*
==========================================================================================
 <query expression>
==========================================================================================
*/

query_expression: query_expression_body;
query_expression_body: non_join_query_expression | joined_table;
non_join_query_expression: (non_join_query_term | joined_table (UNION | EXCEPT) (ALL|DISTINCT |DISTINCT ON)? query_term)
 ((UNION | EXCEPT) (ALL | DISTINCT | DISTINCT ON)? query_term)*;
query_term: non_join_query_term | joined_table;
```

```
non_join_query_term: ( non_join_query_primary
| joined_table INTERSECT (ALL|DISTINCT|DISTINCT ON)? query_primary)
(INTERSECT (ALL|DISTINCT|DISTINCT ON)? query_primary)*;

query_primary: non_join_query_primary | joined_table;
non_join_query_primary: simple_table | LEFT_PAREN non_join_query_expression RIGHT_PAREN;
simple_table: query_specification | explicit_table;
explicit_table: TABLE table_or_query_name;
table_or_query_name: table_name | identifier;
table_name: identifier ( DOT identifier ( DOT identifier )? )?;
query_specification: SELECT set_qualifier? select_list table_expression?;
select_list: select_sublist (COMMA select_sublist)*;
select_sublist: derived_column  | qualified_asterisk;
derived_column: value_expression as_clause?;
qualified_asterisk: (tb_name=Identifier DOT)? MULTIPLY;
set_qualifier: DISTINCT | DISTINCT ON | ALL;
column_reference: (tb_name=identifier DOT)? name=identifier;
as_clause: (AS)? identifier;
column_reference_list: column_reference (COMMA column_reference)*;

/*
===============================================================================================
 <subquery>
Specify a scalar value, a row, or a table derived from a query expression.
===============================================================================================
*/

scalar_subquery: subquery;
row_subquery: subquery;
table_subquery: subquery;
subquery: LEFT_PAREN query_expression RIGHT_PAREN;

 /*
===============================================================================================
 <predicate>
===============================================================================================
 */

Predicate: comparison_predicate | between_predicate | in_predicate
 | pattern_matching_predicate // like predicate and other similar predicates
 | null_predicate | exists_predicate;

 /*
===============================================================================================
 <comparison predicate>
Specify a comparison of two row values.
===============================================================================================
 */

comparison_predicate: left=row_value_predicand c=comp_op right=row_value_predicand;
comp_op: EQUAL | NOT_EQUAL | LTH | LEQ | GTH | GEQ;

 /*
===============================================================================================
 <between predicate>
===============================================================================================
 */

between_predicate: predicand=row_value_predicand between_predicate_part_2;
between_predicate_part_2: (NOT)? BETWEEN (ASYMMETRIC | SYMMETRIC)? begin=row_value_predicand AND
end=row_value_predicand;
```

```
/*
==========================================================================================
 <in predicate>
==========================================================================================
*/

in_predicate: predicand=numeric_value_expression NOT? IN in_predicate_value;
in_predicate_value: table_subquery | LEFT_PAREN in_value_list RIGHT_PAREN;
in_value_list: row_value_expression ( COMMA row_value_expression )*;

/*
==========================================================================================
 <pattern matching predicate>
Specify a pattern-matching comparison.
==========================================================================================
*/

pattern_matching_predicate: f=row_value_predicand pattern_matcher s=Character_String_Literal;
pattern_matcher: NOT? negativable_matcher | regex_matcher;
negativable_matcher: LIKE | ILIKE | SIMILAR TO | REGEXP | RLIKE;
regex_matcher: Similar_To | Not_Similar_To | Similar_To_Case_Insensitive | Not_Similar_To_Case_Insensitive;

/*
==========================================================================================
<null predicate>
Specify a test for a null value.
==========================================================================================
*/

null_predicate: predicand=row_value_predicand IS (n=NOT)? NULL;

/*
==========================================================================================
<quantified comparison predicate>
Specify a quantified comparison.
==========================================================================================
*/

quantified_comparison_predicate: l=numeric_value_expression c=comp_op q=quantifier s=table_subquery;
quantifier: all | some;
all: ALL;
some: SOME | ANY;

/*
==========================================================================================
 <exists predicate>
Specify a test for a non_empty set.
==========================================================================================
*/

exists_predicate: NOT? EXISTS s=table_subquery;

/*
==========================================================================================
<unique predicate>
Specify a test for the absence of duplicate rows
==========================================================================================
 */

unique_predicate: UNIQUE s=table_subquery;

/*
==========================================================================================
```

```
<interval qualifier>
Specify the precision of an interval data type.
==========================================================================================
 */

primary_datetime_field: non_second_primary_datetime_field | SECOND;
non_second_primary_datetime_field: YEAR | MONTH | DAY | HOUR | MINUTE;
extended_datetime_field: CENTURY | DECADE | DOW | DOY | EPOCH | ISODOW | ISOYEAR | MICROSECONDS | MILLENNIUM |
MILLISECONDS | QUARTER | WEEK;

/*
==========================================================================================
<routine invocation>
Invoke an SQL-invoked routine.
==========================================================================================
 */

routine_invocation: function_name LEFT_PAREN sql_argument_list? RIGHT_PAREN;
function_names_for_reserved_words: LEFT | RIGHT;
function_name: identifier | function_names_for_reserved_words;
sql_argument_list: value_expression (COMMA value_expression)*;

 /*
==========================================================================================
<declare cursor>
==========================================================================================
 */

orderby_clause: ORDER BY sort_specifier_list;
sort_specifier_list: sort_specifier (COMMA sort_specifier)*;
sort_specifier: key=row_value_predicand order=order_specification? null_order=null_ordering?;
order_specification: ASC | DESC;
limit_clause: LIMIT e=numeric_value_expression;
null_ordering: NULL FIRST | NULL LAST;

/*
 * Lexer Rules
 */
/*
==========================================================================================
Tokens for Case Insensitive Keywords
==========================================================================================
*/

fragment A: 'A' | 'a';
fragment B: 'B' | 'b';
fragment C: 'C' | 'c';
fragment D: 'D' | 'd';
fragment E: 'E' | 'e';
fragment F: 'F' | 'f';
fragment G: 'G' | 'g';
fragment H: 'H' | 'h';
fragment I : 'I' | 'i';
fragment J : 'J' | 'j';
fragment K: 'K' | 'k';
fragment L: 'L' | 'l';
fragment M: 'M' | 'm';
fragment N: 'N' | 'n';
fragment O: 'O' | 'o';
fragment P: 'P' | 'p';
fragment Q: 'Q' | 'q';
fragment R: 'R' | 'r';
fragment S: 'S' | 's';
fragment T: 'T' | 't';
```

```
fragment U: 'U' | 'u';
fragment V: 'V' | 'v';
fragment W: 'W' | 'w';
fragment X: 'X' | 'x';
fragment Y: 'Y' | 'y';
fragment Z: 'Z' | 'z';

/*
===========================================================================================
Reserved Keywords
===========================================================================================
*/

AS: A S;
ALL: A L L;
AND: A N D;
ANY: A N Y;
ASYMMETRIC: A S Y M M E T R I C;
ASC: A S C;
BOTH: B O T H;
CASE: C A S E;
CAST: C A S T;
CREATE: C R E A T E;
CROSS: C R O S S;
DESC: D E S C;
DISTINCT: D I S T I N C T;
END: E N D;
ELSE: E L S E;
EXCEPT: E X C E P T;
FALSE: F A L S E;
FULL: F U L L;
FROM: F R O M;
GROUP: G R O U P;
HAVING: H A V I N G;
ILIKE: I L I K E;
IN: I N;
INNER: I N N E R;
INTERSECT: I N T E R S E C T;
INTO: I N T O;
IS: I S;
JOIN: J O I N;
LEADING: L E A D I N G;
LEFT: L E F T;
LIKE: L I K E;
LIMIT: L I M I T;
NATURAL: N A T U R A L;
NOT: N O T;
NULL: N U L L;
ON: O N;
OUTER: O U T E R;
OR: O R;
ORDER: O R D E R;
RIGHT: R I G H T;
SELECT: S E L E C T;
SOME: S O M E;
SYMMETRIC: S Y M M E T R I C;
TABLE: T A B L E;
THEN: T H E N;
TRAILING: T R A I L I N G;
TRUE: T R U E;
UNION: U N I O N;
UNIQUE: U N I Q U E;
USING: U S I N G;
WHEN: W H E N;
```

```
WHERE: W H E R E;
WITH: W I T H;

/*
==========================================================================================
Non Reserved Keywords
==========================================================================================
*/

AVG: A V G;
BETWEEN: B E T W E E N;
BY: B Y;
CENTURY: C E N T U R Y;
CHARACTER: C H A R A C T E R;
COLLECT: C O L L E C T;
COALESCE: C O A L E S C E;
COLUMN: C O L U M N;
COUNT: C O U N T;
DAY: D A Y;
DEC: D E C;
DECADE: D E C A D E;
DOW: D O W;
DOY: D O Y;
EPOCH: E P O C H;
EVERY: E V E R Y;
EXISTS: E X I S T S;
EXTERNAL: E X T E R N A L;
EXTRACT: E X T R A C T;
FILTER: F I L T E R;
FIRST: F I R S T;
FORMAT: F O R M A T;
FUSION: F U S I O N;
GROUPING: G R O U P I N G;
HASH: H A S H;
HOUR: H O U R;
INDEX: I N D E X;
INSERT: I N S E R T;
INTERSECTION: I N T E R S E C T I O N;
ISODOW: I S O D O W;
ISOYEAR: I S O Y E A R;
LAST: L A S T;
LESS: L E S S;
LIST: L I S T;
LOCATION: L O C A T I O N;
MAX: M A X;
MAXVALUE: M A X V A L U E;
MICROSECONDS: M I C R O S E C O N D S;
MILLENNIUM: M I L L E N N I U M;
MILLISECONDS: M I L L I S E C O N D S;
MIN: M I N;
MINUTE: M I N U T E;
MONTH: M O N T H;
NATIONAL: N A T I O N A L;
NULLIF: N U L L I F;
PRECISION: P R E C I S I O N;
QUARTER: Q U A R T E R;
RANGE: R A N G E;
REGEXP: R E G E X P;
RLIKE: R L I K E;
SECOND: S E C O N D;
SET: S E T;
SIMILAR: S I M I L A R;
STDDEV_POP: S T D D E V UNDERLINE P O P;
STDDEV_SAMP: S T D D E V UNDERLINE S A M P;
```

```
SUM: S U M;
TABLESPACE: T A B L E S P A C E;
THAN: T H A N;
TIMEZONE: T I M E Z O N E;
TIMEZONE_HOUR: T I M E Z O N E UNDERLINE H O U R;
TIMEZONE_MINUTE: T I M E Z O N E UNDERLINE M I N U T E;
TRIM: T R I M;
TO: T O;
UNKNOWN: U N K N O W N;
VALUES: V A L U E S;
VAR_SAMP: V A R UNDERLINE S A M P;
VAR_POP: V A R UNDERLINE P O P;
VARYING: V A R Y I N G;
WEEK: W E E K;
YEAR: Y E A R;
ZONE: Z O N E;

/*
===============================================================================================
Data Type Tokens
===============================================================================================
*/

BOOLEAN: B O O L E A N;
BOOL: B O O L;
BIT: B I T;
VARBIT: V A R B I T;
INT1: I N T '1';
INT2: I N T '2';
INT4: I N T '4';
INT8: I N T '8';
TINYINT: T I N Y I N T; // alias for INT1
SMALLINT: S M A L L I N T; // alias for INT2
INT: I N T; // alias for INT4
INTEGER: I N T E G E R; // alias - INT4
BIGINT: B I G I N T; // alias for INT8
FLOAT4: F L O A T '4';
FLOAT8: F L O A T '8';
REAL: R E A L; // alias for FLOAT4
FLOAT: F L O A T; // alias for FLOAT8
DOUBLE: D O U B L E; // alias for FLOAT8
NUMERIC: N U M E R I C;
DECIMAL: D E C I M A L; // alias for number
CHAR: C H A R;
VARCHAR: V A R C H A R;
NCHAR: N C H A R;
NVARCHAR: N V A R C H A R;
DATE: D A T E;
TIME: T I M E;
TIMETZ: T I M E T Z;
TIMESTAMP: T I M E S T A M P;
TIMESTAMPTZ: T I M E S T A M P T Z;
TEXT: T E X T;
BINARY: B I N A R Y;
VARBINARY: V A R B I N A R Y;
BLOB: B L O B;
BYTEA: B Y T E A; // alias for BLOB
INET4: I N E T '4';
// Operators
Similar_To: '~';
Not_Similar_To: '!~';
Similar_To_Case_Insensitive: '~*';
Not_Similar_To_Case_Insensitive: '!~*';
```

```
// Cast Operator
CAST_EXPRESSION: COLON COLON;

ASSIGN: ':=';
EQUAL: '=';
COLON: ':';
SEMI_COLON: ';';
COMMA: ',';
CONCATENATION_OPERATOR: VERTICAL_BAR VERTICAL_BAR;
NOT_EQUAL: '<>' | '!=' | '~='| '^=';
LTH: '<';
LEQ: '<=';
GTH: '>';
GEQ: '>=';
LEFT_PAREN: '(';
RIGHT_PAREN: ')';
PLUS: '+';
MINUS: '-';
MULTIPLY: '*';
DIVIDE: '/';
MODULAR: '%';
DOT: '.';
UNDERLINE: '_';
VERTICAL_BAR: '|';
QUOTE: '\'';
DOUBLE_QUOTE: '"';
NUMBER: Digit+;
fragment
Digit: '0'..'9';
REAL_NUMBER
: ('0'..'9')+ '.' ('0'..'9')* EXPONENT?
| '.' ('0'..'9')+ EXPONENT?
| ('0'..'9')+ EXPONENT;

BlockComment: '/*' .*? '*/' -> skip;
LineComment: '--' ~[\r\n]* -> skip;

/*
==========================================================================================
Identifiers
==========================================================================================
*/

Identifier: Regular_Identifier;
fragment
Regular_Identifier: ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|Digit|'_')*;

/*
==========================================================================================
 Literal
==========================================================================================
*/

// Some Unicode Character Ranges
fragment Control_Characters: '\u0001' .. '\u001F';
fragment Extended_Control_Characters: '\u0080' .. '\u009F';
Character_String_Literal: QUOTE ( ESC_SEQ | ~('\\'|'\'') )* QUOTE;
fragment EXPONENT: ('e'|'E') ('+'|'-')? ('0'..'9')+;
fragment HEX_DIGIT: ('0'..'9'|'a'..'f'|'A'..'F');
fragment ESC_SEQ: '\\' ('b'|'t'|'n'|'f'|'r'|'\"'|'\''|'\\') | UNICODE_ESC | OCTAL_ESC;
fragment OCTAL_ESC: '\\' ('0'..'3') ('0'..'7') ('0'..'7') | '\\' ('0'..'7') ('0'..'7') | '\\' ('0'..'7');
fragment UNICODE_ESC: '\\' 'u' HEX_DIGIT HEX_DIGIT HEX_DIGIT HEX_DIGIT;
```

```
/*
==========================================================================================
Whitespace Tokens
==========================================================================================
*/

Space: ' ' -> skip;
White_Space: ( Control_Characters | Extended_Control_Characters )+ -> skip;
BAD: . -> skip;
ErrorChar: .;
```

Bibliographie

[1] A. Mitrović, « Experiences in Implementing Constraint-Based Modeling in SQL-Tutor », in *Intelligent Tutoring Systems*, vol. 1452, B. P. Goettl, H. M. Halff, C. L. Redfield, et V. J. Shute, Éd. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, p. 414-423.

[2] K. VanLehn, « The Behavior of Tutoring Systems », *Int. J. Artif. Intell. Educ.*, vol. 16, n° 3, p. 227-265, janv. 2006.

[3] P. Abrahamsson, O. Salo, J. Ronkainen, et J. Warsta, « Agile Software Development Methods: Review and Analysis », Cornell University Library, ArXiv:1709.08439 [cs.SE], sept. 2017.

[4] K. Beck et E. Gamma, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2000.

[5] S. Fraser, K. Beck, B. Caputo, T. Mackinnon, J. Newkirk, et C. Poole, « Test Driven Development (TDD) », in *Extreme Programming and Agile Processes in Software Engineering*, 2003, p. 459-462.

[6] J. Navarro, « Unit testing with CppUnit. », *Unit testing with CppUnit.*, 2003. [En ligne]. Disponible sur: https://www.codeproject.com/Articles/5660/Unit-testing-with-CPPUnit. [Consulté le: 02-mai-2018].

[7] R. K. Gupta, H. Prajapati, et H. Singh, *Test-Driven JavaScript Development*. Packt Publishing Ltd, 2015.

[8] JUnit, « JUnit », 2002. [En ligne]. Disponible sur: http://junit.org/junit4/. [Consulté le: 23-mai-2016].

[9] T. Nathaniel, « Test::Unit - Ruby Unit Testing », *Test::Unit - Ruby Unit Testing https://ruby-doc.org/stdlib-1.8.7/libdoc/test/unit/rdoc/Test/Unit.html*, 2014.

[10] J. Hunt, « Scala Testing », in *A Beginner's Guide to Scala, Object Orientation and Functional Programming*, Springer, Cham, 2014, p. 365-382.

[11] P. Hamill, *Unit Test Frameworks: Tools for High-Quality Software Development*. O'Reilly Media, Inc., 2004.

[12] « Database Unit Testing for SQL Server tsqlt », *http://tsqlt.org/*, 2016. [En ligne]. Disponible sur: http://tsqlt.org/. [Consulté le: 21-févr-2018].

[13] S. Cohen, « Equivalence of Queries Combining Set and Bag-set Semantics », in *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, New York, NY, USA, 2006, p. 70–79.

[14] P. Baxendale et E. F. Codd, « A Relational Model of Data for Large Shared Data Banks », *Commun. ACM*, vol. 13, n° 6, p. 377-387, 1970.

[15] E. F. Codd, « Extending the Database Relational Model to Capture More Meaning », *ACM Trans Database Syst*, vol. 4, n° 4, p. 397–434, 1979.

[16] H. Harris et B. Nicol, « SQL/DS: IBM's First RDBMS », *IEEE Ann. Hist. Comput.*, vol. 35, n° 2, p. 69-71, avr. 2013.

[17] R. Preger, « The Oracle Story, Part 1: 1977-1986 », *IEEE Ann. Hist. Comput.*, vol. 34, n° 4, p. 51-57, oct. 2012.

[18] « mariaDB », *MariaDB*, 2009. [En ligne]. Disponible sur: https://mariadb.com/, https://mariadb.org/. [Consulté le: 03-mai-2018].

[19] « MySQL », *MySql*, 1995. [En ligne]. Disponible sur: https://dev.mysql.com/doc/. [Consulté le: 03-mai-2018].

[20] PostgreSQL Global Development Group, « PostgreSQL », *PostgreSQL*, 1996. [En ligne]. Disponible sur: https://www.postgresql.org/. [Consulté le: 03-mai-2018].

[21] R. Elmasri et S. B. Navathe, *Fundamentals of Database Systems*, 7th éd. Pearson, 2015.

[22] L. Lavoie et C. Khnaisser, « Théorie relationnelle », *IGE 487 - Modélisation de bases de données*, 2017. .

[23] J. D. Ullman et J. Widom, *A First Course in Database Systems*, 3rd ed. Upper Saddle River, NJ: Pearson/Prentice Hall, 2008.

[24] R. Elmasri et S. Navathe, *Fundamentals of Database Systems*, 6th ed. Boston: Addison-Wesley, 2011.

[25] L. Jiang et Z. Su, « Automatic Mining of Functionally Equivalent Code Fragments via Random Testing », in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, New York, NY, USA, 2009, p. 81–92.

[26] B. S. Baker, « On Finding Duplication and Near-duplication in Large Software Systems », in *Proceedings of 2nd Working Conference on Reverse Engineering*, 1995, p. 86-95.

[27] T. Kamiya, S. Kusumoto, et K. Inoue, « CCFinder: a Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code », *IEEE Trans. Softw. Eng.*, vol. 28, n° 7, p. 654-670, 2002.

[28] I. D. Baxter, C. Pidgeon, et M. Mehlich, « DMS reg: Program Transformations for Practical Scalable Software Evolution », in *Proceedings. 26th International Conference on Software Engineering*, 2004, p. 625-634.

[29] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, et L. Bier, « Clone Detection Using Abstract Syntax Trees », in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, 1998, p. 368-377.

[30] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, et M. Bernstein, « Pattern Matching for Clone and Concept Detection », *Autom. Softw. Eng.*, vol. 3, n° 1-2, p. 77-108, 1996.

[31] J. Krinke, « Identifying Similar Code With Program Dependence Graphs », in *Proceedings Eighth Working Conference on Reverse Engineering*, 2001, p. 301-309.

[32] D. Schuler, V. Dallmeier, et C. Lindig, « A Dynamic Birthmark for Java », in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, 2007, p. 274–283.

[33] R. Belohlavek et V. Vychodil, « Relational Similarity-based Model of Data Part 1: Foundations and Query Systems », *Int. J. Gen. Syst.*, vol. 46, n° 7, p. 671-751, 2017.

[34] R. Dollinger et N. A. Melville, « Semantic Evaluation of SQL Queries », in *2011 IEEE 7th International Conference on Intelligent Computer Communication and Processing*, 2011, p. 57-64.

[35] A. K. Chandra et P. M. Merlin, « Optimal Implementation of Conjunctive Queries in Relational Data Bases », in *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, New York, NY, USA, 1977, p. 77–90.

[36] D. S. Johnson et A. Klug, « Optimizing Conjunctive Queries that Contain Untyped Variables », *SIAM J. Comput. Phila.*, vol. 12, n° 4, p. 25, 1983.

[37] Y. Sagiv et Y. Saraiya, « Minimizing Restricted-fanout Queries », *Discrete Appl. Math.*, vol. 40, n° 2, p. 245-264, 1992.

[38] C. Chekuri et A. Rajaraman, « Conjunctive Query Containment Revisited », *Theor. Comput. Sci.*, vol. 239, n° 2, p. 211-229, 2000.

[39] R. van der Meyden, « The Complexity of Querying Indefinite Data About Linearly Ordered Domains », in *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, New York, NY, USA, 1992, p. 331–345.

[40] A. Y. Levy et Y. Sagiv, « Semantic Query Optimization in Datalog Programs (Extended Abstract) », in *Proceedings of the Fourteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, New York, NY, USA, 1995, p. 163–173.

[41] A. Levy, I. S. Mumick, Y. Sagiv, et O. Shmueli, « Equivalence, Query-reachability and Satisfiability in Datalog Extensions », in *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, New York, NY, USA, 1993, p. 109–122.

[42] Y. Sagiv et M. Yannakakis, « Equivalences Among Relational Expressions with the Union and Difference Operators », *J ACM*, vol. 27, n° 4, p. 633–655, 1980.

[43] S. Chaudhuri et M. Y. Vardi, « Optimization of Real Conjunctive Queries », in *Proceedings of the Twelfth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, New York, NY, USA, 1993, p. 59–70.

[44] S. Cohen, Y. Sagiv, et W. Nutt, « Equivalences Among Aggregate Queries with Negation », *ACM Trans Comput Log.*, vol. 6, n° 2, p. 328–360, avr. 2005.

[45] S. Grumbach, M. Rafanelli, et L. Tininini, « On the Equivalence and Rewriting of Aggregate Queries », *Acta Inform.*, vol. 40, n° 8, p. 529-584, 2004.

[46] S. Cohen, W. Nutt, et Y. Sagiv, « Deciding Equivalences Among Conjunctive Aggregate Queries », *J ACM*, vol. 54, n° 2, 2007.

[47] Y. E. Ioannidis et R. Ramakrishnan, « Containment of Conjunctive Queries: Beyond Relations As Sets », *ACM Trans Database Syst*, vol. 20, n° 3, p. 288–324, 1995.

[48] W. Kim, « On Optimizing an SQL-like Nested Query », *ACM Trans Database Syst*, vol. 7, n° 3, p. 443–469, 1982.

[49] U. Dayal, « Of Nests and Trees: A Unified Approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers », in *Proceedings of the 13th International Conference on Very Large Data Bases*, San Francisco, CA, USA, 1987, p. 197–208.

[50] I. S. Mumick et H. Pirahesh, « Implementation of Magic-sets in a Relational Database System », in *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 1994, p. 103–114.

[51] S. Ceri et G. Gottlob, « Translating SQL Into Relational Algebra: Optimization, Semantics, and Equivalence of SQL Queries », *IEEE Trans. Softw. Eng.*, vol. SE-11, n° 4, p. 324-345, 1985.

[52] L. Baekgaard et L. Mark, « Incremental Computation of Nested Relational Query Expressions », *ACM Trans Database Syst*, vol. 20, n° 2, p. 111–148, 1995.

[53] B. Cao et A. Badia, « SQL Query Optimization Through Nested Relational Algebra », *ACM Trans Database Syst*, vol. 32, n° 3, 2007.

[54] S. Anuj, Paras Nath Barwal, « jOOQ-JAVA OBJECT ORIENTED QUERYING », *JOOQ-JAVA OBJECT ORIENTED QUERYING*, 2014. [En ligne]. Disponible sur: http://esatjournals.net/ijret/2014v03/i09/IJRET20140309049.pdf. [Consulté le: 01-août-2017].

[55] P. Terence, « ANTLR », *http://www.antlr.org/*, 1992. [En ligne]. Disponible sur: http://www.antlr.org/. [Consulté le: 01-août-2017].

[56] T. Parr, *The Definitive ANTLR 4 Reference*. Dallas, Texas: The Pragmatic Bookshelf, 2012.

[57] T. J. Parr et R. W. Quong, « ANTLR: A Predicated-LL(k) Parser Generator », *Softw. Pract. Exp.*, vol. 25, n° 7, p. 789-810, 1995.

[58] T. Parr et K. Fisher, « LL(*): The Foundation of the ANTLR Parser Generator », in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, 2011, p. 425–436.

[59] R. Johnson, « J2EE Development Frameworks », *Computer*, vol. 38, nº 1, p. 107-110, janv. 2005.

[60] R. Kallman *et al.*, « H-store: A High-performance, Distributed Main Memory Transaction Processing System », *Proc VLDB Endow*, vol. 1, nº 2, p. 1496–1499, août 2008.

[61] C. Bauer et G. King, « Java Persistence with Hibernate », *MANNING*, vol. version 2, p. 45.