



Construction automatique de bases de données pour le domaine médical : Intégration de texte et maintien de la cohérence

Nicolas Hiot

► To cite this version:

Nicolas Hiot. Construction automatique de bases de données pour le domaine médical : Intégration de texte et maintien de la cohérence. Informatique [cs]. Université d'Orléans, 2024. Français. <NNT : 2024ORLE1026>. <tel-04794341>

HAL Id: tel-04794341

<https://theses.hal.science/tel-04794341v1>

Submitted on 20 Nov 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



HAL Authorization

UNIVERSITÉ D'ORLÉANS

ÉCOLE DOCTORALE Mathématiques, Informatique, Physique Théorique et Ingénierie des Systèmes (MIPTIS)

Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)

THÈSE présentée par :

Nicolas HIOT 

soutenue le : **1er juillet 2024**

pour obtenir le grade de : **Docteur de l'Université d'Orléans**

Discipline / Spécialité : **Informatique**

Construction automatique de bases de données pour le domaine médical

Intégration de texte et maintien de la cohérence

THÈSE dirigée par :

Mme Mirian HALFELD FERRARI 

Pr

Univ. d'Orléans

RAPPORTEURS :

Mme Zoubida KEDAD 

Pr

UVSQ Univ. Paris-Saclay

M. Hubert NAACKE 

MCF-HDR

Sorbonne Université

JURY :

Mme Salima BENBERNOU 

Pr

Univ. Paris Cité

M. Jacques CHABIN 

MCF

Univ. d'Orléans

Mme Mirian HALFELD FERRARI 

Pr

Univ. d'Orléans

Mme Zoubida KEDAD 

Pr

UVSQ Univ. Paris-Saclay

M. Hubert NAACKE 

MCF-HDR

Sorbonne Université

Remerciements

Tout d'abord, je tiens à exprimer ma plus profonde gratitude à Mirian et Jacques pour l'opportunité qu'ils m'ont offerte en me faisant confiance et en m'encadrant tout au long de ce projet. Travailler à vos côtés a été un véritable plaisir, et vous m'avez transmis bien plus que des connaissances. Votre expertise, votre soutien et vos conseils ont été inestimables. J'espère sincèrement que notre collaboration continuera à se renforcer et à se prolonger.

Je souhaite ensuite remercier Zoubida Kedad et Hubert Naacke pour l'honneur qu'ils m'ont fait en acceptant d'être rapporteurs de cette thèse. Merci également à Salima Benbernou pour avoir présidée ce jury. Vos questions pertinentes, nos échanges et vos conseils avisés ont enrichi mon travail et m'ont permis de progresser dans ma réflexion scientifique.

Un grand merci à toute l'équipe PAMDA pour votre accueil chaleureux, et tout particulièrement à Sophie, en ta qualité de cheffe d'équipe, pour ta gentillesse.

Je souhaite également exprimer ma reconnaissance à l'ensemble des équipes d'Ennov, et plus spécifiquement à Olivier Paris et Joshua Amavi, pour la confiance qu'ils m'ont témoignée tout au long de cette aventure. À l'équipe EnnovLabs : Ivan et Thang, tout juste arrivé, merci pour votre soutien et nos échanges. À l'équipe R&D de Paris : Alexandre, Arnaud, Florian, Jonathan, Lionel et Nhu, vous avez égayé mes journées et m'avez beaucoup appris. Une pensée particulière également pour les chefs de projet : Anaïs, Aphia, Audrey, Katell, Lilly, Marie, Maryna et Pierre, ainsi que pour Anissa, Colm, Laurent, Mathilde et Virginie. Nos échanges riches et constructifs ont non seulement nourri mes réflexions, mais ont aussi apporté un véritable soutien lors des moments plus difficiles.

Je tiens à remercier chaleureusement les membres du LIFO et, en particulier, l'ensemble des doctorants avec qui j'ai eu la chance de partager cette aventure : Antoine, Aymeric, Ciro, Darine, Florian, Joanne, Jordan, Lingchen, Mael, Mathieu, Olivier, Placido, Samuel, Sébastien, Shahrzad, Sofiane et Thibaut. Merci pour votre sincérité, votre gentillesse, votre entraide et votre soutien continu. J'espère que cet esprit de solidarité perdurera encore longtemps. Je vous souhaite à tous beaucoup de succès dans vos travaux respectifs.

À tous ceux avec qui j'ai eu la chance de collaborer et d'échanger durant ces années, merci pour ces expériences enrichissantes qui ont contribué à l'évolution de mes travaux.

Un immense merci à mes amis : Alexis, Anaïs, Antoine, Axel, Coraline, Hicham, Julie, Loanne, Mickaël, Morgane, Nicolas, Pierre et Yann. Merci d'avoir été présents à mes côtés pour me changer les idées, me soutenir et répondre présent. Vos encouragements et les moments que nous avons partagés ont été une source précieuse de réconfort. Une mention spéciale à Axel, qui me suit depuis plus de 20 ans. Merci pour ta fidélité et ton soutien indéfectible à travers les années, sans jamais faillir.

Enfin, je tiens à remercier du fond du cœur toute ma famille pour votre soutien inconditionnel, en particulier à mes grands-parents. Une pensée toute spéciale pour ma sœur, qui a toujours été une oreille attentive, prête à écouter et à me conseiller lorsque j'en avais besoin.

Pour finir, je ne pourrai jamais assez remercier mes parents pour tous les sacrifices qu'ils ont consentis pour me permettre de poursuivre mes études et d'arriver jusqu'à ce point. Votre amour, votre soutien et vos encouragements ont été la clé de ma réussite. Ce manuscrit est autant le vôtre que le mien. Je vous aime profondément !

Table des matières

Liste des figures	v
-------------------	---

Liste des tables	viii
------------------	------

1 Introduction	1
1.1 Un peu de contexte	1
1.2 Axes de recherche	2
1.3 Contributions et organisation du manuscrit	4
1.3.1 Publications	5
1.3.2 Logiciels	6

I Maintenance et cohérence d’une base de données incomplète	7
--	----------

2 Maintien de la cohérence	9
2.1 Introduction	9
2.2 Préliminaires	11
2.2.1 Logique du premier ordre	11
2.2.2 Base de données	11
2.2.3 Base de données graphes	12
2.2.4 Requête sur les graphes de propriétés	14
2.3 Travaux connexes	14
2.3.1 SETUP : Grammaire de graphe pour la mise à jour de base RDFS	16
2.3.2 Dépendances fonctionnelles pour les graphes	18
2.4 Discussion	19

3 Mise à jour incrémentale et cohérente d’une base de données	21
3.1 Contributions	21
3.2 Modélisation de la base de données sous Neo4J	22
3.3 Réduction incrémentale de la redondance	24
3.3.1 Requête sur le graphe	25
3.3.2 Simplification	26
3.4 Chase incrémental et effet de bords	28
3.4.1 Cycles et génération infinie	28
3.4.2 Utilisation du graphe pour le Chase	29
3.4.3 Chase en arrière	31
3.5 Mise à jour incrémentale	32
3.5.1 Insertion	32
3.5.2 Suppression	33
3.6 Evaluation	34
3.6.1 Implémentation sur un modèle relationnel	35
3.6.2 Évaluation des résultats	36
3.7 Conclusion	40

II	Intégration de données textuelles	43
4	Structuration d'information	45
4.1	Introduction	45
4.2	Des textes aux graphes	46
4.3	Préliminaires	49
4.3.1	Grammaires	49
4.3.2	Arbres	51
4.3.3	Textes et arbres	52
4.3.4	Réécriture d'arbres	53
4.4	Une grammaire comme schéma	54
4.5	Structuration automatique	58
4.5.1	Enrichissement des arbres	59
4.5.2	Extraction de la grammaire	64
4.5.3	Construction des classes d'équivalence	67
4.5.4	Réécriture des arbres	70
4.6	Proposition d'implémentation	70
4.6.1	Identification des groupes	72
4.6.2	Identification des relations	78
4.6.3	Identification des collections	79
4.7	Évaluation préliminaire	81
4.7.1	Comportement de la procédure instanceEvolution	82
4.7.2	Étude critique de la grammaire obtenue	85
4.7.3	Intégration de données	87
4.8	Conclusion	88
5	Extraction d'information	91
5.1	Introduction	91
5.2	Corpus et analyse syntaxique	92
5.3	Extraction d'entités ambiguës	94
5.3.1	Entités multi-valuées	95
5.3.2	Extraction par lexiques	95
5.3.3	Cascade de CRFs	98
5.4	Enrichissement et contextualisation	99
5.5	Classification de cas cliniques	101
5.5.1	Système de classification basé sur l'extraction par lexiques	101
5.5.2	Négation et incertitude	102
5.5.3	Évaluation	103
5.6	Interrogation de base de données en langue naturelle	103
5.6.1	Extraction d'entités	105
5.6.2	Construction de DB-query à partir d'entités enrichies	106
5.6.3	Évaluation	108
5.6.4	Construction de graphique	109
5.7	Conclusion	111
6	Conclusion et perspectives	113
6.1	Contributions	113
6.1.1	Maintenance et cohérence d'une base de données incomplète	113
6.1.2	Intégration de données textuelles	114
6.2	Perspectives	114
6.2.1	Maintien de la cohérence de graphes attribués	114
6.2.2	Uniformisation de données pour l'application de processus	115
6.2.3	Base de données floues	116

III	Annexes	117
A	Méthode de calcul et preuves pour la recherche de l'homomorphisme le plus spécifique	119
B	Description des jeux de données (mise à jour cohérente)	125
B.1	Movie	125
B.1.1	Description	125
B.1.2	Prédicats et contraintes	125
B.2	GameOfThrone	126
B.2.1	Description	126
B.2.2	Prédicats et contraintes	127
B.3	Social	128
B.3.1	Description	128
B.3.2	Prédicats et contraintes	129
	Bibliographie	131
	Glossaire	143
	Acronymes	146

Table des figures

1.1	Instance sous forme de graphe	4
2.1	Exemple d'instance d'une base de données graphe et son équivalent en relationnel	13
2.2	Règle de réécriture simplifiée pour l'insertion du fait $CI(A, B)$	16
2.3	Exemple d'ajout du fait $CI(A, B)$	17
2.4	Effets de bord pour l'ajout du fait $CSub(x, y)$	17
2.5	Exemple de modification pour l'application d'une règle	18
2.6	Exemple de dépendances fonctionnelles pour les graphes	19
2.7	Exemple de modification d'un graphe	19
3.1	Modèle de Graphe	22
3.2	Instance $\mathcal{D}$ représentée par un graphe de propriétés	24
3.3	Temps moyen d'exécution (ms) par SGBD et par opération	36
3.4	Nombre de requêtes émises par nombre de valeurs nulles dans l'instance	37
3.5	Résultats des expérimentations pour MySQL	38
3.6	Résultats des expérimentations pour Neo4J	39
4.1	Représentations syntaxiques d'un texte en graphe	46
4.2	Exemple de la structure de graphe utilisée par ABSTRA	48
4.3	Extrait d'un arbre de dérivation pour la règle de production $U \rightarrow X Y Z$	49
4.4	Exemple de dérivation d'une grammaire pour le nombre 0011	50
4.5	Exemple d'un arbre ordonné	51
4.6	Comparaison entre l'arbre syntaxique et l'arbre de dépendance	52
4.7	Exemple d'une règle de réécriture	53
4.8	Exemple d'application d'une règle de réécriture	53
4.9	Règle $\text{ins_elem}(T, A, u.i)$	54
4.10	Règle $\text{del_elem}(T, \epsilon)$	54
4.11	Règle $\text{del_elem}(T, u.i)$	54
4.12	Processus itératif pour la structuration automatique	54
4.13	Exemple de dérivation de la méta-grammaire	57
4.14	Processus de structuration sous forme de diagramme de flux	58
4.15	Comparaison entre les conjonctions en français et en anglais	60
4.16	Règle $\text{fix_coord}(T, u.i)$	60
4.17	Règle $\text{fix_conj}(T, u.i)$	61
4.18	Règle $\text{unnest_ent}(T, u.i)$	61
4.19	Exemple d'enrichissement d'arbre de syntaxe	62
4.20	Exemple de placement d'un sous-arbre entité	63
4.21	Règle $\text{reduce}(T, u.i, S_{\text{labels}})$	64
4.22	Exemple d'arbre enrichie simplifié	64
4.23	Exemple du quotient d'un arbre	65
4.24	Exemple d'arbre enrichi	69
4.25	Exemple de dendrogramme pour le partitionnement hiérarchique	70
4.26	Exemple d'application de la procédure findGroups	74
4.27	Exemple d'application de la fonction findSubgroups	76
4.28	Exemple d'application de la procédure mergeGroups	78

4.29	Exemple de transformation en insérant des relations	78
4.30	Exemple de transformation en insérant des collections	80
4.31	Extrait d'un document du corpus CAS	81
4.32	Nombre de règles de productions qui compose la grammaire pour chaque étape de la procédure	83
4.33	Nombre moyen d'instances pour chaque catégorie	83
4.34	Nombre de structures remarquable différentes pour chaque catégorie	84
4.35	Exemple d'instance	85
5.1	Imbrication d'entités dans le corpus CAS	92
5.2	Exemple d'un transducteur fini	98
5.3	Schéma du système d'identification du profil clinique du patient	102
5.4	Arbre de dépendance d'une version simplifiée de NLQ_{run}	105
5.5	Interface utilisateur pour la recherche en langage naturel	108
5.6	Interface utilisateur pour la construction de graphique	110
B.1	Schéma de la base <i>Movie</i>	125
B.2	Liste des contraintes de la base <i>Movie</i>	126
B.3	Schéma de la base	126
B.4	Liste des contraintes de la base <i>GOT</i>	127
B.4	Liste des contraintes de la base <i>GOT</i>	128
B.5	Schéma de la base <i>Social</i>	128
B.6	Liste des contraintes de la base <i>Social</i>	129
B.6	Liste des contraintes de la base <i>Social</i>	130

Liste des tableaux

2.1	Nomenclature des prédicats RDF/S en logique	15
3.1	Instance des jeux de données utilisée pour l'évaluation des performances	35
4.1	Méta-grammaire S-attribuée $\mathbb{G}$ au format BNF	56
5.1	Nombre d'entités par type dans le corpus d'entraînement	93
5.2	Nombre d'entités imbriquées dans le corpus d'entraînement	93
5.3	Liste des catégories thématiques du MeSH	96
5.4	Liste de lexèmes, leur classe associée et la liste des tokens obtenus avec la fonction <i>token</i>	98
5.5	Exemple des différents niveaux d'annotation pour la cascade de CRFs	99
5.6	Classes du chapitre C du MeSH	102
5.7	Résultat d'expérimentations supplémentaires sur l'impact des post-traitements.	103
5.8	Score obtenu pour l'extraction d'entités pour la recherche en langage naturel	109
5.9	Score obtenu pour l'extraction d'entités pour la construction de graphiques	110
B.1	Description de la base <i>Movie</i>	125
B.2	Description de la base <i>GOT</i>	127
B.3	Description de la base <i>Social</i>	129

Chapitre 1

Introduction

Sommaire

1.1	Un peu de contexte	1
1.2	Axes de recherche	2
1.3	Contributions et organisation du manuscrit	4
1.3.1	Publications	5
1.3.2	Logiciels	6

1.1 Un peu de contexte

L'action DOING réunit des chercheurs issus des domaines des bases de données, de l'intelligence artificielle et du traitement du langage naturel. Son objectif principal est de convertir les données en informations, puis en connaissances. Plus précisément, elle se concentre sur le stockage des informations dans des bases de données graphes. Les bases de données graphes sont privilégiées en raison de leur capacité à modéliser de manière efficace les relations complexes entre les données. Cette modélisation permet une exploitation efficace de ces relations en vue de les transformer en connaissances exploitables.

Les axes de discussion de DOING se concentrent sur deux problématiques principales :

- L'extraction d'informations à partir de données textuelles et leur représentation dans des bases de connaissances.
- Le développement de méthodes intelligentes pour le traitement et la maintenance de ces bases de données, incluant de nouvelles formes de requêtes pour l'analyse efficaces, flexibles et sécurisées adaptées à l'utilisateur, ainsi que des garanties de qualité et de protection de la vie privée.

Les travaux exposés dans cette thèse coïncident avec l'action DOING, le projet de recherche ayant étant initié simultanément à la mise en place de cette action. Les multiples collaborations entreprises dans le cadre de ces travaux sont principalement liées à cette action.

Par ailleurs, ce projet de recherche est mené en collaboration avec la société Ennov, un éditeur de logiciels spécialisé dans le secteur médical. Ennov propose une suite logicielle basée sur une Gestion Electronique de Documents (GED) conçue pour aider les entreprises pharmaceutiques et biomédicales à gérer leurs contenus, automatiser leurs processus métier et se conformer aux normes réglementaires internationales. Cette suite logicielle permet de réduire les risques, faciliter la conformité et améliorer la prise de décision. Ennov est un fournisseur majeur de solutions pour le Regulatory Information Management System (RIM/RIMS) et l'Identification of Medicinal Products (IDMP). Sa suite logicielle comprend des fonctionnalités de gestion de processus et de contenu pour la qualité, les affaires réglementaires, les études cliniques et la pharmacovigilance. Ennov est intéressé à offrir une meilleure automatisation des tâches répétitives, une visibilité accrue dans toute l'organisation et une facilité de mise en œuvre.

1.2 Axes de recherche

Le projet de recherche se focalise sur les bases de données graphes et s'intéresse en particulier aux données issues du domaine médical. Le projet s'articule autour de deux axes principaux.

Dans un premier temps, notre intérêt se focalise sur les bases de données contenant des informations incomplètes. L'incomplétude des données peut provenir de diverses sources. Dans le cadre de cette thèse, où l'exploitation des données textuelles est centrale, il est essentiel de prendre en compte cette incomplétude. On s'intéresse à la maintenance d'une base de données cohérente par rapport à un ensemble de règles. Dans cette optique, nous souhaitons suivre la politique de mise à jour proposée dans [CHL20]. Notre objectif est d'étudier la mise en œuvre de cette politique de manière efficace sur un modèle de données de type graphe. On se pose alors les questions suivantes : Comment imposer un ensemble de contraintes aux bases de données graphe ? Comment garantir la cohérence de ces bases tout au long de leur évolution ? Et comment implémenter cette politique en interaction avec des Système de Gestion de Base de Données (SGBD) ?

Bien qu'il existe des approches proposant des contraintes spécifiques pour les graphes, telles que celles présentées dans [FL19], nous avons opté pour une modélisation du graphe de données qui correspond à la définition logique de nos règles. Cela nous permet de mettre en œuvre la politique de [CHL20] en conservant le pouvoir d'expression des règles tout en améliorant les performances des opérations coûteuses, par rapport au modèle relationnel. La flexibilité intrinsèque des bases de données graphes nous permet en effet de représenter les mêmes données de différentes manières, permettant d'optimiser les interrogations subséquentes.

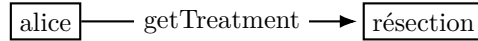
Exemple 1.2.1. Supposons l'existence des prédicats suivants : *getTreatment*, qui associe un patient à un traitement, *forPatho*, qui associe un traitement à la pathologie qu'il concerne et le prédicat *hasPatho*, indiquant qu'un patient a une pathologie. Dans cet exemple, nous pouvons considérer les règles suivantes :

$r_1 : \text{getTreatment}(x, y) \rightarrow \text{forPatho}(y, z)$

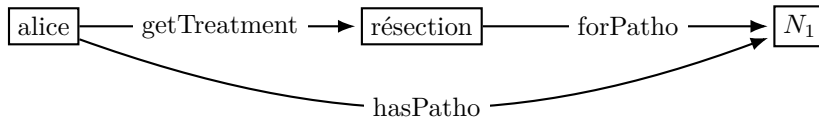
$r_2 : \text{getTreatment}(x, y), \text{forPatho}(y, z) \rightarrow \text{hasPatho}(x, z)$

La règle r_1 stipule qu'un traitement est obligatoirement lié à une pathologie, tandis que la règle r_2 établit que si un patient x reçoit un traitement pour une pathologie z , alors ce patient est atteint de cette pathologie ($\text{hasPatho}(x, z)$).

Considérons le graphe suivant représentant l'ensemble de faits *getTreatment*(alice, résection).



Dans ce contexte, le graphe ne satisfait pas la règle r_1 . Pour respecter cette règle, il faudrait ajouter le fait *forPatho*(résection, N_1). Remarquons que N_1 est une valeur nulle, ce qui signifie qu'une pathologie existe, mais sa valeur exacte est inconnue. Cependant, cet ensemble de faits ne respecterait pas non plus la règle r_2 . Pour obtenir un ensemble de faits cohérent avec les règles énoncées, il serait nécessaire de construire l'ensemble *getTreatment*(alice, résection), *forPatho*(résection, N_1), *hasPatho*(alice, N_1), représenté par le graphe suivant :



Le deuxième volet du projet de recherche se concentre sur les données textuelles, qui sont largement présentes, notamment dans les descriptions de cas cliniques et dans le domaine de la pharmacovigilance/cosmétovigilance. Les cas cliniques fournissent un historique détaillé de la santé d'un patient dès son admission à l'hôpital, comprenant ses antécédents médicaux, les symptômes observés, les diagnostics établis et les traitements recommandés. Quant à la pharmacovigilance, elle consiste à surveiller en continu les effets indésirables des médicaments après leur commercialisation, en se basant principalement sur les retours des patients ou des professionnels de santé. L'objectif consiste à rendre les informations contenues dans les textes exploitables en facilitant leur interrogation et leur analyse. Pour cela, il est nécessaire de structurer et d'agréger les données textuelles selon des abstractions générales qui correspondent à

des entités dans une base de données. Dans ce cadre, nous explorons les questions suivantes : Comment repérer et extraire les données pertinentes présentes dans le texte ? Comment structurer ces informations de manière à les rendre exploitables et enregistrables dans une base de données ? Et en particulier, quelles abstractions génériques (intermédiaires) nous permettront de passer d’une vision détaillée et multiforme représentée par le texte à une vision plus globale et uniforme requise par une base de données ?

Dans cette thèse, nous partons du principe que notre instance initiale est constituée d’un ensemble de textes qui respectent une grammaire. L’objectif est de faire évoluer cette grammaire en la généralisant vers une autre grammaire qui représente des abstractions structurantes correspondant à un schéma de données. Cette évolution est réalisée par une suite de transformations appliquée à l’instance.

Exemple 1.2.2. Prenons en compte le texte suivant, extrait d’un cas clinique du corpus CAS [GCD18] :

Un patient de 78 ans suivi pour cancer prostatique avec métastases ganglionnaires ayant déjà subi une résection endoscopique prostatique avec pulpectomie a été admis en urgence pour insuffisance rénale aiguë obstructive à 330 mmol/l de créatinine avec fièvre et urétérohydronéphrose bilatérale à l’échographie.

Notre méthode consiste à extraire les informations du texte, telles que les entités nommées. Par exemple, nous pouvons identifier des symptômes tels que « insuffisance rénale », « fièvre » ou encore « urétérohydronéphrose ». En supposant que les informations sont déjà structurées par le texte lui-même, notre objectif est de les restructurer afin d’obtenir une représentation uniforme. Par exemple, à partir du texte « Un patient de 78 ans », nous pourrions construire le fait $Patient(N_1, \text{homme}, 78)$. En continuant avec l’extrait de texte précédent, nous pourrions construire l’ensemble de faits suivants :

- | | |
|--|--|
| — $CasClinique(c-2-3)$ | — $getTreatment(p_1, \text{pulpectomie})$ |
| — $hasPatient(c-2-3, p_1)$ | — $passExam(p_1, e_1)$ |
| — $Patient(p_1, \text{homme}, 78)$ | — $measure(e_1, \text{créatinine}, 330 \text{ mmol L}^{-1})$ |
| — $hasPatho(p_1, \text{cancer})$ | — $reveal(e_1, \text{insuffisance rénale})$ |
| — $concernsAnat(\text{cancer}, \text{prostate})$ | — $passExam(p_1, e_2)$ |
| — $leadTo(\text{cancer}, \text{métastases ganglionnaires})$ | — $reveal(e_2, \text{fièvre})$ |
| — $getTreatment(p_1, \text{résection endoscopique})$ | — $passExam(p_1, \text{échographie})$ |
| — $concernsAnat(\text{résection endoscopique}, \text{prostate})$ | — $reveal(\text{échographie}, \text{urétérohydronéphrose})$ |

Ces informations peuvent être représentées par le graphe attribué présenté dans la figure 1.1. Notez que, dans cet exemple, le prédicat *measure* est promu comme attributs de la relation *exam*.

Remarque. Le graphe proposé ici n’est donné qu’à titre d’exemple et n’est qu’une des nombreuses représentations sous forme de graphe possibles. Par exemple, les relations *getTreatment* peuvent être décomposées en un nœud *Treatment* et *TreatmentType*. Cette représentation permettrait de séparer l’instance de son type et faciliterait la construction de relations autour du type. Un raisonnement similaire peut être appliqué pour les examens et les pathologies.

Il convient de noter que le choix des outils de Traitement Automatique des Langues (TAL) utilisés à cette étape, s’est fait en prenant en considération les contraintes spécifiques à notre domaine d’application :

- Les défis liés à la sensibilité des données médicales, qui imposent souvent des restrictions strictes sur leur accès et leur disponibilité. Ces restrictions visent à protéger la vie privée des patients et à prévenir tout accès non autorisé ou usage malveillant de ces informations cruciales pour la santé. Les modèles d’apprentissage automatique supervisé nécessitent un accès direct aux données pour être entraînés, ce qui peut compromettre la confidentialité des informations médicales des patients. Pour pallier cela, des solutions telles que l’apprentissage fédéré ou des approches symboliques plus classiques peuvent être mise en place.
- Dans les domaines sensibles comme le domaine médical, la nécessité de construire des solutions explicables est primordiale. L’aide à la prise de décision doit être appuyée par des méthodes claires et compréhensibles, surtout étant donné que les décisions prises par les systèmes informatiques peuvent avoir des conséquences directes sur la santé et la vie des patients. Une explication claire de la manière dont ces décisions sont prises est essentielle. Cela permet aux médecins et aux patients de comprendre pourquoi une recommandation spécifique a été faite. De plus, les approches explicables

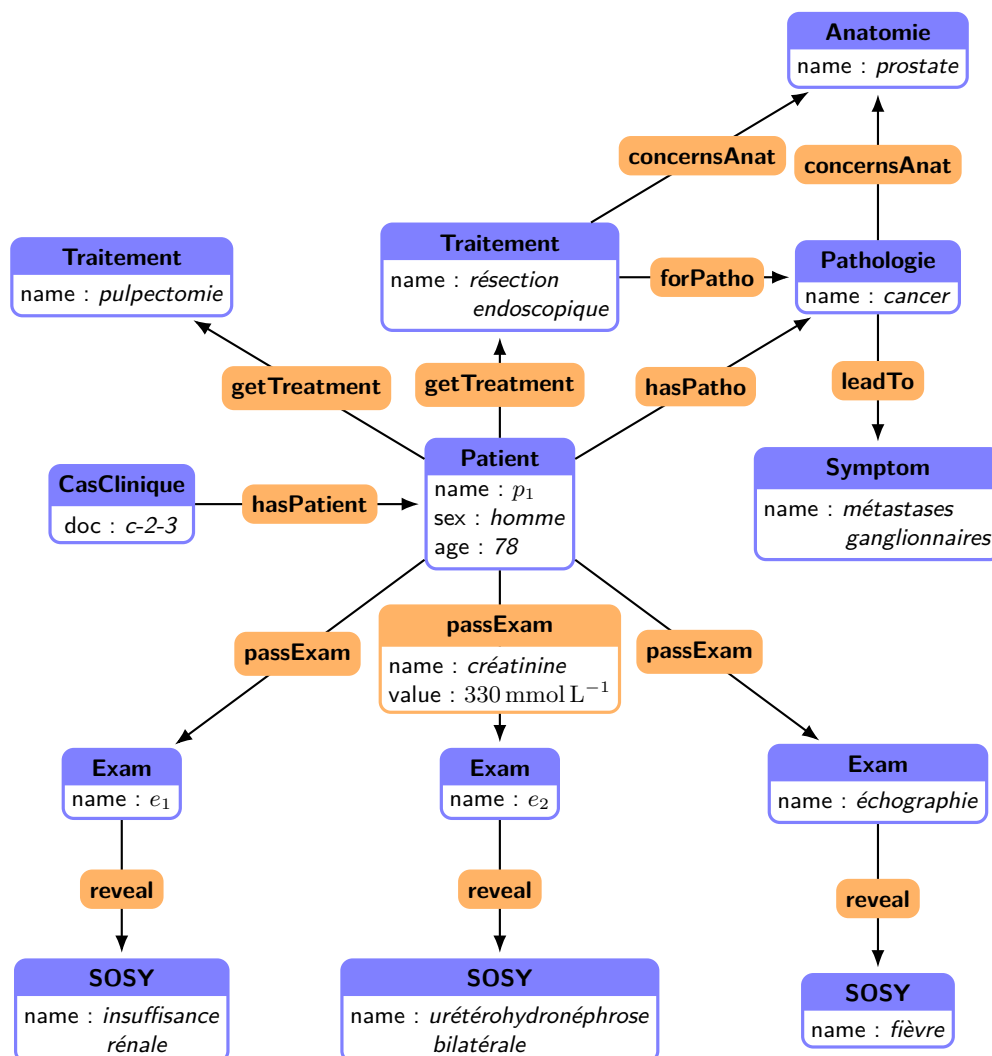


FIGURE 1.1 – Instance sous forme de graphe

permettent de retracer les décisions et de déterminer la partie du processus qui est responsable en cas d'erreur. Favoriser cette transparence contribue à créer une confiance dans le système et à maintenir l'utilisateur au centre du processus décisionnel.

1.3 Contributions et organisation du manuscrit

Les contributions importantes de nos travaux s'organisent de la manière suivante :

Maintenance et cohérence d'une base de données incomplète Présentée dans la partie I, la mise à jour cohérente des bases de données revêt une importance capitale pour garantir la qualité des données. Le chapitre 2 offre une vue d'ensemble de la notion de cohérence dans le contexte des bases de données. Alors que certaines approches se concentrent sur l'application de règles lors de l'interrogation, nous mettons l'accent sur la cohérence dès les mises à jour. Une base de données maintenue cohérente facilite l'analyse et l'interrogation en garantissant un niveau uniforme de qualité des données. Nous nous intéressons particulièrement aux bases de données graphes incomplètes, c'est-à-dire, celles comportant des valeurs nulles liées. Le chapitre 3 propose une méthode de mise à jour incrémentale reposant sur un SGBD graphe et qui garantit le respect des contraintes tout en assurant qu'il n'y a aucune redondance d'information. Ce chapitre présente également une suite d'expérimentations permettant de mesurer les performances de notre solution.

Intégration de données textuelles Nous nous concentrons sur la structuration des données textuelles en vue de l'enregistrement, dans une base de données graphe, des informations essentielles qu'elles apportent. En effet, pouvoir structurer et regrouper des informations de manière cohérente pour les maintenir dans un SGBD présente plusieurs avantages, notamment l'amélioration des capacités de recherche et la simplification de l'analyse des données. Le chapitre 4 présente une solution développée pour répondre à la problématique de structuration automatique des données textuelles. Plus précisément, nous nous penchons sur la définition d'une structure valide par rapport aux bases de données et sur les méthodes pour la construire automatiquement à partir de données textuelles. Pour finir, le chapitre propose une évaluation critique de la solution proposée.

L'extraction d'information (IE) est un domaine étroitement lié au TAL. Elle implique généralement l'identification d'entités nommées, leur résolution ou Entity Linking (c'est-à-dire l'association de la mention à un identifiant unique) et l'extraction de relations entre ces entités. Dans le chapitre 5, nous examinons plusieurs travaux réalisés en collaboration avec des chercheurs spécialisés dans le TAL. Nous présentons les différentes solutions développées dans cette thèse et mises en œuvre au sein d'Ennov. Il est important de souligner que notre intérêt pour les données textuelles découle de notre partenariat avec Ennov. Le domaine médical englobe une vaste gamme d'informations, allant des images aux modèles 3D, en passant par les rapports textuels sur les cas cliniques ou la pharmacovigilance. Cependant, l'écosystème d'Ennov est principalement axé sur les textes. Le chapitre 5 termine avec la présentation de deux cas d'application concrets illustrant l'application pratique des méthodes discutées.

1.3.1 Publications

Les travaux présentés dans cette thèse reposent sur une série de collaborations interdisciplinaires, notamment dans le domaine du TAL. Ils ont également donné lieu aux publications suivantes :

- [AHH20] Joshua Amavi, Mirian Halfeld-Ferrari, and Nicolas Hiot. Natural Language Querying System Through Entity Enrichment. In *ADBIS, TPDL and EDA 2020 Common Workshops and Doctoral Consortium*, volume 1260 of *Communications in Computer and Information Science*, pages 36–48, Lyon, France, August 2020. Springer International Publishing. doi:10.1007/978-3-030-55814-7_3.
- [CEHH20] Jacques Chabin, Cédric Eichler, Mirian Halfeld-Ferrari, and Nicolas Hiot. Graph Rewriting Rules for RDF Database Evolution Management. In *Proceedings of the 22nd International Conference on Information Integration and Web-based Applications & Services*, pages 134–143, New York, NY, USA, November 2020. Association for Computing Machinery. doi:10.1145/3428757.3429126.
- [CEHH21] Jacques Chabin, Cédric Eichler, Mirian Halfeld-Ferrari, and Nicolas Hiot. Graph rewriting rules for RDF database evolution : Optimizing side-effect processing. *International Journal of Web Information Systems*, 17(6) :622–644, December 2021. doi:10.1108/IJWIS-03-2021-0033.
- [CHHL23] Jacques Chabin, Mirian Halfeld-Ferrari, Nicolas Hiot, and Dominique Laurent. Managing Linked Nulls in Property Graphs : Tools to Ensure Consistency and Reduce Redundancy. In *Advances in Databases and Information Systems*, volume 13985 of *Lecture Notes in Computer Science*, pages 180–194, Cham, September 2023. Springer Nature Switzerland. doi:10.1007/978-3-031-42914-9_13.
- [HMB21] Nicolas Hiot, Anne-Lyse Minard, and Flora Badin. DOING@DEFT : utilisation de lexiques pour une classification efficace de cas cliniques. In *Traitement Automatique des Langues Naturelles*, pages 41–53, Lille, France, 2021. ATALA. URL : <https://hal.science/hal-03265924>.
- [MRH⁺20] Anne-Lyse Minard, Andréane Roques, Nicolas Hiot, Mirian Halfeld-Ferrari, and Agata Savary. DOING@ DEFT : cascade de CRF pour l'annotation d'entités cliniques imbriquées. In *6e conférence conjointe Journées d'Études sur la Parole (JEP, 33e édition), Traitement Automatique des Langues Naturelles (TALN, 27e édition), Rencontre des Étudiants Chercheurs en Informatique pour le Traitement Automatique des Langues (RÉCITAL, 22e édition). Atelier DÉfi Fouille de Textes*, pages 66–78, Nancy, France, June 2020. ATALA et AFCEP. URL : <https://hal.science/hal-02784743>.

-
- [SSM⁺22] Agata Savary, Alena Silvanovich, Anne-Lyse Minard, Nicolas Hiot, and Mirian Halfeld-Ferrari. Relation Extraction from Clinical Cases for a Knowledge Graph. In *New Trends in Database and Information Systems*, volume 1652 of *Communications in Computer and Information Science*, pages 353–365, Turin, Italy, August 2022. Springer International Publishing. doi: 10.1007/978-3-031-15743-1_33.

1.3.2 Logiciels

Ces travaux ont abouti à la création de logiciels et de jeux de données pour les expérimentations. Pour garantir la reproductibilité des résultats, tous les logiciels et jeux de données associés sont rendus disponibles publiquement :

- [CEHH20] Jacques Chabin, Cédric Eichler, Mirian Halfeld-Ferrari, and Nicolas Hiot. SetUp : Schema Evolution Through UPdates – SENDUP, 2020. URL : <https://www.univ-orleans.fr/lifo/evenements/sendup-project/index.php/software/setup-schema-evolution-through-updates/>.
- [CHH⁺23] Jacques Chabin, Mirian Halfeld-Ferrari, Nicolas Hiot, Dominique Laurent, and Lucas Moret-Bailly. DataFix, January 2023. URL : <https://gitlab.com/jacques-chabin/DataFix>.
- [CHH24] Jacques Chabin, Mirian Halfeld-Ferrari, and Nicolas Hiot. ArchiTXT, May 2024. URL : <https://github.com/Neplex/ArchiTXT>.

Première partie

Maintenance et cohérence d'une base de données incomplète

Chapitre 2

Maintien de la cohérence

Sommaire

2.1	Introduction	9
2.2	Préliminaires	11
2.2.1	Logique du premier ordre	11
2.2.2	Base de données	11
2.2.3	Base de données graphes	12
2.2.4	Requête sur les graphes de propriétés	14
2.3	Travaux connexes	14
2.3.1	SETUP : Grammaire de graphe pour la mise à jour de base RDFS	16
2.3.2	Dépendances fonctionnelles pour les graphes	18
2.4	Discussion	19

2.1 Introduction

Cette thèse se concentre sur la collecte d'informations provenant de diverses sources, notamment via des processus automatisés d'extraction d'informations à partir de textes. Ces informations peuvent être incomplètes, contenir des erreurs ou ne pas respecter dans la totalité un ensemble de règles métiers. L'objectif fondamental est d'intégrer ces nouvelles informations tout en veillant à ce qu'elles respectent les règles métiers prédéfinies. Lorsque ces règles sont respectées, on parle alors de *base de données cohérente* par rapport à un ensemble de contraintes. L'effort vise à maintenir la cohérence de la base de données à mesure que des mises à jour sont effectuées. Certaines approches se focalisent sur l'application de règles au moment de l'interrogation de la base de données. Toutefois, maintenir la cohérence dès les mises à jour réduit la nécessité de calculs fréquents lors de l'interrogation, car les mises à jour sont estimées moins régulières. De plus, une base de données cohérente offre une plus grande facilité d'analyse. Cela garantit un niveau de qualité de données uniforme pour tous les utilisateurs, même si cela implique une certaine limitation de la flexibilité. En effet, contrairement aux approches qui visent à appliquer un ensemble de contraintes lors de la requête, les contraintes imposées ne sont plus adaptées pour chaque utilisateur mais sont fixées et ne peuvent pas être modifiées. Cela permet tout de même d'avoir un ensemble de contraintes pour chaque exigence métier. Dans ce contexte, cette thèse s'intéresse au maintien de la cohérence (par rapport à des contraintes globales) lors des mises à jour d'une base de données avec des informations incomplètes.

L'importance de pouvoir gérer l'incomplétude se révèle dans l'intégration d'informations provenant de sources textuelles. Dans ces travaux, l'incomplétude prend la forme de valeurs manquantes. Dans le contexte des textes, cela peut être dû à des valeurs non observées, à des omissions liées à une connaissance générale ou simplement à des erreurs d'extraction. Pour l'intégration de bases de données, les valeurs manquantes peuvent résulter d'erreurs de saisie, de données confidentielles ou du fait que certaines informations ne couvrent pas toute l'étendue de notre base. Ainsi, lorsque l'on intègre une base de médicaments ou de patients, les interactions entre les entités, telles que définies par les règles métiers,

peuvent ne pas être explicitées. À titre d'exemple, une contrainte exprimée par la Tuple-Generating Dependency (TGD) $c1 : Pat(x), SOSY(x, y) \rightarrow PrescExam(x, z)$ stipule que si un patient x présente un symptôme y , alors il doit avoir subi un examen z qui a mis en évidence ce symptôme. Ainsi, lorsque la base contient une entrée pour une patiente nommée Lea souffrant de douleurs aux mains ($Pat(Lea)$ et $SOSY(Lea, douleur\ aux\ mains)$), pour être cohérente, la base de données doit également inclure l'atome $PrescExam(Lea, N_1)$. Ici, N_1 représente une valeur inconnue et cet atome indique qu'un examen a été prescrit à Lea, même si l'examen n'est pas encore connu. Reiter propose une sémantique pour les valeurs nulles en utilisant la logique du premier ordre (FOL). Il les décrit comme des individus existants mais inconnus « [...] these nulls represent existing but unknown individuals » [Rei86, p.1]. Ces valeurs inconnues peuvent alors être représentées par des valeurs nulles marquées (ou nommées), permettant leur référencement multiple. Pour illustrer, prenons l'exemple d'une instance de base de données $\mathcal{D}_1 = \{PrescExam(Lea, N_1), ExamResult(Lea, N_1, N_2)\}$. Cette instance indique qu'un examen non spécifié, symbolisé par N_1 , a été prescrit à Lea, et qu'un résultat inconnu, représenté par N_2 , est associé. Ces valeurs nulles reflètent une information partielle et se conforment à l'approche du Closed World Assumption (CWA) où toute information non présente est considérée comme fausse. Dans le contexte de l'Open World Assumption (OWA), la sémantique diffère : les valeurs inconnues sont simplement omises. La logique sous-jacente est à trois valeurs de vérité, vrai, faux, ou inconnu si le fait est manquant dans la base. Les travaux présentés dans cette thèse se focalisent sur le CWA, qui facilite l'analyse des données.

Un autre défi inhérent à la mise à jour réside dans la nécessité d'éviter la redondance d'information tout en maintenant la cohérence. Pour illustrer, considérons l'ajout de la nouvelle information $PrescExam(Lea, x-ray)$ à l'instance $\mathcal{D}_1$ mentionnée précédemment. Dans cette situation, il peut être dangereux de substituer N_1 par $x-ray$ dans $PrescExam(Lea, N_1)$, car N_1 apparaît également dans $ExamResult(Lea, N_1, N_2)$, indiquant ainsi une dépendance liée à N_1 . Le remplacement de N_1 par $x-ray$ impliquant alors la perte du lien entre les deux atomes ou la construction d'une information qui n'est pas fournie par un utilisateur. Cependant, si nous recevons par la suite l'information $ExamResult(Lea, x-ray, join\ inflammation)$, alors il serait raisonnable de remplacer N_1 par $x-ray$, car l'instanciation de N_1 demeure identique dans tous les atomes où elle est présente. En d'autres termes, l'information partielle qui précise que « Lea a subi un examen pour lequel un résultat a été obtenu » se révèle redondante avec l'information détaillée « Lea a subi un examen de type radiographie, ayant comme résultat la détection d'une inflammation ». Dans ce scénario, l'instance obtenue se présenterait comme $\mathcal{D}'_1 = \{ExamResult(Lea, x-ray, join\ inflammation), PrescExam(Lea, x-ray)\}$.

Contributions Basée sur le cadre conceptuel de FOL exposé dans [CHL20], les travaux présentés ici :

1. visent à appliquer cette approche de manière *incrémentale* à des *bases de données graphes*.
2. capitalisent sur les avantages inhérents aux fonctionnalités d'accès, de manipulation et de gestion de données offertes par les Systèmes de Gestion de Base de Données (SGBDs) orientés graphes contrairement à une version en mémoire présentée dans [CHL20].
3. améliorent la création des ensembles d'atomes liés par des valeurs nulles pour le calcul du Core, processus qui permet d'éliminer la redondance d'information et constitue une étape coûteuse dans [CHL20].

Organisation de la partie Pour commencer, la section 2.2 définit les préliminaires et la section 2.3 présente des travaux connexes et expose les distinctions entre l'approche préconisée ici et celle décrite dans [CHL20]. Dans la suite, le chapitre 3 présente en détail l'approche proposée et une évaluation critique illustrant les avantages et les limites de l'exploitation des SGBDs graphes est exposée dans la section 3.6, avant de conclure cette partie.

2.2 Préliminaires

Cette section constitue une vue d'ensemble des concepts et des notions fondamentales qui serviront de base aux chapitres suivants. Dans un premier temps, nous présentons un rappel de certains concepts de la logique du premier ordre (FOL) ainsi que la définition des contraintes et des règles dans ce contexte. Par la suite, nous aborderons les bases de données en adoptant le formalisme FOL (se référer à [AHV95] pour plus d'informations sur cet aspect) et offrirons une brève introduction aux bases de données graphes.

2.2.1 Logique du premier ordre

En FOL, l'information est représentée par un ensemble d'atomes. Dans cette thèse, on utilise l'alphabet $\mathcal{A}$ qui est composé par :

- un ensemble de prédicats $P, Q, R, \dots$
- un ensemble de variables $x, y, z, \dots$
- un ensemble de constantes $a, b, c, \dots$
- un ensemble de valeurs nulles marquées $N_1, N_2, N_3, \dots$
- le quantificateur universel $\forall$ (quel que soit)
- le quantificateur existentiel $\exists$ (il existe)

Definition 2.2.1 (Atome). À partir de l'alphabet $\mathcal{A}$, on définit une formule atomique (ou atome) $P(t_1, \dots, t_n)$ où P est le symbole de prédicat d'arité n et $t_1, \dots, t_n$ sont les termes. Les prédicats représentent les relations entre les termes qui peuvent être des variables, des constantes ou des valeurs nulles. Un fait est un atome composé exclusivement de constantes, tandis qu'un atome instancié est un atome ne contenant aucune variable. Étant donné un atome instancié A , $null(A)$ représente l'ensemble des valeurs nulles de A .

Par exemple, le fait $auteur(Alice)$ comprend le symbole de prédicat $auteur$ d'arité 1 et la constante $Alice$. Si l'on suppose que chaque symbole de prédicat détient une signification (ici, intuitivement claire), ce fait indique que $Alice$ est un auteur.

Definition 2.2.2 (Homomorphisme). Un homomorphisme h entre deux ensembles d'atomes A et B est une application surjective des termes de A vers les termes de B , telle que : (i) si, t est une constante, alors $h(t) = t$; (ii) si, $P(t_1, \dots, t_n) \in A$, alors $P(h(t_1), \dots, h(t_n)) \in B$. S'il existe un homomorphisme inverse h^{-1} tel que $h^{-1}(h(t)) = t$ alors on dit que A est *isomorphe* à B .

Definition 2.2.3 (Simplification). Soit ϕ une formule existentiellement quantifiée de la forme $(\exists X)(\varphi_1(X_1) \wedge \dots \wedge \varphi_n(X_n))$ où X est l'ensemble des variables, $\varphi_i(X_i)$ est une formule atomique et $X_i \subset X$ représente l'ensemble des variables libres dans φ_i . $atoms(\phi)$ est l'ensemble des formules atomiques de ϕ et Φ est l'ensemble de toutes les formules ϕ . Un modèle M d'une formule ϕ est l'ensemble des faits qui ont un homomorphisme de $atoms(\phi)$ vers M . Étant donné deux formules ϕ_1 et ϕ_2 : $\phi_1 \implies \phi_2$ si tout modèle de ϕ_1 est un modèle de ϕ_2 . ϕ_1 et ϕ_2 sont *équivalents*, noté $\phi_1 \iff \phi_2$, si et seulement si $\phi_1 \implies \phi_2$ et $\phi_2 \implies \phi_1$. ϕ_1 est *plus simple que* ϕ_2 , noté $\phi_1 \sqsubseteq \phi_2$ si $\phi_1 \iff \phi_2$ et $atoms(\phi_1) \subseteq atoms(\phi_2)$. Une simplification ϕ_1 de ϕ_2 est *minimale* si $\phi_1 \sqsubseteq \phi_2$ et qu'il n'existe pas de ϕ'_1 tel que $\phi'_1 \sqsubset \phi_1$.

Par exemple, étant donné la formule $\phi = (\exists x, y)(P(a, x) \wedge P(a, y))$, les formules $(\exists x)(P(a, x))$ et $(\exists y)(P(a, y))$ sont équivalentes et sont toutes deux une simplification de ϕ . De plus, [CHL20] démontre que si ϕ_1 et ϕ_2 sont deux simplifications minimales d'une même formule ϕ alors les ensembles $atoms(\phi_1)$ et $atoms(\phi_2)$ sont isomorphes.

Definition 2.2.4 (Core). Une simplification minimale d'une formule ϕ est appelé le Core et s'écrit $core(\phi)$.

2.2.2 Base de données

Definition 2.2.5 (Instance). L'instance d'une base de données $\mathcal{D}$ est définie comme la Skolémisation d'une formule ϕ , représentant un ensemble minimal d'atomes instanciés, pour lequel $core(\phi) = \phi$. La Skolémisation d'une formule ϕ , notée $Sk(\phi)$, revient à substituer l'ensemble des variables par des constantes ou des valeurs nulles, tout en éliminant chaque occurrence du quantificateur existentiel.

Définition 2.2.6 (Requête). Une *requête* est une formule logique de la forme $(\forall X)(Q(X) \leftarrow \phi(X))$, où X représente l'ensemble des variables libres de ϕ . La réponse à une requête effectuée sur une base de données $\mathcal{D}$ correspond à l'ensemble des homomorphismes h tels que $h(\phi(X)) \subseteq \mathcal{D}$.

Définition 2.2.7 (Contrainte). Une *contrainte* (ou règle) sur une base de données est une dépendance génératrice de n-uplets, également appelée Tuple-Generating Dependency (TGD) en anglais, de la forme $(\forall X, Y)(body(X, Y) \rightarrow (\exists Z)head(X, Z))$. Plus simplement, nous écrivons $body(X, Y) \rightarrow head(X, Z)$, où X, Y et Z représentent des ensembles de variables, $body(X, Y)$ est un ensemble d'atomes et $head(X, Z)$ est un atome. Une contrainte c est dite satisfaite par un ensemble I d'atomes instanciés, noté $I \models c$ si pour tout homomorphisme h tel que $h(body(c)) \subseteq I$ il existe un homomorphisme h' tel que $h'(body(c)) = h(body(c))$ et $h'(head(c)) \in I$. Une instance I d'une base de données est dite *cohérente* vis-à-vis d'un ensemble de contraintes $\mathbb{C}$, noté $I \models \mathbb{C}$ si pour toute contrainte c dans $\mathbb{C}$, $I \models c$.

Intuitivement, une contrainte est satisfaite si pour toute instanciation du corps présente dans la base de données, la tête correspondante doit aussi être présente dans la base. Par exemple, la contrainte exprimée par la règle $\forall x \text{ auteur}(x) \rightarrow \exists y \text{ écrit}(x, y)$ signifie que pour chaque individu x considéré comme auteur ($\text{auteur}(x)$) dans l'ensemble de fait, celui-ci doit également contenir au moins une œuvre y écrite par cet auteur ($\text{écrit}(x, y)$). Dans ce contexte, l'ensemble $I_1 = \{\text{auteur}(\text{Alice})\}$ n'est pas *cohérent* contrairement aux ensembles $I_2 = \{\text{auteur}(\text{Alice}), \text{écrit}(\text{Alice}, \text{Pays des merveilles})\}$, $I_3 = \emptyset$ et $I_4 = \{\text{écrit}(\text{Alice}, N_1)\}$.

2.2.3 Base de données graphes

Dans cette section, nous visons à fournir une introduction concise aux bases de données graphes. [Pok15] offre une vue d'ensemble des avantages et des inconvénients de ce type de bases de données. Lorsque nous évoquons les bases de données graphes, nous pouvons identifier deux grandes familles :

Les bases de triplets Notamment représentées par le modèle Resource Description Framework (RDF) développé par le World Wide Web Consortium (W3C). Ces bases enregistrent des triplets sous la forme $\langle x, P, y \rangle$, où chaque triplet représente un atome $P(x, y)$ et est constitué de trois éléments : x , le sujet représenté par l'Uniform Resource Identifier (URI) d'une ressource (un nœud), P est un symbole de prédicat, et y , l'objet, peut être soit un URI, soit un littéral (une valeur de propriétés). RDF est interrogé en utilisant le langage Simple Protocol and RDF Query Language (SPARQL).

Les Graphes de Propriétés Aussi connus sous le nom de Labeled Property Graph (LPG) en anglais, sont des bases de données composées de nœuds reliés par des arcs. Les deux sont labellisés et peuvent contenir des propriétés (ou attributs). Parmi les SGBD de ce type, Neo4J est l'un des plus populaires. Notamment, Neo4J est à l'origine du langage de requête Cypher, qui sert de fondation au nouveau langage du standard ISO : Graph Query Language (GQL), actuellement en cours de développement.

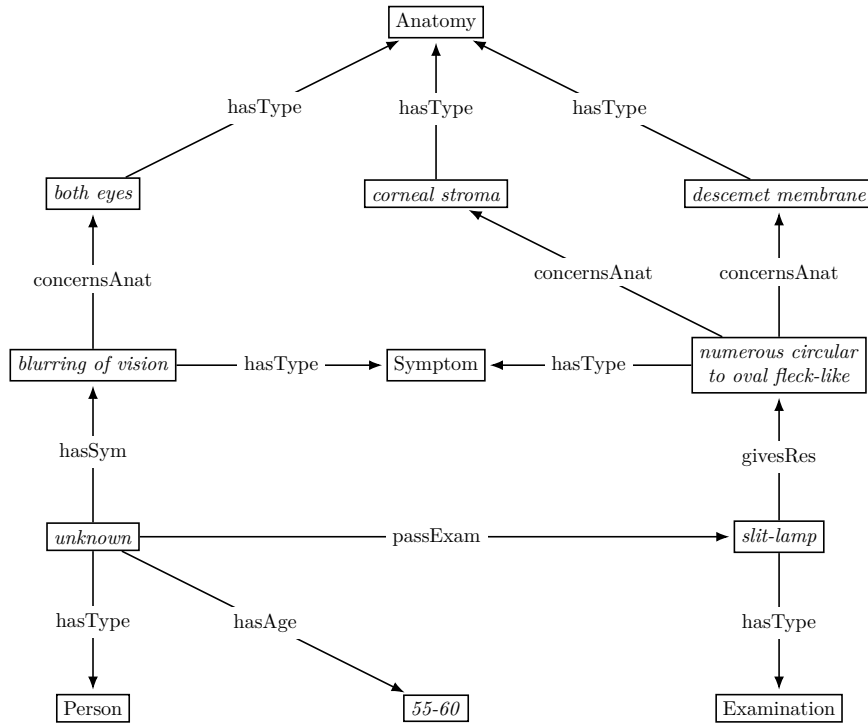
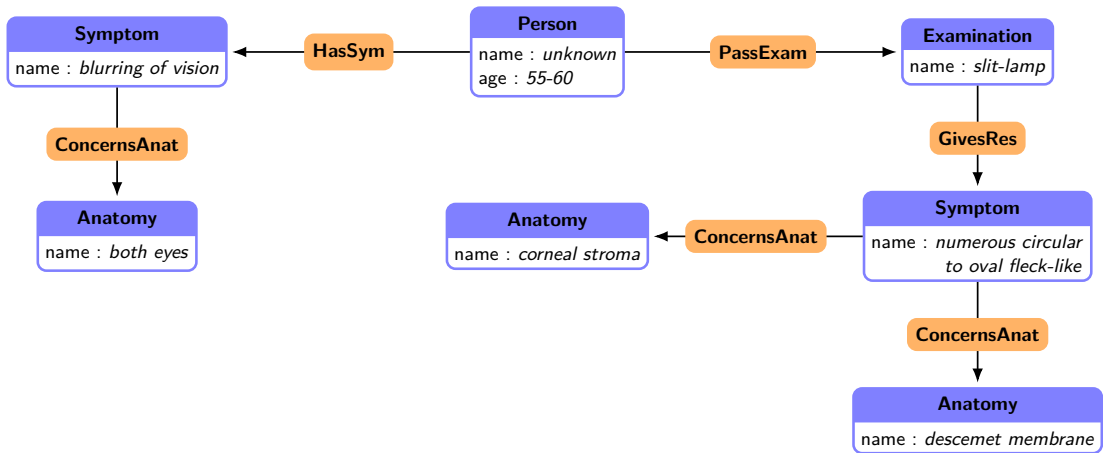
La figure 2.1 offre une comparaison visuelle de ces différents types de bases. La figure 2.1a présente l'instance d'une base de données relationnelle. Cette même instance est représentée dans la figure 2.1b sous la forme d'un graphe RDF (où seule des nœuds et des arcs sont présents). La figure 2.1c représente l'instance sous la forme d'un graphe de propriétés où les nœuds et les arcs peuvent contenir des attributs. Les bases de données graphes sont spécialement conçues pour simplifier les requêtes sur la topologie du graphe. Le modèle de la base de données (ou schéma, pour faire une analogie avec le modèle relationnel, bien qu'aucune règle structurelle ne soit imposée) est étroitement lié aux types de requêtes envisagées. Dans le cadre de Neo4J, par exemple, les nœuds peuvent être étiquetés avec une ou plusieurs étiquettes, appelées *labels*, tandis que les relations ne peuvent en porter qu'une, dénommée *type* (si plusieurs étiquettes sont nécessaires pour une relation, cette dernière sera dupliquée).

Resource Description Framework Schema (RDFS) fournit des relations pour la définition d'ontologies (ou vocabulaires) destinées à structurer des ressources RDF. Il permet de définir le schéma de la base : les classes (correspondant aux types de nœuds) et les propriétés (correspondant aux arêtes). De plus, il offre la possibilité de spécifier des relations de subordination et de subsomption en établissant une hiérarchie entre les éléments au moyen des propriétés `subClass` et `subProp`. RDFS permet également la séparation de l'instance et du schéma.

Anatomy		Symptom		ConcernsAnat		Examination	
id	name	id	name	symptom	anatomy	id	name
0	<i>both eyes</i>	0	<i>blurring of vision</i>	0	0	0	<i>slit-lamp</i>
1	<i>corneal stroma</i>	1	<i>numerous circular to oval fleck-like</i>	1	1		
2	<i>descemet membrane</i>			1	2		

Person			GivesRes		PassExam		HasSym	
id	name	age	exam	symptom	person	exam	person	symptom
0	<i>unknown</i>	<i>55-60</i>	0	1	0	0	0	0

(a) Instance au format relationnel

(b) Instance sous forme de graphe RDF (la propriété *name* est directement affiché dans le nœud individu pour des raisons de lisibilité)

(c) Instance sous forme de graphe de propriétés

FIGURE 2.1 – Exemple d'instance d'une base de données graphe et son équivalent en relationnel

2.2.4 Requête sur les graphes de propriétés

Dans cette partie, notre intérêt se porte principalement sur les graphes de propriétés. Cette section propose une brève introduction à l'interrogation de ces graphes. Cypher est un langage déclaratif de requête conçu pour les graphes de propriétés. Ce langage offre la facilité d'exprimer des requêtes en utilisant le mot-clé **MATCH** pour la recherche de motifs, exprimés sous forme d'art ASCII. Les requêtes sont formulées comme une séquence d'opérations, où le résultat d'une opération devient le paramètre de la suivante. Même si les requêtes portent sur des graphes, les résultats des opérations sont toujours une table d'affectation des variables (avec autant de colonnes que de variables et autant de lignes que de réponses).

Exemple 2.2.1. En considérant l'instance de base de donnée graphe représentée dans figure 2.1c, la récupération de l'ensemble des noms de partie du corps (*Anatomy*) présents dans l'instance s'écrit, en Cypher : **MATCH (x:Anatomy) RETURN x.name**. Le **MATCH** recherche le motif **(x:Anatomy)** où les parenthèses représentent un nœud, **x** est le nom de la variable pour ce nœud et **:Anatomy** indique que l'on cherche uniquement les nœuds qui ont l'étiquette (ou label) *Anatomy*. L'opération **RETURN** termine la requête et renvoie la table d'instanciation suivante :

	x.name
0	<i>both eyes</i>
1	<i>corneal stroma</i>
2	<i>descemet membrane</i>

La requête **MATCH (x:Examination)-[y:GivesRes]->() RETURN x, y** retourne l'ensemble des nœuds labellisés *Examination* qui sont dans une relation *y* ayant le type *GivesRes* représenté par **-[y:GivesRes]->**. Ici, la relation est dirigée indiquant que l'on cherche uniquement les nœuds *Examination* qui sont source de la relation. Le nœud de destination de la relation n'a aucune contrainte et est donc représenté uniquement par **()**. Le langage Cypher permet également de construire des requêtes récursives. Par exemple, étant donné une relation **knows** entre des personnes indiquant quelles se connaissent, pour obtenir l'ensemble des groupes de personnes où chaque membre connaît au moins une autre personne du groupe, on peut écrire en Cypher : **MATCH (x:Person)-[:knows*]-(y:Person) RETURN COLLECT(DISTINCT y) AS persons**. Dans l'expression **-[:knows*]-**, le symbole ***** indique que la relation peut être traversée plusieurs fois de manière récursive. La requête est toujours assurée de se terminer, car Neo4J ne permet pas de traverser la même arête à plusieurs reprises pour un même motif. L'opération **COLLECT(DISTINCT y)** permet de regrouper les instanciations de la variable **y** en fonction des autres variables (équivalent à l'opération **group by** en Structured Query Language (SQL)), ici **x**, en évitant les doublons (**DISTINCT**). L'ensemble est retourné sous forme d'une liste. Neo4J autorise l'utilisation de structure de données (liste, dictionnaire, etc) comme variables dans les requêtes, mais aussi comme valeur de propriété. Il s'agit d'une différence importante avec le modèle relationnel et les graphes RDF qui autorisent uniquement des types simples (chaîne de caractères, entier, etc).

2.3 Travaux connexes

Assurer la validité, la cohérence et l'absence de redondance des données demeure un enjeu crucial et constitue un domaine de recherche actif. La mise à jour contrôlée par un ensemble de contraintes est souvent négligée en faveur de l'efficacité, ce qui peut compromettre la validité des informations dans un cadre réglementaire. Cela peut aussi réduire la qualité des analyses effectuées sur les données qui est un facteur important pour le reporting et l'intelligence artificielle. Les recherches se concentrent principalement sur l'interrogation des données, tandis que la problématique de la mise à jour cohérente des bases de données reste complexe et d'une grande importance [HLS98, ST98, LS02]. Des efforts récents ont été entrepris pour apporter des solutions à cette question dans divers contextes. Des propositions récentes [ACP11, FKAC13, HHU17, HL17], spécifiquement adaptées au monde RDF, examinent les contraintes d'un point de vue conventionnel sur les bases de données, contrairement à la sémantique standard du Web où les contraintes sont simplement des règles d'inférence [LMS08, GOP11, Pat15].

Toutefois, la prise en compte de l'incomplétude des données dans des modèles non relationnels demeure limitée, comme indiqué par [Sir14]. Concernant les bases de données graphes, il convient de souligner une distinction fondamentale. À la différence des bases de données relationnelles, elles opèrent souvent sur

l'hypothèse que les données ne sont pas structurées par un schéma préétabli ni soumises à des contraintes d'intégrité. Toutefois, plusieurs efforts ont été déployés pour pallier cette limitation. Dans ce contexte, [FFTD15] introduit le concept de clés pour les nœuds d'un graphe, établissant ainsi des équivalences semblables aux clés primaires dans le modèle relationnel. De la même manière, [RBP⁺18] aborde la détection d'équivalences entre nœuds dans un graphe RDF, déterminant ces équivalences par la comparaison des propriétés et des valeurs associées à chaque nœud. L'objectif est de préserver la cohérence des relations d'équivalence `sameAs`. Ultérieurement, [FL19] propose le concept de Graph Entity Dependencies (GED), équivalent aux dépendances fonctionnelles, mais appliquées aux graphes. Ces concepts sont approfondis dans [ABD⁺21, BFV⁺18], offrant ainsi une solide introduction à ce sujet. Par ailleurs, [MRGL14] propose une mesure de cohérence sémantique entre un sous-graphe issu d'une base RDF d'origine et un sous-graphe résultant d'une mise à jour. L'idée sous-jacente est d'évaluer si une mise à jour contribue positivement à la cohérence globale de la base.

Dans une base de données graphe, le schéma et l'instance sont intimement liées, ce qui peut rendre complexe la définition et le maintien de la cohérence. Par exemple, l'insertion de *Pat(Lea)* dans un graphe RDF présuppose que *Pat* est une classe existante. Si cela n'est pas le cas, une politique de mise à jour peut engendrer, comme conséquence, l'insertion de cette classe. Autrement dit, une mise à jour de l'instance peut impacter le schéma. Dans ce cadre, [FKAC13] propose une modélisation de RDFS en utilisant FOL. Ces travaux s'appuient sur une formalisation de la sémantique liée à RDFS et définissent des règles d'intégrité visant à garantir la cohérence d'une base RDFS. Cette approche introduit des symboles de prédicat (présenté dans la table 2.1) pour traduire la nomenclature RDF vers le formalisme logique.

RDFS	Prédicats logiques
$\langle x, \text{rdf:type}, \text{rdfs:Class} \rangle$	$Cl(x)$
$\langle x, \text{rdf:type}, \text{rdfs:Property} \rangle$	$Pr(x)$
$\langle x, \text{rdf:type}, \text{rdfs:Resource} \rangle$	$Ind(x)$
$\langle x, \text{rdf:type}, y \rangle$	$CI(x, y)$
$\langle x, \text{rdf:subClassOf}, y \rangle$	$CSub(x, y)$
$\langle x, \text{rdf:subPropertyOf}, y \rangle$	$PSub(x, y)$
$\langle x, \text{rdf:domain}, y \rangle$	$Dom(x, y)$
$\langle x, \text{rdf:range}, y \rangle$	$Rng(x, y)$

TABLE 2.1 – Nomenclature des prédicats RDF/S en logique

Dans ce même contexte, [GMR13] porte l'attention sur les contraintes sémantiques de RDFS uniquement au niveau de l'instance. Les approches présentées dans [FKAC13] et [CEH19] mettent en œuvre des contraintes sous la forme de TGDs, définies par les utilisateurs. D'autres travaux tels que [HHU17] se fondent également sur les contraintes de RDFS, mais ils se concentrent sur une forme simplifiée des TGDs. Les valeurs nulles, représentant des informations incomplètes, sont explorées dans les travaux de [HHU17] et [CEH19]. D'autres aspects tels que l'évolution de schéma sont abordés par [FKAC13] et [GMR13]. [Mah15] et [MFTH15] traitent de l'évolution des ontologies Web Ontology Language (OWL) dans un contexte de monde ouvert (OWA). Cependant, l'approche proposée dans ces travaux ne permet pas le calcul des propriétés transitives et l'utilisation du monde ouvert ne permet pas de garantir la préservation de toutes les contraintes établies. Enfin, [CEH19] se penche sur la sémantique de RDFS et propose de donner la priorité aux mises à jour. Cela signifie que l'approche vise à modifier à la fois l'instance et le schéma pour accommoder les nouvelles informations de manière cohérente. L'objectif de [FKP05], [Got05] et [PS11] est d'obtenir l'ensemble minimal d'une instance de base de données dans le contexte de l'intégration des données. Ils étudient aussi la complexité du calcul de cet ensemble minimal, le Core.

La mise à jour de données incomplètes avec des valeurs nulles marquées reste un problème ouvert [Lib14, CGLT20]. Cette problématique persiste en partie parce que les SGBDs relationnels ne mettent pas en œuvre les modèles théoriques en raison d'un manque de consensus dans ce domaine. De nombreux travaux ont tenté de définir la sémantique des valeurs nulles dans le contexte du modèle relationnel. Parmi ces travaux, [ILJ84] propose deux systèmes pour gérer les informations incomplètes. Le premier s'appuie sur les valeurs nulles du modèle relationnel de Codd [Cod70], tandis que le second traite des valeurs nulles marquées qui peuvent apparaître dans plusieurs tables. Par ailleurs, bien que des bases solides ont été établies pour traiter l'incomplétude des bases de données relationnelles pour l'interrogation

[Zan84, ILJ84, FKUV86, Rei86, Gra91, BB06, Lib16, NK16, GL17], les mises à jour de bases de données incomplètes ont fait l'objet de beaucoup moins d'attention [AG85, FUV83, Win88, Win90, DGLPR09, DGLO⁺17, CHL20].

Dans la suite, nous approfondissons la description de deux travaux, dont une analyse détaillée nous permettra d'élaborer une discussion dans la section 2.4 afin de situer les travaux présentés dans cette partie :

Schema Evolution Through UPdates (SETUP) correspondant à ma première expérience de travail sur les mises à jour des graphes. Il s'agit d'une approche originale basée sur les grammaires de graphes et offrant une solution pour maintenir la cohérence des instances RDFs en conformité avec la sémantique énoncée par [FKAC13].

Graph Entity Dependencies (GED) une définition des contraintes fonctionnelles sur les graphes de propriétés proposée par [FL19].

2.3.1 SETUP : Grammaire de graphe pour la mise à jour de base RDFS

[CEH19, CEHH20] définit un cadre formel pour garantir la cohérence d'un graphe de connaissances RDF. L'approche introduit SETUP, une méthode de maintenance des graphes RDF qui repose sur la réécriture de graphes. L'objectif est de maintenir le respect des contraintes d'intégrité du modèle RDFS tel qu'expliqué dans [FKAC13], [HHU17] ou encore [CHL20]. Les règles de la sémantique RDFS sont formulées sous forme de règles logiques en utilisant les symboles de prédicats définis dans la table 2.1. Par exemple, la règle $(\forall x, y) CI(x, y) \rightarrow Cl(y)$ exprime que pour qu'un élément x soit une instance de y , y doit être une classe. Pour modéliser ces règles de la sémantique, [CEH19] propose l'utilisation de la réécriture de graphes, illustrée par la figure 2.2. Les règles ne peuvent être appliquées que si leur application préserve la cohérence du graphe.

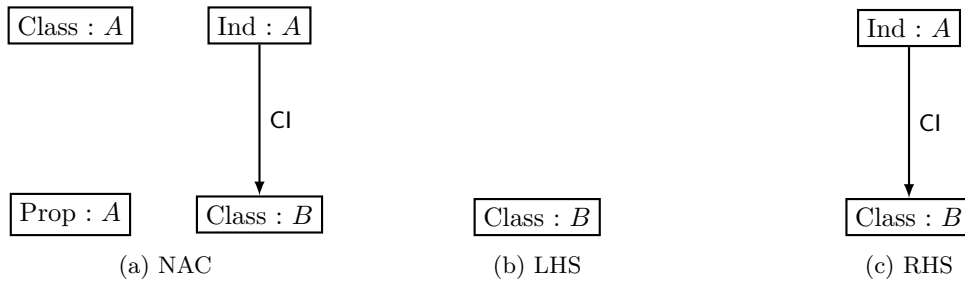


FIGURE 2.2 – Règle de réécriture simplifiée pour l'insertion du fait $CI(A, B)$

Les règles de réécriture sont des outils qui permettent de définir et de manipuler la structure des graphes en spécifiant comment le graphe doit évoluer selon certaines conditions. Elles prennent la forme $LHS \rightarrow RHS$, où Left-Hand Side (LHS) représente un motif dans le graphe et Right-Hand Side (RHS) est le résultat de l'application de la règle. Pour effectuer la réécriture, un morphisme est défini entre LHS et RHS, indiquant comment la règle doit être appliquée. Par exemple, dans la figure 2.2, un morphisme existe sur la classe B . Une règle de réécriture est applicable si un isomorphisme existe entre LHS et le graphe G que l'on construit. En d'autres termes, si le motif spécifié dans LHS se trouve dans G , la règle peut être appliquée pour générer RHS. Cependant, il est possible d'introduire des Negative Application Conditions (NAC), qui permettent de définir des motifs qui, s'ils sont trouvés dans le graphe G , rendent la règle inapplicable. Dans la figure 2.2, les NACs sont utilisées pour éviter l'insertion du fait s'il est déjà présent dans la base ou si le type de A n'est pas le bon.

Exemple 2.3.1. En prenant en compte une base de connaissance cohérente représentée par le graphe G (illustré dans la figure 2.3) et l'insertion du fait $CI(A, B)$, on peut observer qu'en tentant d'appliquer la règle de réécriture présentée dans la figure 2.2, la règle ne peut pas être directement mise en œuvre, car la condition de la NAC est satisfaite (l'élément A est une classe). Pour résoudre cette situation, une intervention supplémentaire est nécessaire pour supprimer le fait concerné (indiqué en rouge). Une fois cette intervention effectuée, la règle de réécriture devient applicable, conduisant à la création de nouveaux

éléments conformément à la règle (représentés en bleu). Cette intervention, appelée *effet de bord*, est une action préalable qui vise à modifier l'état du graphe de manière à permettre l'application de la règle de réécriture.

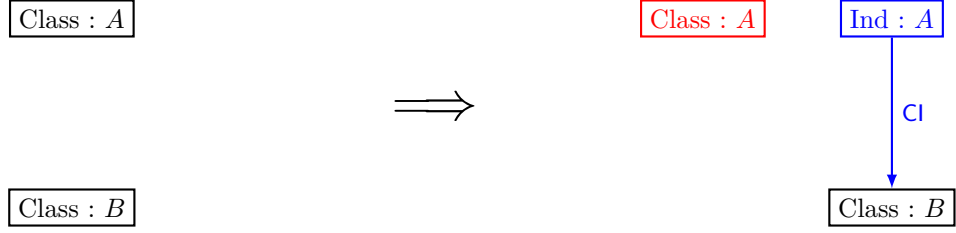


FIGURE 2.3 – Exemple d'ajout du fait $CI(A, B)$

SETUP donne priorité aux mises à jour, en définissant un ensemble d'effets de bord qui doivent être réalisés pour modifier la base afin de maintenir la cohérence après l'ajout d'un fait. Les effets de bord sont un ensemble de règles de réécriture construit à partir des règles sémantiques de RDFS. [CEH19] propose différents niveaux de modification autorisés. Ce niveau peut être paramétré pour chaque utilisateur lui donnant ainsi certains droits de modification. Le niveau le plus élevé est autorisé à modifier le schéma et l'instance pour ajouter un fait. Un niveau moins élevé permet seulement de modifier l'instance tandis que le niveau le plus bas ne peut pas modifier la base et peut donc seulement ajouter ou supprimer un fait si la cohérence de la base est maintenue.

$$\begin{array}{ll}
 Cl(x) & (2.1) \\
 Cl(y) \vee (y = Resource) & (2.2) \\
 \neg CSub(y, x) & (2.3)
 \end{array}
 \qquad
 \begin{array}{ll}
 \forall z CSub(y, z) \rightarrow CSub(x, z) & (2.4) \\
 \forall z CSub(z, x) \rightarrow CSub(z, y) & (2.5) \\
 \forall z CI(z, x) \rightarrow CI(z, y) & (2.6)
 \end{array}$$

FIGURE 2.4 – Effets de bord pour l'ajout du fait $CSub(x, y)$

Exemple 2.3.2. La figure 2.5 représente une base de connaissance initiale (en noir) et les modifications nécessaires (en bleu) pour l'ajout du fait $CSub(C_1, C_2)$. En suivant les effets de bord figure 2.4, l'ajout du fait $CSub(C_1, C_2)$ implique que C_1 et C_2 soient des classes par les effets de bord 2.1 et 2.2 respectivement. Comme C_2 n'est pas présent, le schéma doit être modifié. Le fait $CSub(C_2, Resource)$ est alors ajouté comme effet de bord de l'insertion de $Cl(C_2)$. Par la suite, le fait $CSub(C_1, C_2)$ peut être ajouté en ajoutant aussi le fait $CI(I, C_2)$ par transitivité de la relation CI via l'effet de bord 2.6. Cet exemple montre l'application d'effets de bord pour rendre applicable une règle. Sans ces effets de bord, il aurait été impossible d'ajouter le fait voulu tout en respectant les règles d'intégrité. Ici, l'individu I est bien l'instance d'une classe et de toutes les sous-classes associées. La base avant et après application de la règle est un graphe RDFS valide.

SETUP fait partie du projet ANR réSEaux Numériques de Données sémantiques : Utilité et vie Privée (SENDUP) [ReS, SEN] qui vise à concevoir des mécanismes d'anonymisation pour les données structurées en graphes. Plus précisément, SENDUP se concentre sur les données RDF et cherche à développer des approches pratiques, étayées par des modèles théoriques formels, pour garantir la confidentialité tout en préservant l'utilité des informations lors de la publication ou de l'interrogation de bases de données sous forme de graphes avec une sémantique sous-jacente.

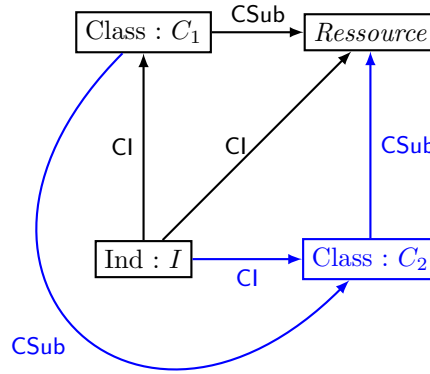


FIGURE 2.5 – Exemple de modification pour l'application d'une règle

2.3.2 Dépendances fonctionnelles pour les graphes

[FFTD15, FWX16, FL19] sont une suite de travaux qui cherchent à définir des GEDs comme les dépendances fonctionnelles. Dans le modèle relationnel, les dépendances fonctionnelles sont définies sur une relation pour un ensemble d'attributs. Pour les graphes, étant généralement dépourvus de schéma, il est nécessaire de définir le périmètre des dépendances fonctionnelles. Les GEDs doivent alors non seulement préciser la cohérence entre les valeurs des attributs, mais également détailler les structures topologiques qui les caractérisent et sont alors la combinaison :

- d'une contrainte topologique, ou motif, sur le graphe qui définit la portée de la dépendance fonctionnelle et
- d'une dépendance fonctionnelle sur les attributs.

Comme pour les langages d'interrogation de graphes (cf. Cypher), un motif est un graphe contenant un ensemble fini de variables pour les nœuds, les arêtes et les propriétés. Dans [FL19], un motif est défini comme un graphe dirigé $Q[\bar{x}] = (V_Q, E_Q, \lambda, \mu)$ où V_Q est un ensemble fini de nœuds, E_Q un ensemble fini d'arêtes, $\lambda : V_Q \cup E_Q \rightarrow L$ est une fonction partielle de labellisation, $\bar{x} \subset \text{VAR}$ est un ensemble de variables tel que $|\bar{x}| = |V_Q|$ ou $|\bar{x}| = |E_Q|$ et $\mu : \bar{x} \rightarrow V_Q \cup E_Q$ est une fonction bijective d'affectation de variables. La recherche du motif Q dans un graphe attribué $G = (V, E, L, F_A)$ ($F_A(v)$ est l'ensemble des propriétés $A_i = a_i$ où $v.A_i$ est une propriété de v et a_i est une constante) consiste à retrouver l'ensemble des homomorphismes h de ces variables tel que le graphe $h(Q)$ soit un sous graphe de G . C'est-à-dire que pour chaque nœud $u \in V_Q$, $L_Q(u) = L(h(u))$ et pour chaque arête $e = (u, p, u')$ dans Q il existe une arête $e' = (h(u), p, h(u'))$ dans G .

Une GEDs est alors définie comme un tuple $(Q[\bar{x}], X \rightarrow Y)$ où $Q[\bar{x}]$ est le motif qui définit la contrainte topologique (le sous graphe sur lequel la contrainte s'applique) et X et Y sont deux ensembles de contraintes sur les variables dans $\bar{x}$. Une contrainte est soit :

- une contrainte sur une valeur de propriété : $x.A = c$;
- une contrainte d'équivalence entre deux propriétés : $x.A = y.B$;
- une contrainte d'identité qui signifie que deux nœuds ou arêtes sont les mêmes : $id(x) = id(y)$.

Où $x, y \in \bar{x}$ sont des variables, A et B sont des propriétés de nœuds ou d'arêtes et c est une constante. Les contraintes ont été étendues par la suite pour inclure des contraintes négatives avec un ensemble prédéfini de prédicats : $\neq, <, \leq, >, \geq$.

Exemple 2.3.3. La figure 2.6 montre deux exemples de dépendance fonctionnelle. C_1 définit que tout auteur d'un document de type thèse est un docteur où Q_1 est la contrainte topologique et $type$ est une propriété des nœuds Document. C_2 exprime le fait que si deux villes sont la capitale d'un même pays, alors elles sont équivalentes et doivent être représentées par le même nœud dans le graphe.

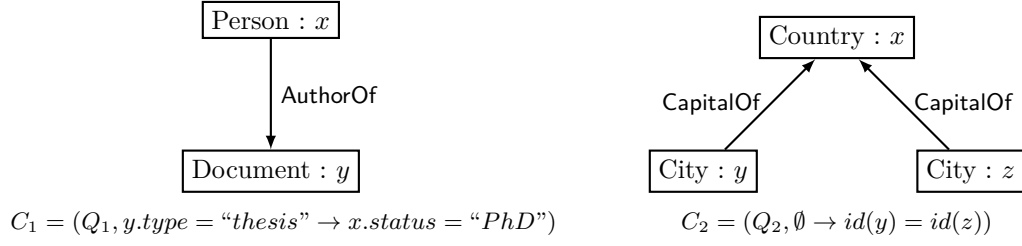


FIGURE 2.6 – Exemple de dépendances fonctionnelles pour les graphes

2.4 Discussion

Concernant SETUP ([CEH19]), une étude menée dans [CEHH20, CEHH21] montre des temps d'exécution élevés pour l'application des règles sur de petites instances avec une première implémentation en mémoire primaire. Ces performances s'expliquent par le calcul des effets de bords et la grande complexité de la réécriture de graphe. Cela rend difficile l'utilisation de cette implémentation de SETUP dans un scénario "en temps réel" ou des mises à jour arrivent régulièrement ou avec des grands graphes. De plus, cette implémentation n'a pas de fonctionnalité de type *fallback* permettant d'annuler les changements effectués sur la base en cas d'échec de la mise à jour, si le niveau de l'utilisateur n'est pas suffisant, ce qui implique de travailler sur une copie de la base. Cependant, il reste efficace pour des mises à jour ponctuelles, des corrections de bases en mode déconnecté ou pour des problématiques liées à la sécurité dans le cadre du projet SENDUP [BEN22]. Bien que SETUP permette la gestion de différents schémas définis sous RDFS, il ne permet pas d'ajouter des contraintes métiers. Définir de nouvelles contraintes impliquerait un travail conséquent et manuel pour leur traduction en règle de réécriture et la construction des effets de bords.

[FL19] constitue une avancée majeure dans la théorie des dépendances fonctionnelles pour les graphes, explorant en détail leur complexité. Contrairement à l'approche du SETUP, cette méthode est adaptable, les dépendances fonctionnelles n'étant pas figées. Comme pour les approches basées sur la réécriture de graphe, l'approche reste couteuse, mais des algorithmes parallèles sont proposés pour le raisonnement sur les GED. Cependant, l'accent est mis sur la vérification des incohérences, sans fournir de solutions pour corriger les instances. La question de l'identité des nœuds dans le contexte de la correction soulève des défis, notamment si deux nœuds sont déclarés équivalents : garder un seul des nœuds ou les fusionner en gérant les propriétés incohérentes n'est pas trivial et rejoint les problématiques rencontrées avec SETUP. Il peut être nécessaire de construire l'identité à partir des GED et des propriétés, de façon similaire aux clés primaires dans le modèle relationnel. Dans les graphes, la notion d'équivalence régulière [EB94] montre bien la complexité de cette tâche. Les GED proposent de bien séparer la topologie de la contrainte qui s'applique alors uniquement sur les propriétés des nœuds ou relations identifiés par la contrainte topologique. De ce fait, une politique de maintien de la cohérence, consisterait à changer les valeurs des attributs, insérer ou supprimer des attributs et, au maximum, à fusionner des nœuds ou des arêtes. La création de nœuds ou d'arêtes reste impossible contrairement à ce qui est proposé avec SETUP.

Exemple 2.4.1. En prenant comme exemple le graphe figure 2.7a et la contrainte $C : \text{Document}(x) \rightarrow \text{AuthorOf}(y, x)$. Les GEDs ne permettraient pas de corriger l'instance contrairement à une approche comme SETUP qui autorise de rajouter le nœud *Person* et l'arête *AuthorOf* manquante pour obtenir l'instance figure 2.7b qui est cohérente par rapport à la contrainte C .

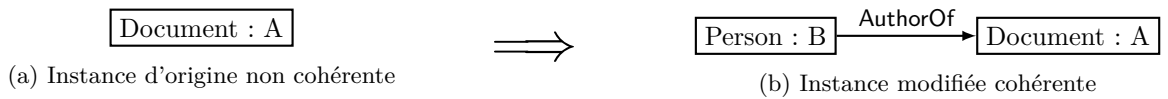


FIGURE 2.7 – Exemple de modification d'un graphe qui implique la création de nouveaux éléments

Dans cette thèse, nous travaillons sur les principes établis dans [CHL20] où les contraintes sont plus générales et non dirigées vers un modèle spécifique (comme les graphes d'attributs ou les graphes RDF). Dans ces travaux, visualiser la correction d'une base par l'introduction de nouveaux nœuds ou arêtes reste envisageable. [CHL20] introduit une sémantique formelle basée sur FOL pour mettre à jour des bases de données incomplètes contenant des valeurs nulles liées, en suivant les principes énoncés dans la sémantique de Reiter [Rei86] (section 2.1). Cette approche de mise à jour est déterministe et exige le respect d'un ensemble de TGDs. Les algorithmes présentés couvrent les opérations d'insertion et de suppression. Pour valider cette proposition, une implémentation en mémoire primaire est proposée. La mise à jour est structurée autour de deux actions fondamentales : *chasing* et *simplification*. La procédure du Chase est utilisée pour garantir la cohérence avec un ensemble de contraintes d'intégrité exprimées sous forme de règles, pouvant engendrer de nouvelles valeurs nulles. Le processus de simplification vise à réduire les redondances en éliminant les valeurs nulles qui peuvent être instanciées sans rompre leurs liens. Il s'agit du calcul du Core [FKP05]. Cette approche présente une flexibilité accrue par rapport à celle présentée dans [CEH19], grâce à l'utilisation de TGDs. Il convient de noter que l'utilisation de contraintes sous forme de TGDs augmente l'expressivité, mais cela s'accompagne de difficultés nécessitant une procédure de poursuite pour calculer la sémantique et générer des effets secondaires lors des mises à jour, entraînant des insertions ou des suppressions supplémentaires pour maintenir la cohérence. Une étude sur la procédure du Chase a été publiée dans [One13] et une évaluation dans [BKM⁺17].

Chapitre 3

Mise à jour incrémentale et cohérente d'une base de données

Sommaire

3.1 Contributions	21
3.2 Modélisation de la base de données sous Neo4J	22
3.3 Réduction incrémentale de la redondance	24
3.3.1 Requête sur le graphe	25
3.3.2 Simplification	26
3.4 Chase incrémental et effet de bords	28
3.4.1 Cycles et génération infinie	28
3.4.2 Utilisation du graphe pour le Chase	29
3.4.3 Chase en arrière	31
3.5 Mise à jour incrémentale	32
3.5.1 Insertion	32
3.5.2 Suppression	33
3.6 Evaluation	34
3.6.1 Implémentation sur un modèle relationnel	35
3.6.2 Évaluation des résultats	36
3.7 Conclusion	40

Ce chapitre est consacré à la description de notre méthode de mise à jour incrémentale sur des bases de données graphes. Il débute par la liste des contributions (section 3.1) et continue sur la modélisation de la base (section 3.2) pour ensuite présenter nos solutions incrémentales pour réduire la redondance (section 3.3) et calculer les effets de bord (section 3.4). Après la présentation des procédures d'insertion et de suppression (section 3.5), nous présenterons des résultats expérimentaux (section 3.6) en comparant l'implémentation dans un SGBD graphe avec une implémentation sur un SGBD relationnel avant de conclure le chapitre (section 3.7).

3.1 Contributions

En travaillant avec des bases de données graphes, la proposition de mise à jour doit opérer en harmonie avec les SGBDs tout en prenant en compte les contraintes propres à ces systèmes. Bien que les travaux exposés dans cette thèse s'alignent sur la politique détaillée dans [CHL20], ils se distinguent par les éléments suivants :

Conception de la base de données

1. Les SGBDs pour les LPGs partent généralement de l'hypothèse que les graphes sont complets, ce qui ne correspond pas à la réalité des données souvent incomplètes. L'approche adoptée par Neo4J, qui considère les valeurs nulles comme des valeurs inexistantes, s'avère insuffisante pour traiter les

valeurs nulles liées. Pour résoudre ce problème, une nouvelle conception de base de données est proposée, traitant les valeurs nulles comme une forme de constante.

2. Notre modélisation vise à éliminer des problèmes d'efficacité constatés lors d'une implémentation sur un SGBD relationnel. Pour cela, la modélisation proposée permet une indexation des valeurs nulles, facilitant ainsi l'identification de tous les atomes directement ou indirectement liés à ces valeurs. Cette approche simplifie grandement les opérations de manipulation des valeurs nulles impactées par une mise à jour pour le calcul du Core.
3. Mise en place de requêtes capables d'identifier les parties du graphe de données concernées par la procédure de mise à jour cohérente. Notamment pour le calcul du Chase et du Core. Ces requêtes sont rendues efficaces par les choix de conceptions de la base de données.

Approche incrémentale

1. Le calcul des mises à jour se limite aux contraintes et aux données spécifiquement concernées. En contraste avec la méthodologie présentée dans [CHL20], qui exige la vérification de l'ensemble de la base de données pour chaque mise à jour.
2. La simplification des valeurs est guidée par les valeurs nulles pouvant potentiellement être simplifiées en raison de la mise à jour. Cette approche diffère de celle de [CHL20], où la simplification prend en compte l'ensemble des valeurs nulles indépendamment de la mise à jour.

3.2 Modélisation de la base de données sous Neo4J

La proposition de mise à jour incrémentale sur des bases de données graphe est implémentée sur le gestionnaire de base de données graphe Neo4J en définissant un ensemble de requêtes Cypher. La maintenance décrite dans [CHL20] relève un coût important pour la récupération des atomes liés par des valeurs nulles (le calcul de l'ensemble LINKEDNULL). L'utilisation de base de données graphe ajoute une possibilité d'optimisation de cette opération. Le modèle de graphe doit mettre en avant ces relations entre les valeurs nulles et faciliter la recherche des homomorphismes.

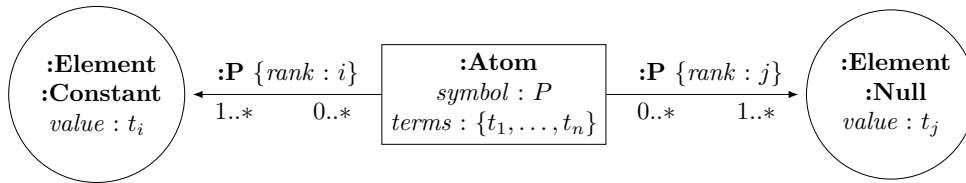


FIGURE 3.1 – Modèle de Graphe

Pour ce faire, on choisit le modèle dans lequel un atome $P(t_1, \dots, t_n)$ est représenté comme un arbre (Figure 3.1) qui a pour racine un nœud *Atom* lié à un ensemble de nœuds *Élément* représentant les termes. Les *Atomes* ont le label **:Atom** et une unique propriété *symbol* représentant le symbole de prédicat. Les *Éléments* ont le label **:Element** et une unique propriété *value* représentant la valeur de l'élément. On leur ajoute aussi un label **:Constant** ou **:Null** respectivement s'il s'agit d'une constante ou d'une valeur nulle. Les *Atomes* sont connectés aux *Éléments* par une relation qui a pour unique propriété le rang auquel apparaît l'élément dans l'atome. Dans cette modélisation (Figure 3.1) proche de la logique, le nœud rectangulaire représente l'*Atome* et les nœuds circulaires les *Éléments*. t_i représente les constantes et t_j les valeurs nulles. Les notations en dessous des arcs représentent leur cardinalité : chaque *Élément* est connecté à au moins un *Atom* alors qu'un *Atom* peut ne pas avoir de termes.

Cette modélisation facilite la récupération des ensembles d'atomes liés ainsi que la recherche d'homomorphismes. En effet, une requête exprimée sous forme logique $Q(X) \leftarrow \phi(X)$ se traduit par la recherche d'un sous graphe dans lequel les variables logiques deviennent naturellement les variables du motif. Rechercher l'ensemble des atomes où N_1 est un terme revient à suivre, dans le sens inverse, l'ensemble des arcs qui arrivent à N_1 . Cependant, dans ce modèle, les atomes sont représentés comme des arbres ce qui rend non négligeable le coût de la reconnaissance de motifs. Cette récupération est importante, car elle permet de reconstruire les atomes sous forme logique afin d'être utilisés dans nos procédures locales. Dans Neo4J, la reconnaissance de motifs est réalisée par un parcours du graphe. Étant donné un nœud *Élément* e , la récupération des nœuds *Atomes* contenant e et ayant P pour symbole de prédicat, implique de :

1. parcourir l'ensemble des arêtes de e
2. pour chaque arête, récupérer le nœud associé
3. pour chaque nœud, vérifier la valeur de la propriété représentant le symbole de prédicat

Si un nœud *Élément* se retrouve lié a beaucoup de nœud *Atomes* (c.-à-d. s'il possède un grand nombre d'arêtes) alors, cette opération peut devenir couteuse. Malheureusement, ceci ne peut pas être résolu par l'utilisation d'un index permettant de parcourir uniquement les arcs nécessaires car, aujourd'hui, Neo4J permet uniquement la création d'index globaux (au graphe) permettant de récupérer, par exemple, l'ensemble des *Atomes* ayant P pour symbole de prédicat. Pour répondre au besoin, il faudrait un index local à chaque nœud (récupérer le sous ensemble correspondant de l'index global) et qui hériterait des attributs des éléments connectés (l'arête doit récupérer le symbole de prédicat du nœud *Atome* associé).

Dans ce contexte, des redondances d'informations et des index sont introduits afin d'optimiser ces différentes opérations :

Limiter le parcours des arêtes

1. La liste ordonnée des termes comme propriété des nœuds *Atomes* permettant de récupérer un atome complet depuis le nœud atome sans avoir à parcourir les arêtes (propriété *terms* du nœud *Atome* dans la figure 3.1) ;
2. Pour chaque arête, on lui associe comme type, le symbole de prédicat de l'*Atome* associé (label P dans la figure 3.1).

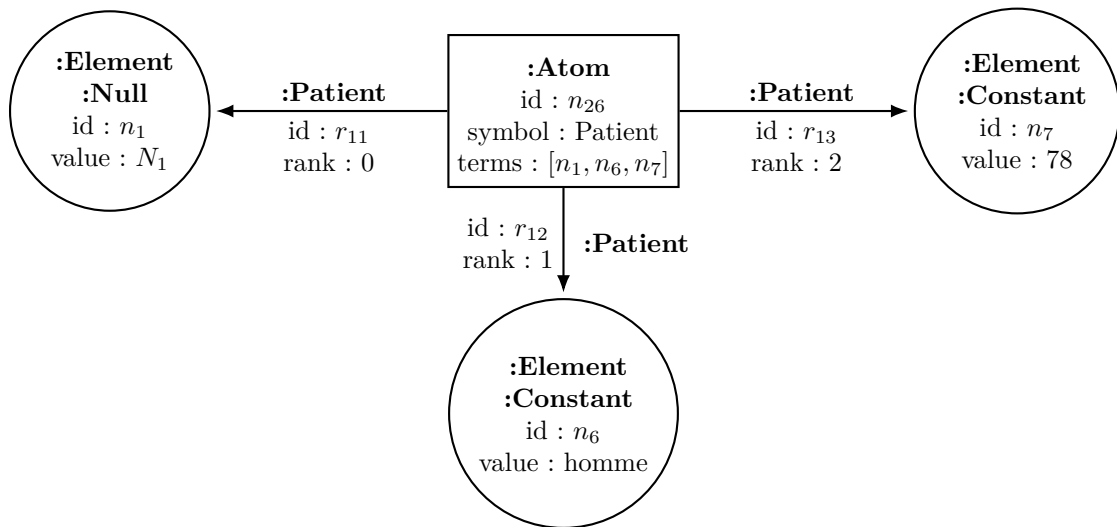
Faciliter la récupération des nœuds

1. Une contrainte d'unicité / index sur les nœuds *Éléments* vérifiant la non-redondance des constantes et des valeurs nulles ;
2. Une contrainte d'unicité / index sur le couple symbole-termes des nœuds *Atomes* vérifiant la non-redondance des atomes ;
3. Un index sur le symbole de prédicat des nœuds *Atomes*.

Exemple 3.2.1. Soit l'ensemble d'atomes suivants provenant de l'exemple 1.2.2 (page 3) :

$$\mathcal{D} = \{Patient(N_1, homme, 78), Exam(N_1, N_2, N_3, 01-01-01), Parameter(N_3, température, N_4), \\ Reveal(N_3, fièvre), hasPatho(N_1, cancer), getTreatment(N_1, résection), Anatomie(prostate), \\ forPatho(résection, cancer), leadsTo(cancer, métastase), concernAna(cancer, prostate)\}$$

En utilisant la modélisation présentée précédemment, l'atome $Patient(N_1, homme, 78)$ est alors représenté par le graphe de propriétés suivant où le nœud n_{26} représente l'atome et les nœuds n_1 , n_6 et n_7 représentent ses termes. La figure 3.2 montre une vue complète de l'instance $\mathcal{D}$.



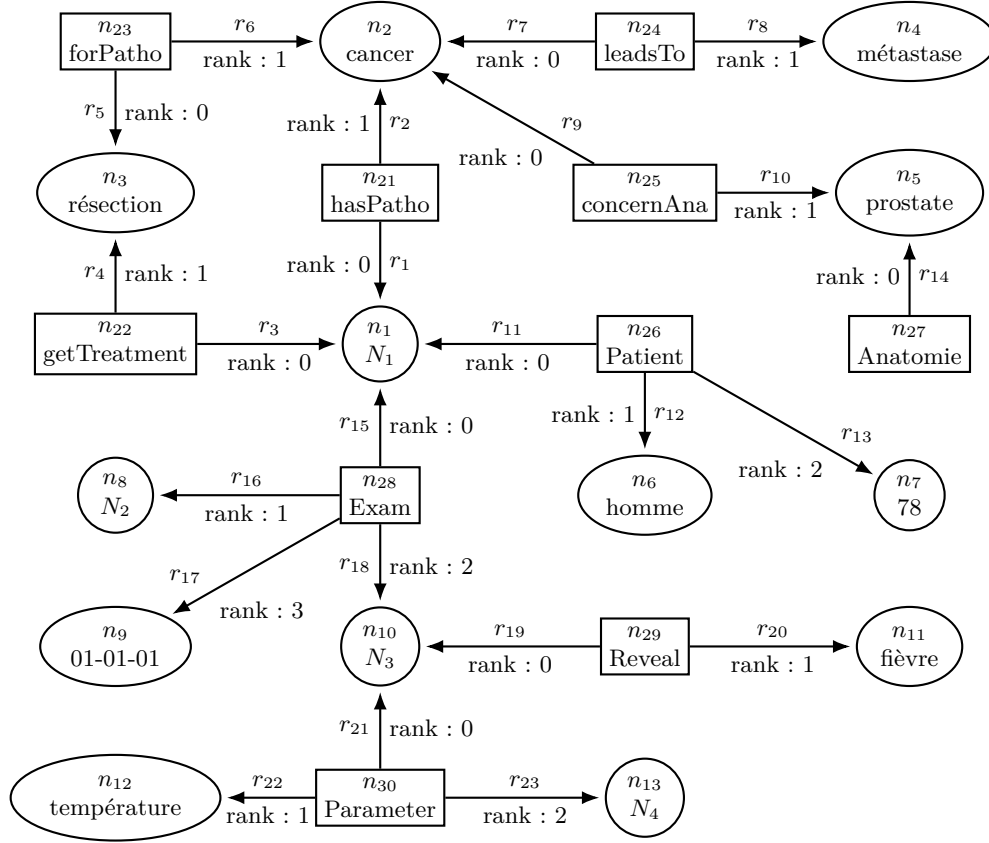


FIGURE 3.2 – Instance $\mathcal{D}$ représentée par un graphe de propriétés. Les nœuds ellipse représentent les *Éléments* et les rectangles les *Atomes*. Les labels et les propriétés d'optimisation ne sont pas affichés

3.3 Réduction incrémentale de la redondance

Comme expliqué dans la section 2.2.2, une base de données $\mathcal{D}$ doit être minimale, c.-à-d. égale à son Core afin d'éviter la redondance des données. La minimalité doit être maintenue à chaque mise à jour et de façon incrémentale. Pour ce faire, des algorithmes incrémentaux sont proposés avec pour objectif de recalculer le Core de la base uniquement en fonction des valeurs nulles impactées par la mise à jour. Formellement, étant donné un ensemble d'atomes I et un ensemble de valeurs nulles ν existant dans I , on recherche un homomorphisme h tel que : pour toutes valeurs nulles $N \notin \nu$, $h(N) = N$, $h(I)$ est minimal et $h(I) \subseteq I$. L'exemple suivant montre cependant que le choix de ν n'est pas arbitraire.

Exemple 3.3.1. En considérant l'ensemble non minimal $I = \{P(a, N_1), P(N_1, N_2), P(a, b), P(b, c), P(a, N_3)\}$, si l'ensemble $\nu = \{N_1\}$, alors seuls les atomes $P(a, N_1)$ et $P(N_1, N_2)$ sont à prendre en compte pour la simplification. Cependant, aucune simplification n'existe en prenant uniquement en compte l'élément N_1 , il est également nécessaire de considérer la valeur nulle N_2 . Pour parvenir à la simplification souhaitée de l'ensemble I , l'ensemble ν doit être élargi pour obtenir $\nu' = \{N_1, N_2\}$. En utilisant ν' , on peut établir un homomorphisme $h = \langle N_1 \mapsto b, N_2 \mapsto c, N_3 \mapsto N_3 \rangle$ qui assure la minimalité de l'ensemble $h(I) = \{P(a, b), P(b, c), P(a, N_3)\}$ par rapport à l'ensemble ν . Il est important de noter que l'atome $P(a, N_3)$ ne fait pas l'objet d'une simplification, car l'élément N_3 ne figure pas dans l'ensemble ν' des valeurs nulles prises en considération.

Comme discuté dans l'exemple 3.3.1, étant donné un ensemble d'atomes I et un ensemble de valeurs nulles liées ν_0 dans I , l'ensemble des valeurs nulles liées ν doit être calculé par extension de ν_0 en considérant toutes les valeurs nulles liées directement ou indirectement aux valeurs nulles de ν_0 . Pour cela, on calcule l'ensemble d'atomes $\text{LinkedNull}_{I,N}$ pour chaque valeur nulle N dans ν_0 . Cet ensemble est

```

1 UNWIND $nulls AS nullPredName
2 MATCH (nullValueNode:Element:Null {value: nullPredName}),
3   pathP = (nullValueNode)-[*1..maxPathLength]-(endNode)
4 WHERE endNode <> nullValueNode AND
5   ALL(n IN nodes(pathP) WHERE NOT (n:Constant))
6 WITH COLLECT(DISTINCT endNode) AS linkedNodes, nullValueNode
7 WITH [n IN linkedNodes WHERE (n:Atom)] AS linkedAtoms,
8   [nullValueNode] + [n IN linkedNodes WHERE (n:Null)] AS linkedNulls
9 UNWIND linkedAtoms AS a RETURN a.symbol as a, a.terms as e, linkedNulls

```

Extrait de code 3.1 – Requête Cypher pour calculer l'ensemble $\text{LinkedNull}_{\mathcal{D}}$

la limite de la séquence $(\text{LinkedNull}_{I,N}^k)_{k \geq 0}$ définie par :

$$\text{LinkedNull}_{I,N}^k = \begin{cases} \{atom \in I \mid N \in \text{nulls}(atom)\} & \text{si } k = 0 \\ \{atom \in I \mid (\exists atom' \in \text{LinkedNull}_{I,N}^{k-1})(\text{nulls}(atom) \cap \text{nulls}(atom') \neq \emptyset)\} & \text{si } k > 0 \end{cases}$$

Proposition 3.3.1. *Pour tout $k \geq 0$, $\text{LinkedNull}_{I,N}^k \subseteq \text{LinkedNull}_{I,N}^{k+1}$ et $\text{LinkedNull}_{I,N}^k \subseteq I$. La séquence $(\text{LinkedNull}_{I,N}^k)_{k \geq 0}$ est donc monotone et bornée par I . Comme I est fini, la séquence a une unique limite qui est un sous-ensemble de I noté $\text{LinkedNull}_{I,N}$.*

La proposition 3.3.1 montre que la redondance peut être uniquement vérifiée sur l'ensemble d'atomes défini par $\bigcup_{N \in \nu_0} \text{LinkedNull}_{I,N}$ et l'ensemble ν de valeurs nulles de cet ensemble.

3.3.1 Requête sur le graphe

L'ensemble $\text{LinkedNull}_{\mathcal{D}}$ (où $\mathcal{D}$ est la base de données) est calculé à l'aide de la requête Cypher présentée dans l'extrait 3.1. Il s'agit d'une requête paramétrée, reconnaissable par la présence d'une variable préfixée du symbole \$. L'opérateur UNWIND (ligne 1) permet de convertir une variable de type liste en une table de résultat. Le MATCH (ligne 2) recherche, dans le graphe, les homomorphismes du motif décrit. Ici, on recherche un nœud, que l'on stockera dans la variable `nullValueNode`, ayant les labels `:Element` et `:Null` et ayant comme valeur de la propriété `value` la valeur stockée dans la variable `nullPredName`. La suite du motif (ligne 3) recherche un chemin `pathP` de taille n , où $1 \leq n \leq \text{maxPathLength}$, partant du nœud précédent. `maxPathLength` est défini comme le double du nombre de valeurs nulles dans la base et permet d'empêcher le parcours de chemins que l'on sait d'avance inutile (un chemin étant composé d'une alternance de nœuds `:Atom` et de nœuds `:Null`, un chemin plus long inclue soit des nœuds `:Constant`, soit un cycle). La condition exprimée par l'opérateur WHERE, vérifie que le nœud final du chemin soit différent du nœud de départ et qu'aucun nœud du chemin `nodes(pathP)` n'ai le label `:Constant`.

L'opérateur WITH (ligne 6) permet de faire une projection d'une table intermédiaire. Sa combinaison avec l'opérateur COLLECT permet de regrouper les variables `endNode` dans une variable `linkedNodes` de type liste en fonction de la variable `nullValueNode`. Cette combinaison est similaire à l'opérateur GROUP BY en SQL. Intuitivement, on recherche pour chaque nulle l'ensemble des nœuds connectés uniquement par des nœuds *Atomes* ou des nœuds *Nulles*. Les ensembles de nœuds sont séparés (ligne 7) en deux listes `linkedAtoms` et `linkedNulls` contenant respectivement l'ensemble des nœuds *Atomes* liés par des valeurs nulles et l'ensemble des nœuds *Nulles*, liant cet ensemble d'atomes. L'opérateur UNWIND final permet de retourner comme résultat de la requête chaque atome associé à l'ensemble des valeurs nulles auquel il est attaché.

Exemple 3.3.2. Dans le graphe d'instance donné dans la Figure 3.2, la récupération de l'ensemble $\text{LinkedNull}_{\mathcal{D},N_1}$ en utilisant la requête de l'extrait 3.1 se traduit par l'exécution pas à pas suivante :

Etape 1 L'opération MATCH WHERE ligne 2 retourne la table d'instanciation suivante de tous les chemins `pathP` partant de N_1 de taille maximale 6 (car 3 valeurs nulles sont présentes dans l'instance) ne contenant que des valeurs nulles et des atomes.

nullValueNode	endNode	pathP
n_1	n_{21}	$[n_1, r_1, n_{21}]$
n_1	n_{22}	$[n_1, r_3, n_{22}]$
n_1	n_{26}	$[n_1, r_{11}, n_{26}]$
n_1	n_{28}	$[n_1, r_{15}, n_{28}]$
n_1	n_8	$[n_1, r_{15}, n_{28}, r_{16}, n_8]$
n_1	n_{10}	$[n_1, r_{15}, n_{28}, r_{18}, n_{10}]$
n_1	n_{29}	$[n_1, r_{15}, n_{28}, r_{18}, n_{10}, r_{19}, n_{29}]$
n_1	n_{30}	$[n_1, r_{15}, n_{28}, r_{18}, n_{10}, r_{21}, n_{30}]$
n_1	n_{13}	$[n_1, r_{15}, n_{28}, r_{18}, n_{10}, r_{21}, n_{30}, r_{23}, n_{13}]$

Etape 2 Le premier WITH ligne 6 permet de regrouper l'ensemble des nœuds destination sous forme d'une liste. On obtient alors, pour chaque **nullValueNode**, la liste de tous les nœuds connectés par des valeurs nulles.

nullValueNode	linkedNodes
n_1	$[n_8, n_{10}, n_{13}, n_{21}, n_{22}, n_{26}, n_{28}, n_{29}, n_{30}]$

Etape 3 Le dernier WITH ligne 7 permet de séparer les atomes des valeurs nulles.

linkedAtoms	linkedNulls
$[n_{21}, n_{22}, n_{26}, n_{28}, n_{29}, n_{30}]$	$[n_1, n_8, n_{10}, n_{13}]$

Etape 4 Pour finir on retourne l'ensemble des atomes associés à l'ensemble **LinkedNull** auquel ils appartiennent. L'opérateur **UNWIND** transpose la liste **linkedAtoms** en lignes de la table dupliquant les autres attributs de la table d'origine pour chaque entrée (ici, **linkedNulls**).

a	e	linkedNulls
hasPatho	$[N_1, \text{cancer}]$	$[n_1, n_8, n_{10}, n_{13}]$
getTreatment	$[N_1, \text{résection}]$	$[n_1, n_8, n_{10}, n_{13}]$
Patient	$[N_1, \text{homme}, 78]$	$[n_1, n_8, n_{10}, n_{13}]$
Exam	$[N_1, N_2, N_3, 01-01-01]$	$[n_1, n_8, n_{10}, n_{13}]$
Reveal	$[N_3, \text{fièvre}]$	$[n_1, n_8, n_{10}, n_{13}]$
Parameter	$[N_3, \text{température}, N_4]$	$[n_1, n_8, n_{10}, n_{13}]$

3.3.2 Simplification

La procédure pour éliminer la redondance sur un ensemble d'atomes I en fonction d'un ensemble de valeurs nulles ν_0 est donnée par la procédure Simplify (page 27). À la ligne 1, les ensembles d'atomes, liés par des valeurs nulles et qui contiennent au moins une valeur nulle présente dans ν_0 , sont calculés. À la ligne 3, pour chaque ensemble d'atomes P , on construit la requête $Q_{core}(X) \leftarrow \phi(X)$ où ϕ est l'ensemble des atomes de P dans lequel chaque valeur nulle N_i est remplacé par la variable correspondante x_i . La requête Q_{core} retourne l'ensemble des homomorphismes sur les valeurs nulles de P . Si plusieurs homomorphismes sont retournés par la requête, cela signifie qu'il existe plusieurs instanciations possibles de P et qu'il faut le simplifier, car P est redondant. Dans ce cas, il faut choisir l'instanciation (ou l'homomorphisme) la *plus spécifique* (ligne 5) et supprimer les atomes redondants (ligne 6). Si l'on ne prend pas l'instanciation la *plus spécifique* il faudra la simplifier à nouveau. La situation est illustrée par l'exemple suivant.

Exemple 3.3.3. En prenant comme exemple l'instance $I = \{B(N_1, N_2), B(a, N_2), B(a, b), C(N_2, N_2), C(N_2, N_3), C(a, b)\}$, avec $P = \{B(N_1, N_2), B(a, N_2), C(N_2, N_2), C(N_2, N_3)\}$ pour $P \in \text{LinkedNull}_{I, N_1}$. On commence par la construction de la requête $Q_{core}^{[P]}(I) \leftarrow B(x_1, x_2), B(a, x_2), C(x_2, x_2), C(x_2, x_3)$, dont les réponses sont présentées dans la table H_P suivante :

H_P	x_1	x_2	x_3
h_1	N_1	N_2	N_3
h_2	N_1	N_2	N_2
h_3	a	N_2	N_3
h_4	a	N_2	N_2

Procédure Simplify(I, ν_0)

```

1  $PSet \leftarrow \{\text{LinkedNull}_{I,N} \mid N \in \nu_0\};$ 
2 pour tous  $P \in PSet$  faire
3   Construire et exécuter la requête  $Q_{core}^{[P]}(I)$  pour récupérer l'ensemble des homomorphismes
    $H = \{h_1, \dots, h_n\};$ 
4   si  $|H| > 1$  alors
5      $h_m \leftarrow \text{PlusSpecifique}(H);$ 
6      $I \leftarrow (I \setminus P) \cup h_m(P);$ 
7 retourner  $I;$ 
```

Étant donnés deux homomorphismes h_1 et h_2 sur le même alphabet Σ : h_1 est dit *moins spécifique* que h_2 , noté $h_1 \prec h_2$, s'il existe un homomorphisme h sur Σ tel que $h \circ h_1 = h_2$ (c.-à-d. qu'il existe une transformation de h_1 vers h_2 nommée h). Soit, p le nombre de valeurs nulles N_j distinctes dans $P = \text{LinkedNull}_{I,N}$ et q le nombre d'homomorphismes h_i retournés par la requête Q_{core} avec h_1 étant l'identité de P . Il est important de noter qu'il peut exister plusieurs homomorphismes plus spécifiques. Le choix n'est pas important et il suffit de récupérer l'un des homomorphismes les plus spécifiques. Pour trouver un des homomorphismes le plus spécifique sur P , on construit la table H_P de p colonnes et q lignes où $H_P[i, j] = h_i(N_j)$. Dans la table H_P , $h_i \preceq h_{i'}$ si et seulement si, pour tout j :

- Si $H_P[i, j] \notin \text{nulls}(P)$ alors $H_P[i, j] = H_P[i', j]$
- Si $H_P[i, j] \in \text{nulls}(P)$ alors pour tout $j' \neq j$ tel que $H_P[i, j] = H_P[i, j']$ on a $H_P[i', j] = H_P[i', j']$

À partir de la table H_P , il est possible de calculer l'homomorphisme le plus spécifique en calculant le nombre de valeurs nulles et de constantes. La preuve et la méthode de calcul sont détaillées dans [CHHL23] et données dans l'annexe A. De plus, il est montré dans [CHHL23] que les homomorphismes h_i les plus spécifiques respectent les propriétés suivantes :

1. h_i est idempotent, c.-à-d. que $h_i \circ h_i = h_i$;
2. h_i possède le nombre maximum de constantes ou de valeurs nulles extérieures à P et possède le moins de valeurs nulles différentes.

Exemple 3.3.4. En considérant la table H_P de l'exemple 3.3.3, l'homomorphisme h_1 représente l'identité de P et est toujours retourné par la requête Q_{core} . En se servant de cette table, il est possible de déterminer les relations suivantes :

- $h_1 \prec h_2$ en raison du nombre inférieur de valeurs nulles distinctes dans h_2 par rapport à h_1 (cette simplification s'opère en éliminant la valeur nulle redondante N_3) ;
- $h_1 \prec h_3$ en raison du nombre supérieur de constantes dans h_3 par rapport à h_1 (où N_1 peut être substitué par la constante a) ;
- $h_2 \prec h_4$ et $h_3 \prec h_4$ pour les mêmes raisons exposées précédemment.

L'homomorphisme le plus spécifique est déterminé comme étant h_4 . Par conséquent, l'application de la procédure Simplify, conduit à simplifier l'instance I comme indiqué ligne 6. On obtient alors :

$$\text{Simplify}(I, \{N_1\}) = \{B(a, N_2), B(a, b), C(N_2, N_2), C(a, b)\}$$

Cette approche basée sur la table H_P est comparable aux techniques d'optimisation de requêtes [CM77]. Cependant, elle diffère de [ASU79, CM77] sur deux points : (i) notre tableau est basé sur les réponses aux requêtes plutôt que sur le corps de la requête ; (ii) nous calculons un homomorphisme le plus spécifique, alors que dans [ASU79] ils rejettent les homomorphismes les moins spécifiques.

3.4 Chasse incrémental et effet de bords

L'algorithme du Chase [ABU79, MMS79] est utilisé pour saturer une base en fonction des TGDs. Dans ces travaux nous utilisons la version du chase présentée dans [CHL20]. Une contrainte est appliquée, c.-à-d. que l'on génère un nouvel atome instancié correspondant à la tête, quand le corps de la contrainte peut être instancié par des atomes dans la base. L'algorithme du Chase se termine quand plus aucune règle ne peut être instanciée ou que plus aucun nouvel atome n'est généré. Notre version incrémentale de l'algorithme consiste à considérer uniquement les règles et leur instanciation sur l'ensemble d'atomes mis à jour.

Exemple 3.4.1 (Application de contraintes). Si on considère l'ensemble de contraintes $\mathbb{C} = \{c_1, c_2\}$ et l'ensemble d'atomes $\mathcal{D} = \{A(a), B(a, b), B(c, b), C(b)\}$ avec :

$$c_1 : A(x) \rightarrow B(x, y)$$

$$c_2 : B(x, y) \rightarrow C(y)$$

L'insertion de l'ensemble $I = \{A(c), A(d)\}$ dans $\mathcal{D}$ implique l'application de la contrainte c_1 uniquement avec les instanciation $h_1 = \langle x \mapsto c \rangle$ et $h_2 = \langle x \mapsto d \rangle$. L'instanciation $h = \langle x \mapsto a \rangle$ n'a pas besoin d'être calculée car $\mathcal{D}$ est déjà cohérente par rapport à $\mathbb{C}$. A ce stade, il n'est pas non plus nécessaire de vérifier si c_2 est applicable. L'application de $h_1(c_1)$ génère un nouvel atome $B(c, N_1)$. Comme il existe déjà une instanciation plus spécifique dans la base : $B(c, b)$, ce nouvel atome n'a pas besoin d'être ajouté. L'application de $h_2(c_1)$ génère un nouvel atome $B(d, N_1)$. Comme il n'existe pas dans $\mathcal{D}$, on l'ajoute et on recommence l'algo avec les nouveaux atomes insérés. Avec l'insertion de $B(d, N_1)$, on déclenche c_2 avec l'instanciation $h_3 = \langle x \mapsto d, y \mapsto N_1 \rangle$ qui génère à son tour l'atome $C(N_1)$. Plus aucun atome ne peut être généré et on obtient donc l'ensemble $\mathcal{D}' = \{A(a), B(a, b), A(c), B(c, b), C(b), A(d), B(d, N_1), C(N_1)\}$.

3.4.1 Cycles et génération infinie

Dans nos travaux, les TGDs peuvent contenir des variables existentielles dans la tête qui généreront de nouvelles valeurs nulles lors de leur application. Cela peut engendrer une génération infinie d'atomes si des cycles existent dans les contraintes. Il est important de s'assurer que cette opération termine. Pour faire face à cette problématique, on associe à chaque valeur nulle, un entier appelé le degré de N et dénoté par $\delta(N^k) = k$. À chaque mise à jour, le degré de toutes les valeurs nulles apparaissant dans $\mathcal{D}$ est fixé à 0. Pendant le Chase, à chaque itération, les nouvelles valeurs nulles générées se voient attribuer un degré égal à $\delta + 1$, où δ est le degré maximal des valeurs nulles présentes dans les atomes du corps de la contrainte, ou 0 si aucune valeur nulle n'apparaît dans le corps de la contrainte. On fixe donc un degré maximal δ_{max} qui permet l'interruption du Chase dès qu'une valeur nulle N dépasse ce seuil, tel que $\delta(N) > \delta_{max}$.

Exemple 3.4.2 (Génération infinie d'atomes). Si on considère l'ensemble d'atomes $I = \{A(a, b)\}$ sur lequel on applique l'ensemble de contraintes $\mathbb{C} = \{c_1, c_2\}$ avec pour degré maximal $\delta_{max} = 1$.

$$c_1 : A(x, y) \rightarrow B(y, z)$$

$$c_2 : B(x, y) \rightarrow A(y, z)$$

L'application de la règle c_1 avec l'instanciation $h_1 = \langle x \mapsto a, y \mapsto b \rangle$ génère l'atome $B(b, N_1^0)$. Ce dernier permet l'application de la contrainte c_2 avec l'instanciation $h_2 = \langle x \mapsto b, y \mapsto N_1^0 \rangle$ et qui génère l'atome $A(N_1^0, N_2^1)$, qui permet la génération de $B(N_2^1, N_3^2)$ via c_1 . Il est facile de voir que la génération des atomes est infinie. Cependant, comme le degré de N_3^2 dépasse le degré maximal δ_{max} on arrête cette branche de la génération. Si d'autres règles peuvent être appliquées on continue la génération de ces dernières. L'ensemble résultat I' est invalide car une valeur nulle dépasse le degré maximal.

Dans la suite, avant de rejeter une mise à jour I , à cause d'un degré trop élevé, nous vérifierons si l'ensemble obtenu après l'application du chase I' n'est pas simplifiable. Si la simplification de I' engendre un ensemble I'' où les valeurs nulles "invalides" ont été supprimées, alors I'' peut être considéré et la mise à jour acceptée. Si I'' reste invalide malgré la simplification, alors la mise à jour devra être rejetée.

3.4.2 Utilisation du graphe pour le Chase

La procédure Chase4Insert définit l'algorithme incrémentale du Chase. L'ensemble $ToIns$ représente les effets de bord de l'insertion de l'ensemble I dans $\mathcal{D}$ tel que si $\mathcal{D}' = \mathcal{D} \cup ToIns : I \subseteq \mathcal{D}'$ et $\mathcal{D}' \models \mathbb{C}$. La boucle (ligne 2) correspond à l'application du Chase incrémentale qui respecte les conditions suivantes :

- ① Une contrainte $c \in \mathbb{C}$ est appliquée *uniquement* si au moins un atome de son corps fait parti des effets de bord $ToIns$. Cette condition vise à éviter le déclenchement inutile de règles ;
- ② Un atome est généré *uniquement* s'il n'existe pas, dans $\mathcal{D}$, un atome plus spécifique ou équivalent. Cette condition vise à prévenir la production d'atomes redondants, ce qui impliquerait une simplification ultérieure et réduit le nombre de générations nécessaires. Elle a également une importance cruciale pour la terminaison du Chase, car elle permet d'éviter certains cycles de générations qui n'auraient pu être simplifiés ultérieurement (voir l'exemple 3.4.2).
- ③ Un atome est généré *uniquement* s'il ne dépasse pas le degré maximum (voir section 3.4.1) ;

Procédure Chase4Insert($\mathcal{D}, \mathbb{C}, \delta_{max}, I$)

```

1  $ToIns := I$  ;
2 tant que il existe une contrainte  $c \in \mathbb{C}$  et un homomorphisme  $h$  tel que  $h(body(c)) \subseteq \mathcal{D} \cup ToIns$ 
   qui respecte les conditions :
   ①  $h(body(c)) \cap ToIns \neq \emptyset$ 
   ②  $\nexists h'' \supseteq h$  tel que  $h''(head(c)) \in \mathcal{D} \cup ToIns$ 
   ③  $\delta(h'(head(c))) \leq \delta_{max}$  où  $h' \supseteq h$  instancie les variables
      existentielles de la tête par de nouvelles valeurs nulles
3 faire
4    $ToIns := ToIns \cup \{h'(head(c))\}$  ;
5 retourner  $ToIns$  ;
```

Cypher Soit une contrainte $c : L_1(\alpha_1), \dots, L_m(\alpha_m) \implies L_0(\alpha_0)$. Nous introduisons les notations suivantes : α est l'ensemble des variables du corps de la contrainte, α^* est l'ensemble des variables substituées, $\alpha_i \subseteq \alpha$ est un sous-ensemble, $\alpha|_i \in \alpha$ denote le nom de la i -ème variable de α et $CONST$ est l'ensemble des constantes distinctes dans c . La recherche de l'application de c dans la base $\mathcal{D}$ se traduit par une requête $Q_{chase}^{[c]}(\mathcal{D})$ présentée dans l'extrait 3.2 qui : (i) vérifie l'existence d'une instance dans $\mathcal{D}$ et (ii) retourne une réponse non vide correspondant à l'instanciation de la tête uniquement si elle n'existe pas déjà dans la base. En FOL, cette requête s'écrit $Q_{chase}^{[c]} \leftarrow L_1(\alpha_1), \dots, L_m(\alpha_m), \neg L_0(\alpha_0)$. L'idée est que si il existe un homomorphisme h_t tel que $h_t(body(c)) \subseteq \mathcal{D}$ alors $Q_{chase}^{[c]}$ retourne une réponse non vide uniquement si pour toute extension h'_t de $h_t : h'_t(L_0(\alpha_0)) \notin \mathcal{D}$.

La requête $Q_{chase}^{[c]}(\mathcal{D})$ commence par rechercher les instanciations de $sub(body(c))$ (ligne 2) où sub est un homomorphisme tel que $sub(body(c)) \subseteq ToIns$. Intuitivement, on recherche les extension sub' de la substitution sub dans $\mathcal{D}$. Le **WHERE**, ligne 4, permet de vérifier l'instanciation du corps de la contrainte c par rapport à la substitution sub . Si la contrainte possède des constantes elles sont aussi vérifiées dans cette clause. L'opérateur **NOT EXISTS**, ligne 5, exécute une sous-requête qui réutilise les variables introduites dans le premier **MATCH**. Cette sous-requête recherche les instanciations de $sub'(head(c))$ dans $\mathcal{D}$ et permet de vérifier le critère ②. Si la réponse de cette sous-requête est vide, l'instanciation sub' est retournée. La requête retourne la substitution comme un mapping entre le nom des variables dans c et leur instanciation. Les variables existentielles présentes dans la tête de la contrainte sont exclues de la clause **RETURN**, et engendreront la génération de nouvelles valeurs nulles lors de l'insertion de la tête dans l'ensemble $ToIns$ (cf. clause ③). Le degré de ces nouvelles valeurs nulles étant égal au degré maximal des valeurs nulles présentes dans le corps, augmenté de 1.

```

1 UNWIND $subs AS sub ①
2 MATCH (∀i, 1 ≤ i ≤ m) (ai:Atom {symbol: "Li"}) ,
3     (∀j, 0 ≤ j < |αi|) (ai) -[:'Li' {rank: j}] -> (αi|j:Element)
4 WHERE (∀k ∈ C) α|k.value = k AND (∀k ∈ α*) α|k.value = sub.α|k } ③
5 AND NOT EXISTS {
6     MATCH (a:Atom {symbol: "L0"}) ,
7     (∀i, 0 ≤ i ≤ |α0|) (ai) -[:'Li' {rank: j}] -> (ei) } ②
8 } RETURN DISTINCT {(∀k, 0 ≤ kN ≤ |α|) α|k:xk.value}
9

```

 Extrait de code 3.2 – Format des requêtes Q_{chase}

Exemple 3.4.3 (Chase). Etant donné une contrainte $c : getTreatment(x, y), forPatho(y, z) \rightarrow hasPatho(x, z)$. En supposant l'insertion du fait $getTreatment(N_1, résection)$, une étape du chase implique la construction de la requête suivante pour la contrainte $c : Q_{chase}^{[c]} \leftarrow getTreatment(N_1, résection), forPatho(résection, z), \neg hasPatho(N_1, z)$ qui se traduit par la requête Cypher suivante à l'aide du modèle donné par l'extrait 3.2 où $k_N = 1$ (z), $k_{CONST} = 2$ (x et y) et $subs = \{ \langle x \mapsto N_1, y \mapsto résection \rangle \}$.

```

1 UNWIND $subs AS sub
2 MATCH (a1:Atom {symbol: "getTreatment"}),
3     (a1)-[:getTreatment {rank: 0}]->(x:Element),
4     (a1)-[:getTreatment {rank: 1}]->(y:Element),
5     (a2:Atom {symbol: "forPatho"}),
6     (a2)-[:forPatho {rank: 0}]->(y:Element),
7     (a2)-[:forPatho {rank: 1}]->(z:Element)
8 WHERE x.value = sub.x AND y.value = sub.y
9 AND NOT EXISTS {
10     MATCH (a:Atom {name: "hasPatho"}),
11     (a)-[:hasPatho {rank: 0}]->(x:Element),
12     (a)-[:hasPatho {rank: 1}]->(z:Element)
13 } RETURN DISTINCT {"x":x.value, "y":y.value, "z":z.value} AS sub

```

L'exécution de cette requête sur le graphe d'instance donné figure 3.2 se traduit par l'exécution pas à pas suivante :

Etape 1 L'opération UNWIND débute en construisant une table d'instanciation à partir de la liste **subs** fournie en paramètre.

sub
$\{x : N_1, y : résection\}$

Etape 2 La clause MATCH permet la récupération du sous-graphe correspondant au corps de la contrainte pour chaque substitution **sub**. À ce stade, l'instanciation complète de toutes les variables du corps de la contrainte est obtenue.

sub	a1	a2	x	y	z
$\{x : N_1, y : résection\}$	n_{22}	n_{23}	n_1	n_3	n_2

Etape 3 Par la suite, une sous-requête est exécutée afin de vérifier l'existence de la tête de la contrainte pour l'instanciation récupérée à l'étape précédente.

sub	a1	a2	x	y	z	a
$\{x : N_1, y : résection\}$	n_{22}	n_{23}	n_1	n_3	n_2	n_{21}

Ici, pour l'unique instanciation identifiée, la sous-requête n'est pas vide. Cela indique qu'un atome satisfaisant la contrainte est déjà présent dans la base de données, et aucune autre opération n'est donc requise. La requête $Q_{chase}^{[c]}$ renvoie alors un résultat vide.

3.4.3 Chasse en arrière

Il peut arriver que la suppression d'un atome instancié a rende la base de données incohérente lorsqu'il est une conséquence d'une contrainte c . En effet, pour toute contrainte c , s'il existe un homomorphisme h tel que $h(\text{head}(c)) = a$ et une extension de h notée h' tel que $h'(\text{body}(c)) \subseteq \mathcal{D}$, la suppression de a engendre une incohérence. Comme la base ne respecte plus la contrainte c , on doit régénérer la tête en utilisant le Chase afin de maintenir la cohérence. Cependant, il est possible que l'on régénère un atome que l'on veut supprimer. Dans ce cas on utilise le Chase en arrière pour supprimer récursivement les atomes qui interviennent dans le corps pour empêcher le déclenchement de la contrainte. Pour éviter le non-déterminisme, il est supposé que l'atome à supprimer a été marqué du symbol '−' lors de la conception de la règle.

Comme pour les insertions, la procédure Chase4Delete calcule, de manière incrémentale, les effets de bord. *ToDel* représente l'ensemble des atomes à supprimer et *ToIns* l'ensemble des nouveaux atomes à insérer. L'idée est de vérifier si c génère un atome isomorphe à un atome en cours de suppression (ligne 3). Pour ce faire, on utilise la requête $Q_{\text{chase}}^{[c]}$ où l'on instancie la tête de la règle au lieu du corps. Si on génère un atome isomorphe à un atome dans *ToDel*, au moins un atome dans $h(\text{body}(c))$ doit être supprimé afin d'empêcher c d'être déclenchée. Cet atome est ensuite ajouté à l'ensemble des atomes à supprimer *ToDel* (ligne 5). Lorsque l'application de la règle donne un nouvel atome a' non-isomorphe à a , il est inséré et peut à son tour déclencher d'autres règles. Les effets de bord sont calculés à l'aide d'un Chase en avant (ligne 7) avec la procédure Chase4Insert et stockés dans l'ensemble *NewToIns*. Ligne 8, si les effets de bord contiennent un atome de *ToDel* à supprimer et/ou que le degré maximal $\delta_{\max}$ est dépassé : l'application de la contrainte c est invalide et l'atome à supprimer dans le corps est ajouté à *ToDel* (ligne 11). Sinon, les nouveaux atomes sont ajoutés à *ToIns* (ligne 9).

Procédure Chase4Delete($\mathcal{D}, \mathbb{C}, \delta_{\max}, \text{del}_{\text{iso}}$)

```

1 ToIns  $\leftarrow \emptyset$  ;
2 ToDel  $\leftarrow \text{del}_{\text{iso}}$  ;
3 tant que  $\exists c \in \mathbb{C}$  et  $h$  tel que  $h(\text{head}(c)) \in \text{ToDel}$  et  $h(\text{body}(c)) \subseteq (\mathcal{D} \setminus \text{ToDel}) \cup \text{ToIns}$  faire
4   si il existe  $h'$  tel que  $h'(\text{body}(c)) = h(\text{body}(c))$  et  $h'(\text{head}(c))$  est isomorphe à  $h(\text{head}(c))$ 
5     alors
6       ToDel  $\leftarrow \text{ToDel} \cup \{h'(\text{body}^-(c))\}$  ;
7     sinon
8       NewToIns  $\leftarrow \text{Chase4Insert}(\mathcal{D}, \mathbb{C}, \delta_{\max}, \text{ToIns} \cup \{h'(\text{head}(c))\})$ ;
9       si NewToIns  $\cap \text{ToDel} = \emptyset$  et  $\delta(N) < \delta_{\max}$  pour tous les  $N$  dans NewToIns alors
10        ToIns  $\leftarrow \text{ToIns} \cup \text{NewToIns}$ ;
11      sinon
12        ToDel  $\leftarrow \text{ToDel} \cup \{h'(\text{body}^-(c))\}$  ;
13 retourner ToDel, ToIns

```

Exemple 3.4.4 (Chasse en arrière). Soit la contrainte $c_1 : \text{getTreatment}(x, y), \text{forPatho}^-(y, z) \rightarrow \text{hasPatho}(x, z)$. En supposant la suppression du fait $\text{hasPatho}(N_1, \text{cancer})$, une étape du chase implique la construction de la requête suivante pour la contrainte c_1 : $Q_{\text{chase}}^{[c_1]} \leftarrow \text{getTreatment}(N_1, y), \text{forPatho}(y, \text{cancer}), \neg \text{hasPatho}(N_1, \text{cancer})$. Cette requête se traduit en requête Cypher à l'aide du modèle présenté dans l'extrait 3.2 où $k_N = 1$ (y), $k_{\text{CONST}} = 2$ (x et z), $h = \langle x \mapsto N_1, z \mapsto \text{cancer} \rangle$ et $\text{subs} = \{h\}$. La procédure débute avec *ToIns* = $\emptyset$ et *ToDel* = $\{\text{hasPatho}(N_1, \text{cancer})\}$.

La première étape consiste à vérifier qu'aucune contrainte n'est plus satisfaite après avoir supprimé l'atome. Pour ce faire, on utilise la requête $Q_{\text{chase}}^{[c_1]}((\mathcal{D} \setminus \text{ToDel}) \cup \text{ToIns})$ avec la paramètre *subs*. On obtient une réponse non vide, indiquant que la contrainte n'est plus satisfaite, avec l'instanciation $h' = \langle x \mapsto N_1, y \mapsto \text{résection}, z \mapsto \text{cancer} \rangle$. L'instanciation retournée de la tête, $h'(\text{head}(c))$, est isomorphe à $\text{hasPatho}(N_1, \text{cancer})$. Par conséquent, il est nécessaire de supprimer l'un des éléments du corps de la règle pour assurer la cohérence tout en maintenant la suppression demandée. Ici, l'atome marqué $\text{forPatho}(\text{résection}, \text{cancer})$ est supprimé. Le symbol de prédicat *forPatho* supprimé n'apparaît pas dans

la tête d'une règle, signalant la fin de la procédure avec $ToIns = \emptyset$ et $ToDel = \{hasPatho(N_1, cancer), forPatho(résection, cancer)\}$.

Supposons maintenant l'ajout de la contrainte $c_2 : getTreatment(x, y), \rightarrow forPatho(y, z)$. Après la suppression du fait $forPatho(résection, cancer)$, il est maintenant nécessaire de vérifier la contrainte c_2 . La requête $Q_{chase}^{[c_2]}((\mathcal{D} \setminus ToDel) \cup ToIns)$ retourne alors une instantiation $h'' = \langle x \mapsto N_1, y \mapsto \text{résection}, z \mapsto N_4 \rangle$. Comme $h''(head(c_2)) = forPatho(résection, N_4)$ n'est pas isomorphe à $forPatho(résection, cancer)$, l'atome est ajouté, et un chase en avant est effectué pour calculer ses effets de bord. Cela déclenche à nouveau la contrainte c_1 avec l'instanciation $\langle y \mapsto \text{résection}, z \mapsto N_4 \rangle$ produisant l'atome $hasPatho(N_1, N_4)$. Comme le nouvel atome n'est pas isomorphe à un atome de $ToDel$, et aucune autre contrainte n'est à vérifier, la procédure se termine avec $ToDel = \{hasPatho(N_1, cancer), forPatho(résection, cancer)\}$ et $ToIns = \{forPatho(résection, N_4), hasPatho(N_1, N_4)\}$.

3.5 Mise à jour incrémentale

Cette section vise à détailler l'ensemble des opérations incrémentales nécessaires pour maintenir la cohérence d'une base de données lors de mises à jour. Considérons $\mathcal{D}$ comme une instance de base de données cohérente, mais incomplète, et U comme un ensemble de mises à jour. La notation $\mathcal{D} \diamond U$ représente l'insertion ou la suppression des mises à jour dans ou à partir de $\mathcal{D}$. Dans [CHL20], une approche "from-scratch" génère une nouvelle instance de base de données notée $\mathcal{D}' = core(upd(\mathcal{D} \diamond U))$, où upd est la procédure de mise à jour utilisée. L'approche présentée ici est une version *incrémentale* du processus de mise à jour, désignée par $upd|_U$. Ainsi, la nouvelle instance de base de données est définie comme $\mathcal{D}' = core|_{NullBucket}(upd|_U(\mathcal{D} \diamond U))$, où $NullBucket$ est l'ensemble des valeurs nulles impactées par la mise à jour ($upd|_U$) appliquée à $\mathcal{D} \diamond U$.

3.5.1 Insertion

On définit la procédure Insert qui permet l'insertion d'un ensemble d'atomes instanciés I dans l'instance de la base de données $\mathcal{D}$ selon l'ensemble de contraintes $\mathbb{C}$ de façon incrémentale. Cette procédure commence par calculer les effets de bord résultants de l'insertion de l'ensemble I d'atomes instanciés dans $\mathcal{D}$, en utilisant le mécanisme du Chase (ligne 1). Le Core de l'ensemble $\mathcal{D} \cup ToIns$ est ensuite calculé (ligne 3) en prenant en compte uniquement : (a) les valeurs nulles stockées dans NULLBUCKET (récupérées via Q_{bucket} , ligne 2) et (b) les valeurs nulles associées. $Q_{bucket}(I)[S]$ recherche les valeurs nulles présentes dans les atomes moins spécifiques qu'un atome de l'ensemble S . Lors d'une insertion, les degrés des valeurs nulles (ou profondeur dans l'arbre de génération des TGDs) sont initialisés à 0. Lorsqu'une contrainte est appliquée, les valeurs nulles générées se voient attribuer un degré $\delta + 1$ (où δ correspond au degré maximal des valeurs nulles dans le corps de la contrainte) ou 0 s'il n'y a pas de valeurs nulles dans le corps. Si $\delta(N) \geq \delta_{max}$, l'insertion est refusée et $\mathcal{D}$ reste inchangée. Dans l'algorithme Insert, une vérification est effectuée pour s'assurer que toutes les valeurs nulles dans l'instance simplifiée ont un degré inférieur à δ_{max} (vérification via Q_{degree} , ligne 4). Si cette condition est satisfaite, les degrés des valeurs nulles sont remis à 0 (via Q_δ , ligne 5), et $\mathcal{D}'$ est retournée comme instance modifiée.

Exemple 3.5.1. Soit l'ensemble de contraintes $\mathbb{C} = \{c_1, c_2, c_3, c_4, c_5, c_6\}$, $\delta_{max} = 3$ et l'instance $\mathcal{D}$ donné figure 3.2 (page 24).

- $c_1 : getTreatment(x, y), forPatho^-(y, z) \rightarrow hasPatho(x, z)$
- $c_2 : Exam(x, y, z, u) \rightarrow Reveal(z, v)$
- $c_3 : Exam(x, y, z, u) \rightarrow Patient(x, v, w)$
- $c_4 : getTreatment(x, y) \rightarrow forPatho(y, z)$
- $c_5 : getTreatment(x, y) \rightarrow Patient(x, v, w)$
- $c_6 : hasPatho(x, y) \rightarrow Patient(x, v, w)$

En exécutant la procédure Insert pour l'insertion de l'ensemble $I = \{Exam(N_1, N_5, E_1, 01-01-01), Parameter(température, 38), Reveal(E_1, fièvre)\}$, durant le Chase, à la ligne 1, la contrainte c_3 est déclenchée et produit l'atome $Patient(N_1, N_6^1, N_7^1)$. On obtient alors $ToIns = \{Exam(N_1, N_5^0, E_1, 01-01-01), Parameter(température, 38), Reveal(E_1, fièvre), Patient(N_1, N_6^1, N_7^1)\}$.

Procédure Insert($\mathcal{D}, \mathbb{C}, \delta_{max}, I$)

```

1  $ToIns \leftarrow \text{Chase4Insert}(\mathcal{D}, \mathbb{C}, \delta_{max}, I)$  ;
2  $NullBucket \leftarrow \{N_j \mid N_j \text{ est une valeur nulle obtenue par } Q_{bucket}(\mathcal{D} \cup ToIns)\}$  ;
3  $\mathcal{D}' \leftarrow \text{Simplify}(\mathcal{D} \cup ToIns, NullBucket)$  ;
4 si  $Q_{degree}^{[NullBucket, \delta_{max}]}(\mathcal{D}')$  alors
    // L'insertion est valide, on remet la profondeur des valeurs nulles a 0
5      $Q_{\delta}^{[NullBucket, 0]}(\mathcal{D}')$  ;
6     retourner  $\mathcal{D}'$  ;
7 sinon
    // L'insertion n'est pas possible sans générer trop de valeurs nulles.
    // Elle est refusée et on annule la modification
8     retourner  $\mathcal{D}$  ;
```

Pour calculer le Core de $\mathcal{D} \cup ToIns$, la procédure commence par calculer l'ensemble $NullBucket$ à la ligne 2 en utilisant la requête Q_{bucket} . On obtient l'ensemble suivant de valeurs nulles à vérifier $NullBucket = \{N_1, N_2, N_3, N_4, N_5, N_6, N_7\}$. On récupère à la fois les valeurs nulles nouvellement créées et les valeurs nulles potentiellement simplifiables. La procédure *Simplify* (ligne 3) simplifie l'instance en $\mathcal{D}' : Patient(N_1, N_6^1, N_7^1)$ se simplifie avec $Patient(N_1, \text{homme}, 78)$.

$$\mathcal{D}' = \{Patient(N_1, \text{homme}, 78), Exam(N_1, N_5, E_1, 01-01-01), Parameter(E_1, \text{température}, 38), \\ Reveal(E_1, \text{fièvre}), hasPatho(N_1, \text{cancer}), getTreatment(N_1, \text{résection}), forPatho(\text{résection}, \text{cancer}), \\ leadsTo(\text{cancer}, \text{métastase}), concernAna(\text{cancer}, \text{prostate}), Anatomie(\text{prostate})\}$$

Étant donné qu'aucune valeur nulle ne dépasse le degré maximum (ligne 4), la nouvelle instance est validée. Il est à noter que comme le degré des valeurs nulles n'est utilisé que durant le Chase, il est remis à 0 à la ligne 5 avant de retourner l'instance modifiée.

3.5.2 Suppression

La procédure Delete effectue la suppression d'un ensemble d'atomes instanciés I dans une instance de base de donnée $\mathcal{D}$ de façon incrémentale. À la ligne 1, la requête Q_{Iso} récupère tous les atomes de $\mathcal{D}$ isomorphes à un atome de l'ensemble I . Par exemple, si $I = \{P(a, N_1)\}$ et $\mathcal{D} = \{P(a, N_5)\}$, la requête Q_{Iso} renverrait $\{P(a, N_5)\}$. À la ligne 2, la procédure Chase4Delete est utilisée pour calculer de manière incrémentale les effets secondaires. Cette fonction génère deux ensembles d'atomes, $ToDel$ et $ToIns$, représentant respectivement les atomes à supprimer et à insérer comme effets secondaires. Une fois que ces effets secondaires ont été appliqués à $\mathcal{D}$ pour obtenir $\mathcal{D}'$ (ligne 3), la nouvelle instance est simplifiée de manière similaire à l'insertion, en calculant le Core. Les valeurs nulles impactées sont générées à la ligne 4, et l'instance simplifiée est calculée à la ligne 5. Contrairement aux insertions, les suppressions ne sont jamais rejetées : la procédure est garantie de se terminer, produisant au pire des cas une instance vide. Ceci est assuré par la procédure Chase4Delete, qui marquera les nouveaux atomes générés dépassant le degré de valeurs nulles maximum δ_{max} comme étant à supprimer, garantissant ainsi la terminaison de la procédure.

Exemple 3.5.2. Soit l'ensemble de contraintes $\mathbb{C} = \{c_1, c_2, c_3, c_4, c_5, c_6\}$ définie dans la section 3.5.1, $\delta_{max} = 3$ et l'instance $\mathcal{D}$ donné figure 3.2 (page 24).

En exécutant la procédure Delete pour la suppression de l'ensemble $I = \{hasPatho(N_5, \text{cancer})\}$. La première étape à la ligne 1 est de récupérer l'ensemble des atomes isomorphes à la suppression (ici on cherche à supprimer toutes les instances de *hasPatho* pour la pathologie cancer). On obtient alors $del_{iso} = \{hasPatho(N_1, \text{cancer})\}$. Le Chase à la ligne 2 commence par déclencher la contrainte c_1 ajoute la suppression de l'atome *forPatho*(résection, cancer). Cette suppression entraîne le déclenchement de la contrainte c_4 . Comme *getTreatment*(N_1 , résection) est dans $\mathcal{D}$, c_4 produit l'atome *forPatho*(résection, N_6^1) puis

Procédure Delete($\mathcal{D}, \mathbb{C}, \delta_{max}, I$)

```

1  $del_{iso} \leftarrow Q_{iso}^{[I]}(\mathcal{D})$  ;
2  $ToDel, ToIns \leftarrow \text{Chase4Delete}(\mathcal{D}, \mathbb{C}, \delta_{max}, del_{iso})$  ;
3  $\mathcal{D}' \leftarrow (\mathcal{D} \cup ToIns) \setminus ToDel$  ;
4  $NullBucket \leftarrow \{N_j \mid N_j \text{ est une valeur nulle obtenue par } Q_{bucket}^{[ToIns \cup ToDel]}(\mathcal{D}')\}$  ;
5  $\mathcal{D}' \leftarrow \text{Simplify}(\mathcal{D}', NullBucket)$  ;
6  $Q_{\delta}^{[NullBucket, 0]}(\mathcal{D}')$  ;
7 retourner  $\mathcal{D}'$ 
```

$hasPatho(N_1, N_6^1)$ via la contrainte c_1 . On obtient alors les ensembles $ToDel = \{hasPatho(N_1, cancer), forPatho(résection, cancer)\}$ et $ToIns = \{forPatho(résection, N_6^1), hasPatho(N_1, N_6^1)\}$.

Pour calculer le Core de $(\mathcal{D} \setminus ToDel) \cup ToIns$, la procédure commence par calculer l'ensemble $NullBucket$ à la ligne 4 en utilisant la requête Q_{bucket} . On obtient l'ensemble suivant de valeurs nulles à vérifier $NullBucket = \{N_1, N_2, N_3, N_4, N_6\}$. La procédure Simplify (ligne 5) simplifie l'instance en $\mathcal{D}'$. L'instance est finalement retournée après remise à zéro du degré des valeurs nulles (ligne 6).

$$\mathcal{D}' = \{Patient(N_1, homme, 78), Exam(N_1, N_2, N_3, 01-01-01), Parameter(N_3, température, N_4), \\ Reveal(N_3, fièvre), hasPatho(N_1, N_6), getTreatment(N_1, résection), forPatho(résection, N_6), \\ leadsTo(cancer, métastase), concernAna(cancer, prostate), Anatomie(prostate)\}$$

3.6 Evaluation

Les performances de la mise à jour incrémentale sont évaluées à l'aide de benchmarks sur trois jeux de données décrits de façon complète dans l'annexe B et accessibles sur le dépôt Git [CHH⁺23] :

Movie¹ est une base de données fournie par Neo4J ; C'est un graphe de propriété représentant des nœuds films associé à des nœuds représentants des acteurs et des réalisateurs. Une partie du graphe se concentre sur les critiques de cinéma où des nœuds *avis* sont attachés aux films et à l'auteur. Cette base a été convertie, en FOL, avec 8 symboles de prédicats ayant une arité 2 à 4 et 17 contraintes ont été construites.

GOT² pour *GameOfThrone* est une base de données LPG fournie par Neo4J ; Cette base est utilisée pour de l'analyse de graphe et décrit les personnages de la série de livres *A Song of Ice and Fire* et mets en évidence leur interaction et leurs affiliations. Elle est constituée de 19 prédicats ayant une arité 2 à 7 et 26 contraintes.

Social³ est un graphe étiqueté (sans propriétés) généré artificiellement par le Linked Data Benchmark Council (LDBC) pour l'évaluation d'algorithmes de graphes ; Elle synthétise un réseau social, comme un forum, dans lequel des personnes interagissent entre elles via des posts et des commentaires. Cette base se constitue de 8 prédicats d'arité 1 à 2 et 39 contraintes.

Remarque. Conformément à la licence Creative Commons CC-BY 4.0 (<https://creativecommons.org/licenses/by/4.0/deed.fr>) du Linked Data Benchmark Council (LDBC) et bien que le jeu de données **Social** provienne du LDBC, il est essentiel de noter que nous ne sommes en aucuns cas associés au LDBC et que nos résultats ne respectent pas leurs normes ni ne sont approuvés par le LDBC. Veuillez trouver les attributions et les réserves pertinentes sur leur site web <https://ldbncouncil.org/>.

La table 3.1, montre les 9 instances qui ont été générées dont 8 qui contiennent des valeurs nulles. **Social** est un jeu de données qui ne contient, initialement, aucune valeur nulle. Cela a permis de construire des instances contenant un nombre variable de valeurs nulles afin de mesurer l'impact du nombre de valeurs nulles sur les performances.

1. <https://github.com/neo4j-graph-examples/movies>
2. <https://github.com/neo4j-graph-examples/graph-data-science>
3. <https://ldbncouncil.org/benchmarks/snb/>

Instances	Nombre d'atomes	Nombre de nulls	Nombre de règles	Null/Atome (τ)
<i>Movie</i>	604	340	12	0.56
<i>GOT</i>	24 818	17 232	32	0.69
<i>Social</i> _{1K}	2248	190	39	0.08
<i>Social</i> _{10K}	16 559	1183	39	0.07
<i>Social</i> _{10K} ^{10N}	16 559	0	39	0.00
<i>Social</i> _{10K} ^{50N}	16 559	50	39	0.00
<i>Social</i> _{10K} ^{100N}	16 559	100	39	0.01
<i>Social</i> _{10K} ^{500N}	16 559	500	39	0.03
<i>Social</i> _{10K} ^{1000N}	16 559	1000	39	0.06

TABLE 3.1 – Instance des jeux de données utilisée pour l'évaluation des performances

Les benchmarks se composent d'un ensemble d'exécutions ayant chacune un jeu de paramètres différents. Ces exécutions sont construites pour chacune des instances de la table 3.1 en faisant varier le type de mise à jour (insertion ou suppression) et sa taille (1, 5, 10 et 20 atomes). La taille de l'instance est aussi variable, en augmentant artificiellement le nombre d'atomes en les dupliquant n -fois (échelle 1, 2 et 5), avec renommage des termes (on obtient donc des graphes à n composantes connexes). Les exécutions sont réalisées 10 fois après 3 itérations de préchauffage afin de précharger le système, les index et le cache de la base de données qui ne sont pas mesurés. Entre chaque itération, l'instance est remise à zéro et la mémoire est nettoyée pour assurer une cohérence dans le temps d'exécution. L'ensemble du code nécessaire pour reproduire ces tests de performances est disponible sur le dépôt Git [CHH⁺23].

Système Ces benchmarks sont implémentés en Java et exécutés avec Java 16, avec une base de données Neo4J 4.4. Le programme a été exécuté dans des conteneurs Docker installé en version 20.10.21 sur une machine virtuelle Rocky Linux 8.7 avec 4 vCPU et 16 GB de mémoire primaire (8 GB réservés pour la base de données et 5 GB pour la procédure) et avec 1 GB s⁻¹ de débit en lecture/écriture sur le disque. La base de données et la procédure de mise à jour s'exécutent sur le même serveur. Bien que cette configuration permet de comparer différentes instances et implémentations, les résultats obtenus ne sont pas représentatifs des performances en conditions réelles sur un serveur plus performant, sans virtualisation et où le SGBD serait isolé.

3.6.1 Implémentation sur un modèle relationnel

À des fins de comparaison, les algorithmes ont aussi été implémentés sous MySQL 8. L'implémentation de la procédure en FOL sous un modèle relationnel est triviale : étant donné un atome instancié $P(t_1, \dots, t_n)$, notre modèle relationnel consiste à créer une table dont le schéma est $R_P[A_1, \dots, A_n]$ où chaque attribut A_i est une valeur textuelle. C'est-à-dire que chaque symbole de prédicat correspond à une table, chaque atome représente un tuple de la table et chaque terme est représenté par un attribut textuel, préfixé du symbole ' $_$ ' pour les valeurs nulles. La traduction des requêtes logiques vers SQL est triviale. La requête $Q_{chase}^{[c]}$ suit l'idée décrite pour les graphes (section 3.4) et s'écrit en SQL avec la clause NOT EXIST.

Remarque. La construction de l'ensemble LinkedNull ne peut être exprimée par une unique requête SQL et est donc réalisée par la procédure FindLinkedNull suivante. En effet, l'utilisation d'une requête SQL récursive pourrait être envisagée, mais n'est cependant pas appropriée ici. Pour ce faire, une table supplémentaire doit être introduite pour stocker les paires de valeurs nulles liées. L'opération se traduirait alors par :

1. une requête SQL récursive pour calculer la fermeture transitive de cette nouvelle table et ;
2. un balayage de l'ensemble de la base de données pour récupérer tous les atomes correspondants.

De plus, la table supplémentaire doit être maintenue après chaque mise à jour, ce qui peut engendrer un cout non négligeable.

Procédure FindLinkedNull($\mathcal{D}$, $NullBucket$)

```

1  $newNull \leftarrow \emptyset$  ;
2  $allNulls \leftarrow \emptyset$  ;
3  $linkedNullSet \leftarrow \emptyset$  ;
4 tant que  $NullBucket \neq \emptyset$  faire
5    $allNulls \leftarrow allNulls \cup NullBucket$  ;
6   pour tous table  $R_P$  dans le schéma faire
7      $Q_N^{[R_P]} \leftarrow \text{SELECT } * \text{ FROM } R_P \text{ WHERE ANY } (A_1, \dots, A_n) \text{ IN } NullBucket$  ;
8     pour tous tuple  $u$  dans  $Q_N^{[R_P]}(\mathcal{D})$  faire
9        $linkedNullSet \leftarrow linkedNullSet \cup \{P(u)\}$  ;
10       $newNull \leftarrow newNull \cup (nulls(u) \setminus allNulls)$  ;
11    $NullBucket \leftarrow newNull$  ;
12    $newNull \leftarrow \emptyset$  ;
13 retourner  $linkedNullSet$  ;

```

3.6.2 Évaluation des résultats

En raison des besoins élevés en mémoire primaire de la méthode "from-scratch" proposée dans [CHL20], la comparaison est limitée à la base de données **Movie**. Avec l'instance **Movie** à l'échelle 1, ces approches de mise à jour restent comparables bien qu'un avantage pour l'approche incrémentale se dessine, en particulier pour MySQL. Dans nos expérimentations, pour la mise à jour d'une base de données, nous obtenons en moyenne un temps de 9017 ms pour la version en mémoire, 151 ms pour MySQL et 2380 ms pour Neo4J. En considérant une instance cinq fois plus grande, nous obtenons une moyenne de 888 966 ms pour la version en mémoire, 595 ms pour MySQL et 2706 ms pour Neo4J. Ces résultats démontrent l'efficacité de l'approche incrémentale et de l'utilisation d'un SGBD pour la mise à jour efficace d'instances de grande taille. En particulier, on remarque une augmentation beaucoup plus légère du temps d'exécution.

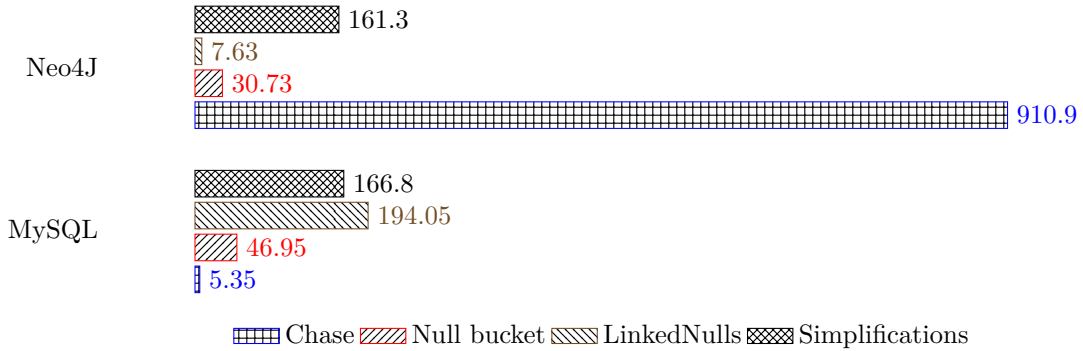


FIGURE 3.3 – Temps moyen d'exécution (ms) par SGBD et par opération

La figure 3.3 montre, pour chacun des SGBD, le temps d'exécution moyen par opération sur l'ensemble des instances (les outsiders présentant plus de 30 s de différences sont supprimés). La récupération des ensembles **LinkedNull** s'est avérée être l'opération la plus coûteuse de la politique de mise à jour avec le modèle relationnel. La modélisation du graphe, présenté dans la section 3.2 (page 22), se concentre sur l'optimisation de cet aspect. Les résultats montrent une nette amélioration de l'extraction des ensembles **LinkedNull** : elle est 25 fois moins coûteuse dans le modèle graphe que dans le modèle relationnel (soit une réduction de 96 %). Cependant, l'utilisation d'un graphe éclaté complexifie la recherche de motif pour l'algorithme du Chase qui est une opération connue pour être très coûteuse [BKM⁺17]. Cette baisse de performance était attendue, mais elle s'avère plus importante que le gain obtenu sur l'extraction des

ensembles LinkedNull.

Analyse détaillée des performances Dans les figures suivantes, on analyse les performances des algorithmes incrémentaux en fonction du nombre de valeurs nulles dans la base (en abscisse), de la taille totale de l'instance (indiqué par ---) sur l'axe de droite et du taux (τ) de valeurs nulles par faits (indiqué par). Les courbes indiquent les valeurs moyennes obtenues pour toutes les itérations correspondantes. Pour améliorer la lisibilité, les graphiques ne tiennent pas compte des résultats pour les occurrences de **GOT** avec plus de 17 000 valeurs nulles. Les figures 3.5 et 3.6 montrent respectivement les résultats obtenus pour MySQL et Neo4J. On remarque que le type de mise à jour (insertion ou suppression) et le taux de valeurs nulles par faits τ ne semblent pas avoir d'impact significatif sur les résultats, quel que soit le SGBD.

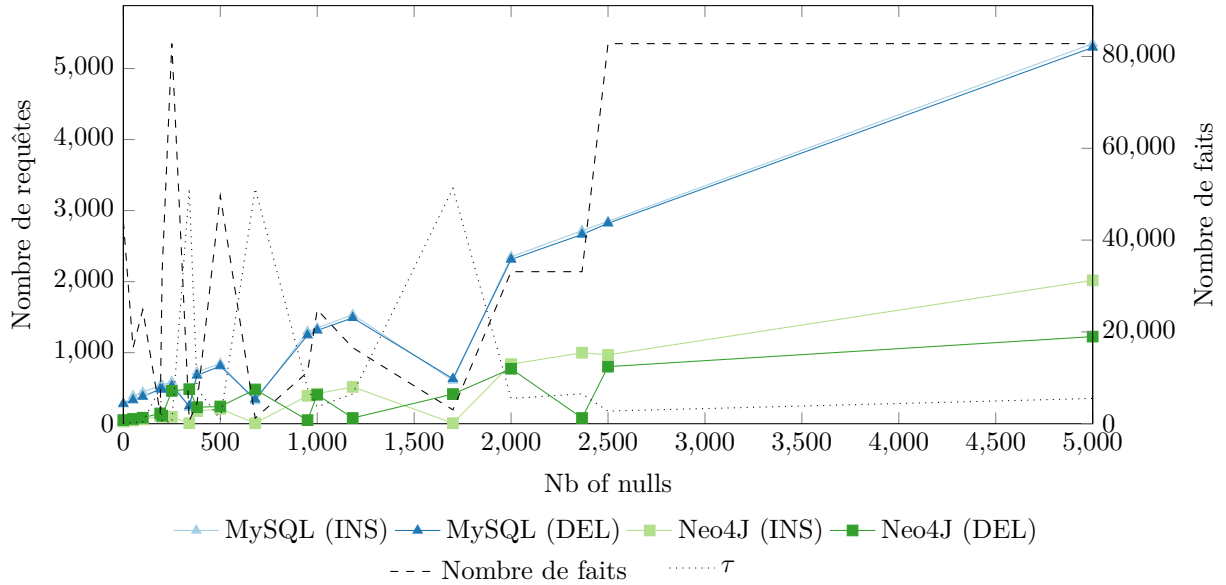
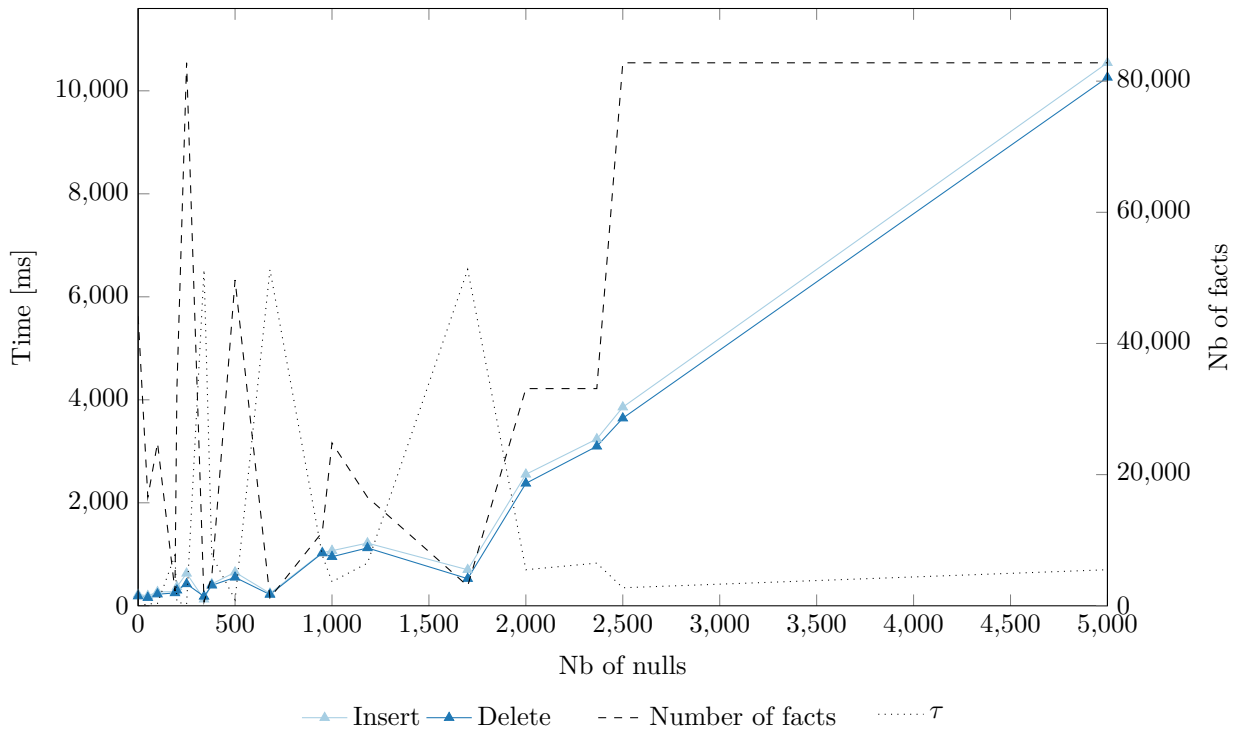


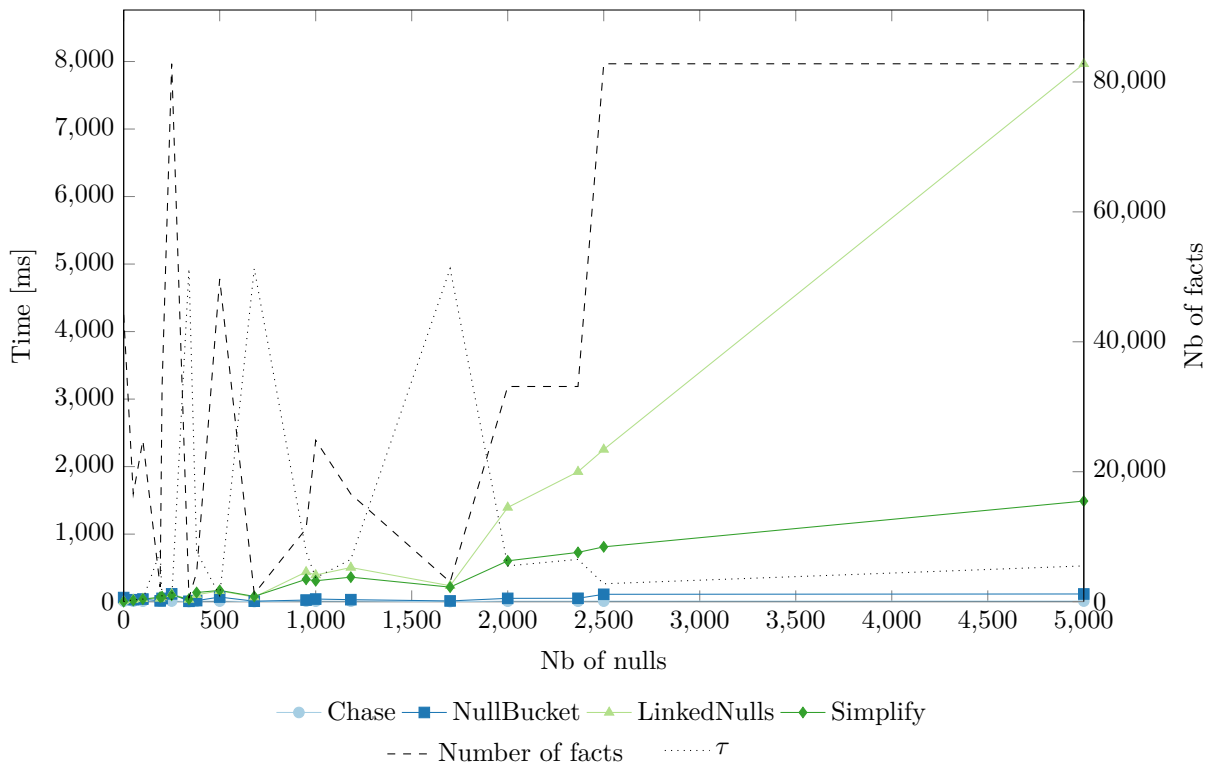
FIGURE 3.4 – Nombre de requêtes émises par nombre de valeurs nulles dans l'instance

La figure 3.4 montre l'évolution du nombre de requêtes soumises à la base de données. Le nombre de requêtes augmente avec le nombre de valeurs nulles dans la base, à l'exception de trois pics où la taille de l'instance est très petite. C'est le cas pour la base de données **Movies** : les pics coïncident avec les situations où seule cette base de données est concernée et on note quand même une augmentation du nombre de requêtes avec un plus grand nombre de valeurs nulles. Cette proportionnalité par rapport au nombre de valeurs nulles s'explique par le fait que la préservation de la cohérence implique la génération de nouvelles données liées par leurs valeurs nulles. Ainsi, en raison de notre méthode de construction, des bases de données plus importantes impliquent davantage de valeurs nulles liées (c.-à-d., des ensembles LINKEDNULL plus importants). Les bases de données plus grandes ont probablement plus de valeurs nulles impactées qui peuvent être simplifiées pendant le calcul du Core, ce qui augmente le nombre de requêtes Q_{core} nécessaires. La différence entre MySQL et Neo4J peut s'expliquer par la construction des ensembles LINKEDNULL qui nécessite plus de requête pour MySQL à cause de l'algorithme FindLinkedNull.

D'après les figures 3.5a et 3.6a on remarque que, pour MySQL, le temps d'exécution dépend du nombre de valeurs nulles alors que pour Neo4J il dépend de la taille de l'instance. Afin de comprendre cette différence, on s'intéresse aux détails des opérations présentées dans les figures 3.5b et 3.6b. Ces graphiques permettent de valider que l'implémentation sous MySQL est grandement impactée par la construction des ensembles LINKEDNULL (indiqué par la courbe $\text{---}\blacktriangle\text{---}$) comme décrit dans la section 3.6.1 et observé dans la figure 3.3. Comme il s'agit de la récupération des ensembles d'atomes liée par des valeurs nulles, il est normal que cette opération soit impactée par leur nombre dans la base. Pour Neo4J, on observe à nouveau que l'opération la plus couteuse est le calcul du Chase (courbe $\text{---}\bullet\text{---}$). Cette dernière est dépendante de la taille de l'instance, car elle consiste à récupérer les instanciations du corps d'une règle. Dans ce cas, les constantes et les valeurs nulles sont traitées de la même manière et il est donc cohérent que cette opération ne dépende pas du nombre de valeurs nulles présentes dans l'instance.

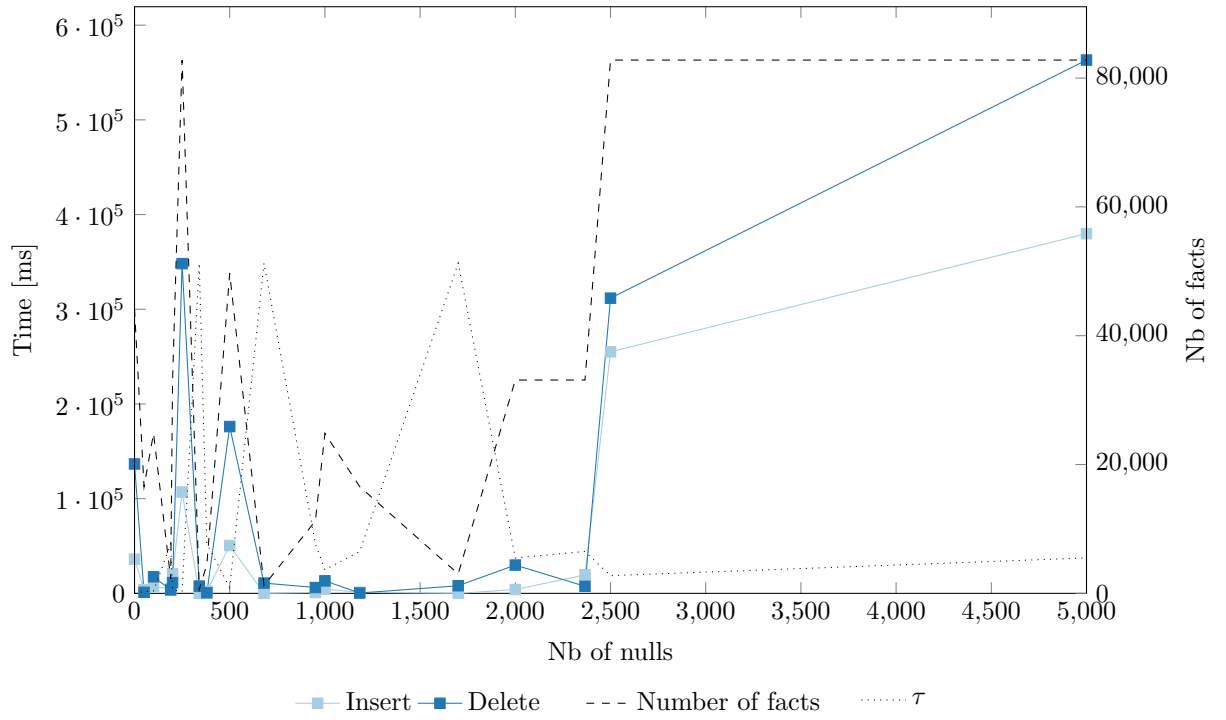


(a) Temps d'exécution par nombre de valeurs nulles dans l'instance

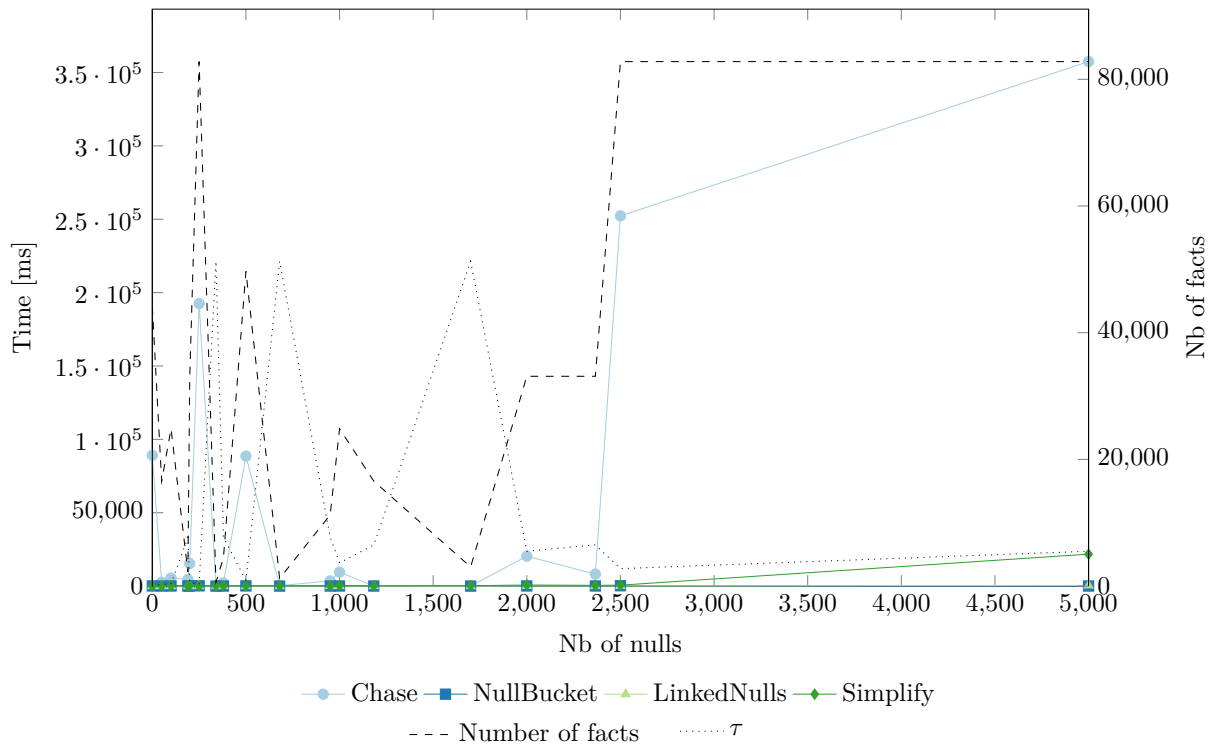


(b) Temps d'exécution de chaque opération par nombre de valeurs nulles dans l'instance

FIGURE 3.5 – Résultats des expérimentations sur 540 scénarios, obtenue par moyenne sur 10 itérations pour MySQL



(a) Temps d'exécution par nombre de valeurs nulles dans l'instance



(b) Temps d'exécution de chaque opération par nombre de valeurs nulles dans l'instance

FIGURE 3.6 – Résultats des expérimentations sur 540 scénarios, obtenue par moyenne sur 10 itérations pour Neo4J

3.7 Conclusion

Les outils d'analyse des données deviennent essentiels dans différents domaines d'application et leur qualité repose sur la cohérence des données. Les travaux présentés dans cette partie du manuscrit visent à améliorer la maintenance cohérente des bases de données incomplètes en proposant des routines incrémentales en amont des systèmes de gestion de base de données qui donne priorité à la mise à jour. Ils étendent les travaux antérieurs de [CHL20], qui proposaient une méthode en mémoire et nécessitant de recalculer l'ensemble de la base. Ces travaux sont fondés sur la sémantique proposée dans [Rei86] pour les données inconnues, afin d'aborder le problème du maintien de la cohérence d'un point de vue logique. Il est possible de rencontrer des valeurs nulles dans les réponses aux requêtes, ce qui implique que, pour le moment, rien dans notre base de données ne nous permet de fournir leur instanciation. Elles peuvent également indiquer une connexion avec d'autres données.

Afin de pouvoir traiter les applications à grande échelle, nous devons viser des solutions efficaces de traitement des données [Sir14], en mettant les solutions incrémentales sur le devant de la scène, en particulier lorsque l'on travaille avec de nouveaux modèles de données (comme c'est le cas dans le contexte XML [BH03, ABH⁺04, BPV04]). Dans le contexte des bases de données graphes, l'approche de [FTX⁺21] propose une méthode pour "incrémentaliser" les algorithmes de graphes abstraits dans un modèle de point fixe. Comme nous l'avons vu précédemment, nous pouvons résumer notre méthode par l'expression suivante $\mathcal{D}' = \text{core}_{\text{NullBucket}}(\text{upd}_U((\mathcal{D} \diamond U))$ où U est l'ensemble des mises à jour requises par l'utilisateur. Cet ensemble est *augmenté* par le biais d'un processus d'inférence qui génère des effets secondaires. La proposition présentée dans [FTX⁺21] nécessite un ensemble *complet* de mises à jour en entrée. En d'autres termes, notre opération de point fixe implique des changements dans l'ensemble des mises à jour, alors que dans [FTX⁺21], l'ensemble des mises à jour est fixe. Leur objectif est de calculer de manière incrémentale de nouvelles réponses sur un graphe mis à jour et non de mettre à jour le graphe de manière incrémentale. Comme le calcul de Core n'est pas un point fixe, il n'entre pas dans le champ d'application de [FTX⁺21].

Dans la littérature, il est courant de limiter l'expressivité des contraintes pour éviter une génération infinie lors du Chase. L'utilisation de $\delta_{\max}$ dans ces travaux permet de traiter des contraintes sous forme de TGDs sans en réduire leur pouvoir d'expressivité. En outre, nous utilisons des simplifications pour maintenir l'instance de la base de données aussi petite que possible et pour éviter la présence de valeurs nulles redondantes, c.-à-d. que la maintenance de la base de données consiste à conserver son Core (qui suit les idées de [FKP05]) dont la mise en œuvre est assurée par une routine de simplification exécutée en association avec des routines de mise à jour.

L'approche a été implémentée et étudiée dans le cadre d'un modèle de base de données graphe et dans le cadre du modèle traditionnel de base de données relationnelle. Les résultats des expériences soulèvent des questions sur la représentation des valeurs nulles dans une base de données graphe. En effet, ces travaux sont aussi un pas vers la mise à jour incrémentale des LPG avec des données incomplètes. Ils illustrent l'impact de la conception des schémas de graphes sur les requêtes et, par conséquent, sur les performances d'une approche de mise à jour incrémentale qui repose sur deux requêtes principales : l'une qui recherche les valeurs nulles liées et l'autre qui recherche les atomes redondants ayant besoin d'être simplifiés. Le modèle de graphe de propriété joue un rôle de plus en plus important aujourd'hui, la gestion des valeurs nulles dans un tel modèle est liée à la définition du schéma et aux questions d'optimisation des requêtes qui doivent être explorées plus avant. Notre schéma de graphe est conçu pour optimiser la recherche des valeurs nulles liées qui est une opération couteuse dans le modèle relationnel et bien qu'anticipé, les performances du Chase dans le modèle graphe rendent son utilisation inappropriée pour de la modification en temps réel. Cela s'explique en partie par l'éclatement des atomes comme des sous graphes qui rend difficile l'exploration de ces derniers.

Une approche plus conventionnelle pour représenter les graphes de propriétés pourrait être envisagée, en particulier si l'on parvient à résoudre la problématique de l'identité dans les graphes, une question explorée par [FL19]. Il est aussi possible d'optimiser le schéma : des techniques visant à réduire la transversalité des arêtes sont mises en œuvre dans [ALQ⁺21]. Une alternative consiste aussi à exploiter la multidimensionnalité des graphes en superposant plusieurs représentations. L'adoption d'un modèle de base de données hybride, tel que suggéré par [HKJ⁺18], pourrait également être envisagée, à condition qu'il offre la flexibilité nécessaire pour gérer diverses représentations des données dans chaque modèle. Pour finir, une piste intéressante serait d'explorer l'adaptation des index utilisés dans les bases de données

3.7. CONCLUSION

graphes pour les bases de données relationnelles. Cependant, le défi réside dans la capacité à maintenir ces représentations de manière efficace. Dans ce contexte, notre approche incrémentale se positionne comme un outil précieux, étant conçue pour être indépendante du modèle de donnée.

Deuxième partie

Intégration de données textuelles

Chapitre 4

Structuration d'information

Sommaire

4.1	Introduction	45
4.2	Des textes aux graphes	46
4.3	Préliminaires	49
4.3.1	Grammaires	49
4.3.2	Arbres	51
4.3.3	Textes et arbres	52
4.3.4	Réécriture d'arbres	53
4.4	Une grammaire comme schéma	54
4.5	Structuration automatique	58
4.5.1	Enrichissement des arbres	59
4.5.2	Extraction de la grammaire	64
4.5.3	Construction des classes d'équivalence	67
4.5.4	Réécriture des arbres	70
4.6	Proposition d'implémentation	70
4.6.1	Identification des groupes	72
4.6.2	Identification des relations	78
4.6.3	Identification des collections	79
4.7	Évaluation préliminaire	81
4.7.1	Comportement de la procédure instanceEvolution	82
4.7.2	Étude critique de la grammaire obtenue	85
4.7.3	Intégration de données	87
4.8	Conclusion	88

4.1 Introduction

L'explosion de la production de données textuelles, alimentée par la numérisation croissante de notre monde, a atteint des proportions phénoménales ces dernières années. Cette abondance de données textuelles représente une mine d'informations précieuses pour la prise de décision, la recherche et le développement, mais leur véritable valeur réside dans la capacité à les traiter efficacement. Le traitement de ces données textuelles est essentiel pour extraire des informations significatives, détecter des tendances, automatiser des tâches, et prendre des décisions éclairées, ce qui en fait un enjeu majeur dans le domaine de l'intelligence artificielle et de l'analyse de données. C'est notamment le cas dans le domaine médical avec les résumés de cas cliniques ou la pharmacovigilance.

Dans ce contexte, le Traitement Automatique des Langues (TAL) est un domaine qui joue un rôle fondamental en utilisant des approches syntaxiques, sémantiques ou à base d'apprentissages, pour extraire de l'information dans des textes. Cela regroupe notamment : la reconnaissance d'entités nommées, l'extraction de relations, les liaisons d'entités nommées avec des individus d'une base de données, l'identification de la coréférence, de la temporalité et de la causalité. Bien que tout cela constitue une étape

majeure, ce n'est pas suffisant pour structurer l'information afin de l'exploiter dans des bases de données traditionnelles. La structuration des données textuelles présente plusieurs avantages, notamment l'amélioration des capacités de recherche, la simplification de l'analyse des données, l'amélioration de la qualité des données grâce à une vérification de cohérence, l'intégration et la réutilisation des informations, la promotion de la collaboration au sein des équipes et la visualisation simplifiée des données à l'aide de graphiques ou de diagrammes.

Organisation La section 4.2 présente un aperçu des diverses approches dans la littérature qui tentent de répondre à cette problématique. La section 4.3 définit l'ensemble des concepts nécessaire à la formalisation des travaux présentés ici. Le chapitre continue ensuite par la définition formelle de l'extraction de schéma (section 4.4) et la description, étape par étape, du processus proposé pour l'extraction de schéma et la structuration automatique d'information (section 4.5). Une proposition d'implémentation du processus est proposée dans la section 4.6 et une évaluation critique des résultats obtenus est réalisée dans la section 4.7. La section 4.8 conclut ce chapitre.

4.2 Des textes aux graphes

Dans cette section, nous abordons succinctement diverses méthodes de représentation de l'information textuelle sous forme de graphe. Nous débutons par des approches où le graphe est simplement une annotation sur le texte, puis examinons des propositions visant à généraliser des informations contenues dans le texte.

Une première façon intuitive de représenter un texte comme un graphe est de visualiser uniquement la syntaxe. Une phrase est composée d'une séquence de mots qu'il est alors possible de représenter sous forme de graphe. Par exemple, la phrase « Alice aime les pommes » peut être représentée par le graphe donné dans la figure 4.1a. Sur ces graphes, en cumulant plusieurs phrases, il est ensuite possible de retrouver des éléments important en utilisant des algorithmes de centralité (comme page rank [PBMW99]), ou des chemins indivisibles. [GZH10] propose notamment d'utiliser ces graphes de mots pour construire des résumés de texte.



FIGURE 4.1 – Représentations syntaxiques d'un texte en graphe

Pour apporter plus d'information que la séquence de mot, le TAL apporte aussi un ensemble d'outils pour la représentation de textes sous forme de graphes. L'arbre de syntaxe représente la structure grammaticale d'une phrase en organisant les mots en fonction de leurs rôles syntaxiques et de leurs relations hiérarchiques. D'autre part, l'arbre de dépendance (figure 4.1b) modélise les relations de dépendance entre les mots d'une phrase, où chaque nœud représente un mot et les arêtes indiquent les relations de gouverneur-dépendant. Il est à noter que ces structures sont différentes selon la langue étudiée. Certains travaux, tendent vers une construction des arbres de dépendance agnostique à la langue du texte à l'aide de Universal Dependencies [Niv15].

Afin d'ajouter de la sémantique dans ces graphes, il est aussi possible de s'appuyer sur les techniques d'extraction d'information (IE) comme la détection d'entité nommée [Pat15]. On peut imaginer représenter un document comme un nœud connecté à un ensemble de nœuds représentant les entités identifiées. Selon la granularité voulue, on pourrait ajouter le découpage du texte en phrase et en mots. Cette approche permet de capturer les relations entre les entités et leur contexte textuel et permet de construire des liens entre les phrases par les entités qui y sont présentes. La puissance des graphes réside dans leur multidimensionnalité, qui permet de représenter simultanément différente modélisation. Cette combinaison d'approches permet une compréhension plus profonde des textes en utilisant de nombreux facteurs.

Ontologies Les ontologies sont un ensemble de concepts interconnectés par des relations permettant de représenter et de raisonner sur un domaine de connaissance spécifique et sont le fondement du web sémantique. Elles se composent d'un ensemble de prédicats utilisés pour représenter les données dans un graphe de connaissance RDF (section 2.2.3, page 12). Dans ce contexte, la tâche d'annotation sémantique (« entity linking ») consiste à lier des parties d'un texte (en leur attribuant des URIs) à des entités dans une ontologie. Cette opération peut être réalisée de façon indépendante ou dans la continuité de l'extraction d'entités [AGOLOV20] (nous étudierons différentes techniques dans le chapitre 5 pour réaliser cette tâche). On retrouve notamment des techniques à base de similarité et de règles [LCL⁺18, AHH20] et d'apprentissage automatique [SHW14, FLP⁺24] dont beaucoup des approches récentes se basant sur des réseaux de neurones [MPS⁺19, SSPV20, LMI⁺20, BDK⁺21, Nav24]. Le processus est souvent le même :

1. Identifier les mentions (partie du texte) qui peuvent être réalisés via l'extraction d'entité nommée ;
2. Identifier les entités candidates dans la base de connaissances ;
3. Classement des entités.

On retrouve des approches qui traitent les entités une à une ou des approches plus globales qui peuvent aider pour la désambiguïsation. La littérature tend à passer des approches séquentielles plus classiques à un processus unique d'annotation via l'apprentissage profond. Ces nouvelles approches ne s'intéressent souvent qu'à l'identification des mentions d'entité et non aux relations. Elles décourraient aussi le lien sémantique important entre les entités et la base de connaissance [HTZ⁺19]. Ces systèmes sont souvent très dépendants de la qualité des bases de connaissances utilisées dans lesquels il y a manque souvent des informations et dont les données ne sont pas toujours nettoyées [WKB18]. De plus, les bases de connaissances sont dynamiques et volatiles, les approches statiques à base d'apprentissage ne permettent pas toujours la mise à jour rapide des modèles.

Toutes ces techniques s'appuient sur des bases de connaissances utilisant des ontologies préétablies et relativement fixe. Cependant, il n'est pas toujours possible de représenter l'information avec ces ontologies. Pour surmonter la complexité et le coût de leur création manuelle, une branche de l'intelligence artificielle s'intéresse à l'apprentissage d'ontologie. La construction d'une ontologie consiste à identifier automatiquement ou semi-automatiquement des termes et des concepts pertinents, leur hiérarchie ainsi que leurs relations. Ces concepts peuvent ensuite être utilisés pour construire une nouvelle ontologie ou en enrichir d'autres. L'apprentissage d'ontologie peut aussi inclure la détection de règles d'inférences. La découverte d'ontologie nécessite souvent une intervention humaine [NJLC20] et la construction complètement automatique reste difficile voire impossible [HER11, WLB12, BM15]. Ici on s'intéresse principalement à l'apprentissage d'ontologie à partir de textes [Kon19, Wāt20, ACG20]. Comme pour l'annotation sémantique, on trouve des approches à base de règles et d'apprentissage automatique.

Historiquement, on trouve des approches, similaires à l'annotation sémantique, à base de motifs syntaxiques (comme « X is a Y ») [Mor99] mais aussi des approches basées sur l'analyse des arbres syntaxiques et de dépendances [GGA⁺02, Niv04]. Ces approches s'intéressent notamment à l'extraction des triplets sujet / verbe / objet. [Bud99], utilise la fréquence de co-occurrence entre les unités lexicales pour identifier les concepts et les relations. Ces approches peuvent ensuite être complétées par du raisonnement logique pour identifier la hiérarchie des concepts, les relations ou construire des règles d'inférences [ZM93]. [ACG20] présente une vue détaillée des différents outils disponibles. L'approche présentée dans cette thèse s'appuie sur certaines de ces techniques. Cependant, les approches classiques comme Text-to-Onto [MV01], Text2Onto [CV05], et TextStorm [APC02] dépendent beaucoup de l'intervention humaine et ne sont finalement que des outils d'aide plus que des processus automatiques. De plus, les techniques qui se basent sur des motifs prédéfinis offrent de bonnes performances [SSM⁺22], mais restent limitées, car ces motifs sont souvent dépendants du domaine étudié.

À partir de 2010, de nombreux travaux ont émergé, mettant l'accent sur les techniques d'apprentissage profond. Ces systèmes offrent généralement des performances supérieures aux systèmes historiques, comme documenté par [ACG20]. Dans cette thèse, nous ne nous intéressons pas à ces systèmes, car ils sont souvent considérés comme des boîtes noires. Il est essentiel, en particulier dans le domaine médical, d'être capable d'expliquer les prises de décision et de fournir un système en lequel les utilisateurs peuvent avoir confiance [RSG16]. Bien que de nombreux travaux tentent de répondre à cette problématique avec l'intelligence artificielle explicative, ce sujet ne sera pas détaillé ici. Une approche plus conventionnelle permet à un humain, même en cas d'automatisation totale, de vérifier et d'intervenir si nécessaire pour corriger les données à chaque étape du processus. Les travaux présentés dans cette thèse ont également l'ambition de viser la construction d'un système adaptable à différents domaines, évitant ainsi de dé-

pendre de l'apprentissage pour un domaine spécifique. De plus, pour les approches basées sur les réseaux de neurones, la demande en données d'apprentissage est souvent élevée comme leur cout financier et environnemental [SGM19].

Dans le cadre des textes, les ontologies peuvent être considérées comme un schéma universel d'annotation, permettant ainsi de rendre les informations extraites depuis les textes interopérables, c'est-à-dire compatibles et échangeables entre différents systèmes et applications. Leur valeur réside dans leur capacité à connecter diverses sources de données et à exprimer la connaissance en établissant des liens sémantiques entre les différents concepts et informations. En revanche, les bases de données offrent une structure homogène où les données sont organisées selon un schéma défini, facilitant ainsi le stockage et la manipulation efficace des données, notamment pour gérer de grands volumes. Les bases de données sont particulièrement adaptées pour l'analyse approfondie et le raisonnement sur les données. De plus, compte tenu de l'intervention humaine quasi systématique dans le processus d'extraction d'ontologie (en dehors de la validation), cette thèse s'intéresse uniquement à la structuration des données en vue de leur exploitation dans une base de données.

ABSTRA ABSTRA [BMU22] est un système conçu pour présenter un ensemble de jeux de données de manière homogène à un utilisateur, dans le but de l'aider à mieux les comprendre et de lui permettre de choisir celui à utiliser et comment il est structuré. L'objectif principal est de représenter les jeux de données comme un ensemble de classes de haut niveau et les relations entre elles. Le système est capable d'analyser un grand nombre de modèles de données structurés ou semi-structurés, tels que RDF, JSON, XML, CSV, des bases relationnelles ou des graphes de propriétés. Le système ABSTRA se compose de trois parties principales. La première partie vise à représenter les jeux de données de manière homogène en introduisant une structure intermédiaire inspirée du modèle Entité-Association (E/A) [Che76]. Cette structure comprend des sous-enregistrements (« sub-record ») qui représentent les données, des enregistrements (« record ») représentant des objets et des collections. Un exemple de cette structure est illustré dans la figure 4.2. Dans un deuxième temps, le système identifie les nœuds de données importants ainsi que les collections principales. Par exemple, si un objet est composé de sous-objets, seul l'objet parent peut être important si les sous-objets ne servent qu'à structurer les informations au sein de l'objet. Cependant, les sous-objets peuvent également être importants s'ils apparaissent ailleurs dans les données. Enfin, ABSTRA classe les collections principales (les types d'objets) pour leur attribuer une étiquette, telle que *Person*, *Product*, etc., permettant ainsi de construire une vision de haut niveau du jeu de données.

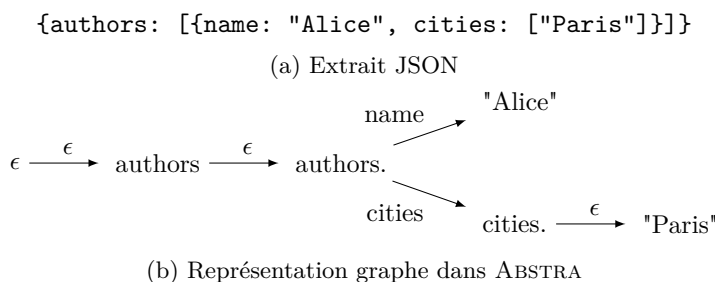


FIGURE 4.2 – Exemple de la structure de graphe utilisée par ABSTRA [BMU22]

Dans ABSTRA et dans nos travaux, on s'intéresse à l'extraction de schéma. Toutefois, contrairement à l'approche présentée dans [BMU22], qui vise à identifier un schéma résumé de données structurées ou semi-structurées, notre objectif est de structurer automatiquement des données non-structurées issues du texte de manière à ce qu'elles soient conformes à un schéma valide et ainsi pouvoir les enregistrer dans une base de données. Cette tâche consiste alors, non seulement à identifier le schéma, mais aussi à construire l'instance associée. Bien que les travaux de [BMU22] diffèrent de ceux présentés dans cette thèse, nous constaterons dans la section 4.4 que la structure de données intermédiaire utilisée dans ABSTRA est similaire à celle proposée dans notre approche. Cette ressemblance témoigne de la nécessité actuelle d'envisager des méta-modèles en tant qu'abstraction pour différents modèles de données. Ce concept apparaît aussi dans [GLDMRA19] et [MAA⁺24].

4.3 Préliminaires

Cette section a pour objectif de définir les concepts fondamentaux qui seront utilisés dans la suite de ce chapitre et se décompose en quatre parties distinctes. La première partie se focalise sur la notion de grammaire. Une grammaire permet de définir un langage en utilisant un ensemble de règles de production qui dérivent les mots du langage sur un alphabet donné. La seconde partie de cette section s'attarde sur les arbres, qui sont des structures de données hiérarchiques. Dans une troisième partie, nous montrerons comment les arbres jouent un rôle important dans la représentation des structures linguistiques. Cette section se termine par la définition des opérations de transformations et d'éditations appliquées aux arbres.

4.3.1 Grammaires

Definition 4.3.1 (Grammaire hors contexte). Une Grammaire hors contexte (CFG) est définie par un quadruplet (N, T, R, S) où N est un ensemble fini de symboles non-terminaux ; T est un ensemble fini de symboles terminaux ; R un ensemble fini de règles de production et $S \in N$ est le symbole initial. Une règle de production est définie par la forme $X \rightarrow \alpha$, avec $X \in N$ et α est une chaîne de symboles terminaux et non-terminaux.

Nous introduisons la notion de CFG condensée qui admet aussi des règles de production de la forme $X \rightarrow \alpha^+$ où α^+ est une expression régulière qui répète une ou plusieurs fois la chaîne α . Cela est équivalent aux règles $X \rightarrow \alpha$ et $X \rightarrow \alpha X$.

Un arbre de dérivation, également appelé arbre d'analyse ou arbre de syntaxe, décrit comment le symbole de départ d'une grammaire G dérive un mot du langage. Si un non-terminal U est associé à une règle de production $U \rightarrow X Y Z$, l'arbre de dérivation peut contenir un nœud interne étiqueté par U , comportant trois enfants, à savoir X , Y et Z , disposés de gauche à droite comme représenté dans la figure 4.3. Chaque nœud interne représente un symbole non-terminal de la grammaire G , tandis que les feuilles symbolisent les symboles terminaux de G . Les liaisons entre les nœuds illustrent la manière dont les symboles sont dérivés les uns des autres. La construction de l'arbre de dérivation s'effectue de manière récursive en suivant les règles de production de la grammaire. Elle débute avec un nœud racine correspondant au symbole initial de la grammaire, et à chaque niveau de l'arbre, les nœuds sont remplacés par des symboles conformément aux règles de production.

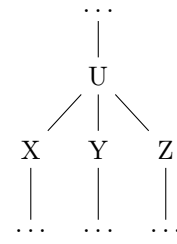


FIGURE 4.3 – Extrait d'un arbre de dérivation pour la règle de production $U \rightarrow X Y Z$

Definition 4.3.2 (Arbre de dérivation). Étant donné une CFG $G = (N, T, R, S)$, un arbre de dérivation de la grammaire est un arbre qui respecte les propriétés suivantes :

1. Le nœud racine de l'arbre est étiqueté par le symbole de départ S , noté $l(\epsilon) = S$;
2. Chaque feuille de l'arbre f est étiquetée par un symbole terminal, c'est-à-dire que $l(f) \in T$;
3. Chaque nœud interne n de l'arbre est étiqueté par un symbole non-terminal, c'est-à-dire que $l(n) \in N$;
4. Si U est un non-terminal utilisé comme étiquette d'un nœud interne n et $X_1, \dots, X_n$ sont les étiquettes des enfants de n de gauche à droite, alors il existe une règle de production $U \rightarrow X_1 \dots X_n$ dans R . Les étiquettes $X_1, \dots, X_n$ représentent une séquence de symboles terminaux et non-terminaux.

Exemple 4.3.1. Soit la grammaire $G = (\{P\}, \{0, 1\}, R, P)$ où R est l'ensemble de règles de productions ci-dessous et qui produit des nombres binaires. La figure 4.4a montre un arbre de dérivation pour 0011.

$$P \rightarrow 0 \qquad P \rightarrow 1 \qquad P \rightarrow P 0 \qquad P \rightarrow P 1$$

Le principe des grammaires attribuées peut se résumer de la manière suivante : nous considérons une grammaire hors contexte. À chacun de ses non-terminaux, on attache deux ensembles de symboles : (1) les attributs synthétisés, qui transmettent l'information depuis les feuilles d'un arbre de dérivation jusqu'à la racine et (2) les attributs hérités, qui transportent l'information en sens inverse. À chaque production, nous associons un ensemble de règles sémantiques qui spécifient comment calculer les attributs de sortie ;

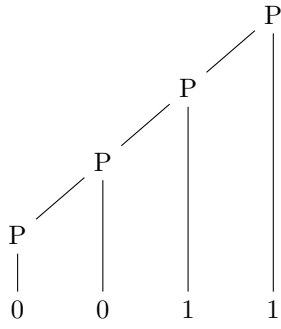
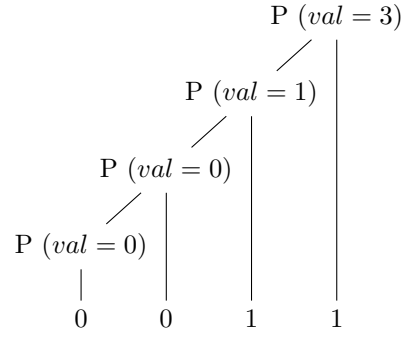
(a) Arbre de dérivation pour la grammaire G définie dans l'exemple 4.3.1(b) Arbre de dérivation pour la grammaire G' définie dans l'exemple 4.3.2 avec les valeurs de l'attribut synthétisé val

FIGURE 4.4 – Exemple de dérivation d'une grammaire pour le nombre 0011

c'est-à-dire les attributs synthétisés du non-terminal membre gauche de la production et les attributs hérités des non-terminaux du membre droit de la production, en fonction des attributs d'entrée de la production (les autres attributs). La définition suivante formalise cette notion.

Définition 4.3.3 (Grammaire attribuée). Une grammaire attribuée [Knu68], ou grammaire avec attributs, est une CFG $G = (N, T, R, S)$ à laquelle on ajoute, pour chaque règle de production $r \in R$ un ensemble de règles sémantiques noté Φ_r et pour chaque symbole $X \in (N \cup T)$ on associe un ensemble fini d'attributs $A(X)$ constitué de deux sous-ensembles disjoints d'attributs :

- $A_{\uparrow}(X)$ pour les attributs synthétisés où $\forall X \in T, A_{\uparrow}(X) = \emptyset$
- $A_{\downarrow}(X)$ pour les attributs hérités où, pour le symbole initial S , $A_{\downarrow}(S) = \emptyset$

Chaque attribut $a \in A(X)$ possède un ensemble (potentiellement infini) de valeurs possibles V_a duquel une valeur est sélectionnée pour chaque apparition de X dans l'arbre de dérivation. r est une règle de production de la forme $X_0 \rightarrow X_1, \dots, X_n$ où $n \geq 1$, $X_0 \in N$ et $X_i \in (N \cup T)$ pour $1 \leq i \leq n$. Une règle sémantique $\varphi_{a_j} \in \Phi_r$ est une fonction $\varphi_{a_j} : \prod_{i=1}^n V_{a_i} \rightarrow V_{a_j}$ où $a_j \in A_{\uparrow}(X_j)$ si $j = 0$, $a_j \in A_{\downarrow}(X_j)$ si $j > 0$ et $a_i \in A(X_i)$. En d'autres termes, une règle sémantique sur $r : X_0 \rightarrow X_1, \dots, X_n$ est de la forme $a_0 \leftarrow \varphi(a_1, \dots, a_n)$ et calcule la valeur d'un attribut de X_0 à partir des attributs des symboles $X_1, \dots, X_n$. Si a_0 est un attribut de X_0 , c'est un attribut synthétisé. Si a_0 est un attribut de X_j ($1 \leq j \leq n$), c'est un attribut hérité.

Exemple 4.3.2. Soit la grammaire attribuée G' construite à partir de G (exemple 4.3.1). La grammaire G' associe la valeur décimale à un nombre binaire. Cela est formalisé par les règles de production suivantes en associant un attribut synthétisé val au non terminal P et en fournissant des règles pour calculer la valeur de l'attribut val par rapport à la valeur de l'attribut val' précédemment calculé et associé au côté droit de la règle de production. Dans les règles de production, les attributs sont indiqués en indice et les règles sémantiques sont présentées, entre crochets, à droite de la règle de production. La figure 4.4b montre, pour chaque niveau, la valeur calculée de l'attribut val pour 0011.

$$\begin{array}{llll}
 P_{val} \rightarrow 0 & [val \leftarrow 0] & P_{val} \rightarrow P_{val'} 0 & [val \leftarrow 2 * val'] \\
 P_{val} \rightarrow 1 & [val \leftarrow 1] & P_{val} \rightarrow P_{val'} 1 & [val \leftarrow 2 * val' + 1]
 \end{array}$$

Les grammaires S -attribuées sont des grammaires attribuées qui contiennent uniquement des attributs synthétisés. Elles ont l'avantage d'être plus simple et plus facilement vérifiable en utilisant uniquement une propagation du bas vers le haut. Elles sont souvent plus adaptées pour des tâches de compilation ou d'analyse sémantique bien que leur pouvoir d'expression soit réduit par rapport aux autres grammaires attribuées qui acceptent les attributs hérités.

Définition 4.3.4 (Méta-grammaire). Une méta-grammaire $\mathbb{G} = (N, T, R, S)$ est une grammaire S -attribuée qui définit la syntaxe des règles de production d'un ensemble de CFG condensées. Autrement dit, chaque mot du langage reconnu par $\mathbb{G}$ sera la liste des règles de productions d'une CFG condensées.

N est l'ensemble fini de méta-non-terminaux, T l'ensemble fini de méta-terminaux, R l'ensemble fini des règles de productions et $S \in N$ le symbole initial. On définit l'attribut synthétisé spécial γ qui permet de vérifier à chaque niveau de l'arbre de dérivation que la dérivation produit bien une CFG condensée valide. Si $S_\gamma = \top$, alors la dérivation de $\mathbb{G}$ est considérée valide, inversement, si $S_\gamma = \perp$ la dérivation de $\mathbb{G}$ est considérée invalide. On distingue alors deux types de règles sémantiques dans une règle de production $r \in R$:

- Les règles de la forme $a \leftarrow \alpha$ où a est l'attribut qui est évalué et α est une formule sur les attributs dans r ;
- Les règles de la forme $\gamma \leftarrow \beta$ où β est une formule logique sur les attributs dans r .

Pour des raisons de lisibilité, on omettra la partie $\gamma \leftarrow$ et les règles de la forme $a \leftarrow a$ dans la suite de ce chapitre.

4.3.2 Arbres

Définition 4.3.5 (Arbre ordonné). Formellement, un arbre T est une paire $T = (D, l)$ où D est le domaine de l'arbre et l est une fonction de d'étiquetage telle que $l : D \rightarrow \Sigma \cup \{\lambda\}$ où Σ est l'ensemble des labels de l'arbre et λ un symbole spécial. Le domaine D d'un arbre est un sous ensemble de $(\mathbb{N})^*$ (c.-à-d. un ensemble de séquence d'entier de la forme $x.y.z$) qui respecte les propriétés suivantes :

1. D est clos sur les préfixes, c'est-à-dire que pour $u, u' \in (\mathbb{N})^*$ si u est un préfixe de u' et $u' \in D$, alors $u \in D$, et
2. Pour tout $u \in (\mathbb{N})^*$ et $j \in \mathbb{N}$ si $u.j \in D$ alors pour tous $i \in \mathbb{N}$ tel que $0 \leq i < j$ on a $u.i \in D$.

Chaque élément de D est appelé *position*. Pour un nœud n à la position p , $|p|$ définit la longueur de la séquence, aussi appelée *profondeur* de n . La racine d'un arbre est à la position ϵ et est représentée par le symbole spécial λ tel que $l(\epsilon) = \lambda$. Un arbre vide est donc défini par $T = (\{\epsilon\}, \langle \epsilon \mapsto \lambda \rangle)$. On note $v \prec u$ si $\exists i \in \mathbb{N}$ tel que $v = u.i$ et on appelle le nœud à la position v *parent* du nœud à la position u inversement u est *enfant* de v . On note $v \prec^* u$ si $\exists v'$ tel que $u = v.v'$, on dit que v est *préfixe* de u et on appelle un nœud à la position u *descendant* du nœud à la position v . On appelle *feuille* un nœud à une position u tel que $u.0 \notin D$, en d'autres termes, c'est un nœud qui n'a pas d'enfants.

Définition 4.3.6 (Sous-arbre). Étant donné un arbre $T = (D, l)$, un sous-arbre de T à la position $u \in D$ est noté $T|_u = (D', l')$ et respecte les propriétés suivantes :

1. $D' \subseteq D$ tel que $\forall v \in D' \ u \preceq^* v$ et
2. $l' = \langle v \mapsto l(v) \mid v \in D' \rangle$.

Soit un sous-arbre $t = T|_u$, on note $t' = P_i^t$ le i -ème sous-arbre parent de t tel que $t' = T|_v$, $u = vw$ et $|w| = i$.

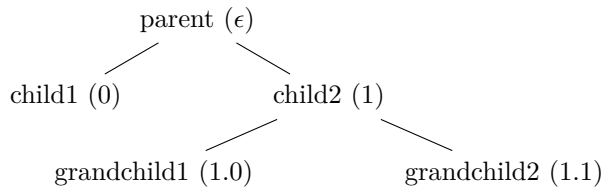


FIGURE 4.5 – Exemple d'un arbre ordonné

Exemple 4.3.3. La figure 4.5 illustre un arbre $T = (D, l)$ ordonné où $D = \{\epsilon, 0, 1, 1.0, 1.1\}$. Dans la figure, les positions sont représentées entre parenthèses. Nous avons par exemple $l(0) = \text{child1}$. $T|_1 = (D', l')$ est un sous-arbre de T . Remarquez que $T|_1$ n'est pas un arbre, car $D' = \{1, 1.0, 1.1\}$ ne respecte pas les conditions requises (définition 4.3.5). Ici, $P_1^{T|_1}$ coïncide avec T .

4.3.3 Textes et arbres

Les textes possèdent une structure complexe, cependant il est possible de les représenter comme des arbres. Il existe deux structures très utilisées dans la littérature :

Arbre syntaxique (Figure 4.6a) Aussi appelé arbre en constituant, est l'arbre de dérivation de la grammaire du texte. En linguistique, il représente la structure syntaxique d'une phrase. Il met en évidence les relations hiérarchiques entre les mots ou groupe de mots et les catégories auxquelles ils appartiennent où Part of Speech (PoS) en anglais : nom (NN), verbe (VBD), adjectif (ADJ), déterminant (DT), etc. Les feuilles de l'arbre sont les unités lexicales (les mots) et les nœuds intermédiaires représente les structures abstraites comme une phrase verbale (VP) ou un groupe nominal (NP).

Arbre de dépendance (Figure 4.6b) Cette représentation met en évidence des liens syntaxique direct entre les unités lexicales représentant une forme de sémantique. Dans l'arbre de dépendance, chaque unité lexicale est un nœud de l'arbre et les arêtes sont labellisés par des relations syntaxiques comme *sujet* ou *objet* (entre un nom et un verbe), ou encore *modifier* entre un adjectif et un nom. Cet arbre est souvent utilisé dans les tâches d'extraction d'information pour récupérer une sémantique entre les mots notamment, car des liens peuvent exister entre des mots très éloignés ce qui est moins évidents dans un arbre syntaxique.

Ces arbres sont obtenus à l'aide d'apprentissages automatiques entraînés à partir d'un corpus appelé banques d'arbres. L'ensemble des labels utilisés dépend des corpus utilisés et de la langue. Pour les arbres de dépendance, Universal Dependencies¹ est un guide d'annotation très répandu.

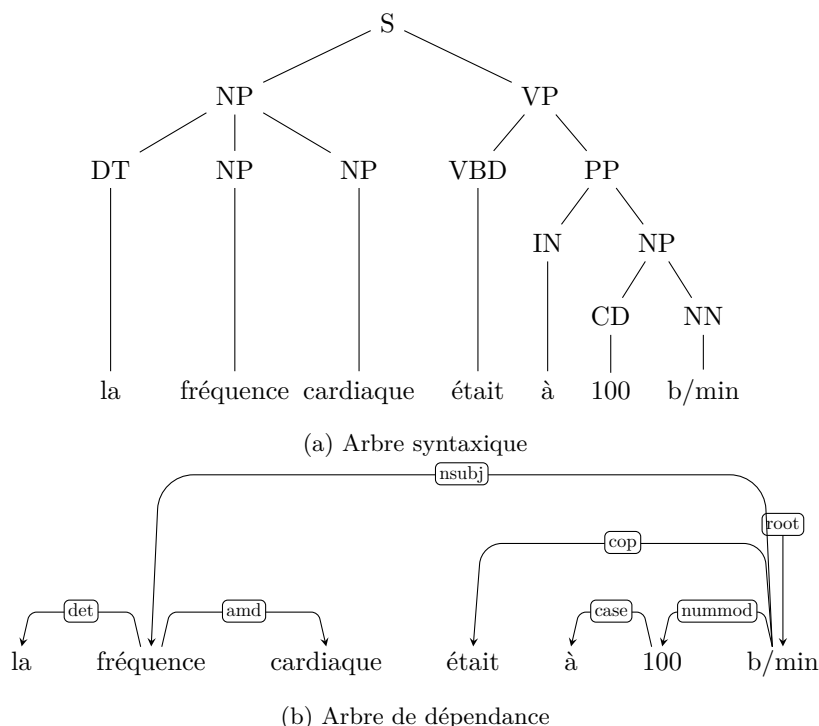


FIGURE 4.6 – Comparaison entre l'arbre syntaxique (obtenu avec CoreNLP) et l'arbre de dépendance (obtenu avec SpaCy) pour la phrase « [...] la fréquence cardiaque était à 100 b/min » issue du corpus CAS [GCD18]

1. <https://universaldependencies.org/>

4.3.4 Réécriture d'arbres

Definition 4.3.7 (Haie). Une haie est définie comme une séquence d'arbres, qui peut éventuellement être vide, représentée sous la forme $h = [t_0, \dots, t_n]$ où $|h|$ représente le nombre d'arbres contenus dans cette haie (ici $|h| = n + 1$). Une substitution, notée σ , est une application bijective d'un ensemble de variables V vers un ensemble de haies et d'un ensemble de labels vers un ensemble de sous-arbres, étendue de manière homomorphe aux arbres.

Definition 4.3.8 (Règle de réécriture). Une règle de réécriture sur un arbre décrit comment un arbre t peut être réécrit en t' pour une position u donnée. Elle est constituée d'une partie gauche, appelée Left-Hand Side (LHS), représentant un motif et d'une partie droite, appelée Right-Hand Side (RHS), représentant la transformation et s'écrit $LHS \rightarrow RHS$. Le motif est un sous-arbre construit sur l'ensemble $\Sigma \cup V \cup \{\lambda\}$ où Σ est l'ensemble des labels de t , V est un ensemble de variables et λ est le symbole racine. Il existe un morphisme entre les variables de LHS et RHS permettant la transformation.

La transformation s'applique sur une substitution σ du motif de la partie gauche par un sous-arbre de t à une position donnée u . En d'autres termes, il existe une correspondance entre les éléments du motif (LHS) et un sous-arbre de t . On note l'application de la règle sur l'arbre t donnant t' avec la substitution σ à la position $u : t \mapsto_{[u, LHS \rightarrow RHS, \sigma]} t'$ tel que $t|_u = \sigma(LHS)$ et $t'|_u = \sigma(RHS)$. Une règle de réécriture peut contenir un ensemble de conditions d'application en plus du motif. Ces conditions spécifient les circonstances dans lesquelles la règle de réécriture peut être appliquée. Les conditions peuvent inclure des contraintes sur les attributs des nœuds ou des arêtes, des contraintes topologiques, etc.

Exemple 4.3.4 (Règle de réécriture). Étant donné la règle de réécriture $rule(u.i)$ et la contrainte $|\sigma(x)| = i$ donnée figure 4.7. U représente le nœud à la position u de l'arbre cible, $x \in V$ est une variable et $\{X, Y, A, B, C, D\} \subseteq \Sigma$ représentent des étiquettes (des non-terminaux) de l'arbre. La règle est applicable sur un arbre T s'il existe un sous-arbre $t = T|_u$ avec une substitution σ tel que $\sigma(x) = [t|_0, \dots, t|_{i-1}]$ et $l(u.i) = A$. En cas d'application de la règle, le sous-arbre A est supprimé de T .

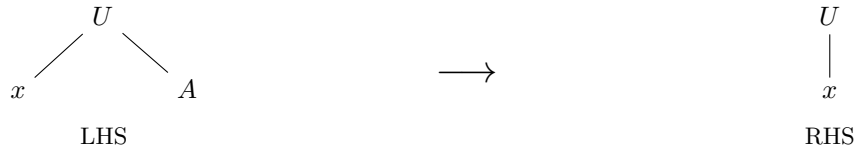


FIGURE 4.7 – Exemple d'une règle de réécriture

Si on considère l'arbre T donné figure 4.8a, avec l'application de la règle $rule(0.2)$, où $u = 0$ et $i = 2$, on obtient $\sigma(x) = [C, B]$. Le résultat de l'application et l'arbre figure 4.8b.

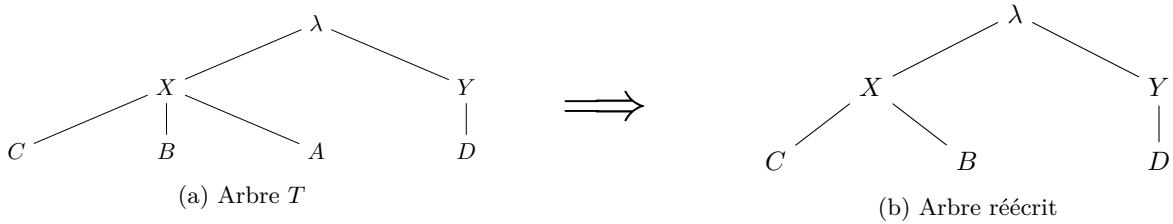


FIGURE 4.8 – Exemple d'application d'une règle de réécriture

On introduit les opérations de réécriture d'arbre élémentaires qui seront utilisés dans ce chapitre. La première règle, `ins_elem` (figure 4.9), permet l'insertion d'un nœud d'étiquette A dans l'arbre T à la position $u.i$. Comme discuté dans l'exemple 4.3.4, la condition $|\sigma(x)| = i$ assure qu'il y a i éléments dans la haie $\sigma(x)$ impliquant que le nœud A se trouve bien à la position $u.i$. Il n'y a aucune contraintes sur y qui a pour rôle de déplacer les éléments à droite de A .

Les figures 4.10 et 4.11 présente la règle de réécriture `del_elem` utilisée pour supprimer un sous-arbre dans un arbre T à une position donnée $u.i$. La règle est récursive : si l'on supprime l'unique enfant a d'un nœud b , c'est le nœud b qui est supprimé (figure 4.11). Pour cela, on utilise la condition $|\sigma(x)| + |\sigma(y)| \neq 0$

pour vérifier si la substitution est non-vidé. Si l'on se retrouve à devoir supprimer l'arbre au complet (figure 4.10), on le remplace par un arbre vide contenant uniquement le symbole racine λ .

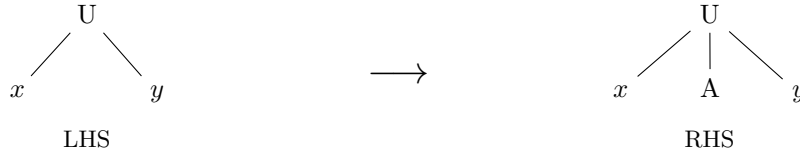


FIGURE 4.9 – $\text{ins_elem}(T, A, u.i)$ où $A \in \Sigma, |\sigma(x)| = i$

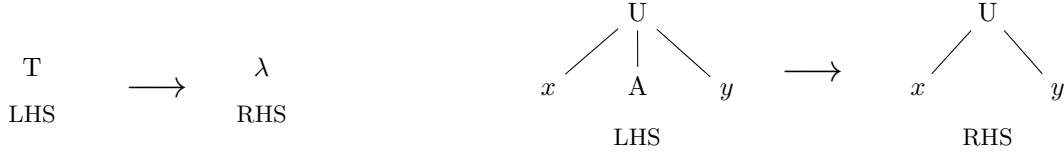


FIGURE 4.10 – $\text{del_elem}(T, \epsilon)$

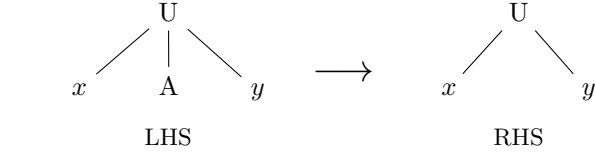


FIGURE 4.11 – $\text{del_elem}(T, u.i)$ où $|\sigma(x)| = i$ si $|\sigma(x)| + |\sigma(y)| \neq 0$, sinon $\text{del_elem}(X, u)$

4.4 Une grammaire comme schéma

Dans cette thèse notre objectif est de transformer des données textuelles en une instance de graphe (ou d'arbre) respectant une grammaire cible G . La construction de cette grammaire cible implique un processus d'extraction de schémas, commençant par la transformation des phrases en arbres syntaxiques. Le processus consiste en plusieurs itérations et est résumé visuellement dans la Figure 4.12, où, sur l'axe vertical, à chaque étape i , nous considérons :

- Une instance I_i sous forme d'une forêt d'arbres syntaxiques enrichis.
- Un arbre quotient S_i construit à partir des arbres de I_i .
- Une grammaire d'arbre G_i (pour les arbres dans I_i) dérivée de l'arbre quotient S_i .

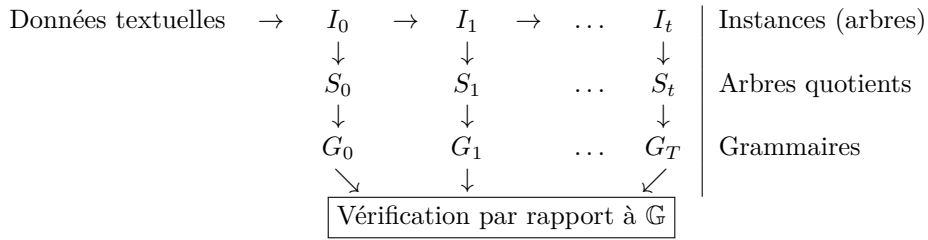


FIGURE 4.12 – Processus itératif pour la structuration automatique

L'axe horizontal de la figure 4.12 représente la progression du processus de l'étape i à l'étape $i + 1$. Cette progression est basée sur le raisonnement suivant :

Initialisation Le processus commence par la transformation des phrases en arbres syntaxiques, ce qui donne lieu à une instance désignée par I_0 .

Enrichissement Une seconde étape consiste à enrichir les arbres de l'instance I_0 en incorporant les informations extraites par une analyse du texte. Il peut s'agir d'insérer des entités, des noms de relations ou d'autres informations pertinentes dans les arbres.

Évolutions successives :

1. **Évolution de l'instance.** L'évolution de l'instance I_i vers l'instance I_{i+1} suit un processus de transformation. Au cours de cette étape, les branches de l'arbre sont regroupées ou transformées sur la base de mesures de similarité. Il peut s'agir de réorganiser la structure des arbres afin d'améliorer leur cohérence ou de les aligner plus efficacement sur la représentation souhaitée.

2. **Évolution de la grammaire.** L'évolution de G_i vers G_{i+1} est déclenchée en vérifiant si la grammaire G_i est conforme à la méta-grammaire générale $\mathbb{G}$. À l'étape i , si G_i n'est pas conforme à $\mathbb{G}$, le processus se poursuit en transformant les structures des arbres de I_i donnant lieu à l'instance I_{i+1} qui engendre un nouveau graphe quotient S_{i+1} et une nouvelle grammaire G_{i+1} . Le processus se termine lorsque nous trouvons une grammaire qui satisfait $\mathbb{G}$.

L'objectif de la méta-grammaire $\mathbb{G}$ est d'établir la description générale des grammaires qui serviront pour structurer des textes. Lorsqu'on considère une instance I , les non-terminaux d'une grammaire G sont définis pour représenter les concepts suivants, qui seront appliqués aux arbres de I :

Arbre d'entité Un arbre de profondeur 1 dont la racine a un nom d'entité comme étiquette et dont les enfants sont des terminaux.

Groupe Un arbre de profondeur 2 dont tous les enfants sont des arbres d'entités.

Relation Un arbre de profondeur 3 ayant exactement deux groupes avec des étiquettes distinctes comme enfants.

Collection Un arbre de profondeur 3 n'ayant pour enfants que des groupes équivalents ou un arbre de profondeur 4 n'ayant pour enfants que des relations équivalentes.

Il est intéressant de remarquer que ces concepts s'alignent sur l'approche présentée dans [BMU22]. Les arbres d'entités peuvent être assimilés au concept de *sub-records* qui se réfèrent à une paire attribut-valeur. Dans I_t , l'instance finale de la figure 4.12, les groupes correspondent aux *records* de [BMU22], alors que dans les grammaires G_i (obtenues dans le processus de construction de G de la figure 4.12), ils servent de formats intermédiaires. Le concept *collection* est identique à celui introduit dans [BMU22].

Méta-grammaire La table 4.1 présente notre méta-grammaire $\mathbb{G}$. Il s'agit d'une grammaire attribuée qui définit tous les schémas (ou les structurations) valides pour nos données. Il est important de rappeler qu'ici un schéma de base de données est représenté par une grammaire hors contexte. Dans cette méta-grammaire $\mathbb{G}$ les méta-non-terminaux sont représentés par des termes entre chevrons $\langle \cdot \rangle$. Les règles sémantiques sont présentées dans la partie droite de la table 4.1, entre crochets $[]$.

La première méta-règle de production de $\mathbb{G}$ (4.1) indique que la grammaire cible G est définie par une règle initiale suivie par une liste de règles, éventuellement vide. La règle initiale engendrée par $\mathbb{G}$ (méta-règle 4.2) contient le symbole *ROOT* (non-terminal initial de G) du côté gauche. Son côté droit est défini par les méta-règles 4.3-4.8 qui spécifie la construction d'une série de non-terminaux de G . Ces non-terminaux sont : *ENT*, *GROUP*, *REL* and *COLL*, représentant, respectivement, des entités, des groupes d'entités, des relations entre des groupes et des collections de groupes ou de relations. Pour distinguer chaque structuration spécifique à une grammaire G , on associe un attribut *name* à chaque non-terminal. Les attributs de $\mathbb{G}$ sont synthétisés et représentent des listes de noms (*name*) utilisés comme identificateur :

- eL (ou eL') : liste contenant les noms des entités,
- gL (ou gL') : liste contenant les noms des groupes,
- rL (ou rL') : liste contenant les noms des relations,
- cgL (ou cqL') : liste contenant les noms des collections de groupes,
- crL (ou crL') : liste contenant les noms des collections de relations.

Leur initialisation commence du bas vers le haut. Les règles syntaxiques permettent, notamment de vérifier qu'un nom est unique. Par exemple, lors de l'application de la méta-règle 4.19-4.20, la liste eL reçoit des noms d'entité qui sont uniques. Cela est aussi le cas pour toutes les autres listes : l'unicité du nom d'un nouveau non-terminal est assurée par les règles sémantiques. Il est également essentiel de noter que ces règles sémantiques garantissent que chaque non-terminal apparaissant du côté droit d'une règle de production de la grammaire G doit avoir une règle le définissant. Par exemple, lors de l'application de la méta-règle 4.1, tous les éléments de la liste gL' doivent être présents dans gL . Dans l'exemple de la figure 4.13, si $gL' = A$ et $gL = A, B$, le non-terminal $GROUP_A$ présent dans la règle $ROOT \rightarrow GROUP_A COLL_1$ à gauche de l'arbre de dérivation (voir figure 4.13) est défini par la règle $GROUP_A \rightarrow ENT_1$, comme illustré dans la partie droite de cet arbre de dérivation (voir figure 4.13).

$$\epsilon ::= \langle \text{root}_{eL',gL',cgL',rL',crL'} \rangle \text{ EOL } \langle \text{ruleList}_{eL,gL,cgL,rL,crL} \rangle \quad [eL' \subseteq eL; gL' \subseteq gL; cgL' \subseteq cgL; rL' \subseteq rL; crL' \subseteq crL] \quad (4.1)$$

$$\langle \text{root}_{eL,gL,cgL,rL,crL} \rangle ::= \lambda \rightarrow \langle \text{rootList}_{eL,gL,cgL,rL,crL} \rangle \quad (4.2)$$

$$\langle \text{rootList}_{eL,gL,cgL,rL,crL} \rangle ::= \epsilon \quad [eL \leftarrow \emptyset; gL \leftarrow \emptyset; cgL \leftarrow \emptyset; rL \leftarrow \emptyset; crL \leftarrow \emptyset] \quad (4.3)$$

$$| \text{ENT}_{\text{name}} \langle \text{rootList}_{eL',gL,cgL,rL,crL} \rangle \quad [\text{name} \notin eL'; eL \leftarrow \{\text{name}\} \cup eL'] \quad (4.4)$$

$$| \text{GROUP}_{\text{name}} \langle \text{rootList}_{eL,gL',cgL,rL,crL} \rangle \quad [\text{name} \notin gL'; gL \leftarrow \{\text{name}\} \cup gL'] \quad (4.5)$$

$$| \text{REL}_{\text{name}} \langle \text{rootList}_{eL,gL,cgL,rL',crL} \rangle \quad [\text{name} \notin rL'; rL \leftarrow \{\text{name}\} \cup rL'] \quad (4.6)$$

$$| \text{COLL}_{\text{name}} \langle \text{rootList}_{eL,gL,cgL',rL,crL} \rangle \quad [\text{name} \notin cgL'; cgL \leftarrow \{\text{name}\} \cup cgL'] \quad (4.7)$$

$$| \text{COLL}_{\text{name}} \langle \text{rootList}_{eL,gL,cgL,rL,crL'} \rangle \quad [\text{name} \notin crL'; crL \leftarrow \{\text{name}\} \cup crL'] \quad (4.8)$$

$$\langle \text{ruleList}_{eL,gL,cgL,rL,crL} \rangle ::= \epsilon \quad [eL \leftarrow \emptyset; gL \leftarrow \emptyset; cgL \leftarrow \emptyset; rL \leftarrow \emptyset; crL \leftarrow \emptyset] \quad (4.9)$$

$$| \langle \text{entity}_{\text{name}} \rangle \text{ EOL } \langle \text{ruleList}_{eL',gL,cgL,rL,crL} \rangle \quad [\text{name} \notin eL'; eL \leftarrow \{\text{name}\} \cup eL'] \quad (4.10)$$

$$| \langle \text{group}_{\text{name},eL'} \rangle \text{ EOL } \langle \text{ruleList}_{eL,gL',cgL,rL,crL} \rangle \quad [\text{name} \notin gL' \wedge eL' \subseteq eL; gL \leftarrow \{\text{name}\} \cup gL'] \quad (4.11)$$

$$| \langle \text{relation}_{\text{name},gL'} \rangle \text{ EOL } \langle \text{ruleList}_{eL,gL,cgL,rL',crL} \rangle \quad [\text{name} \notin rL' \wedge gL' \subseteq gL; rL \leftarrow \{\text{name}\} \cup rL'] \quad (4.12)$$

$$| \langle \text{collGrp}_{\text{name},grpName} \rangle \text{ EOL } \langle \text{ruleList}_{eL,gL,cgL',rL,crL} \rangle \quad [\text{name} \notin cgL' \wedge grpName \in gL; cgL \leftarrow \{\text{name}\} \cup cgL'] \quad (4.13)$$

$$| \langle \text{collRel}_{\text{name},relName} \rangle \text{ EOL } \langle \text{ruleList}_{eL,gL,cgL,rL,crL'} \rangle \quad [\text{name} \notin crL' \wedge relName \in rL; crL \leftarrow \{\text{name}\} \cup crL'] \quad (4.14)$$

$$\langle \text{group}_{\text{name},eL} \rangle ::= \text{GROUP}_{\text{name}} \rightarrow \langle \text{entList}_{eL} \rangle \quad (4.15)$$

$$\langle \text{collGrp}_{\text{name},grpName} \rangle ::= \text{COLL}_{\text{name}} \rightarrow \text{GROUP}_{\text{grpName}}^+ \quad (4.16)$$

$$\langle \text{relation}_{\text{name},gL} \rangle ::= \text{REL}_{\text{name}} \rightarrow \text{GROUP}_{\text{name1}} \text{ GROUP}_{\text{name2}} \quad [\text{name1} \neq \text{name2}; gL \leftarrow \{\text{name1}, \text{name2}\}] \quad (4.17)$$

$$\langle \text{collRel}_{\text{name},relName} \rangle ::= \text{COLL}_{\text{name}} \rightarrow \text{REL}_{\text{relName}}^+ \quad (4.18)$$

$$\langle \text{entList}_{eL} \rangle ::= \text{ENT}_{\text{name}} \quad [eL \leftarrow \{\text{name}\}] \quad (4.19)$$

$$| \text{ENT}_{\text{name}} \langle \text{entList}_{eL'} \rangle \quad [\text{name} \notin eL'; eL \leftarrow \{\text{name}\} \cup eL'] \quad (4.20)$$

$$\langle \text{entity}_{\text{name}} \rangle ::= \text{ENT}_{\text{name}} \rightarrow \langle \text{data} \rangle \quad (4.21)$$

TABLE 4.1 – Méta-grammaire S-attribuée $\mathbb{G}$ au format BNF. Cette grammaire définit des grammaires qui seront vues comme des schémas de base de donnée

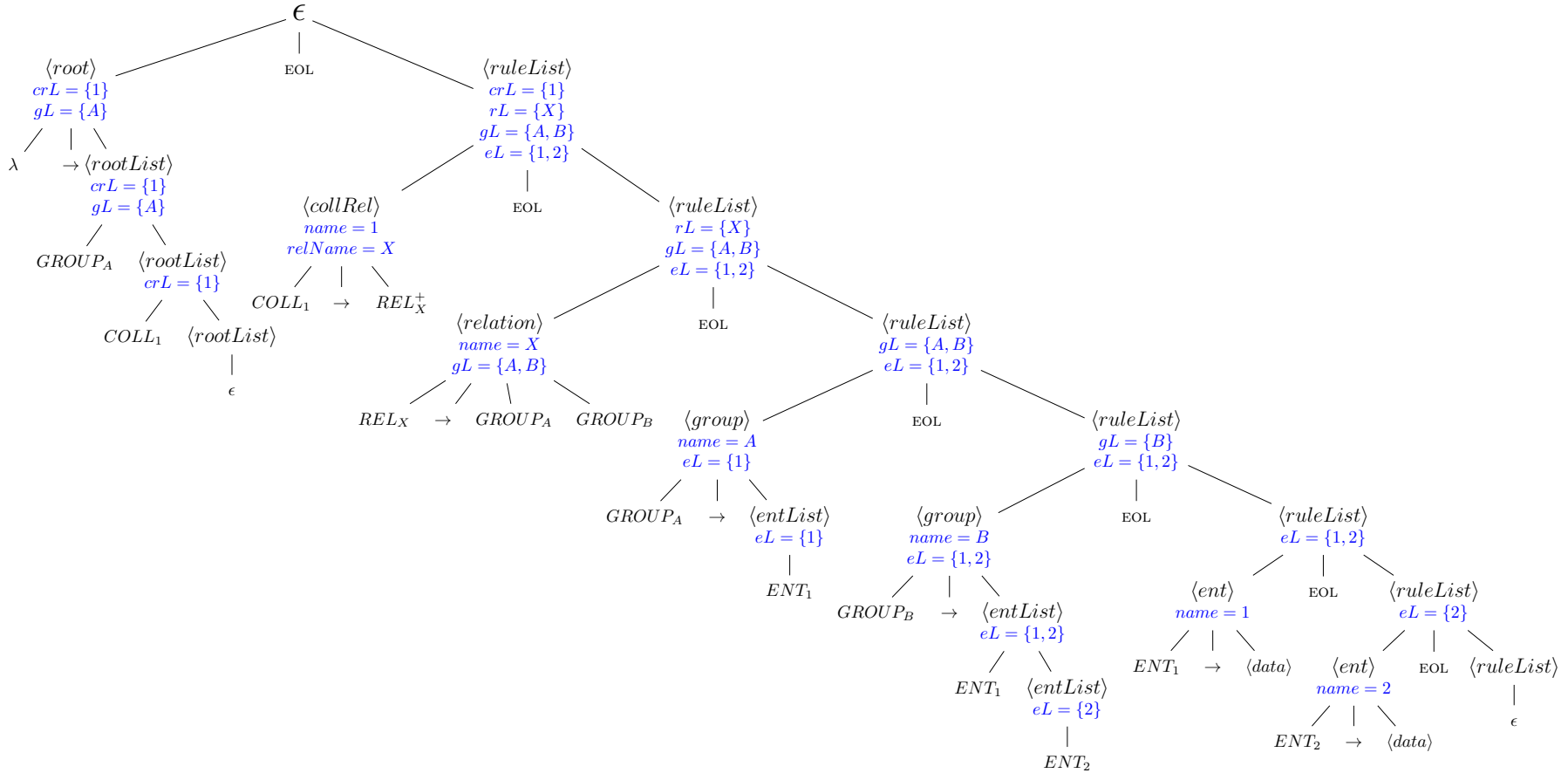


FIGURE 4.13 – Exemple de dérivation de la méta-grammaire

Exemple 4.4.1. La figure 4.13 illustre la dérivation de la grammaire G via la méta-grammaire $\mathbb{G}$. En bleu on note, pour chaque nœud, la valeur de chaque attribut en omettant, pour des raisons de visibilité, γ et les attributs ayant pour valeur un ensemble vide. Les règles de production de la grammaire G sont les suivantes :

$$\begin{aligned}
 & ROOT \rightarrow GROUP_A COLL_1 \\
 & COLL_1 \rightarrow REL_X^+ \\
 & GROUP_A \rightarrow ENT_1 \\
 & ENT_1 \rightarrow \langle data \rangle \\
 & REL_X \rightarrow GROUP_A GROUP_B \\
 & GROUP_B \rightarrow ENT_1 ENT_2 \\
 & ENT_2 \rightarrow \langle data \rangle
 \end{aligned}$$

La règle de production $ROOT \rightarrow GROUP_A COLL_1$ est obtenue par l'application de la méta-règle 4.1, suivie de 4.2. Ce qui nous donne $ROOT \rightarrow \langle rootList \rangle$. Le côté droit de cette règle est un méta-non-terminal. Dans la suite, il est spécifié par l'application de la méta-règle 4.5 qui engendre $ROOT \rightarrow GROUP_A \langle rootList \rangle$, puis 4.8 nous donne $ROOT \rightarrow GROUP_A COLL_1 \langle rootList \rangle$ et 4.3 produit $ROOT \rightarrow GROUP_A COLL_1$. Voir la figure 4.13 avec l'arbre de dérivation. Les méta-règles 4.9-4.14 permettent de définir un ensemble de règles. En effet, le méta-non-terminal $\langle ruleList \rangle$ a déjà été utilisé dans 4.1. Chaque règle produite permet de définir un non-terminal de G . Les règles sémantiques poseront des contraintes, obligeant que chaque non-terminal de notre grammaire cible G soit défini afin de construire une grammaire valide. Dans notre exemple, nous avons $GROUP_A \rightarrow \langle entList \rangle$ et ensuite $GROUP_A \rightarrow ENT_1$.

4.5 Structuration automatique

Les arbres syntaxiques constituent la fondation de la structuration automatique de l'information. L'objectif final est d'obtenir des arbres contenant uniquement les entités hiérarchisées conformément à la structure de l'arbre de syntaxe et valide selon la méta-grammaire $\mathbb{G}$. Plusieurs traitements sont nécessaires pour la préparation des textes à la suite de l'analyse syntaxique. L'objectif est de simplifier les arbres sur lesquels nous travaillons et d'incorporer les informations extraites, telles que les entités et les relations, au sein de l'arbre de syntaxe. En effet, dans le domaine du traitement automatique du langage, la tâche d'extraction d'information inclue généralement une étape d'extraction d'entités nommées suivie par une étape d'extraction de relations entre ces entités. Une entité nommée est une expression linguistique référentielle, souvent associée aux noms propres et aux descriptions définies. Elle est souvent définie comme une représentation d'un objet ou d'un concept du monde physique. Ainsi, les entités nommées peuvent être considérées comme des instances d'une classe. Elles se composent donc d'un type (la classe) et d'une valeur (représentant l'instance). Par exemple, « Paris » peut représenter une entité de type « Ville » avec la valeur « Paris ». Lors de l'extraction des informations dans un texte, une fois que les entités sont identifiées, l'extraction de relations est une tâche qui consiste à détecter si deux entités ou plus sont liées et à classer le lien. Les entités représentent les unités d'information, les données, qui seront structurées en vue de leur enregistrement en base de données.

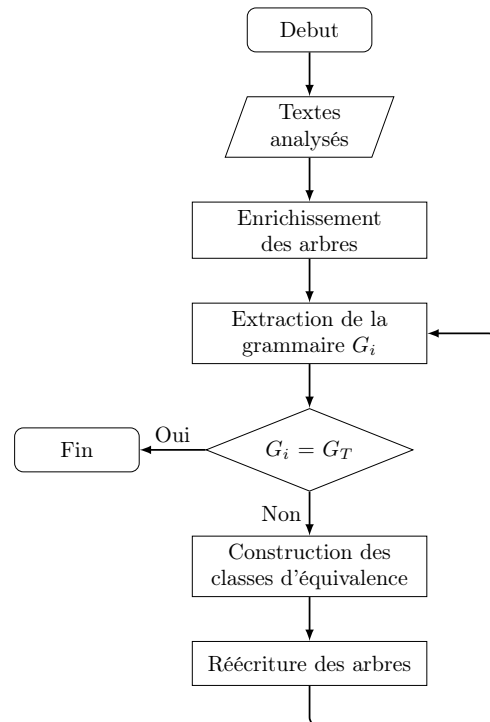


FIGURE 4.14 – Processus de structuration sous forme de diagramme de flux

Définition 4.5.1 (Entité nommée). Une entité nommée est définie comme un tuple $E = (entityName, startToken, endToken)$ où $entityName$ est le nom ou le type de l'entité nommée, $startToken$ est l'indice du token qui marque le début de l'entité, et $endToken$ est l'indice du token qui marque la fin de l'entité. $L_{tokens}(E)$ correspond à la séquence de tokens faisant partie de l'entité, définie comme $L_{tokens}(E) = [startToken, \dots, endToken]$.

La figure 4.14 illustre, à l'aide d'un diagramme de flux, les différentes étapes mises en œuvres pour structurer les données textuelles, c'est-à-dire pour obtenir une grammaire représentant le schéma de nos données. La première étape constitue une phase de préparation des arbres syntaxiques afin d'incorporer les entités et les relations au sein de l'arbre syntaxique. Cette étape vise aussi la simplification de ses arbres en supprimant les structures redondantes. Comme indiqué dans la section 4.4, une seconde étape cruciale consiste à vérifier que nous sommes en mesure de construire un schéma valide qui couvre l'ensemble de nos arbres. Pour cela, il est impératif de pouvoir construire automatiquement la grammaire de schéma à partir des arbres d'instances. Pour finir, si le schéma construit n'est pas valide, il est nécessaire de réécrire les arbres. Pour cela, on commence, dans la troisième étape par calculer l'ensemble des classes d'équivalences pour les non-terminaux. Dans la quatrième étape, la réécriture cherche à unifier les sous-arbres équivalents et à leur donner une structure valide par rapport à la méta-grammaire $\mathbb{G}$.

Les différents traitements proposés dans notre approche reposent sur des règles de réécriture d'arbre (section 4.3.4). Dans la suite, nous présentons de façon en détailée les différentes étapes présentées sur la figure 4.14 (sections 4.5.1 à 4.5.4).

4.5.1 Enrichissement des arbres

L'étape de préparation des textes constitue une phase importante. Les arbres de syntaxe enrichie et simplifiée obtenue après cette phase sont la base de la structuration automatique. La représentation des arbres d'entités et les simplification effectuée sont guidée par la structure que l'on souhaite obtenir.

Procédure `enrich_tree( $T = (D, l)$ , $Ents$ )`

```

// Réécriture des coordinations
1 pour chaque  $p \in D$  faire
2   | fix_coord( $T, p$ );
3   | fix_conj( $T, p$ );
// Enrichissement des arbres
4 Trier l'ensemble  $Ents$  par longueur ( $|E|$ ) de façon décroissante;
5 pour chaque  $E \in Ents$  faire
6   | ins_ent( $T, E$ );
7 pour chaque entité englobante de  $T$  à la position  $u.i$  faire
8   | unnest_ent( $T, u.i$ );
// Simplification des arbres
9 tant que  $\exists p \in D$  tel que  $height(p) = 1$  et  $l(p) \neq ENT$  faire
10  | del_elem( $T, p$ );
11 pour chaque  $p \in D$  faire
12  | reduce( $T, p, \{ENT\}$ );

```

La procédure `enrich_tree` résume les transformations nécessaires au nettoyage et à l'insertion des ensembles d'entités ($Ents$) au sein d'un arbre syntaxiques (T) afin de construire un arbre enrichi. La procédure commence par réécrire les arbres en simplifiant les sous-arbres de coordination (ligne 1). Le tri à la ligne 4 permet de traiter correctement les entités imbriquées afin que leur insertion se fasse du haut vers le bas de l'arbre. La ligne 6 se charge d'ajouter les entités dans l'arbre et la ligne 8 convertit les entités imbriquées. Après l'enrichissement des arbres syntaxiques, l'algorithme s'occupe de la simplification en commençant par supprimer les branches ne contenant pas d'entités (ligne 10) avant de réduire l'arbre (ligne 12).

Dans la suite, cette sous-section présentera de façon détaillée les différentes opérations nécessaires à l'enrichissement des arbres syntaxique.

Conjonctions de coordinations

La structure des arbres syntaxiques est dépendante de la langue étudiée. Plus particulièrement, on s'intéresse à la représentation des conjonctions de coordination, lesquelles se révèlent être d'excellents candidats pour la représentation des collections. La figure 4.15 illustre la disparité structurelle entre le français (4.15a) et l'anglais (4.15b) dans les banques d'arbres utilisées. Le français est annoté avec une structure en profondeur, tandis que l'anglais présente une structure à plat qui sépare bien les composants de la coordination en les mettant au même niveau. Ainsi, la structure de l'anglais se rapproche davantage de la représentation des collections, lesquels sont représentés par un sous-arbre où chaque enfant constitue un élément de la collection. Une première étape dans la préparation des textes implique la transformation (pour le français) des conjonctions vers la structure, plus adaptée, de l'anglais.

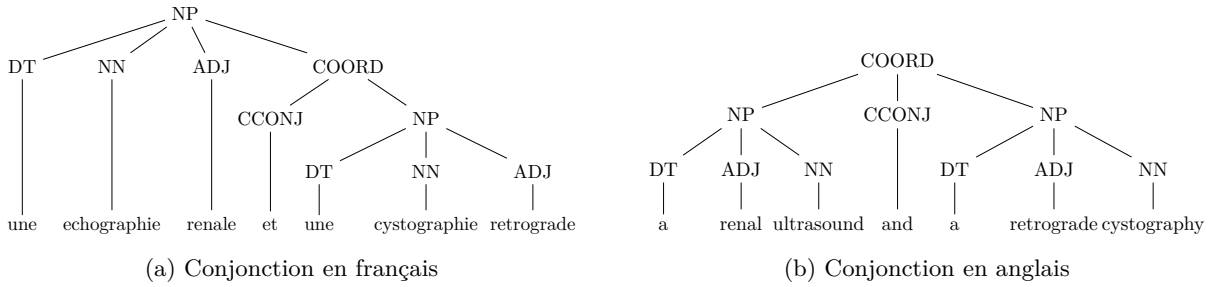


FIGURE 4.15 – Comparaison entre les conjonctions en français et en anglais

Pour ce faire, on introduit deux règles de réécriture : `fix_coord` (figure 4.16) et `fix_conj` (figure 4.17). `fix_coord` est chargée de mettre à plat la structure imbriquée des sous-arbres de conjonction. La règle `fix_conj` est utilisée pour fusionner plusieurs conjonction imbriquées. Par exemple, la phrase « A et B et C » sera représenté par un sous-arbre de conjonction T_1 entre A et T_2 où T_2 est un sous-arbre de conjonction entre B et C. Comme une collection est représentée à plat, cette règle a pour objectif de transformer cette double conjonction en un seul sous-arbre ayant pour enfants A, B et C.

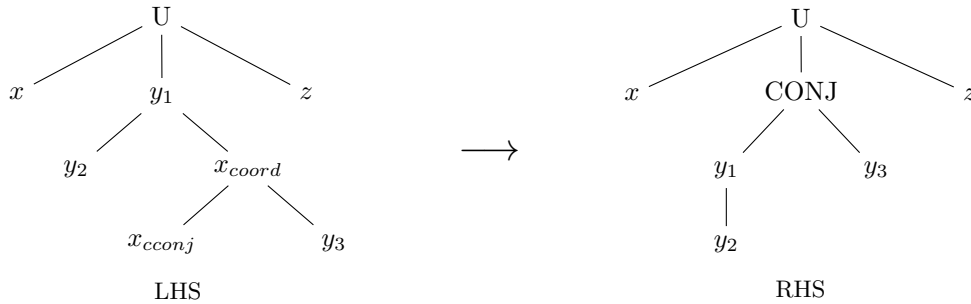
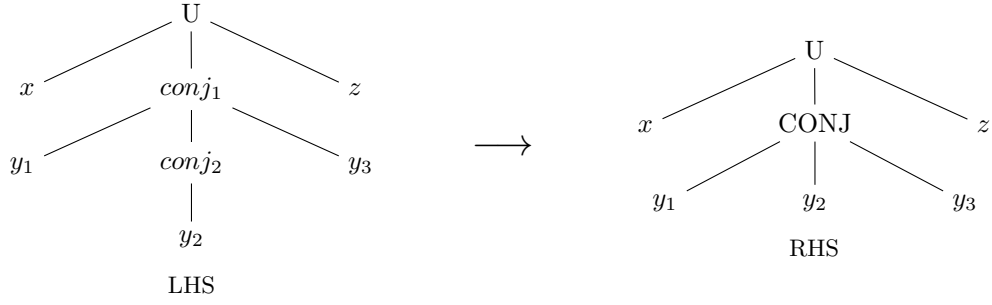


FIGURE 4.16 – `fix_coord`($T, u.i$) où $|\sigma(x)| = i$, $|\sigma(x_{coord})| = 1$, $|\sigma(x_{cconj})| = 1$, $t(\sigma(x_{coord})) = \text{'COORD'}$ et $t(\sigma(x_{cconj})) = \text{'CCONJ'}$

Enrichissement des arbres

L'intégration d'une entité dans un arbre syntaxique se concrétise par la création d'un sous-arbre d'entité avec pour racine un nœud étiqueté *ENT* auquel est attribué le type de l'entité (Personne, Pays, Maladie, etc.) et des feuilles représentant les tokens qui composent l'entité. L'enrichissement des arbres syntaxiques consiste à intégrer l'ensemble des sous-arbres d'entité.

Définition 4.5.2 (Sous-arbre ordonné d'entité). Soit un arbre syntaxique $T = (D, l)$, une entité nommée $E = (\text{entityName}, \text{startToken}, \text{endToken})$ et $L_{\text{tokens}}(E)$ la liste des indices des tokens de E donnée dans la définition 4.5.1 (page 58). Un sous-arbre ordonné d'entité est défini par un tuple $E_T = (\text{entityName}, L_{\text{tree}}(T, E))$. $L_{\text{tree}}(T, E)$ est la séquence des positions des tokens de E dans l'arbre T tel


 FIGURE 4.17 – $\text{fix_conj}(T, u.i)$ où $t(\sigma(\text{conj}_1)) = \text{'CONJ'}$, $t(\sigma(\text{conj}_2)) = \text{'CONJ'}$ and $|\sigma(\text{conj}_2)| = 1$

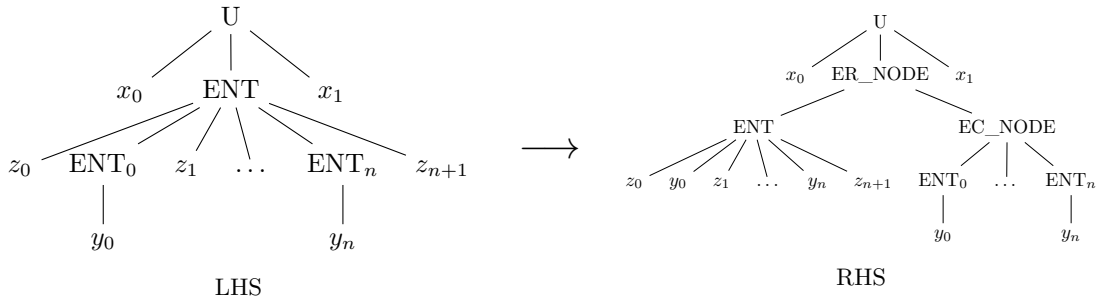
que $L_{\text{tree}}(T, E) = [u.b, \dots, u.e] = [\text{treePos}(T, \text{startToken}), \dots, \text{treePos}(T, \text{endToken})]$ où u est la position commune à tous les tokens de E et $\text{treePos} : \mathbb{T} \times \mathbb{N} \rightarrow D$ est une fonction qui associe pour chaque indice d'un token sa position dans l'arbre. $|E|$ est la taille de l'entité (ou nombre de tokens) telle que :

$$|E| = |L_{\text{tokens}}(E)| = |L_{\text{tree}}(T, E)| = (\text{endToken} - \text{startToken}) + 1$$

Dans certaines situations, les entités peuvent être imbriquées, c'est-à-dire qu'elles peuvent être contenues dans une autre, résultant en deux nœuds entités d'ont l'un est ancêtre de l'autre. Cependant, cette structure est considérée comme invalide selon la méta-grammaire $\mathbb{G}$, où il est spécifié que les nœuds entités ne peuvent pas avoir une entité comme parent. Ainsi, pour s'aligner sur la méta-grammaire, il a été choisi de représenter cette imbrication sous la forme d'une relation entre l'entité englobante et la collection des entités contenues (c.-à-d. un arbre ayant deux enfants : l'entité englobante et l'entité englobée).

Definition 4.5.3. Lorsqu'un nœud du type entité a pour ancêtre un nœud qui est aussi du type entité, la règle de réécriture donnée par la figure 4.18 est appliquée pour introduire un nouveau nœud ER_NODE à la place de l'entité englobante. Ce nœud symbolise la relation entre l'entité englobante et les entités englobées. Ce sous-arbre a pour enfants :

- (i) l'arbre d'entité englobante, qui a pour feuille le texte complet couvert par l'entité englobante et
- (ii) un nœud EC_NODE , symbolisant une collection et possédant comme enfants l'ensemble des arbres d'entités englobées.


 FIGURE 4.18 – $\text{unnest_ent}(T, u.i)$ où $|\sigma(x_0)| = i$

Exemple 4.5.1. La figure 4.19 illustre un exemple d'enrichissement d'un arbre syntaxique pour la phrase : « John Smith takes 500mg of Paracetamol ». Nous considérons l'arbre ordonné présenté dans la figure 4.19a ainsi que les entités suivantes :

- | | | |
|----------------------------|--------------------|---------------------------|
| 1. « John Smith » (Person) | 3. « 500 » (Value) | 5. « Paracetamol » (Drug) |
| 2. « 500 mg » (Dosage) | 4. « mg » (Unit) | |

Pour la première entité, « John » est situé à la position 000 dans l'arbre de syntaxe et « Smith » à la position 010. Le nœud représentant cette entité doit donc être inséré comme enfant du nœud à la position

0, qui correspond au préfixe commun à « John » et « Smith » (respectivement aux positions 000 et 010). Les entités 3 et 4 sont des entités imbriquées dans l'entité 2, comme illustrée dans la figure 4.19b. Par conséquent, il est nécessaire de corriger l'arbre de la figure 4.19b en s'introduisant la structure de relation présentée dans la définition 4.5.3.

On introduit alors le nœud *ER_NODE* à la position de l'entité englobante, c'est-à-dire, comme enfant du nœud *NP* situé à la position 110 dans l'arbre de syntaxe (figure 4.19a). Ainsi, dans la figure 4.19c, *ER_NODE* est racine d'un sous-arbre qui contient :

1. comme premier fils, le sous-arbre ayant l'entité *ENT_DOSAGE* comme racine et deux feuilles : « 500 » à la position 110000 et « mg » à la position 110001 ;
2. comme deuxième fils, le sous-arbre ayant l'entité *EC_NODE* comme racine et comme fils :
 - (a) l'entité *ENT_VALUE* ayant « 500 » comme feuille et
 - (b) l'entité *ENT_UNIT* ayant « mg » comme feuille.

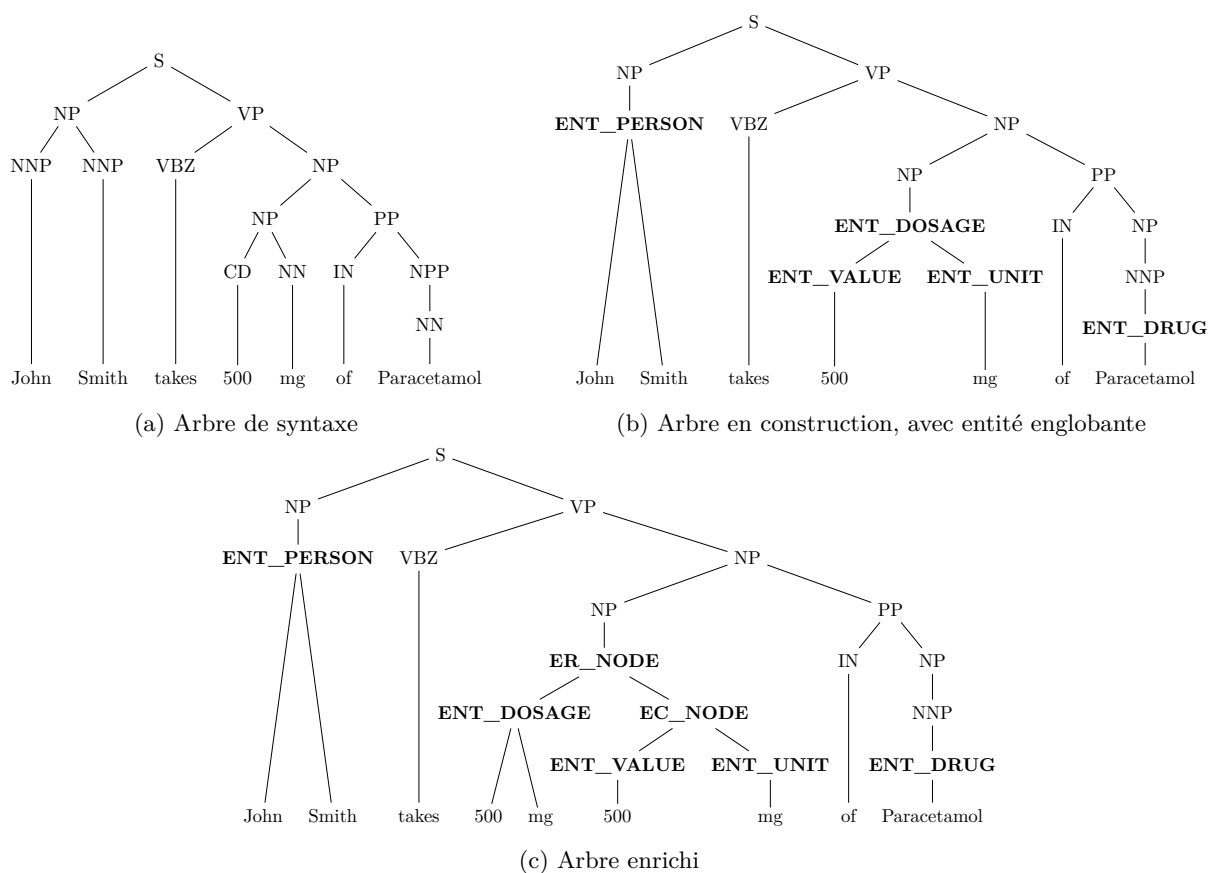


FIGURE 4.19 – Exemple d'enrichissement d'arbre de syntaxe

La procédure *ins_ent* (page 64) met en place l'enrichissement d'un arbre par l'insertion d'une entité. Pour comprendre le rôle de cette procédure, prenons comme exemple l'insertion de l'entité $E_T = (ENT_1, [u.1.0, u.1.1, u.2.0, u.2.1])$ dans l'arbre de la figure 4.20a. Nous rappelons que les labels des nœuds internes sont des non-terminaux. La question consiste à savoir dans quelle position de l'arbre, nous allons insérer le sous-arbre entité avec une nouvelle étiquette, *ENT1*, pour représenter une entité nommée. Dans ce cas simple, l'entité se retrouve comme enfant à la position *u.1* et est le résultat de la fusion des nœuds aux position *u.1* et *u.2*. Le résultat est celui de la figure 4.20b. Néanmoins, dans une situation légèrement différente, nous souhaitons insérer l'entité $E_T = (ENT_2, [u.1.1, u.2.0])$ dans l'arbre de la figure 4.20a. L'idée ici est de l'insérer comme frère à droite de *u.1*, c'est-à-dire, à la position *u.2*. En faisant cela, nous obtenons l'arbre de la figure 4.20c.

L'objectif est de faire évoluer un arbre *T* pour qu'il y contienne un sous-arbre où la racine est étiquetée par le nom de l'entité et ses enfants (feuilles) représentent les tokens qui constituent de cette entité.

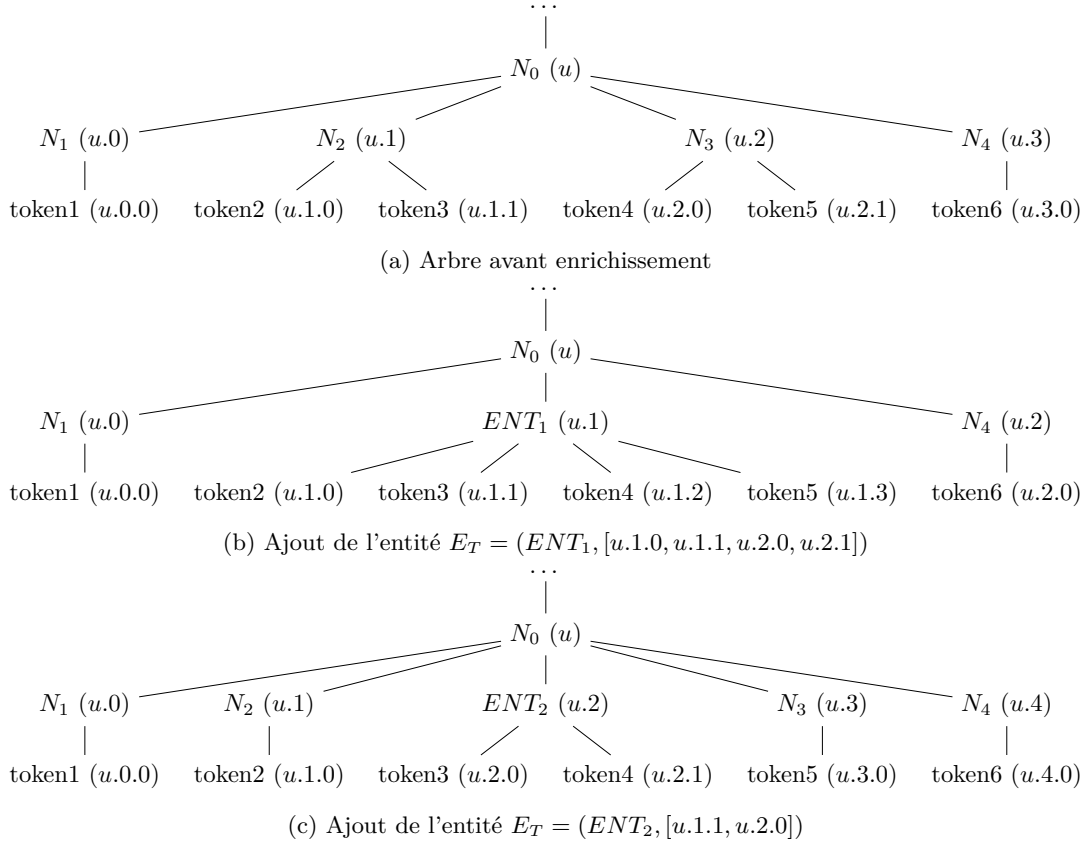


FIGURE 4.20 – Exemple de placement d'un sous-arbre entité

Autrement dit, il s'agit de modifier T pour y placer un sous-arbre entité. Sois u le plus grand préfixe commun des positions dans $L_{tree}(T, E)$. Le sous-arbre entité sera placé comme un fils de la position u . Pour savoir dans quelle position $u.i$ de T nous devons placer le sous-arbre entité, il faut prendre en compte la position $p \in D$ du premier token de E . Dans l'arbre original, soit $T|_{u.x_0}$ le sous-arbre, fils de u , qui contient le premier token de E . Nous allons ajouter le sous-arbre entité comme frère à droite de $T|_{u.x_0}$ et supprimer les sous-arbres contenant les positions originales des tokens. Ainsi, dans le cas de l'entité ENT_1 , l'insertion du sous-arbre d'entité est faite à la position $u.2$, mais la suppression de $T|_{u.1.0}$ entraîne la suppression de $T|_{u.1}$ donnant comme résultat l'arbre de la figure 4.20b. Pour l'entité ENT_2 , le sous-arbre d'entité est inséré à la position $u.2$ et le sous-arbre $T|_{u.1.1}$ est supprimé.

La procédure `ins_ent` a pour but d'effectuer l'évolution de T pour y placer le sous-arbre entité. Elle commence par trouver le plus grand préfixe commun des positions de tous les tokens d'une entité E . La première position $u.b$ sert de guide pour le placement du sous-arbre entité : ce sous-arbre sera placé à la position $u.(x_0 + 1)$ de l'arbre T (ligne 3). Cet ajout peut impacter les positions à droite de $u.(x_0 + 1)$, affectant ainsi les positions des autres tokens. A la ligne 4 les nouvelles positions des tokens sont calculées. La boucle de la ligne 6 nous permet de parcourir les tokens de E en les ajoutant comme feuilles du sous-arbre entité et, en même temps, de les supprimer de leur position originale. Cette série d'insertions et suppressions se fait de droite à gauche, pour éviter de recalculer les positions des tokens à chaque tour de boucle.

Simplifications

La simplification des arbres est faite en deux étapes :

1. On supprime les sous-arbres qui ne contiennent pas d'entité. Ces sous-arbres sont repérables en regardant les parents de toutes les feuilles de T . S'ils ne sont pas labellisés par le nom d'une entité alors ils doivent être supprimés. Par exemple, dans la figure 4.19c, le sous-arbre $T|_{1.1.1.0}$.
2. On supprime les nœuds qui ne sont pas étiquetés comme entité et qui ont un seul fils. Par exemple,

Procédure $\text{ins_ent}(T, E)$

```

1  $[u.b, \dots, u.e] \leftarrow L_{tree}(T, E)$  ; //  $u$  est le plus grand préfixe commun des positions des tokens de  $E$ 
   dans l'arbre  $T$ 
2  $u.x_0.x_1 \dots x_n \leftarrow u.b$  ; // Pour  $u.b$ , nous explicitons chaque composant de  $b$ :  $x_0, x_1, \dots$ 
   //  $u.x_0$  est la racine du sous-arbre, fils de  $u$ , qui contient le premier token de  $E$ 
3  $\text{ins\_elem}(T, \text{entityName}, u.(x_0 + 1))$  ; // On ajoute à droite de la position  $u.x_0$  un nouveau noeud
   étiqueté avec le nom de l'entité à ajouter. C'est donc la racine du sous-arbre entité.
4  $[u.b', \dots, u.e'] \leftarrow L_{tree}(T, E)$  ; // On recalcule les positions des tokens, éventuellement modifiées
   par l'insertion précédente. On obtient ainsi une nouvelle liste de positions
5  $j \leftarrow |E| - 1$ ;
6 tant que  $j \geq 0$  faire
   // On parcourt toutes les positions de la liste  $[u.b', \dots, u.e']$  en partant de la fin
7    $pos \leftarrow [u.b', \dots, u.e'][j]$ ; //  $pos$  est la position d'un token
8    $\text{ins\_elem}(T, l(pos), u.(x_0 + 1).0)$ ; // On insère le token comme feuille du sous-arbre entité
9    $\text{del\_elem}(T, pos)$  ; // On supprime la feuille qui originellement correspondait au token qui vient
   d'être rajouté
    
```

dans la figure 4.19c, le noeud à la position 0. Pour ce faire, on utilise successivement la règle de réécriture *reduce* présentée dans la figure 4.21.

Exemple 4.5.2. Dans l'arbre donné dans la figure 4.19c, on remarque que pour le noeud *ENT_PERSON* le noeud parent *NP* peut être supprimé. Le sous-arbre du noeud *VBZ* ne contient aucune entité et peut donc être supprimé. Sa suppression rend le noeud parent *VP* supprimable à son tour, car il ne possède alors plus qu'un seul enfant. En appliquant le raisonnement à tout l'arbre, on obtient l'arbre donné dans la figure 4.22.

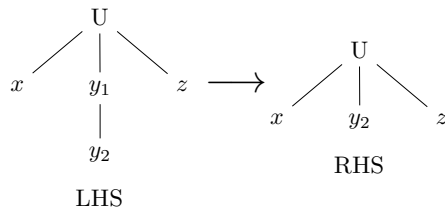


FIGURE 4.21 – $\text{reduce}(T, u.i, S_{labels})$ où $|\sigma(x)| = i$ et $|\sigma(y_1)| = 1$ avec $|\sigma(y_2)| = 1$ si $S_{label} = \emptyset$; $t(u.i) \notin S_{labels}$ si $S_{label} \neq \emptyset$

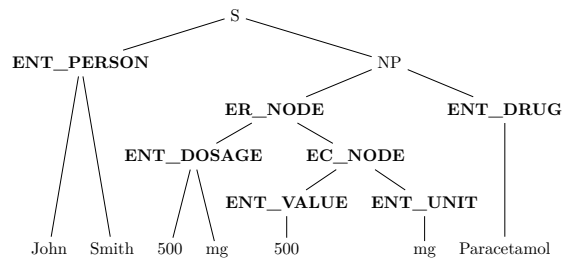


FIGURE 4.22 – Exemple d'arbre enrichie simplifiée

4.5.2 Extraction de la grammaire

Pour réaliser la migration de l'instance d'une grammaire vers une autre, il est indispensable de pouvoir récupérer la grammaire à partir de l'instance afin de pouvoir la vérifier à l'aide de la méta-grammaire $\mathbb{G}$. Étant donné une instance I reconnue par une grammaire G , il est possible de récupérer la grammaire G à partir de I en calculant l'arbre quotient S . Dans la suite, nous introduisons cette notion. Rappelons d'abord qu'en mathématiques, une *partition* d'un ensemble X est un regroupement de ses éléments en sous-ensembles non-vides et mutuellement disjoints. Toute *relation d'équivalence* sur un ensemble définit une partition de cet ensemble, et toute partition définit une relation d'équivalence. Une *famille d'ensembles* F est une partition d'un ensemble X si et seulement si toutes les conditions suivantes sont remplies : (i) $\emptyset \notin F$; (ii) $\bigcup_{A \in F} A = X$ et (iii) $\forall A, B \in F (A \neq B) \Rightarrow (A \cap B = \emptyset)$. Les ensembles de F sont appelés *blocs*. En théorie des graphes, un graphe quotient Q d'un graphe G est un graphe dont les sommets sont des blocs d'une partition des sommets de G et où un bloc A est adjacent à un bloc B si au moins un sommet de A est adjacent à un sommet de B par rapport à l'ensemble des arêtes de G . Pour construire

la grammaire à partir d'un arbre, on introduit la définition d'*arbre quotient*, ce qui correspond à un graphe quotient sans cycles et sans sommets ayant des multiples parents. Dans la construction d'un arbre quotient la définition de la fonction **Succ** ci-dessous est nécessaire.

Définition 4.5.4. Étant donné un arbre $T = (D, l)$ et une relation d'équivalence R . Soit $D/R = \{C_0, \dots, C_n\}$ l'ensemble des classes d'équivalences de T , on définit la fonction **Succ** tel que :

$$\text{Succ}(C) = \{C' \mid \exists u \in C, v \in C' \text{ tq. } u \prec v\} \quad (4.22)$$

Pour réaliser l'extraction des grammaires G_i nous utilisons la relation d'équivalence R_l entre les labels d'un arbre T , définie par $(\forall x \in D, y \in D) x R_l y \iff l(x) = l(y)$. Intuitivement, on obtient alors comme classes d'équivalences un ensemble de positions pour chaque étiquette présente dans l'arbre T . Par exemple, dans la figure 4.23a, $C_\lambda = \{\epsilon\}$ et $C_X = \{0, 1\}$ sont les classes d'équivalence pour les étiquettes λ et X respectivement. La fonction **Succ** permet de calculer la hiérarchie des classes d'équivalence où pour une classe donnée C , la fonction retourne l'ensemble des classes d'équivalences qui ont au moins un élément enfant d'un élément de C . Nous pouvons également écrire que $\text{Succ}(C) = \bigcup_{u \in C} \{C' \mid \exists i \ u.i \in C'\}$. L'exemple suivant illustre cette fonction.

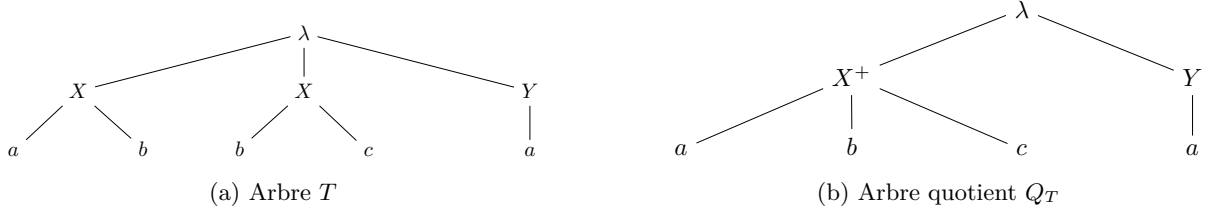


FIGURE 4.23 – Exemple du quotient d'un arbre

Exemple 4.5.3. Étant donné l'arbre $T = (D, l)$ donné en figure 4.23a. La première étape de la construction de l'arbre quotient Q_T commence par la récupération des classes d'équivalences de D données par la relation R_l . On obtient alors les classes suivantes :

$$\begin{array}{lll} C_\lambda = \{\epsilon\} & C_X = \{0, 1\} & C_Y = \{2\} \\ C_a = \{00, 20\} & C_b = \{01, 10\} & C_c = \{11\} \end{array}$$

La construction de Q_T implique la récupération de la hiérarchie des ensembles équivalents. L'algorithme commence par la classe C_λ et parcourt récursivement ses successeurs. On a, pour chaque classe, les successeurs suivants :

$$\begin{array}{lll} \text{Succ}(C_\lambda) = \{C_X, C_Y\} & \text{Succ}(C_X) = \{C_a, C_b, C_c\} & \text{Succ}(C_Y) = \{C_a\} \\ \text{Succ}(C_a) = \emptyset & \text{Succ}(C_b) = \emptyset & \text{Succ}(C_c) = \emptyset \end{array}$$

Dans l'exemple 4.5.3, la figure 4.23a représente une instance des données. L'arbre quotient permet de construire la CFG qui reconnaît l'instance.

Avec les classes d'équivalence, la construction de l'arbre quotient implique désormais l'association de chaque classe à une position d'un arbre ordonné. Nous introduisons la fonction **QDom** qui permet de créer le domaine de cet arbre quotient, en utilisant la fonction **Succ**. Elle a pour objectif d'associer à chaque classe successeur de C une position relative $u.p$ dans l'arbre quotient, où u est la position de la classe C .

Remarque. Une classe d'équivalence peut aussi apparaître comme successeur de différentes classes. Dans l'exemple 4.5.3, C_a apparaît comme successeur de C_Y et de C_X . Dans ce cas, nous aurons différentes positions associées à C_a .

Étant donné un arbre T , la fonction **Quotient** construit l'arbre quotient, noté $Q_T = (Q_D, Q_l)$. Cette construction commence par la racine qui représente la classe d'équivalence C_λ . Un ensemble *classes* contenant des couples de la forme (*classe d'équivalence, position dans Q_T*) guide la construction (ligne 3). Ainsi, initialement, *classes* contient le couple (C_λ, ϵ) et, par conséquent, l'arbre quotient est un nœud à la position ϵ avec l'étiquette λ . Ensuite, pour chaque couple (CL, u) extrait de *classes*, la fonction **QDom** calcule l'ensemble de ses couples successeurs et les ajoute à l'ensemble *classes* (ligne 7). La nouvelle

Fonction QDom(C_i, u)

```

1  $S \leftarrow \emptyset$ ;
2  $p \leftarrow 0$ ;
3 pour chaque  $C_j \in \text{Succ}(C_i)$  faire
4    $S \leftarrow S \cup \{(C_j, u.p)\}$ ;
5    $p \leftarrow p + 1$ ;
6 retourner  $S$ 

```

position u est ajoutée à Q_D , le domaine de Q_T . Dans les lignes 9 à 15 la fonction vérifie si la classe d'équivalence CL contient plusieurs positions ayant le même père. C'est le cas, dans l'exemple 4.5.3, de $C_X = \{0, 1\}$ qui contient deux positions dont le père est à la position ϵ . Dans ce cas, l'étiquette associée à la position u sera $l(u)$ décorée du symbole $+$ (ici, X^+).

Comme on cherche à construire une CFG condensée, si dans un arbre de dérivation, des nœuds frères sont étiquetés par le même non-terminal, cela doit être indiqué dans l'arbre quotient de manière condensée, à l'aide du symbole $+$. Pour cela, il est nécessaire de vérifier si la classe d'équivalence d'un non-terminal N possède plusieurs positions ayant le même parent. Cette démarche implique également une *fusion* des éventuelles dérivations distinctes de N . Comme le non-terminal N peut être associé à des règles de production différentes, en fusionnant ces dérivations, on accepte des arbres de dérivation incomplets. L'exemple suivant illustre la construction de l'arbre quotient.

Fonction Quotient($T = (D, l)$)

```

1  $Q_D \leftarrow \{\epsilon\}$  ; // Initialisation du domaine le l'arbre quotient avec la position de la racine
2  $Q_l \leftarrow \langle \epsilon \mapsto \lambda \rangle$  ; // L'étiquette  $\lambda$  associée à la racine
3  $classes \leftarrow \{(\{\epsilon\}, \epsilon)\}$  ; // ensemble contenant les couples classe-position qu'il reste à traiter
4 tant que  $classes \neq \emptyset$  faire
5   Prendre un couple  $(CL, u)$  de l'ensemble  $classes$ ;
   // L'ensemble est mis à jour :  $classes \leftarrow classes \setminus \{(CL, u)\}$ 
6    $successeurs \leftarrow \text{QDom}(C, u)$  ; // ensemble des couples classe-position qui succèdent à  $CL$ 
7    $classes \leftarrow classes \cup successeurs$  ;
8    $Q_D = Q_D \cup \{u\}$ ;
9   si  $u \neq \epsilon$  alors
10    Soit  $u' \prec u$  ; //  $u'$  est la position du père de  $u$ 
11     $FILS \leftarrow \{v \mid \forall v \in CL, u' \prec v\}$  ; // Ensemble des positions dans  $CL$  ayant  $u'$  comme père
12    si  $|FILS| > 1$  alors
13       $Q_l(u) = l(u) + '+'$ ;
14    sinon
15       $Q_l(u) = l(u)$ ;
16 retourner  $Q_T = (Q_D, Q_l)$ 

```

Exemple 4.5.4. À partir de la hiérarchie obtenue dans l'exemple 4.5.3, on construit le domaine de l'arbre en associant, à chaque successeur, une position relative à son parent. La fonction QDom associe à chaque classe qui succèdent CL une position qui représente un enfant de u et permet de calculer le domaine de l'arbre quotient.

Pour calculer les positions à l'aide de la fonction QDom, on commence à la position ϵ et on obtient $QDom(C_\lambda, \epsilon) = \{(C_X, 0), (C_Y, 1)\}$. Après des applications successives de la fonction, le résultat est le

suivant :

$$\begin{array}{ll}
 \text{QDom}(C_X, 0) = \{(C_a, 00), (C_b, 01), (C_c, 02)\} & \text{QDom}(C_Y, 1) = \{(C_a, 10)\} \\
 \text{QDom}(C_a, 00) = \emptyset & \text{QDom}(C_a, 10) = \emptyset \\
 \text{QDom}(C_b, 01) = \emptyset & \text{QDom}(C_c, 02) = \emptyset
 \end{array}$$

En construisant le domaine Q_D , il est facile de se rendre compte que le nœud correspondant à la classe C_a sera dupliqué dans l'arbre, car il est lié à deux positions : 00 et 10. Avec le domaine construit par $QDom$ on peut construire l'arbre donné en figure 4.23b. Dans l'arbre de la figure 4.23a le non-terminal X apparaît dans l'arbre T dans deux positions distinctes et indique des dérivations différentes. Le nœud correspondant à la classe d'équivalence C_X dans Q_T se voit donc attribué le symbole '+' signifiant que le symbole X peut être répété plusieurs fois comme enfant de λ (voir la définition 4.3.1 des CFG condensée). Cette information est déduite de l'ensemble des positions de $C_X = \{0, 1\}$ (ligne 9 de l'algorithme Quotient) où deux positions partagent le même préfixe (ici, ϵ).

En particulier, dans la fonction Quotient, après un premier tour de la boucle **tant que** à la ligne 4, nous avons $classes = \{(C_X, 0), (C_Y, 1)\}$ et nous considérons le couple $(C_X, 0)$. Dans ce cadre, nous aurons $successeurs = \{(C_a, 00), (C_b, 01), (C_c, 02)\}$ (ligne 6). À la ligne 9, comme $u = 0$, la condition est vérifiée et u' est ϵ . Nous avons donc $FILS = \{0, 1\}$, car les deux positions, 0 et 1, dans C_X ont le même père. Q_l est donc $\langle 0 \mapsto X^+ \rangle$ car $l(0) = X$ dans T et $|FILS| > 1$.

À partir de l'arbre quotient Q_T , il est possible de reconstruire la CFG condensée G_T qui, selon la définition 4.3.1, peut contenir des non-terminaux annotés par le symbole '+', où T représente une dérivation incomplète de G_T (c'est-à-dire que certains symboles non-terminaux ne sont pas présents). Cette incomplétude met en évidence un ensemble de données manquantes. En effet, si une règle de production de G_T est $X \rightarrow a b c$, l'arbre T de la figure 4.23a sera accepté comme une dérivation valable seulement si nous considérons que c ou a peuvent être des valeurs manquantes. Il est important de rappeler que les arbres enrichis proviennent de l'analyse syntaxique des textes en langue naturelle. Le manque d'informations peut être dû à une omission ou une erreur lors de l'analyse et peut être représenté par des valeurs nulles. La définition suivante établie de manière formelle la transformation d'un arbre quotient en grammaire.

Remarque. Nous avons examiné dans la première partie de cette thèse comment nous pouvons gérer de telles valeurs au sein d'une base de données. Nous reviendrons en détails sur ce point dans la conclusion.

Définition 4.5.5 (Grammaire quotient). Étant donné un arbre T et son quotient $Q_T = (Q_D, Q_l)$, la CFG condensée $G_T = (N, T, P, \lambda)$ qui reconnaît T est obtenue via Q_T où :

- λ est le symbole de départ ;
- l'ensemble de non-terminaux N , éventuellement décorés par '+', est l'ensemble d'étiquettes $Q_l(u)$ pour toute position $u \in Q_D$ qui n'est pas une feuille ;
- l'ensemble de terminaux T est l'ensemble d'étiquettes $Q_l(u)$ pour toute position $u \in Q_D$ qui est une feuille ;
- l'ensemble P de règles de production contient, pour toute position $u \in Q_D$ qui n'est pas une feuille, les règles de la forme $Q_l(u) \rightarrow Q_l(u.0), \dots, Q_l(u.i)$.

Exemple 4.5.5. Étant donné l'arbre Q_T de l'exemple 4.5.4 donné en figure 4.23b, on obtient la grammaire G_T suivante :

$$\lambda \rightarrow X^+ Y \qquad X \rightarrow a b c \qquad Y \rightarrow a$$

4.5.3 Construction des classes d'équivalence

Rappelons maintenant le concept de sous-arbre d'entité donné par la définition 4.5.2 (page 60). Les sous-arbres d'entité représentent l'information et sont donc la base de comparaison des sous-arbres. La représentation textuelle d'un objet du monde réel peut avoir plusieurs formes de surface matérialisées par différents sous-arbres d'entité. De plus, la langue naturelle contient souvent de l'information implicite ou manquante. Ainsi, deux sous-arbres peuvent contenir un ensemble différent de sous-arbres d'entité, mais quand même représenter le même objet. Par exemple, un *traitement* ne contient pas toujours une fréquence

ou un dosage (« Le patient prend 500 mg de Paracétamol » ou « Le patient prend du Paracétamol tous les jours »). Les arbres syntaxiques de ces phrases ne seront pas identiques. Néanmoins, les deux représentent un *traitement*. Dans notre approche, nous allons considérer qu'ils sont équivalents.

Pour déterminer l'équivalence entre deux sous-arbres, il n'est pas suffisant de considérer uniquement l'ensemble des étiquettes des entités présentes dans les sous-arbres. La langue naturelle est ambiguë et une même forme de surface (et donc un même arbre d'entité) peut représenter deux objets différents. Il est donc nécessaire de prendre aussi en compte le contexte. Pour ce faire nous utilisons le concept d'équivalence régulière comme introduite par [WR83] pour les graphes. L'idée est que deux sommets d'un graphe sont jugés équivalents si leur voisinage est équivalent. Par exemple, deux personnes peuvent être jugées *équivalentes* (comme catégorie d'objet représenté, par exemple un patient) si elles sont toutes les deux liées à des sommets *équivalents* comme une maladie, un traitement, etc sans pour autant que ces nœuds soient les mêmes.

Afin de définir la relation d'équivalence, nous définissons une mesure de similarité entre deux sous-arbres. On note $\mathbb{T}$ l'ensemble de tous les arbres et $\mathbb{ST}$ l'ensemble de tous les sous-arbres. Une mesure de similarité est une fonction symétrique $f : \mathbb{ST} \times \mathbb{ST} \rightarrow [0, 1]$ où $f(x, x) = 1$ pour tout $x \in \mathbb{ST}$. Plusieurs mesures f de similarité peuvent être utilisés comme Jaccard, Levenshtein, Jaro, ou encore la distance d'édition pour les arbres [ZS89]. La similarité contextuelle d'arbres est notée $sim_f(x, y)$, entre deux sous-arbres enrichis $x = T|_u$ et $y = T'|_v$, est définie comme une moyenne pondérée des similarités donnée récursivement par la fonction f pour chaque sous-arbre parent. Les poids sont inversement proportionnels à la distance entre les sous-arbres parents et les sous-arbres x et y . L'idée sous-jacente est que plus un sous-arbre est distant d'un autre, moins il a de valeur dans son contexte. La mesure $sim_f(x, y)$ est donnée par l'équation 4.23 où P_i^x (respectivement P_i^y) est le i -ème sous-arbre parent de x (respectivement y) et $depth_{min}$ est la profondeur minimum entre les sous-arbres x et y tel que $depth_{min} = \min\{|u|, |v|\}$.

$$sim_f(x, y) = \frac{\sum_{i=0}^{depth_{min}} \frac{1}{i+1} \cdot f(P_i^x, P_i^y)}{\sum_{j=0}^{depth_{min}} \frac{1}{j+1}} \quad (4.23)$$

Axiome 4.5.1. La fonction sim_f est une moyenne pondérée de f . Par conséquent, sim_f est symétrique, bornée par l'intervalle $[0, 1]$ et pour tout $x \in \mathbb{ST}$, $sim_f(x, x) = 1$.

Exemple 4.5.6. Soit l'arbre donné figure 4.24. On commence par définir une fonction de similarité. Dans cet exemple on utilisera simplement l'indice de Jaccard [Jac01] sur les noms d'entité présents dans le sous-arbre. L'indice de Jaccard est défini comme suit :

$$J(X, Y) = \frac{|X \cap Y|}{|X \cup Y|}$$

Il est facile de voir que les deux sous-arbres ER_NODE_1 et ER_NODE_2 sont identiques. On obtient alors $f(ER_NODE_1, ER_NODE_2) = 1$. Cependant, comme leur contexte est différent : l'un est associé à un médicament (Paracétamol) et l'autre à une fréquence (every day) on s'attend à une similarité différente de 1. La fonction sim_f (équation 4.23) permet de calculer cette différence. On commence par calculer la similarité entre les sous-arbres NP_1 et NP_2 et on obtient $f(NP_1, NP_2) = 0.75$. Puis on continue récursivement jusqu'à atteindre la racine. A partir du nœud $CONJ$, les sous-arbres sont identiques avec une similarité de 1. On obtient alors :

$$sim_f(ER_NODE_1, ER_NODE_2) = \frac{\overbrace{1 \times 1}^{ER_NODE} + \overbrace{0.5 \times 0.75}^{NP} + \overbrace{0.33 \times 1}^{CONJ} + \overbrace{0.25 \times 1}^S}{1 + 0.5 + 0.33 + 0.25} = \frac{1.955}{2.08} \simeq 0.94$$

Ce qui permet de montrer que les deux sous-arbres sont proches sans pour autant être équivalents car il ne sont pas mis en relation avec les mêmes objets.

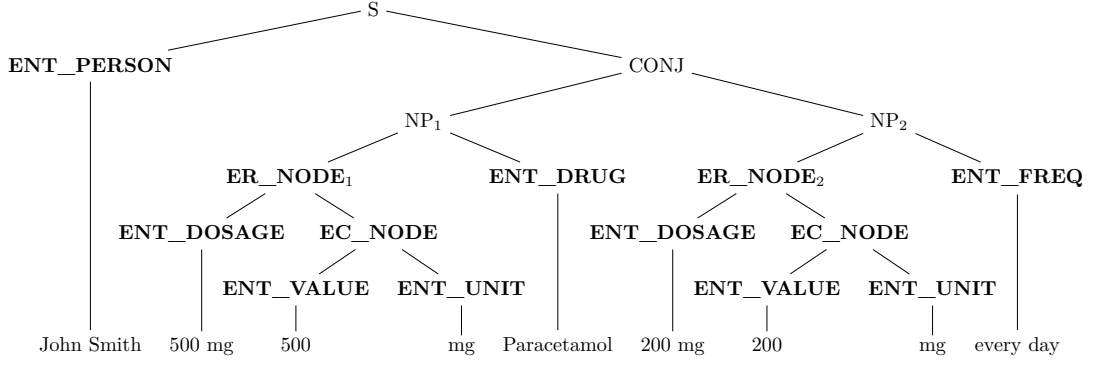


FIGURE 4.24 – Exemple d'arbre enrichi

Avec cette mesure de similarité, il est possible de définir la relation de similarité entre deux sous-arbres enrichis. On dira alors que deux sous-arbres sont similaires si leur similarité est supérieure à un taux donné en paramètre. La décision de ce taux n'est pas arbitraire et il doit être déterminé pour chaque ensemble de données.

Définition 4.5.6 (Similarité de sous-arbres). Soient deux arbres enrichis T_1 et T_2 avec deux sous-arbres respectifs $st_1 = T_1|_u$ et $st_2 = T_2|_v$ et un seuil noté $\tau \in [0, 1]$, on dit que st_1 et st_2 sont τ -similaires, noté $st_1 \sim_\tau st_2$, si et seulement si $\text{sim}_f(st_1, st_2) \geq \tau$.

Proposition 4.5.1. La τ -similarité est une relation de similarité réflexive et symétrique.

Démonstration. Pour tout x , on a $\text{sim}_f(x, x) = 1$ et $1 \geq \tau$ pour toute valeur possible de τ , alors, on peut déduire que la relation de τ -similarité est *réflexive*. De plus, si on a $x \sim_\tau y$, alors $\text{sim}_f(x, y) \geq \tau$. Comme la fonction sim_f est symétrique ($\text{sim}_f(x, y) = \text{sim}_f(y, x)$) on a donc $\text{sim}_f(y, x) \geq \tau$ et par extension $y \sim_\tau x$. Nous pouvons conclure que la relation de τ -similarité est *symétrique*. $\square$

Définition 4.5.7 (Equivalence de sous-arbres). Soient deux arbres enrichis T_1 et T_2 avec deux sous-arbres respectifs $st_1 = T_1|_u$ et $st_2 = T_2|_v$ et la relation de τ -similarité définie précédemment (définition 4.5.6). On définit la relation de τ -équivalence entre st_1 et st_2 notée $st_1 \equiv_\tau st_2$ tel que st_1 est τ -équivalent à st_2 si st_1 est τ -similaire à st_2 ou si st_1 est τ -équivalent à un sous-arbre st_3 lui-même τ -équivalent à st_2 . La proposition est synthétisée par l'équation 4.24.

$$(\forall x \in \mathbb{ST}, y \in \mathbb{ST}) x \equiv_\tau y \iff x \sim_\tau y \vee (\exists z \in \mathbb{ST}) x \equiv_\tau z \wedge y \equiv_\tau z \quad (4.24)$$

Proposition 4.5.2. La τ -équivalence est une relation d'équivalence, c'est-à-dire réflexive, symétrique et transitive.

Démonstration. Soit st_1 un sous-arbre enrichi. La réflexivité de la τ -équivalence découle directement de la réflexivité de la τ -similarité.

Soient st_1 et st_2 deux sous-arbres enrichis et supposons que $st_1 \equiv_\tau st_2$. On a alors deux cas :

- $st_1 \sim_\tau st_2$: qui implique, par symétrie, $st_2 \sim_\tau st_1$ et donc $st_2 \equiv_\tau st_1$.
- $\exists z(st_1 \equiv_\tau z \wedge st_2 \equiv_\tau z)$: qui peut se réécrire en $\exists z(st_2 \equiv_\tau z \wedge st_1 \equiv_\tau z)$ et qui donne, par définition, $st_2 \equiv_\tau st_1$.

On peut donc en déduire que la τ -équivalence est *symétrique*.

Soient st_1 , st_2 et st_3 trois sous-arbres enrichis et supposons que $st_1 \equiv_\tau st_2$ et $st_2 \equiv_\tau st_3$. Par symétrie, nous avons $st_1 \equiv_\tau st_2$ et $st_3 \equiv_\tau st_2$. En utilisant la définition de la τ -équivalence (équation 4.24), nous pouvons remplacer z par st_2 . On obtient alors, par définition, $st_1 \equiv_\tau st_3$. On peut conclure que $(st_1 \equiv_\tau st_2 \wedge st_2 \equiv_\tau st_3) \implies st_1 \equiv_\tau st_3$. La τ -équivalence est donc *transitive*.

La τ -équivalence respecte les trois propriétés d'une relation d'équivalence : *réflexive*, *symétrique* et *transitive*. $\square$

Définition 4.5.8 (Classe d'équivalences). On note $[x]_\tau$ l'ensemble τ -équivalent de x (partition ou classe d'équivalence) donné par la clôture transitive de $\equiv_\tau$ tel que $y \in [x]_\tau \iff y \equiv_\tau x$.

Soit $T = (D, l)$ un arbre ordonné. $D/\equiv_\tau$ est l'ensemble quotient (ou partitionnement) de D par $\equiv_\tau$ tel que $D/\equiv_\tau = \{[x]_\tau \mid x \in D\}$, c-à-d. que $D/\equiv_\tau$ est l'ensemble de tous les ensembles τ -équivalent de D .

Le partitionnement d'un ensemble, basé sur une distance, peut être calculé à l'aide d'un regroupement hiérarchique en lien simple et la mesure de similarité sim_f . [CM10] montre qu'un regroupement hiérarchique en lien simple correspond à un partitionnement selon une relation d'équivalence. Le regroupement hiérarchique implique la construction d'une hiérarchie des éléments selon une mesure de similarité. Initialement, chaque individu forme une classe distincte. À chaque étape, les classes *proches* sont fusionnées deux à deux en fonction de la mesure de similarité (ou de dissimilarité). Une hiérarchie est qualifiée *en lien simple* lorsque la similarité entre deux classes est évaluée comme étant le maximum de la similarité (ou le minimum de la dissimilarité) entre chaque paire d'éléments de ces classes respectives. Cette hiérarchie est fréquemment représentée graphiquement à l'aide d'un dendrogramme (cf. figure 4.25). Il s'agit d'un arbre binaire décrivant les fusions de classes, où les classes sont disposées sur l'axe des abscisses et l'axe des ordonnées représente la dissimilarité entre les classes (plus la valeur est élevée, plus les classes sont éloignées).

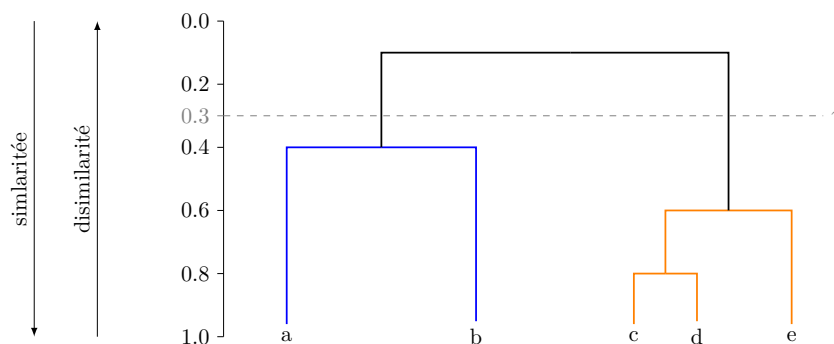


FIGURE 4.25 – Exemple de dendrogramme pour le partitionnement hiérarchique

Une classification par un taux τ consiste à fusionner des classes tant que la similarité demeure supérieure à τ . Visuellement, il est possible de tracer une droite sur le dendrogramme et de conserver toutes les classes situées en dessous de cette droite. Dans la figure 4.25, en fixant $\tau = 0.3$, deux classes émergent, à savoir $\{a, b\}$ (en bleu) et $\{c, d, e\}$ (en orange). Si le seuil est augmenté à $\tau = 0.5$, trois classes distinctes apparaissent, à savoir $\{a\}$, $\{b\}$ et $\{c, d, e\}$. En continuant d'augmenter le seuil, par exemple à $\tau = 0.7$, quatre classes distinctes seraient obtenues, à savoir $\{a\}$, $\{b\}$, $\{c, d\}$ et $\{e\}$.

4.5.4 Réécriture des arbres

La dernière étape du processus de structuration consiste à réécrire les arbres d'instances. L'objectif est de réécrire l'arbre d'instances de façon à ce que l'instance soit reconnue par un schéma valide selon la méta-grammaire $\mathbb{G}$. Pour cela, on cherche à identifier les nœuds correspondants à des *groupes*, des *relations* ou les *collections*. L'ensemble des sous-arbres équivalents peuvent être réécrits pour correspondre à la structure associée à leur catégorie. Cette réécriture est un processus itératif où à chaque modification de l'arbre, on vérifie si ce dernier produit un schéma valide ou non. Comme il n'existe pas de schéma unique, la procédure de réécriture dans cette étape peut varier en fonction des objectifs définis. Une proposition d'objectifs et une implémentation des procédures correspondantes sont données dans la section suivante.

4.6 Proposition d'implémentation

Dans cette section, nous décrivons en détail une implémentation pour effectuer l'évolution de l'instance, illustrée par la séquence des I_i dans la figure 4.12 (page 54). On rappelle que l'évolution de l'instance I_i vers l'instance I_{i+1} suit un processus de transformation. Au cours de cette transformation, les sous-arbres de l'arbre sont regroupés ou transformés sur la base d'une mesure de similarité. Il peut s'agir de réorganiser la structure des arbres afin d'améliorer leur cohérence ou de les aligner plus efficacement sur la représentation souhaitée. Chaque instance I_i engendre une grammaire G_i . À l'étape i , si G_i n'est pas conforme à $\mathbb{G}$, le processus se poursuit en transformant les structures des sous-arbres de I_i donnant lieu à l'instance I_{i+1} qui engendre une nouvelle grammaire G_{i+1} . Le processus se termine lorsque nous trouvons une grammaire qui satisfait $\mathbb{G}$.

Procédure instanceEvolution($T = (D, l)$, epochs, τ , minSupport)

```

1 simp ← TRUE;
2 i ← 0;
3 tant que  $i \leq \text{epochs}$  and  $\text{simp} = \text{TRUE}$  faire
4   i ← i + 1;
5   // Supprimer les nœuds redondants
6   pour tous  $p \in D$  tel que  $l(p) \notin \{\text{'ENT'}, \text{'LEAF'}\}$  faire reduce( $T, p, \emptyset$ ) ;
7   // Identifier les nœuds groupes
8   findGroups( $T, \tau, \text{minSupport}$ );
9   // Ajouter des nœuds pour séparer un sous-groupe fréquent
10  simp ← findSubgroups( $T, \tau, \text{minSupport}$ );
11  // Fusionner des nœuds voisins pour augmenter un groupe
12  si simp = FALSE alors simp ← mergeGroups( $T, \tau, \text{minSupport}$ ) ;
13  // Identifier les collections de groupes
14  si simp = FALSE alors simp ← findCollections( $T, \tau, \text{minSupport}$ ) ;
15  // Identifier les relations
16  si simp = FALSE alors simp ← findRelationship( $T, \tau, \text{minSupport}$ ) ;
17  // Identifier les collections de relations
18  si simp = FALSE alors simp ← findCollections( $T, \tau, \text{minSupport}$ ) ;
19  // Supprimer les nœuds non étiquetés en bas
20  si simp = FALSE alors
21    pour chaque  $u, j \in$  où  $l(u, j) = \text{'ENT'}$  faire
22      simp ← reduce( $T, u, \{\text{'ENT'}, \text{'GROUP'}, \text{'REL'}, \text{'COLL'}\}$ );
23  // Aplatir l'arbre en haut
24  si simp = FALSE alors
25    pour chaque  $p \in D$  tel que  $\text{depth}(p) = 2$  faire
26      simp ← reduce( $T, p, \{\text{'ENT'}, \text{'GROUP'}, \text{'REL'}, \text{'COLL'}\}$ );

```

La procédure principale instanceEvolution se charge d'orchestrer les différentes opérations d'édition et implémente l'évolution de l'instance, indiquée par la boucle de la figure 4.14 (page 58). On suppose qu'on a en entrée un unique arbre en tant que forêt enracinée d'arbre d'instances. La phase de prétraitement qui consiste à enrichir et à simplifier les arbres de syntaxe a déjà été réalisée. On définit quatre paramètres supplémentaires :

- la fonction de similarité f ;
- le seuil τ de similarité ;
- le support minimum qui détermine quand un élément doit être considéré fréquent ;
- l'epoch qui borne le nombre de cycles que l'on autorise.

L'idée générale consiste à unifier les sous-arbres en (i) éliminant les variations de structurelle et (ii) en maximisant la fréquence d'apparition de ces structures. L'objectif est de minimiser le nombre de règles de production de la grammaire qui sera extraite de l'instance. La procédure instanceEvolution permet de converger vers une solution qui est valide par rapport à la méta-grammaire $\mathbb{G}$. La procédure instanceEvolution est indépendante de l'extraction et de la vérification de la grammaire à chaque étape. Cela est rendu possible, car les opérations de transformation s'appliquent exclusivement aux parties invalides de l'arbre. Lorsque plus aucune transformation n'est possible, la grammaire résultante est par conséquent valide. L'utilisation de l'epoch assure la terminaison de la procédure, même si cela implique que certaines parties de l'arbre ne sont pas valides à la fin de l'exécution.

La procédure est constituée d'une boucle générale (ligne 3) qui construit les instances successivement. À chaque itération, l'algorithme commence par appliquer les simplifications décrites dans la section 9

(page 9) en utilisant la règle de réécriture *reduce* (ligne 5). Dans la suite, on identifie les nœuds candidats pour représenter des groupes (opération *findGroups*). Une fois ces premières étapes effectuées, la procédure tente successivement d'effectuer des transformations de l'instance. La variable *simp* permet d'enregistrer si une simplification a été faite. Si une opération modifie l'arbre, les suivantes ne sont pas exécutées. En effet, on recommence le cycle pour recalculer les classes d'équivalences qui ont pu changer. Dans la procédure *instanceEvolution*, les opérations sont listées en suivant un ordre logique, car, il faut d'abord détecter les groupes, les unifier, puis les regrouper en collections ensuite, il est possible d'identifier des relations entre les groupes et finalement des collections de relations. Par exemple, si à l'itération i les opérations *findSubgroups* et *mergeGroups* ne font pas de transformations sur l'arbre I_i , alors, c'est l'opération suivante : *findCollections* qui sera exécuté. Si cette dernière modifie l'instance I_i , on passe à l'instance I_{i+1} et on recommence la boucle. Ainsi, dans l'ordre, on retrouve les opérations suivantes :

1. *findSubgroups* pour construire des sous-groupes fréquents ;
2. *mergeGroups* pour construire des groupes plus grands en incorporant d'autres informations, mais qui maintient sa similarité avec sa classe d'équivalence ainsi qu'une fréquence au-dessus de support minimum ;
3. *findCollections* pour construire des collections de groupes (ces collections peuvent être utilisées lors de l'étape suivante) ;
4. *findRelationship* pour construire des relations entre deux groupes ;
5. *findCollections* pour construire des collections de relations.

Si aucune transformation est faite avant la ligne 12 de la procédure *instanceEvolution*, nous procédons à des opérations qui transforment l'arbre de manière plus radicale en supprimant des nœuds que nous n'avons pas pu structurer via les opérations précédentes. Autrement dit, si aucune modification n'a pu être faite par les cinq opérations citées ci-dessus, on cherche à supprimer des structures trop complexes qui ne sont pas catégorisables. Notamment, les arbres peuvent avoir une grande profondeur, mais nos grammaires autorisent uniquement une profondeur maximale de 5. L'idée est donc de mettre à plat des structures trop profondes en supprimant un à un les niveaux. L'algorithme construit les instances en commençant par le bas de l'arbre et en remontant, on commence alors par détruire l'arbre par le bas en supprimant les niveaux intermédiaires situés au-dessus des entités qui n'ont pas pu être catégorisés (ligne 13). Quand l'ensemble des nœuds présents au-dessus des entités ont été catégorisés, la ligne 16 supprime alors l'ensemble des niveaux supérieurs qui n'ont pas été catégorisée en tant qu'*entité*, *groupe*, *relation* ou *collection*.

Remarque. Dans notre approche, chaque instance est considérée comme une forêt enracinée, composée d'arbres syntaxiques distincts pour chaque phrase. Les opérations, correspondant à une réécriture, sont appliquées individuellement à chaque phrase, bien que le calcul des classes d'équivalence et du support soit réalisé sur l'ensemble de la forêt. Pour l'opération à la ligne 16, seul le niveau de profondeur 2 est supprimé, conservant ainsi la séparation entre les phrases (représentées par la profondeur 1). Au-delà, seule l'opération *findCollections* peut construire des collections entre plusieurs phrases. Ce traitement indépendant des phrases assure que les regroupements se font au sein d'une même phrase et donc d'une même unité sémantique, évitant ainsi la construction de regroupements inter-phrases non alignés sur la sémantique du texte

Dans la suite, on étudie en détail l'implémentation des opérations de transformations mises en place.

4.6.1 Identification des groupes

Dans cette section on s'intéresse aux diverses opération utilisée pour identifier et réécrire les sous-arbres correspondant à un groupement d'entités. Pour rappel, un groupe est représenté par un sous-arbre contenant uniquement des sous-arbres d'entités de type distinct. Intuitivement, dans le modèle relationnel les groupes correspondent à un tuple où les types d'entités correspondent aux différents attributs de la table. L'identification des groupes est réalisé par trois procédures :

- *findGroups* pour identifier les nœuds candidats dans l'arbre,
- *findSubgroups* pour construire des sous groupes fréquents et maximiser le support et
- *mergeGroups* pour fusionner des groupes et maximiser la taille des groupes.

Trouver des groupes

La procédure `findGroups` est utilisée pour identifier les groupements d'entités qui sont fréquents dans l'arbre. Pour cela, on commence par calculer le partitionnement des sous-arbres (ligne 1) qui se trouvent au-dessus des sous-arbres entités (section 4.5.3). Comme on recherche des éléments fréquents (c'est-à-dire que leur support est supérieur à minSupport), on filtre les partitions obtenues qui ont un support trop faible. L'ensemble equivalent_st contient alors l'ensemble des classes d'équivalence de sous-arbres dont la cardinalité est supérieur à minSupport .

Procédure `findGroups( $T = (D, l), \tau, \text{minSupport}$ )`

```

// Identifier les clusters de sous-arbres équivalents et garder uniquement les fréquents
1  $\text{equivalent\_st} \leftarrow \{ \text{equiv\_pos} \mid \text{equiv\_pos} \in D / \equiv_{\tau}, |\text{equiv\_pos}| \geq \text{minSupport} \}$ ;
2  $k \leftarrow 0$ ;
3 tant que  $\text{equivalent\_st} \neq \emptyset$  faire
4    $ST\_min \leftarrow \text{findSTwithMinLeavesTree}(\text{equivalent\_st})$ ;
5    $\text{equivalent\_st} \leftarrow \text{equivalent\_st} \setminus ST\_min$ ;
6   pour chaque  $st \in ST\_min$  tel que  $\{e \mid e \in st, l(e) \neq \text{'GROUP'}\} = \emptyset$  faire
7      $j \leftarrow 0$ ;
8     Construire  $T_{GROUP} = (D_{GROUP}, t_{GROUP})$  un nouvel arbre avec  $t_{GROUP}(\epsilon) = \text{'GROUP'}_k$ ;
9     pour chaque  $l \in \text{leaves}(st)$  faire
10        $\text{ins\_elem}(T_{GROUP}, l, j)$ ;
11        $j \leftarrow j + 1$ ;
12     Soit  $u$  tel que  $T|_u = st$ ;
13      $\text{ins\_elem}(T, t_{GROUP}, u + 1)$ ;
14      $\text{del\_elem}(T, u)$ ;
15    $k \leftarrow k + 1$ ;

```

La fonction `findSTwithMinLeavesTree` renvoie la classe d'équivalence la plus grande dont la hauteur moyenne de ses sous-arbres est la plus faible. Intuitivement, on veut travailler avec les classes les plus fréquentes (donc les plus grandes) mais en partant du bas de l'arbre et en remontant vers la racine. La boucle **pour chaque** de la ligne 6 à 14, construit, pour chaque sous-arbre st appartenant à ST_min , un nouveau sous-arbre T_{GROUP} dont la racine est labellisée $GROUP_k$ (parfois réduit en G_k dans les figures) où $GROUP$ est un symbole non-terminal et k un attribut identifiant les regroupements équivalents (voir la définition 4.3.3 des grammaires attribuée). Les enfants du sous-arbre T_{GROUP} sont exclusivement les sous-arbres d'entités originalement présents dans st . Néanmoins, il est important de noter que cette construction ne se fait pas à partir des sous-arbres ayant déjà un sous-arbre labellisé $GROUP$. La modification de l'arbre initiale est implémentée dans les lignes 13 (en insérant le nouveau sous-arbre T_{GROUP} comme frère à droite de st) et 14 (en supprimant le sous-arbre st). Le choix de l'insertion avant la suppression est dû au caractère récursif de la suppression (règle `del_elem`, figure 4.11). Cela évite la disparition du nœud parent avant l'insertion, déjà prévue, du nouvel enfant dans le cas où le parent ne contient qu'un seul enfant.

Exemple 4.6.1. Prenons, comme exemple, l'arbre T donné dans la figure 4.26a où chaque E_i représente un sous-arbre d'entités. Nous supposons :

$$D / \equiv_{\tau} = \{\{0, 10, 2100\}, \{1\}, \{2\}, \{11\}, \{110, 211\}, \{21\}, \{210\}, \{21000\}\}$$

Si $\text{minSupport} = 2$ alors $\text{equivalent_st} = \{\{0, 10, 2100\}, \{110, 211\}\}$. La boucle **tant que**, à la ligne 3, commence avec $ST_min = \{110, 211\}$, choisi par la procédure `findSTwithMinLeavesTree`, car le sous-arbre à la position 2100 est le plus profond, ce qui augmente la moyenne de profondeurs de sa classe d'équivalence. La procédure `findGroups` modifie l'étiquette des nœuds 110 et 211 en $GROUP_0$. On obtient alors l'arbre T' donnée figure 4.26b. La seconde classe d'équivalence choisie est $\{0, 10, 2100\}$. Le label $GROUP_1$ est assigné à chaque position de cette classe équivalence (racine d'un sous-arbre). Le nœud à la position 21000 est supprimé conformément à la définition des sous-arbres groupes. On obtient alors l'arbre T'' comme illustré dans la figure 4.26c

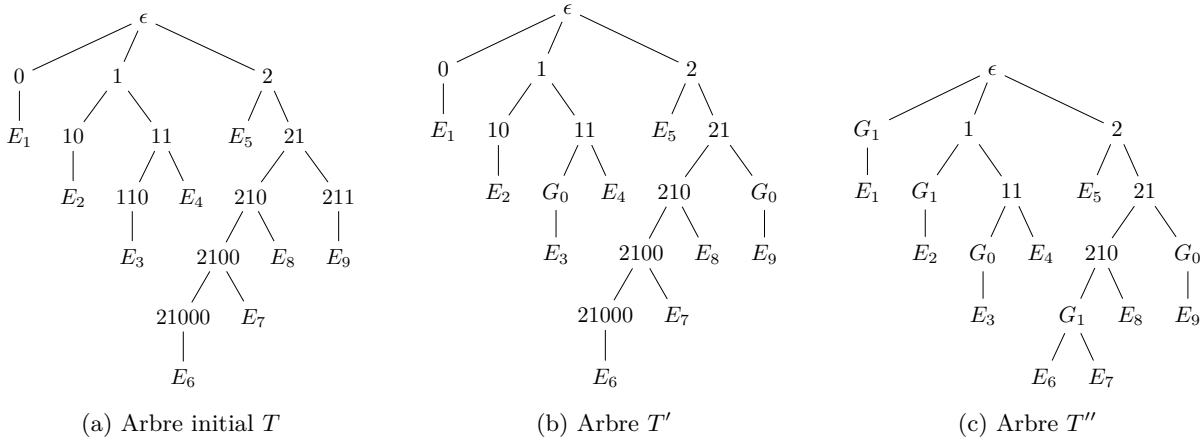


FIGURE 4.26 – Exemple d'application de la procédure findGroups

Séparation en sous-groupes fréquents

La procédure findGroups identifie uniquement des nœuds déjà présents dans l'arbre. Cependant, il est possible que la structure interne ne soit pas suffisante et on peut alors obtenir des nœuds ayant beaucoup d'enfants et pouvant alors inclure un sous-ensemble plus fréquent d'entités. Afin de minimiser le nombre de groupes distinct et de maximiser leurs fréquences, la procédure findSubgroups est chargée d'identifier les sous-ensembles d'entités qui pourraient former un groupe plus fréquent. En d'autres termes, findSubgroups s'assure qu'aucun groupe ne contient un autre groupe plus fréquent. De plus, cette transformation est également appliquée aux sous-arbres dont la racine n'a pas encore d'étiquette correspondant à un non-terminal de la grammaire. Si, pour un sous arbre st , il est possible de construire un sous-arbre st_i plus fréquent à partir d'un sous-ensemble des arbres d'entités qui sont des descendants de st , la transformation suivante est opérée : le sous-arbre st est remplacé par un nouveau sous-arbre non étiqueté dont les enfants comprennent le nouveau sous-arbre st_i ainsi que les sous-arbres présents dans st mais absents dans le nouveau st_i .

Pour faire cela, la procédure findSubgroups récupère l'ensemble S_{ENT} des sous-arbres entité de chaque sous-arbre st à tester (ligne 3). On construit ensuite tous les sous-ensembles possibles d'entités comportant au moins deux éléments (Le « power set » $\mathcal{P}$). Tous les ensembles de taille égale seront traités simultanément dans la boucle **pour chaque** qui commence à la ligne 8, en débutant par l'évaluation des ensembles les plus grands (ligne 6). **combinaison** construit, pour l'ensemble st , l'ensemble des parties de taille k de st . À chaque itération, nous créons une copie de l'arbre T pour former un nouvel arbre T_j . Dans T_j , nous modifions st en retirant l'étiquette de sa racine (ligne 10), puis en lui ajoutant un nouvel enfant étiqueté *GROUP*. Ce nœud *GROUP* devient le parent des sous-arbres entité présents dans **combinaison**. Parallèlement, tous les éléments de **combinaison** sont retirés des enfants de st (boucle à la ligne 11). Ensuite, nous calculons la classe d'équivalence du nouveau sous-arbre groupe construit dans T_j (ligne 15). L'exemple ci-dessous illustre la situation.

Exemple 4.6.2. Considérons l'arbre T représenté dans la figure 4.27, où la racine est étiquetée U . Soit st le sous-arbre de T avec une racine étiquetée *GROUP*, dans lequel nous cherchons à déterminer si un sous-groupe pourrait être plus fréquent que le groupe initial. Ici, le nombre de sous-arbres entité dans st est de 4. Supposons que nous soyons à l'étape où $k = 3$. L'ensemble **combinaison** est alors représenté par $\{E_1, E_2, E_3\}, \{E_1, E_2, E_4\}, \{E_1, E_3, E_4\}, \{E_2, E_3, E_4\}$. Nous allons donc construire les quatre arbres illustrés dans la figure 4.27b.

Il est important de souligner que chaque T_j correspond à une analyse réalisée sur un sous-ensemble de **combinaison**. Une fois que tous les sous-ensembles de taille équivalente ont été examinés, nous identifions parmi tous les arbres T_j celui que nous désignons comme $T_{j_{max}}$, qui présente la fréquence maximale pour le sous-arbre dont la racine est étiquetée *GROUP*. Si cette fréquence est supérieure à $minSupport$ et à la fréquence de st dans T , alors T est remplacé par $T_{j_{max}}$. Une fois le sous-arbre fréquent créé, les entités restantes sont remise en jeu. S'il n'est pas possible de construire un ensemble de k entités fréquent alors on recommence avec $k - 1$.

Procédure findSubgroups($T = (D, l), \tau, \text{minSupport}$)

```

1 groupFound  $\leftarrow$  FALSE;
2 pour tous sous-arbre  $st$  tel que  $st = T|_u$  et  $l(u) \notin \{'ENT', 'REL', 'COLL'\}$  faire
3    $S_{ENT} \leftarrow \{T|_{u.i} \mid \forall i \, l(u.i) = 'ENT'\}$ ; // Ensemble des arbres d'entités enfants de  $st$ 
4    $group\_support \leftarrow |[st]_\tau|$ ;
5    $k \leftarrow |S_{ENT}|$ ;
6   tant que  $k > 1$  faire
7      $j \leftarrow 0$ ;
8     // Pour chaque sous-ensemble de taille  $k$  dans  $\mathcal{P}(S_{ENT})$ 
9     pour chaque combinaison  $combi_j \in \text{combination}(S_{ENT}, k)$  faire
10       $T_j \leftarrow \text{copy}(T)$ ; //  $T_j$  est l'arbre où l'on remplace l'ensemble  $combi_j$  par un groupe
11       $l_{T_j}(u) = ""$ ;
12       $\text{ins\_elem}(T_j, GROUP, u.0)$ ;
13      pour tous  $st_E \in combi_j$  faire
14         $\text{del\_elem}(T_j, u.i)$  tel que  $T_j|_{u.i} = st_E$ ;
15         $\text{ins\_elem}(T_j, st_E, u.0.0)$ ;
16       $support_j \leftarrow |[T_j]_{u.0}|$ ; //  $[T_j]_{u.0}$  est l'ensemble équivalent qui contient le sous-arbre  $T_j|_{u.0}$ 
17       $j \leftarrow j + 1$ ;
18      // Choisir le groupe qui a le plus grand support
19       $j_{max}$  est un indice tel que  $support_{j_{max}} = \max(support_j)$ ;
20      si  $support_{j_{max}} \geq \text{min\_support}$  et  $support_{j_{max}} > group\_support$  alors
21        // On continue de combiner les arbres d'entités restants dans  $st$ 
22        groupFound  $\leftarrow$  TRUE;
23         $S_{ENT} \leftarrow S_{ENT} \setminus combi_{j_{max}}$ ;
24         $k \leftarrow \min(|S_{ENT}|, k)$ ; // La taille de la combinaison suivante  $k$  est réduite lorsque  $|S_{ENT}| < k$ 
25         $T \leftarrow T_{j_{max}}$ ; // L'arbre modifié est sauvegardé
26      sinon
27         $k \leftarrow k - 1$ ;
28 retourner groupFound ;

```

Exemple 4.6.3. Poursuivant l'exemple 4.6.2, imaginons que T soit remplacé par T_2 , indiquant ainsi que c'est l'ensemble $combi_2 = \{E_1, E_2, E_4\}$ que nous considérons comme un nouveau groupe. Ce remplacement signifie que T_2 contient le sous-arbre $GROUP$ (correspondant à $combi_2$) le plus fréquent, en comparaison à ceux de T_0 , T_1 , et T_3 , et que la fréquence de ce sous-arbre est supérieure à celle de st dans T .

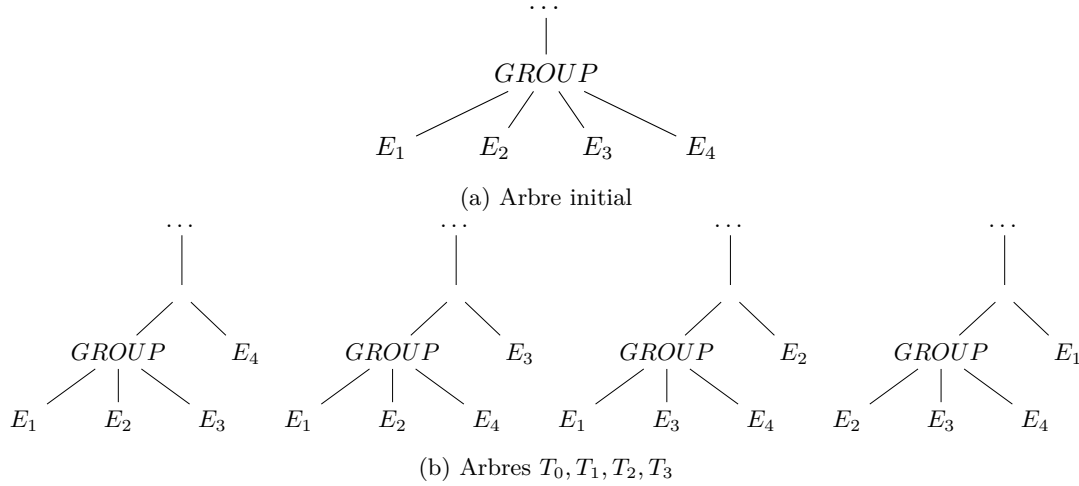


FIGURE 4.27 – Exemple d'application de la fonction findSubgroups

Fusionner en groupe fréquent

L'objectif de la fonction mergeGroups est d'augmenter la taille des groupes par la fusion des groupes frères ou par l'ajout, dans un groupe, des sous-arbres entité qui sont frères de ce groupe. Tout comme pour la fonction précédente, l'objectif est d'effectuer des transformations uniquement si nous obtenons des sous-arbres plus fréquents que dans la version originale. Cette fusion sera tentée uniquement pour des groupes qui ne sont pas déjà dans une relation ou une collection (ligne 2).

Pour chaque sous-arbre st à tester, S_{GROUP} (ligne 3) représentera l'ensemble des enfants de st étiquetés $GROUP$, tandis que S_{ENT} (ligne 4) sera l'ensemble des enfants de st étiquetés ENT . Nous explorons toutes les combinaisons possibles entre les éléments de ces deux ensembles, en commençant par les plus grandes. À la ligne 5, $|S_{GROUP}/\equiv_\tau|$ représente le nombre d'éléments dans S_{GROUP} modulo ceux qui sont équivalents. En d'autres termes, nous n'essayons pas de fusionner des groupes qui appartiennent à la même classe d'équivalence. À la ligne 8, $combinaison$ est l'ensemble des sous-ensembles de $ST_{ENT} \cup S_{GROUP}$ de taille k . À la ligne 10, un nouvel ensemble ST_{ENT} est construit. Il contient l'ensemble des sous-arbres entités présents dans $combij$ (le sous-ensemble de $combinaison$ sur lequel on travaille). Il est important de remarquer que ST_{ENT} n'est pas construit si plusieurs sous-arbres entité identiques existent dans $combij$. On construit un nouvel arbre T_j de la manière suivante :

1. T_j est une copie de T ;
2. Dans T_j , on remplace les sous-arbres qui apparaissent dans $combij$ par un seul nouveau sous-arbre dont la structure est la suivante :
 - (a) La racine est étiquetée $GROUP$;
 - (b) Les enfants sont tous les sous-arbres entités présents dans ST_{ENT} .

Comme dans la procédure précédente, chaque T_j correspond à une analyse réalisée sur un sous-ensemble de $combinaison$. Une fois que tous les sous-ensembles de taille identique ont été examinés, nous identifions parmi tous les arbres T_j celui que nous désignons comme $T_{j_{max}}$, qui présente la fréquence maximale pour le sous-arbre dont la racine est étiquetée $GROUP$. Si cette fréquence est supérieure à $minSupport$ et à la fréquence la plus élevée parmi celles des sous-arbres étiquetés $GROUP$ de $combij$, alors T est remplacé par $T_{j_{max}}$.

Exemple 4.6.4. Soit un arbre T contenant le sous-arbre st qui est représenté dans la figure 4.28a. La racine du sous-arbre st est à la position u et son étiquette ne correspond pas encore à un symbole de la grammaire. Nous allons essayer de fusionner des enfants de st . Nous avons $S_{GROUP} = \{u0, u1, u2\}$, $S_{ENT} = \{u3, u4\}$ et $k = 4$. Remarquer que $u0$ et $u2$ (les deux sous-arbres labellisés $GROUP_1$) sont dans la même classe d'équivalence. Pour $k = 3$ nous avons :

$$combination(S_{GROUP} \cup S_{ENT}, 3) = \{\{u0, u1, u3\}, \{u0, u1, u4\}, \{u0, u3, u4\}, \{u1, u2, u3\}, \\ \{u1, u2, u4\}, \{u1, u3, u4\}, \{u2, u3, u4\}\}$$

Procédure mergeGroups($T = (D, l), \tau, \text{minSupport}$)

```

1 groupFound  $\leftarrow$  FALSE;
2 pour tous sous-arbre st tel que  $st = T|_u$  and  $l(u) \notin \{\text{'ENT'}, \text{'GROUP'}, \text{'REL'}, \text{'COLL'}\}$  faire
3    $S_{GROUP} = \{T|_{u.i} \mid \forall i \, l(u.i) = \text{'GROUP'}\}$ ; // Ensemble des arbres groupe enfants de st
4    $S_{ENT} = \{T|_{u.i} \mid \forall i \, l(u.i) = \text{'ENT'}\}$ ; // Ensemble des arbres d'entités enfants de st
5    $k \leftarrow |S_{GROUP}| \equiv_{\tau} |S_{ENT}|$ ;
6   tant que  $k > 1$  faire
7      $j \leftarrow 0$ ;
8     // Pour chaque sous-ensemble de taille k dans  $\mathcal{P}(S_{GROUP} \cup S_{ENT})$ 
9     pour tous combinaison  $combi_j \in \text{combination}(S_{GROUP} \cup S_{ENT}, k)$  faire
10       $ST = \{x \mid \forall x \, l(x) = \text{'GROUP'}\}$ ;
11       $ST_{ENT} = \{T|_{u.i} \mid \forall x \, l(x) = \text{'ENT'}\} \cup \{x \mid \forall g \in ST, \forall x \in g\}$ ;
12      si  $ST_{ENT}$  contient plusieurs fois la même entité alors SKIP ;
13       $subgroup\_support_j \leftarrow \max(|g| \mid \forall g \in ST)$ ;
14       $T_j \leftarrow \text{copy}(T)$ ; //  $T_j$  est l'arbre où l'on remplace l'ensemble  $combi_j$  par un groupe
15       $\text{ins\_elem}(T_j, GROUP, u.0)$ ;
16      pour tous  $st_E \in ST_{ENT}$  faire
17         $\text{del\_elem}(T_j, u.i)$  tel que  $T_j|_{u.i} = st_E$ ;
18         $\text{ins\_elem}(T_j, st_E, u.0.0)$ ;
19         $support_j \leftarrow |[T_j|_{u.0}]_{\tau}|$ ; //  $[T_j|_{u.0}]$  est l'ensemble équivalent qui contient le sous-arbre  $T_j|_{u.0}$ 
20         $j \leftarrow j + 1$ ;
21      // Choisir le groupe qui a le plus grand support
22       $j_{max}$  est un indice tel que  $support_{j_{max}} = \max\{support_j \mid support_j \geq subgroup\_support_j\}$ ;
23      si  $support_{j_{max}} \geq \text{min\_support}$  alors
24        // On continue de combiner les arbres d'entités restants dans st
25        groupFound  $\leftarrow$  TRUE;
26         $S_{ENT} \leftarrow S_{ENT} \setminus combi_{j_{max}}$ ;
27        // La taille de la combinaison suivante k est réduite lorsque  $|S_{ENT}| < k$ 
28         $k \leftarrow \min(|S_{ENT}|, k)$ ;
29         $T \leftarrow T_{j_{max}}$ ; // L'arbre modifié est sauvegardé
30      sinon
31         $k \leftarrow k - 1$ ;
32 retourner groupFound ;

```

Toutes ses combinaisons sont testées en créant un nouvel arbre contenant le nouveau groupe et en calculant le support de ce dernier. La figure 4.28b montre le sous-arbre st_0 créé pour la combinaison $\{u0, u1, u3\}$ avec $ST_{ENT} = \{E_1, E_2, E_3, E_4, E_5, E_9\}$ et la figure 4.28c celui pour la combinaison $\{u2, u3, u4\}$ avec $ST_{ENT} = \{E_6, E_7, E_8, E_9, E_{10}\}$. L'arbre T_0 correspond à l'arbre T où le sous-arbre st a été remplacé par st_0 , idem pour T_1 . Si l'on suppose que l'on accepte la transformation proposée par l'arbre T_0 , nous allons changer T en T_0 et continuer la procédure mergeGroups en tentant de fusionner $u1$ et $u2$ dans ce nouvel arbre.

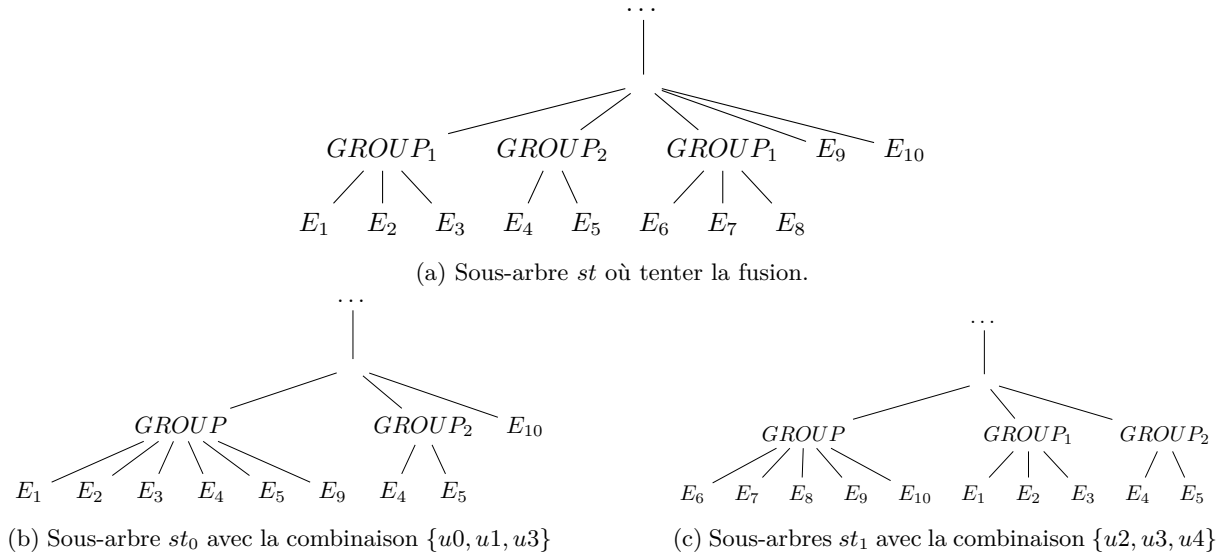


FIGURE 4.28 – Exemple d'application de la procédure mergeGroups

4.6.2 Identification des relations

La fonction findRelationship a pour objectif de créer des relations sur des sous-arbres non encore labellisés. Il existe deux manières de procéder à cette création :

1. Le sous-arbre possède uniquement deux enfants labellisés *GROUP*. Dans ce cas, la racine du sous-arbre sera labellisée *REL*.
2. Le sous-arbre possède un enfant labellisé *GROUP* et un autre labellisé *COLL*. Dans ce cas, le sous-arbre sera complètement restructuré. Ses enfants auront pour racine un nœud étiqueté *REL* et deux enfants : celui ayant pour racine le nœud *GROUP* et un autre qui était initialement l'enfant du nœud labellisé *COLL*. En effet, pour chaque sous-arbre $coll_i$, enfant du nœud étiqueté *COLL*, nous créons un sous-arbre labellisé *REL* ayant $coll_i$ comme l'un de ses enfants. Cette transformation peut être visualisée comme une opération de distributivité.

Cette fonction est illustrée par la figure 4.29.

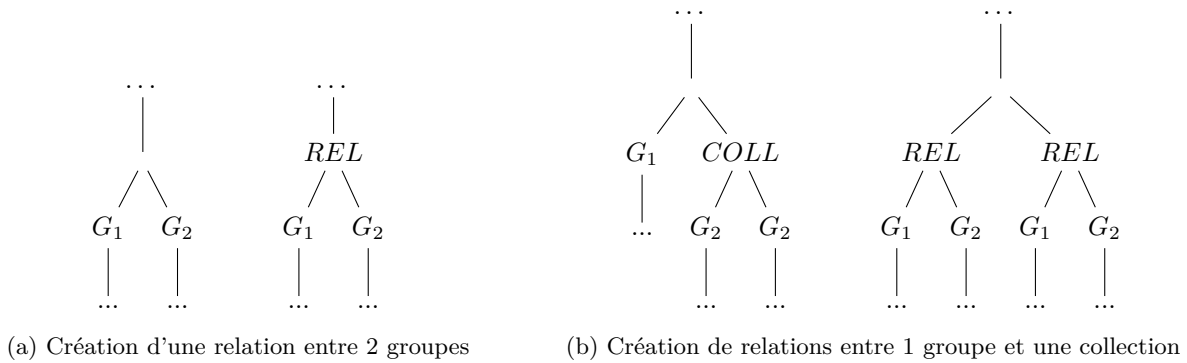


FIGURE 4.29 – Exemple de transformation en insérant des relations (procédure findRelationship)

Procédure findRelationship($T = (D, l), \tau, \text{minSupport}$)

```

1 relationshipFound  $\leftarrow$  FALSE;
2 pour chaque sous-arbre  $st$  tel que  $st = T|_u, |st| = 2$ , et
    $l(u) \notin \{ENT, GROUP, REL, COLL\}$  faire
3   si  $l(u.0) = GROUP$  et  $l(u.1) = GROUP$  alors
4     // tous les enfants de  $st$  sont des nœuds groupe, il est donc marqué comme relation
5      $l(u) \leftarrow REL$ ;
6     relationshipFound  $\leftarrow$  TRUE;
7   sinon si ( $l(u.0) = COLL$  et  $l(u.1) = GROUP$ ) ou ( $l(u.0) = GROUP$  et
8      $l(u.1) = COLL$ ) alors
9      $group \leftarrow u.l$  tel que  $l(u.l) = GROUP$ ;
10     $coll \leftarrow u.m$  tel que  $l(u.m) = COLL$ ;
11     $pos \leftarrow 2$ ;
12    pour tous  $i \in [0, \dots, |T|_{coll} - 1]$  faire
13      // Ajout d'une relation pour chaque paire  $\langle group, T|_{coll.i} \rangle$ 
14      ins_elem( $T, REL, u.pos$ );
15      ins_elem( $T, T|_{group}, u.pos.0$ );
16      ins_elem( $T, T|_{coll.i}, u.pos.1$ );
17      del_elem( $T, coll.i$ );
18    del_elem( $T, group$ );
19    relationshipFound  $\leftarrow$  TRUE;
20 retourner relationshipFound ;

```

4.6.3 Identification des collections

Le rôle de la fonction findCollections est de regrouper dans une collection des groupes (ou des relations) lorsqu'ils sont enfants d'un même nœud et font partie d'une même classe d'équivalence. Les explications qui vont suivre sont données pour créer des collections de groupes, mais elles sont les mêmes pour créer des collections de relations. Pour chaque sous-arbre st non encore labellisé (par exemple celui de la figure 4.30a), la fonction va se réaliser à travers trois boucles :

- La première boucle (ligne 4) va créer un sous-arbre labellisé *COLL* contenant comme enfant tous les sous-arbres de st labellisés *GROUP* contenus dans une même classe d'équivalence lorsqu'il y en a au moins deux (figure 4.30b).
- La seconde boucle (ligne 12) va regrouper dans une même collection des sous-arbres collection contenant des enfants issues de la même classe d'équivalence (figure 4.30c).
- La troisième boucle (ligne 18) va ajouter à un sous-arbre collection les groupes enfants de st qui sont équivalents aux enfants de la collection (figure 4.30d).

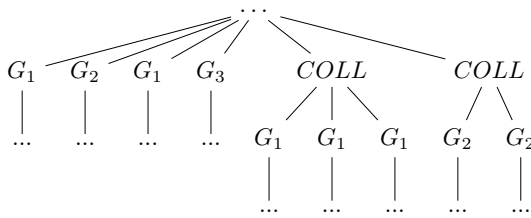
Exemple 4.6.5. La figure 4.30a montre un sous-arbre dans lequel la fonction findCollections va faire des transformations. Dans cet exemple, les indices correspondent aux différentes classes d'équivalence. Lors de la première boucle de la fonction il y a la création d'un nouveau sous-arbre collection contenant les deux groupes qui sont équivalents figure 4.30b. La figure 4.30c montre la transformation du sous-arbre après la seconde boucle, les deux collections contenant des groupes équivalents sont regroupés en un sous-arbre. La figure 4.30d montre que le groupe $GROUP_2$ équivalent aux groupes d'une des collections va être ajouté à celle-ci.

Procédure findCollections($T = (D, l), \tau, \text{minSupport}$)

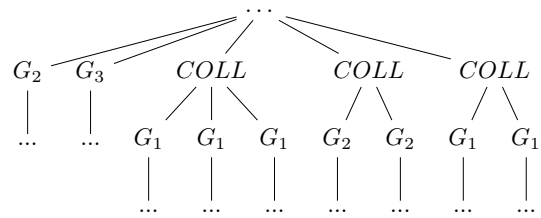
```

1 collectionFound  $\leftarrow$  FALSE;
2 pour chaque sous-arbre st tel que  $\text{st} = T|_u$  et  $l(u) \notin \{\text{'ENT'}, \text{'GROUP'}, \text{'REL'}, \text{'COLL'}\}$  faire
3   pos  $\leftarrow |st|$ ;
4   // Construction d'une collection de groupes ou de relations
5   pour chaque sous-ensemble subSet tel que  $\exists x \forall u.i \in \text{subSet} \text{ où } t|_{u.i} \in [x]_\tau$ ,
6      $l(u.i) \in \{\text{'GROUP'}, \text{'REL'}\}$  et  $|\text{subSet}| \geq 2$  faire
7       collectionFound  $\leftarrow$  TRUE;
8       ins_elem( $T, \text{'COLL'}, u.\text{pos}$ );
9       pour tous  $u.i \in \text{subSet}$  faire
10         ins_elem( $T, T|_{u.i}, u.\text{pos}.0$ );
11       pour tous  $u.i \in \text{subSet}$  (en allant du plus grand au plus petit) faire
12         del_elem( $T, u.i$ );
13   pos  $\leftarrow |st|$ ;
14   // Fusion de collections
15   pour chaque sous-ensemble subSet tel que  $\exists x \forall u.i \in \text{subSet} \text{ où } T|_{u.i} \in [x]_\tau$ ,
16      $l(u.i) \in \{\text{'COLL'}\}$  et  $|\text{subSet}| \geq 2$  faire
17       collectionFound  $\leftarrow$  TRUE;
18       ins_elem( $T, \text{'COLL'}, u.\text{pos}$ );
19       pour tous  $u.i.j$  tel que  $u.i \in \text{subSet}$  faire
20         ins_elem( $T, T|_{u.i.j}, u.\text{pos}.0$ );
21         del_elem( $T, u.i.j$ );
22   // Ajout d'éléments à une collection
23   pour tous group, coll tel que  $\exists x, i, j \text{ où } l(u.i) = \text{'GROUP'}, l(u.j) = \text{'COLL'}, T|_{u.i} \in [x]_\tau$  et
24      $\forall k T|_{u.j,k} \in [x]_\tau$  faire
25     collectionFound  $\leftarrow$  TRUE;
26     group_pos =  $|T|_{u.j}|$ ;
27     ins_elem( $T, T|_{u.i}, u.j.\text{group\_pos}$ );
28     del_elem( $T, u.i$ );
29 retourner collectionFound ;

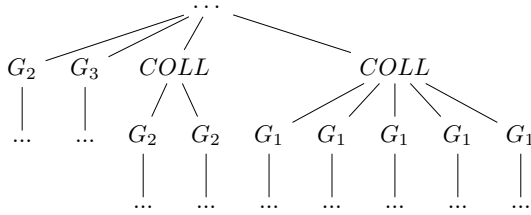
```



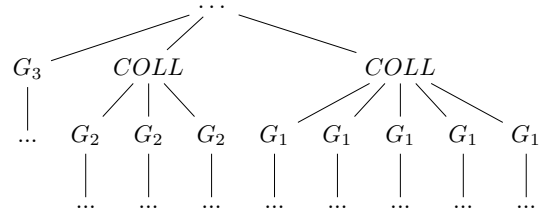
(a) Sous-arbre par la création de collection



(b) Sous-arbre après la première boucle



(c) Sous-arbre après la seconde boucle



(d) Sous-arbre à la fin de la procédure

FIGURE 4.30 – Exemple de transformation en insérant des collections (procédure findCollections)

4.7 Évaluation préliminaire

Dans cette section, nous entreprenons une évaluation critique, préliminaire, de notre stratégie. Il est important de souligner qu'il n'existe pas de modèle universel pour évaluer une stratégie et que, souvent, les approches décrites dans la littérature ne sont pas directement comparables. Notre évaluation critique est donc fondée sur l'observation du comportement de notre prototype par rapport aux objectifs que nous voulons atteindre. Dans la suite, la section 4.7.1 présente une étude du comportement de la procédure `instanceEvolution` et la section 4.7.2 analyse la grammaire obtenue après application de la procédure sur notre jeu de données utilisé comme preuve de concept. Cette analyse nous permettra d'identifier des points forts et des points faibles de notre approche.

L'évaluation de notre approche est faite sur un cas d'usage que nous utilisons comme preuve de concept. On utilise le corpus CAS [GCD18], un corpus textuel composé de cas cliniques réels et fictifs. Un cas clinique est un récit concis détaillant l'historique médical, les symptômes, les diagnostics et les traitements d'un patient depuis son admission à l'hôpital ou sa consultation médicale. Ce corpus est annoté manuellement par différents annotateurs et se compose de 100 textes, 8098 entités nommées réparties en 10 catégories. Les entités peuvent être imbriquées les unes dans les autres. La figure 4.31 montre un extrait du corpus avec les annotations représentées par des boîtes. Dans les textes on retrouve la description d'examen avec le résultat obtenu, la détection de symptômes, les traitements et prescriptions réalisés. L'analyse repose sur les paramètres d'expérience suivants :

- on exécute l'algorithme sur un sous-ensemble du corpus composé de 150 phrases et 457 entités ;
- on utilise l'indice de Jaccard comme présenté dans la section 4.5.3 pour notre mesure de similarité ;
- on fixe τ à 0.5 ;
- on autorise 50 itérations de l'algorithme.

L'implémentation a été réalisée en Python et utilise CoreNLP pour l'analyse syntaxique. Le code source du projet nécessaire pour la reproduction des expériences est disponible sur Git [CHH24]. Pour le corpus, [GCD18] donne les modalités d'accès aux données.

Lors de la première visite médicale à 11 semaines de sa deuxième grossesse, la patiente souffrait de nausées et de vomissements depuis une dizaine de jours. Une perfusion intraveineuse continue de solution saline à 0.9 % avec du dextrose à 5 % avait permis de la réhydrater. Elle était repartie avec une ordonnance de doxylamine et de pyridoxine (six comprimés par jour). Une semaine plus tard, la patiente s'est présentée au service d'urgence pour une récurrence des nausées avec une moyenne de quatre épisodes de vomissements par jour. [GCD18, filehtml-24-cas]

FIGURE 4.31 – Extrait d'un document du corpus CAS

Pour analyser notre approche, il est nécessaire de rappeler les idées majeures qui ont guidé notre démarche. Notre structuration des données textuelles consiste à proposer une grammaire qui joue le rôle d'un schéma, indiquant les informations essentielles à stocker. Les non-terminaux de cette grammaire offrent une manière de regrouper ces informations essentielles. Notre objectif est donc d'agréger des informations similaires, sur la base d'une notion de similarité préétablie, afin d'obtenir comme résultat un schéma compréhensible et une instance valide, résumant les informations du texte initial. Pour atteindre cet objectif, notre procédure `instanceEvolution` (page 71) construit itérativement l'instance en agrégeant ou en séparant les branches de l'arbre. Pour ce faire, elle applique successivement un ensemble d'opérations qui sont numérotées comme suit :

- | | |
|---|---|
| 1. <code>findSubgroups</code> | 5. <code>findCollections</code> (relations) |
| 2. <code>mergeGroups</code> | 6. <code>reduce</code> (bottom) |
| 3. <code>findCollections</code> (groupes) | 7. <code>reduce</code> (top) |
| 4. <code>findRelationship</code> | |

Il est important de remarquer que ces opérations sont ordonnées en fonction de leur niveau de généralisation. Au cours des différentes itérations de la procédure `instanceEvolution`, nous alternons entre des étapes nécessitant des modifications pour rendre les détails visibles, imposant ainsi des spécialisations et des étapes où des agrégations sont nécessaires pour proposer des généralisations. Le comportement de la procédure `instanceEvolution` peut donc être caractérisé par des allers-retours entre ces deux actions : réduire le nombre de règles de production pour généraliser, puis augmenter ce nombre (au besoin, en annulant certaines agrégations précédentes) pour mieux spécifier avant de proposer d'autres généralisations.

La procédure `instanceEvolution` continue à l'itération suivante si une modification est appliquée à l'arbre d'instance. Cela implique que pour une étape i , si on construit de nouveaux groupes, la détection de relations et de collections ne pourra se faire qu'à l'étape $i + 1$. Afin de pouvoir étudier l'évolution de l'arbre en comparant une à une les itérations, on ajoute les aspects suivants :

- L'instance initiale I_0 ne contient pas de structures remarquables (*groupes*, *relations* et *collections*). Afin de comparer l'impact des réécritures, l'itération 1 représente l'arbre I_0 sur lequel on applique seulement l'opération `findGroups`.
- À la fin de chaque itération de la procédure, on annote les nœuds candidats pour être des *relations* ou des *collections* qui ne nécessitent pas de réécriture (cela inclut l'itération 1). Par exemple, un nœud ayant exclusivement des nœuds *groupes* équivalents comme enfant sera alors annoté comme une *collection*. C'est-à-dire que l'on annote uniquement des nœuds qui ont déjà une structure valide par rapport à la méta-grammaire, mais qui ne sont pas encore étiquetés. La recherche des nœuds *relations* et *collections* permet de mesurer à chaque itération leur nombre potentiel même si l'opération de réécriture associée ne s'est pas déclenchée. Cet étiquetage est normalement à appliquer uniquement à la fin de l'exécution de la procédure.

Dans ce contexte, nous devons réfléchir au type des résultats que nous cherchons à obtenir. Pour cela, les aspects suivants servent de guide pour analyser le comportement de la procédure `instanceEvolution` et du résultat obtenu :

1. Une grammaire compréhensible doit être concise par rapport à une version initiale qui exprime l'instance originale elle-même. Autrement dit, notre procédure doit réduire le nombre de règles de production signifiant que l'instance a été unifiée. Il est donc important de surveiller l'impact de notre procédure sur le nombre de règles de production.
2. Nos agrégations reposent sur les concepts de *groupes*, *relations* et *collections*, qui constituent les structures génériques et fondamentales de notre schéma. On s'attend alors à ce que le nombre d'instances de ces structures augmente au fur et à mesure des itérations. Il est donc important d'évaluer le nombre d'instances associées à chacune d'entre elles. Cela implique de déterminer combien de fois, par exemple, le nom terminal $GROUP_9$ apparaît dans l'arbre (notre instance). Habituellement, il serait naturel de considérer qu'une structure est justifiée si elle présente un nombre significatif d'instances. Toutefois, le nombre d'instances peut varier considérablement et des structures relativement peu courantes peuvent s'avérer essentielles. On s'intéresse alors à surveiller l'impact de notre procédure sur le nombre moyen d'instances associées à nos structures génériques.
3. Nos structures fondamentales suivent une hiérarchie définie. Par exemple, une *collection* représente une agrégation de groupes, offrant ainsi une généralisation plus poussée. Nous estimons qu'il est donc important d'observer l'impact de notre procédure sur le nombre de structures différentes pour chaque catégorie que nous avons, à savoir les *groupes*, les *relations* et les *collections*.

Dans le reste de cette section, nous présentons les résultats de nos observations sur ces trois aspects.

4.7.1 Comportement de la procédure `instanceEvolution`

La figure 4.32 illustre le comportement de la procédure `instanceEvolution` par rapport au nombre de règles de production. Elle montre le nombre de règles de production de la grammaire G_i ($\bullet\text{---}\bullet$) avec sa droite de tendance ($\text{---}\text{---}$). La courbe $\text{---}\blacktriangle\text{---}$ montre le nombre de nœuds non catégorisés et $\text{---}\text{---}$ est sa droite de tendance. La partie basse du graphique indique l'opération de transformation appliquée pour chaque étape i ($\blacksquare$). On observe que la procédure `instanceEvolution` progresse efficacement vers l'objectif de minimiser la taille de la grammaire tout en structurant l'instance. Nous passons ainsi de 178 règles de production différentes à 68. Au terme des 50 itérations, seulement 34 nœuds n'ont pas pu être catégorisés, par rapport à 155 initialement.

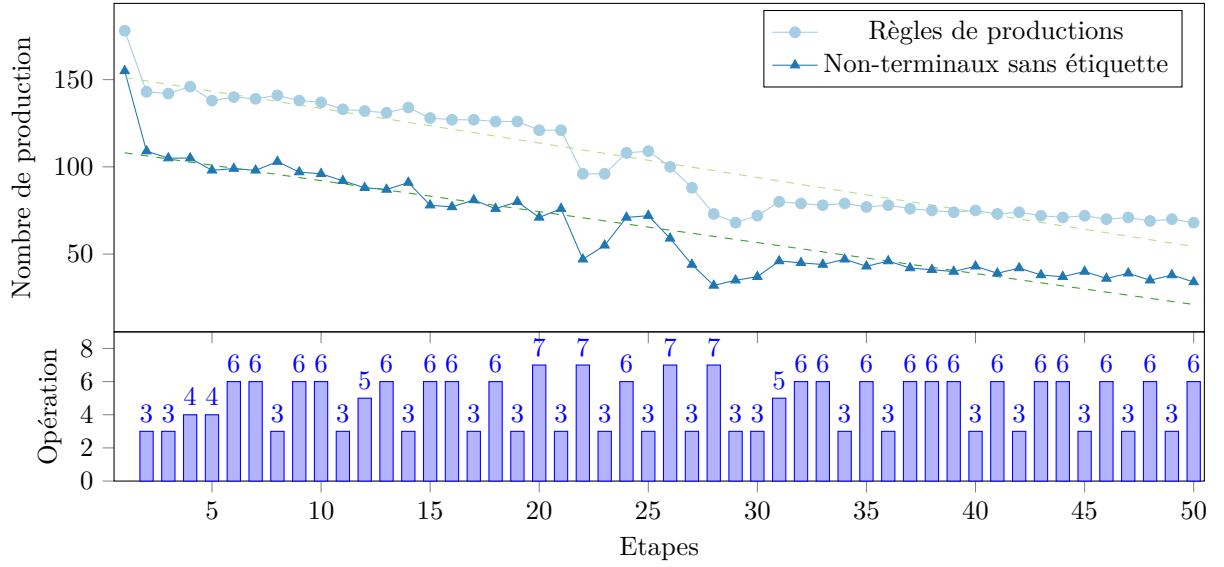


FIGURE 4.32 – Nombre de règles de productions qui compose la grammaire pour chaque étape de la procédure

Une analyse approfondie de la figure 4.32 nous permet de remarquer des moments où le nombre de règles de production reste relativement constant. Ce phénomène se produit, par exemple, aux itérations 21 et 25. Dans ces circonstances, la procédure *instanceEvolution* applique des opérations que nous pouvons qualifier de plus *généralistes* qui se remarque par un changement important du nombre de règles de productions. Cependant, le résultat de ces opérations ne produit pas d'arbres valides ou fréquents et une restructuration est alors nécessaire. C'est à ce stade que nous observons le va-et-vient discuté précédemment. Aux étapes 24, 30 et 31, le nombre de règles de productions augmente rapidement, car la procédure ajoute les structures plus spécifiques (c'est-à-dire, elle modifie des sous-arbres, en les spécifiant différemment, et pour cela, elle augmente le nombre de règles de production), pour permettre de reconnaître, lors d'une itération suivante, une structure plus générique, comme une collection.



FIGURE 4.33 – Nombre moyen d'instances pour chaque catégorie (*groupe*, *relation*, *collection*)

La figure 4.33 illustre le comportement de la procédure `instanceEvolution` par rapport au nombre d'instances associées à chacune des structures (aspect 2 ci-dessus). La courbe $\text{---}\text{+}\text{---}$ montre le nombre de classes d'équivalence qui sont construites à chaque étape. On remarque que ce nombre diminue avec le temps (passant de 25 classes d'équivalence à 17) ce qui signifie que les structures s'unifient au fur et à mesure des étapes et que les transformations opérées permettent de rapprocher des groupes distincts.

Les courbes $\text{---}\bullet\text{---}$, $\text{---}\blacktriangle\text{---}$ et $\text{---}\blacksquare\text{---}$ illustrent respectivement le nombre moyen d'instances pour chaque *groupe*, *relation* et *collection*. Ces nombres restent plutôt stables, malgré la diminution du nombre de nœuds non étiquetés. Au début de l'exécution, le nombre moyen d'instances est de 34.2 pour les groupes et de 2.2 pour les relations. À la fin de l'exécution, on observe en moyenne 29.8 instances pour les groupes et 3 pour les relations. La diminution du nombre d'instances pour les groupes peut s'expliquer par leur promotion au rang de relations, augmentant ainsi le nombre moyen d'instances par relation. Concernant la baisse du nombre d'instances pour les collections, cela s'explique par la fusion successive des collections. Après 50 itérations, on obtient en moyenne 1.4 instances par collection. Dans l'idéal, le schéma ne devrait contenir qu'une seule collection par *groupe* et par *relation*. On devrait alors obtenir 1 instance par collection, signifiant que toutes les collections ont bien été fusionnées les unes avec les autres.

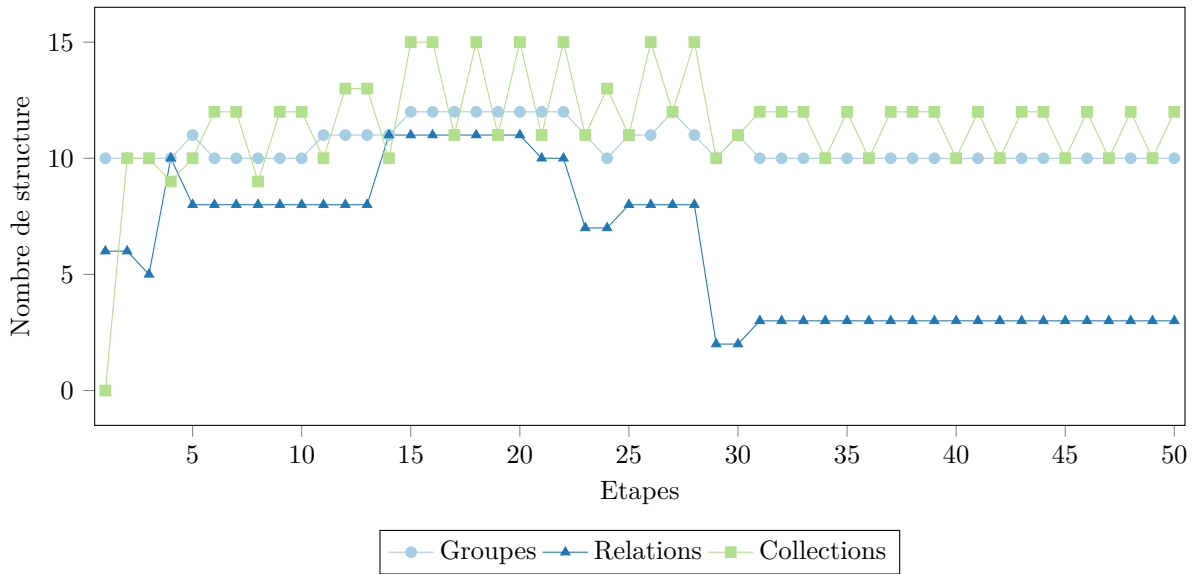


FIGURE 4.34 – Nombre de structures remarquables différentes pour chaque catégorie (*groupe*, *relation*, *collection*)

La figure 4.34 met en lumière le comportement de la procédure `instanceEvolution` par rapport au nombre de structures différentes pour chaque catégorie (aspect 3 ci-dessus). Ainsi, dans la figure 4.34, $\text{---}\bullet\text{---}$ représente le nombre de groupes, $\text{---}\blacktriangle\text{---}$ le nombre de relations et $\text{---}\blacksquare\text{---}$ le nombre de collections.

Dans l'instance I_0 (à la première itération), il est possible de construire 10 groupes différents et d'annoter 6 relations et 0 collections. Au bout de l'itération 28, la procédure a construit 11 groupes, 8 relations et 15 collections. Il s'agit de l'itération qui contient le moins de règles de production. Les itérations 29 et 30 fusionnent les collections après la réduction de l'arbre par le haut (comme expliqué dans la section 4.6, page 70), exécutée durant l'itération 28. On observe alors une chute significative du nombre de relations, où seulement 2 relations sont identifiées avant qu'une troisième ne soit découverte à l'itération 31. Cela peut s'expliquer par le changement au niveau des groupes : à l'itération 29, nous passons de 15 à 10 groupes. La suppression de ces groupes entraîne également la suppression des relations associées. En revanche, le nombre moyen d'instances par relation augmente, ce qui implique que deux classes d'équivalence correspondant à des groupes ont fusionné, entraînant ainsi la fusion de relations (nous avons donc moins de relations différentes, mais plus d'instances pour chaque relation). Nous observons le même phénomène à l'itération 21. Cela est caractéristique de l'unification de l'instance.

La procédure se termine avec la construction de 10 groupes, 3 relations et 12 collections (on obtient alors quasiment une collection pour chaque groupe et relation). Nous remarquons que le nombre de

groupes et de collections reste stable après l'itération 31. Cela indique que nous avons identifié une structure qui représente bien les données et que les étapes suivantes servent uniquement à supprimer ou à réécrire l'instance sans avoir d'impact significatif sur le schéma obtenu.

Stratégie La stratégie employée, qui vise à repérer les éléments fréquents, semble converger vers une solution satisfaisante. Cependant, elle est particulièrement sensible aux paramètres sélectionnés, tels que le support minimal ou τ . La figure 4.35 illustre un exemple de l'échec de cette méthode sur le même jeu de données, avec $\tau = 0.7$ plutôt que $\tau = 0.5$ comme mentionné précédemment. De manière intuitive, on pourrait espérer que l'opération `mergeGroups` fusionne le groupe 7 avec l'entité *Anatomie*. Cependant, même si la structure de l'arbre suggère cette association, elle n'est pas réalisée. L'intégration de l'entité *Anatomie* dans le groupe 7 éloigne trop le nouveau groupe des autres instances, créant ainsi un autre ensemble équivalent dont le support est trop faible. Cela est notamment dû au fait que les éditions sont traitées individuellement et non comme un ensemble. En effet, si l'ensemble des groupes de la classe d'équivalence était modifié simultanément, cette association aurait peut-être été réalisée.

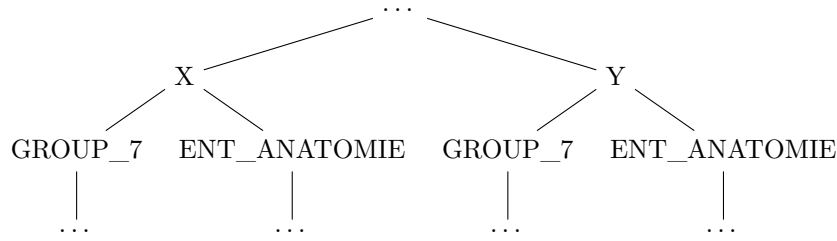


FIGURE 4.35 – Exemple d'instance

Dans la suite, nous étudions la grammaire de schéma obtenue à l'issue de la procédure `instanceEvolution`.

4.7.2 Étude critique de la grammaire obtenue

Dans notre approche, la procédure `instanceEvolution` effectue des modifications sur l'arbre afin de construire l'instance cible à partir de laquelle la grammaire (ou le schéma) cible est extraite, comme expliqué dans la section 4.5.2. Notez que cette grammaire est obtenue après suppression des nœuds non étiquetés ainsi que les entités orphelines n'étant pas rattachées à un nœud *groupe*. Après l'exécution étudiée dans la section précédente avec les paramètres d'expériences définis, nous obtenons la grammaire suivante :

$$\begin{array}{ll}
 COLL_0 \rightarrow GROUP_0^+ & COLL_1 \rightarrow GROUP_1^+ \\
 COLL_2 \rightarrow GROUP_2^+ & COLL_3 \rightarrow GROUP_3^+ \\
 COLL_4 \rightarrow GROUP_4^+ & COLL_6 \rightarrow GROUP_6^+ \\
 COLL_8 \rightarrow GROUP_8^+ & COLL_9 \rightarrow GROUP_9^+ \\
 COLL_{10} \rightarrow GROUP_{10}^+ & COLL_{11} \rightarrow GROUP_{11}^+ \\
 COLL_{11 \leftrightarrow 8} \rightarrow REL_{11 \leftrightarrow 8}^+ & COLL_{2 \leftrightarrow 6} \rightarrow REL_{2 \leftrightarrow 6}^+ \\
 GROUP_0 \rightarrow ENT_{Dose} ENT_{Frequence} & GROUP_1 \rightarrow ENT_{Examen} ENT_{Valeur} \\
 & ENT_{Mode} ENT_{Substance} \\
 & ENT_{Sosy} ENT_{Traitement} \\
 GROUP_3 \rightarrow ENT_{Dose} ENT_{Examen} & \\
 & ENT_{Sosy} ENT_{Substance} \\
 GROUP_8 \rightarrow ENT_{Examen} ENT_{Sosy} & \\
 GROUP_{10} \rightarrow ENT_{Substance} & \\
 REL_{10 \leftrightarrow 9} \rightarrow GROUP_{10} GROUP_9 & \\
 REL_{2 \leftrightarrow 6} \rightarrow GROUP_2 GROUP_6 & \\
 & GROUP_4 \rightarrow ENT_{Anatomie} ENT_{Examen} \\
 & ENT_{Sosy} \\
 & GROUP_6 \rightarrow ENT_{Dose} \\
 & GROUP_9 \rightarrow ENT_{Mode} \\
 & GROUP_{11} \rightarrow ENT_{Anatomie} \\
 & REL_{11 \leftrightarrow 8} \rightarrow GROUP_{11} GROUP_8
 \end{array}$$

En raison de l'utilisation de noms peu expressifs, le schéma obtenu peut sembler non intuitif. Néanmoins, il est valide conformément à la méta-grammaire.

Groupes En analysant en détails les différents groupes identifiés et en considérant leur sémantique, nous constatons des associations prévisibles telles que : le groupe 1 (*Examen, Valeur*) représentant le résultat d'un examen, le groupes 0 (*Dose, Fréquence, Mode, Substance, Sosy, Traitement*) représentant un traitement ou encore les groupes 1, 3, 8 et 4 représentant un résultat d'examen. On identifie les points suivants :

1. Il est intéressant de noter que certains groupes correspondent à ce qui serait intuitivement interprété comme une relation. Par exemple, le groupe 8 (*Examen, Sosy*) et le groupe 4 (*Anatomie, Examen, Sosy*) représentent une relation entre un examen (groupe 1) et un symptôme. Dans ce cas, l'entité *Anatomie* peut faire référence à la localisation de l'examen ou du symptôme. Avec une connaissance du domaine d'application, il serait préférable de construire deux groupes distincts : un premier pour représenter l'examen (*Examen, Anatomie*) et un second pour représenter le symptôme (*Sosy, Anatomie*). Le lien entre ces deux groupes serait alors représenté par une relation. Cependant, en analysant le corpus, on remarque que ces informations sont souvent interdépendantes et qu'il existe peu d'autres liens avec d'autres entités dans le corpus. Il est alors cohérent de les regrouper systématiquement dans un unique groupe, pour minimiser la taille de la grammaire, remplaçant ainsi le besoin de construire deux groupes distincts et une relation.
2. Une prescription se compose intuitivement d'un ensemble d'éléments (*Traitement* ou *Substance, Dose, Mode, Fréquence*) pour un symptôme ou une maladie (*Sosy*). Dans le corpus, un traitement est presque systématiquement associé à un symptôme. Il est donc naturel de retrouver le groupe 0 qui représente un traitement associé à un symptôme. La critique à faire ici est que, du point de vue des bases de données, le symptôme pourrait être représenté comme un groupe distinct, ce qui permettrait d'établir des relations entre l'examen et le symptôme, ainsi qu'entre le symptôme et le traitement. Néanmoins, notre approche n'intègre pas cette connaissance du domaine. Il est donc tout à fait naturel qu'en suivant le raisonnement implémenté, et étant donné que le symptôme est généralement représenté par une seule entité (parfois associée à une *Anatomie* ou à une *Valeur*), que l'on retrouve cette association. Comme pour le point précédent, cette construction permet de minimiser la taille de la grammaire en évitant la construction d'une relation supplémentaire. Il est à noter que si notre corpus comptait avec plus d'instances de l'entité *Sosy* isolée, la procédure *instanceEvolution* aurait pu favoriser la création du groupe *Sosy* et donc des relations par la suite.
3. On remarque que les groupes 1, 3, 8 et 4 exprime sémantiquement la même chose : un résultat d'examen. Mais chaque groupe a une forme de surface différente. On pourrait critiquer que ces groupes n'ont pas été rapprochés sémantiquement (sous un unique non-terminal) mais il est aussi possible de reconnaître qu'il s'agit en réalité de type d'examen différent. Certains sont simplement évoqué (groupe 4), certains résulte en une valeur (groupe 1) et d'autres mesures une certaines doses (groupe 3). En particulier, le groupe 3 est ambigu et peut représenter également un traitement suite à un examen dans certaines phrases. Pour faire le lien avec le point précédent, la présence de l'entité *Sosy* pourrait être également représenté par une relation.
4. Nous remarquons qu'un grand nombre de groupes sont unitaires. Les groupes 9 (*Mode*), 10 (*Substance*) et 11 (*Anatomie*) correspondent à des entités associées à un ensemble fini de valeur. Il est donc assez naturel de les isolés, malgré le fait qu'on retrouve également ces entités dans des groupes plus larges. Par contre, les entités *Dose* et *Fréquence* ne sont pas associé à un domaine fini.

Relations En ce qui concerne les relations, on remarque qu'uniquement trois relations ont été identifiées. La première relation, $11 \leftrightarrow 8$, relie l'entité *Anatomie* et le groupe 8 (*Examen, Sosy*). Elle correspond en réalité à une forme plus structurée du groupe 4, qui n'a pas été unifiée.

Les relations $10 \leftrightarrow 9$ entre les entités *Dose* et *Substance*, ainsi que la relation $2 \leftrightarrow 6$ entre *Fréquence* et *Dose*, représentent un sous-ensemble des entités associées au traitement (groupe 0) et auraient pu être intuitivement représentées par le même non-terminal. Cette constatation met en lumière la principale limitation de la procédure *instanceEvolution* : elle identifie les structures en fonction de leurs usages plutôt que de leur sémantique. Cette limitation est particulièrement notable dans ce cas, car le regroupement intuitif correspondant à un traitement contient de nombreuses entités, dont certaines ne sont pas toujours

présentes. Ainsi, bien que leur sémantique soit similaire, la similarité des sous-arbres correspondants est relativement faible.

L'utilisation d'une mesure de similarité contextuelle a été pensée pour cette situation, mais le groupe correspondant au traitement est souvent beaucoup plus large que son contexte (c'est-à-dire les entités qui gravitent autour du sous-arbre). Il serait pertinent d'explorer d'autres mesures de similarité pour tenter de répondre à cette problématique. L'application de la procédure *instanceEvolution* sur un jeu de données où les groupements sont plus petits et mieux équilibrés pourrait alors conduire à de meilleurs résultats.

En conclusion de cette évaluation critique, il est important de souligner que la composante principale de notre stratégie est syntaxique. Le prototype fonctionne en utilisant uniquement la structure de l'arbre, tandis que les informations sémantiques se limitent aux entités nommées. Le résultat obtenu est intéressant, car les associations générées sont cohérentes par rapport aux textes et permettent de produire une structure homogène qui pourrait être interrogée ou utilisée pour des analyses ultérieures. Plusieurs pistes peuvent être envisagées pour enrichir ou améliorer cette première version de notre approche :

- Prendre en considération des contraintes comme les dépendances fonctionnelles entre des entités qui viendraient améliorer le schéma obtenu en évitant des redondances dans l'instance de la base. Par exemple, supposons que la dépendance fonctionnelle $ENT_{Substance} \rightarrow ENT_{Mode}$ est valable dans notre domaine d'application. La règle associée à $GROUP_0$ pourrait être réécrite en supprimant $ENT_{Substance} ENT_{Mode}$ de son côté droit. Ensuite, il serait nécessaire d'ajouter une nouvelle relation entre $GROUP_0$ et le non-terminal qui regrouperait les deux entités supprimées. Il est intéressant de remarquer que des contraintes moins restrictives que les dépendances fonctionnelles pourraient donner lieu à la même modification. Pour évaluer si une telle modification serait intéressante, il serait nécessaire d'observer le nombre de valeurs identiques (ici, associés à la paire $ENT_{Substance} ENT_{Mode}$) par rapport au nombre d'instances de $GROUP_0$.
- Explorer de nouvelles fonctions objectif pourrait être bénéfique. Par exemple, plutôt que de se concentrer uniquement sur la minimisation de la taille de la grammaire, il pourrait être avantageux de chercher à maximiser la similarité au sein des groupes. Il est aussi possible de combiner plusieurs objectifs.

4.7.3 Intégration de données

L'algorithme opère sur des arbres quelconques, ce qui permet l'insertion de données préalablement structurées. Par conséquent, outre la structuration de données semi-structurées, il est possible d'intégrer de manière itérative de nouvelles données à une base existante. En se basant sur l'implémentation proposée qui repose sur la fréquence d'apparition des différentes structures, on peut supposer que les nouvelles données seront structurées de manière cohérente avec les données existantes dès lors que nous disposons d'un nombre suffisant d'exemples et que le nombre de nouvelles données est inférieur à celui des données existantes. Pour vérifier cette théorie, nous considérons une instance composée des groupes suivants :

$$\begin{aligned} SOSY &= \{ENT_{Sosy}, ENT_{Anatomie}, ENT_{Substance}\} \\ Traitement &= \{ENT_{Dose}, ENT_{Frequence}, ENT_{Mode}, ENT_{Substance}\} \\ Examen &= \{ENT_{Examen}, ENT_{Anatomie}\} \end{aligned}$$

L'instance comprend également des relations entre les entités *SOSY* et *Traitement*, ainsi que des relations entre les groupes *Examen* et *SOSY*. Finalement, l'instance est composée de 50 exemples de chaque structure et nous cherchons à structurer 10 phrases supplémentaires. Les phrases prises en compte ne concerne que des traitements, il n'y a donc pas de mention d'examen ou de résultats. Il est important de noter que les regroupements dans l'instance initiale contiennent toujours l'ensemble de toutes les entités, ce qui diminue la similarité par rapport aux textes ou certaines entités peuvent être manquantes. Pour tenter de corriger cela, on utilise un taux relativement faible ($\tau = 0.2$) et un support minimum de 30.

Après exécution de la procédure, on obtient la grammaire de schéma suivante :

$$\begin{array}{ll}
COLL_0 \rightarrow GROUP_0^+ & COLL_1 \rightarrow GROUP_1^+ \\
COLL_2 \rightarrow GROUP_2^+ & COLL_3 \rightarrow GROUP_3^+ \\
COLL_4 \rightarrow GROUP_4^+ & \\
COLL_{0 \leftrightarrow 1} \rightarrow REL_{0 \leftrightarrow 1}^+ & COLL_{1 \leftrightarrow 2} \rightarrow REL_{1 \leftrightarrow 2}^+ \\
GROUP_0 \rightarrow ENT_{Dose} ENT_{Frequence} & GROUP_1 \rightarrow ENT_{Anatomie} ENT_{Sosy} \\
& ENT_{Mode} ENT_{Substance} \\
GROUP_2 \rightarrow ENT_{Examen} ENT_{Anatomie} & GROUP_3 \rightarrow ENT_{Mode} ENT_{Substance} \\
GROUP_4 \rightarrow ENT_{Substance} & \\
REL_{0 \leftrightarrow 1} \rightarrow GROUP_0 GROUP_1 & REL_{1 \leftrightarrow 2} \rightarrow GROUP_1 GROUP_2
\end{array}$$

On observe que les groupes et les relations fournis en entrée sont bien retrouvés. De plus, 12 nouvelles instances du groupe 1 sont identifiées, parmi lesquelles 9 contiennent l'entité *SOSY* et 3 incluent le couple *Anatomie-SOSY*. De nouveaux regroupements sont également découverts, avec 7 instances du *GROUP_4* composées uniquement de l'entité *Substance*, et 2 instances du *GROUP_3*, l'une contenant l'entité *Mode* et l'autre le couple *Mode-Substance*. On remarque également que le *GROUP_3* aurait pu être identifié comme *GROUP_0* car il correspond à un sous-ensemble. Pour le *GROUP_4*, le fait de séparer l'entité *Substance* n'est pas illogique, car elle peut être ambiguë entre *GROUP_0* et *GROUP_1*. Ces résultats mettent en lumière le potentiel de la structuration de nouvelles données dans une instance existante.

4.8 Conclusion

Dans ce chapitre, nous avons exploré une approche reposant sur la réécriture d'arbre pour résoudre le défi de la structuration automatique de l'information en vue de son enregistrement dans une base de données. Notre approche peut se résumer aux contributions suivantes :

- La formalisation d'un schéma de base de données valide à l'aide d'une méta-grammaire S-attribuée.
- La proposition d'un algorithme itératif, composé d'étapes clés pour la structuration automatique des données et l'extraction d'une grammaire cible correspondant au schéma envisagé pour une base de données.
- L'adoption d'une approche symbolique hybride intégrant des techniques statistiques pour guider la sélection des règles de réécriture à appliquer.
- Le développement d'un prototype mettant en œuvre notre approche, avec pour objectif la minimisation de la taille de la grammaire. Pour cela, l'implémentation vise à optimiser la taille des groupes identifiés tout en réduisant leur variabilité à l'aide d'une mesure de fréquence.
- La réalisation d'une preuve de concept, accompagnée d'une analyse critique de notre approche, confirmant sa capacité à atteindre l'objectif fixé tout en évaluant la qualité du schéma de base de données produit.

L'approche développée dans ce chapitre représente une étape significative pour l'intégration structurée de données textuelles dans une base de données. Sa caractéristique principale réside dans sa capacité à être déployée sans nécessiter de données d'apprentissage, la rendant ainsi facilement adaptable à divers domaines d'application. De plus, sa transparence permet à l'utilisateur de suivre et de valider chaque étape du processus de structuration, permettant d'assurer la confiance dans le système et la qualité des données produites.

La structuration automatique de données peut parfois aboutir à des instances incomplètes, où certaines entités peuvent manquer dans l'instance d'un groupe. Cette incomplétude peut être imputée à divers facteurs tels que des omissions dans le texte ou des erreurs lors du processus d'extraction. Cette notion d'instances incomplètes est alors représentée en utilisant des valeurs nulles, ce qui fait écho avec le concept de mise à jour de base de données incomplète présenté dans le chapitre 3. Les méthodes mises en œuvre pour garantir la cohérence dans le chapitre 3 ajoutent ainsi une étape supplémentaire visant à assurer la qualité de l'instance obtenue.

Perspectives Il est possible d'améliorer la procédure actuelle, comme déjà mentionné, par différentes extensions. Par exemple, en prenant en considération des règles métier spécifiques, en intégrant un processus de détection automatique des dépendances fonctionnelles [PEM⁺15] ou en incorporant des relations entre les entités, extraites au préalable (aujourd'hui, l'approche se base uniquement sur les entités extraites depuis le texte et suppose que l'annotation sémantique et la résolution de la coréférence a déjà été effectuée). En outre, notre approche laisse prévoir des perspectives que nous illustrons dans la suite :

- La structure de l'arbre obtenu à la fin de l'application de la procédure `instanceEvolution` est générique (c.-à-d., quelles n'est pas dépendante d'un modèle de données). Elle peut ainsi être utilisée pour représenter des données dans différents formats (semi-structurées ou structurées), de manière proche de ce qui est proposée dans [BMU22]. Cela ouvre la possibilité d'utiliser cette approche dans le cadre de l'intégration des données.
- L'instance obtenue grâce à notre méthode peut être intégrée dans une base de données existante en utilisant un mapping de schéma.
- Dans la section 4.7.3, nous avons exploré la possibilité de mettre en œuvre un processus incrémental d'intégration de données. On suppose l'existence d'une grammaire G conforme à la méta-grammaire et d'une instance volumineuse et valide par rapport à G représentée par l'arbre T . Il suffit alors d'appliquer la procédure `instanceEvolution` sur un nouvel arbre T' construit à partir de l'arbre existant T , auquel de nouveaux sous-arbres sont ajoutés pour représenter de nouvelles données.

Chapitre 5

Extraction d'information

Sommaire

5.1	Introduction	91
5.2	Corpus et analyse syntaxique	92
5.3	Extraction d'entités ambiguës	94
5.3.1	Entités multi-valuées	95
5.3.2	Extraction par lexiques	95
5.3.3	Cascade de CRFs	98
5.4	Enrichissement et contextualisation	99
5.5	Classification de cas cliniques	101
5.5.1	Système de classification basé sur l'extraction par lexiques	101
5.5.2	Négation et incertitude	102
5.5.3	Évaluation	103
5.6	Interrogation de base de données en langue naturelle	103
5.6.1	Extraction d'entités	105
5.6.2	Construction de DB-query à partir d'entités enrichies	106
5.6.3	Évaluation	108
5.6.4	Construction de graphique	109
5.7	Conclusion	111

5.1 Introduction

Ce chapitre synthétise les travaux menés simultanément à l'approche présentée dans le chapitre 4, qui se concentre sur la structuration automatique des données textuelles. Les travaux de ce chapitre portent sur l'extraction d'informations à partir des textes, notamment la reconnaissance des entités nommées. L'extraction ou la reconnaissance d'entités nommées [Pat15], Named-Entity Recognition (NER) en anglais, est une tâche commune du TAL qui consiste à identifier la partie d'un texte non structuré qui représente une entité nommée. Cette tâche peut être réalisée soit par des techniques basées sur des grammaires, soit par un modèle statistique tel que l'apprentissage automatique (voir [JM08] pour une introduction complète dans ce domaine). Les approches statistiques sont largement utilisées dans l'industrie, car elles offrent de bons résultats. Cependant, ces approches nécessitent beaucoup de données pour leur apprentissage, ce qui implique des coûts élevés pour leur récupération, leur annotation et l'apprentissage. Dans le cadre d'Ennov, les solutions mises en place concernent des données du domaine médical contenant des informations sensibles, telles que des données de patients ou des secrets industriels. L'utilisation de l'apprentissage à partir de ces données présente des risques liés à la protection de la vie privée et au secret de fabrication [FJR15, SSM19]. Dans ce contexte, la compartimentation des données est cruciale. Une approche fréquemment adoptée consiste à créer un modèle générique et à l'affiner pour chaque utilisateur ou source de données [RSG16]. Cette pratique permet également de réduire les coûts, ce qui explique son adoption courante, notamment avec des modèles tels que Bidirectional Encoder Representations from Transformers (BERT) [DCLT19] ou Generative Pre-trained Transformer (GPT)

[RNSS18, BMR⁺20]. Cette approche soulève les mêmes problématiques en matière d'accès aux données, car le modèle générique doit être construit sur des données publiques ou synthétiques qui ne sont pas toujours disponibles. Dans ce contexte, l'apprentissage fédéré émerge comme une solution potentielle, permettant la création d'un modèle global à partir d'entraînements locaux. Cependant, la construction de modèles locaux par source de données se heurte au manque de données annotées disponibles localement. Une solution possible consiste à combiner l'utilisation de modèles d'analyse génériques avec des approches symboliques spécifiques. Cette approche offre l'avantage d'utiliser un système réutilisable, mais en capacité d'être localement spécialisé.

Extraction itérative d'entités L'extraction d'entités peut représenter une tâche complexe en raison de la diversité des types d'entités, de leur valeur, de leur taille ou de leur imbrication. Dans cette thèse, nous proposons de décomposer le processus d'extraction en sous-tâches plus simples, permettant d'exploiter une combinaison de diverses méthodes. Initialement, on se focalise sur l'extraction d'informations basiques à l'aide de grammaires ou de lexiques. Puis, progressivement, au travers d'une série d'étapes, on s'attelle à la combinaison, la correction, le filtrage et l'enrichissement de ces entités élémentaires afin de parvenir à des entités enrichies sémantiquement. Ainsi, dans la phrase « Marie est entrée à l'hôpital », on commencera par déterminer que *Marie* est une personne avant d'ajouter l'information plus précise que *Marie* est une patiente. Cette approche permet de tirer parti d'outils génériques pour l'analyse syntaxique et l'extraction d'entités simples, tels que l'extraction de nombres ou de dates. Dans la suite, il est possible d'appliquer des techniques d'apprentissage, en se basant sur les entités extraites plutôt que sur le texte brut. Cela permet de construire un processus d'extraction ne nécessitant pas l'accès à de larges ensembles de données du domaine étudié.

Organisation Ce chapitre commence par une description (section 5.2) du corpus étudié dans cette thèse. La section 5.3 présente les techniques mises en œuvre pour la reconnaissance d'entités nommées et la section 5.4 présente la notion de contextualisation des entités. Enfin, le chapitre se termine sur la présentation de deux cas d'usage direct des techniques présentées dans ce chapitre :

- la classification de cas cliniques (section 5.5) et
- les interfaces d'interrogation en langue naturelle (section 5.6).

La section 5.7 conclut ce chapitre.

5.2 Corpus et analyse syntaxique

Les travaux présentés dans cette thèse se basent sur le corpus CAS – Corpus français de cas cliniques [GCD18] - utilisé dans plusieurs éditions de DÉfi Fouille de Textes (DEFT) [CGGH20, GGI21], une campagne d'évaluation des systèmes dédiés au traitement de texte médical en français. Le corpus est composé de 167 cas cliniques en français (100 cas pour l'entraînement et 67 pour les tests) et contient 13 548 entités. Les cas cliniques proviennent de publications dans la littérature scientifique et d'échantillons pédagogiques destinés aux étudiants en médecine. Les cas sont anonymisés et décrivent des situations réelles ou fictives issues de différents domaines médicaux (cardiologie, urologie, oncologie, etc.). Certaines parties du corpus ont été annotées manuellement à partir de zéro (par deux annotateurs), tandis que d'autres ont été pré-annotées automatiquement (en utilisant des CRFs) puis corrigées manuellement.

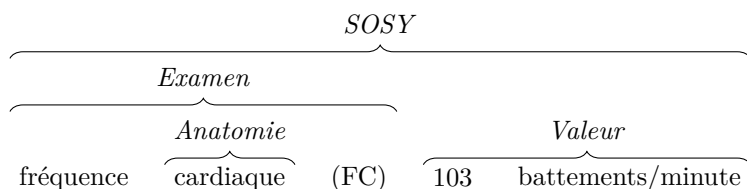


FIGURE 5.1 – Imbrication d'entités dans le corpus CAS

Le corpus contient des annotations pour 13 types d'entités : 2 pour les expressions temporelles (*Date*, *Moment*), 4 pour les objets et faits médicaux (*Anatomie*, *Examen*, *Pathologie* et *Signe ou Symptôme* -

SOSY), et 7 pour le traitement des patients (*Dose*, *Durée*, *Fréquence*, *Mode*, *Substance*, *Traitement* et *Valeur*). Des entités plus courtes peuvent être imbriquées dans des entités plus grandes, par exemple une entité *Examen* peut contenir la partie du corps (*Anatomie*) sur laquelle l'examen est réalisé (figure 5.1). Le tableau 5.1 montre le nombre d'entités de chaque type et montre les imbrications possibles. On ne s'intéresse pas aux entités temporelles ici.

Entité	Support	Entités imbriquées
<i>SOSY</i>	1831	<i>Substance</i> , <i>Examen</i> , <i>Anatomie</i> , <i>Valeur</i>
<i>Anatomie</i>	1608	
<i>Examen</i>	1218	<i>Substance</i> , <i>Anatomie</i>
<i>Substance</i>	1024	
<i>Valeur</i>	588	
<i>Moment</i>	451	
<i>Dose</i>	392	
<i>Traitement</i>	374	<i>Anatomie</i>
<i>Pathologie</i>	369	<i>Anatomie</i> , <i>Valeur</i>
<i>Mode</i>	243	
<i>Fréquence</i>	243	

TABLE 5.1 – Nombre d'entités par type dans le corpus d'entraînement

Dans le tableau 5.2, nous présentons les imbrications les plus fréquentes dans le corpus. La première colonne contient le type de l'entité englobante, la deuxième colonne le type de l'entité imbriquée, la troisième colonne le nombre de fois où ces deux entités sont imbriquées et les deux dernières colonnes le pourcentage d'entités imbriquées par rapport au nombre d'entités 1 ou 2 dans le corpus tel que :

$$P_{ent_1} = \text{nombre } ent_2 \text{ imbriquée dans } ent_1 / \text{nombre } ent_1$$

$$P_{ent_2} = \text{nombre } ent_2 \text{ imbriquée dans } ent_1 / \text{nombre } ent_2$$

Nous remarquons entre autres que les entités *SOSY* englobent très souvent d'autres entités, en particulier des entités *anatomie* et que les entités *anatomie* sont très souvent imbriquées (au total 1496 sont imbriquées, parfois dans plusieurs entités à la fois, ce qui représente 93% des entités *anatomie*). Ces entités *sosy* sont souvent longues, avec en moyenne 4,9 mots par entité, contre 1,5 mots par entité pour les entités *anatomie*. Cet aspect nous a incités à construire un système en cascade, en suivant les travaux de [AHG07]. Ainsi, différents modèles seront entraînés pour apprendre à reconnaître un ou deux types d'entités.

<i>ent₁</i>	<i>ent₂</i>	Nombre d'imbrications	<i>P_{ent₁}</i>	<i>P_{ent₂}</i>
<i>SOSY</i>	<i>Anatomie</i>	980	54 %	61 %
<i>SOSY</i>	<i>Examen</i>	440	24 %	36 %
<i>SOSY</i>	<i>Valeur</i>	409	22 %	70 %
<i>SOSY</i>	<i>Substance</i>	80	4 %	8 %
<i>Pathologie</i>	<i>Anatomie</i>	151	41 %	9 %
<i>Pathologie</i>	<i>Valeur</i>	18	5 %	3 %
<i>Traitement</i>	<i>Anatomie</i>	111	30 %	7 %
<i>Examen</i>	<i>Anatomie</i>	370	30 %	23 %
<i>Examen</i>	<i>Substance</i>	23	2 %	2 %

TABLE 5.2 – Nombre d'entités imbriquées dans le corpus d'entraînement

Afin de pouvoir traiter les textes, on commence par réaliser une analyse syntaxique composée :

- d'une tokenisation permettant de déterminer les unités lexicales (les mots) ;
- de l'étiquetage des parties du discours qui détermine le rôle du mot dans une phrase (verbe, sujet, etc) ;
- de la lemmatisation pour déterminer la forme canonique des mots ;

- de la racinisation qui retire les suffixes et préfixes des mots et
- de l'analyse en dépendance qui donne l'arbre de dépendance entre les unités lexicales.

L'ensemble de l'analyse syntaxique est réalisée avec l'aide de *SpaCy* [Spa, HMLB20, PA21]. Pour le français, on utilise le modèle `fr_core_news_md` (3.7.0) qui est construit sur un Convolutional Neural Network (CNN) [HM17] à partir des corpus français Sequoia [CS12, BCC⁺20] et WikiNER [NRR⁺13, Lea17]. Pour l'anglais, on utilise le modèle `en_core_sci_md` (0.5.2) de ScispaCy [NKBA19, All20] entraîné sur les corpus OntoNotes [WHM⁺11, WPM⁺13] et GENIA [KOTT03, MC08].

5.3 Extraction d'entités ambiguës

L'extraction d'informations dans un texte implique souvent l'identification et la classification des entités nommées [AA16, GKG18, NJM21]. Elles représentent des éléments spécifiques dans un texte qui ont un nom propre ou une désignation particulière. Ces éléments peuvent inclure des personnes, des lieux, des organisations, des dates, des quantités, des mesures, etc. Une entité nommée est alors représentée par une plage de texte ainsi qu'un type. Dans ce cadre, l'annotation sémantique est un processus du TAL qui vise à associer une valeur aux entités, notamment des références dans une base de connaissances ou une base de données, mais aussi des valeurs numériques ou encore des dates.

Les approches récentes dans la littérature pour l'extraction d'entités se reposent généralement sur l'apprentissage automatique [NJM21]. Bien que ces approches offrent généralement de bonnes performances, elles nécessitent d'avoir de grands volumes de données annotées pour leur apprentissage. Depuis quelques années, les modèles basés sur l'apprentissage profond, tels que les CNN et les modèles à base de transformateurs comme BERT [DCLT19] ou les Large Language Models (LLMs) tels que GPT [RNSS18, BMR⁺20], ont montré des résultats prometteurs dans une variété de tâches de traitement automatique du langage naturel. Cependant, l'utilisation de modèles génériques peut poser des défis dans des domaines spécifiques tels que le médical, où le vocabulaire peut être plus complexe et où des données annotées sont souvent rares et coûteuses à obtenir. Dans cette thèse, nous explorons l'idée d'associer des solutions génériques pour l'extraction d'informations avec des approches plus *traditionnelles* adaptées au domaine médical.

Lexiques Lorsque le champ des valeurs à extraire est restreint, connu à l'avance et disponible dans une base de données, l'utilisation de lexiques s'avère pertinente surtout quand les données d'entraînement sont difficilement accessibles. L'extraction d'entités au moyen de lexiques est une approche courante dans le domaine du traitement automatique du langage naturel [NJM21] et qui remonte aux premiers systèmes d'extraction d'entités [Rau91]. Certaines études comme [PKM14] s'intéressent à utiliser des lexiques pour améliorer les systèmes basés sur des réseaux de neurones. Les lexiques ont l'intérêt d'être peut-être coûteux, de ne pas nécessiter d'entraînement et sont d'autant plus intéressants quand on travaille sur un domaine où le vocabulaire est contrôlé comme pour le médical [CMO12]. Ainsi, la variation linguistique est réduite et les ressources souvent déjà disponibles. Ces lexiques sont essentiellement des répertoires comprenant des termes spécifiques et leurs variations lexicales associées. Par exemple, on pourrait créer un lexique des noms de maladies avec leurs acronymes ou encore construire une liste de personnes associées à leur nom, leur prénom, leur adresse email, etc.

Dans la section 5.3.2, nous listerons certaines ressources disponibles pour le domaine médical. Nous présenterons également une méthode efficace pour rechercher et identifier des entités dans le texte en s'appuyant sur ces lexiques.

Grammaires locales Lorsque les données appartiennent à un domaine infini ou avec une grande variété de valeurs, les grammaires locales sont plus adaptées que les lexiques. Elles sont particulièrement efficaces pour des informations plus ou moins structurées tel que des nombres ou des dates. Cette approche repose sur la définition de règles syntaxiques et grammaticales qui décrivent les schémas récurrents des entités que l'on souhaite extraire. Par exemple, une grammaire locale peut être construite pour identifier des dates sous la forme « jour / mois / année » ou encore des valeurs sous la forme « nombre unité ». L'utilisation de grammaires locales pour l'extraction d'entités a été largement étudiée dans la littérature scientifique.

Nous commençons cette section par une description des entités utilisées dans cette thèse puis, nous présenterons les méthodes mises en place pour l'enrichissement des entités.

5.3.1 Entités multi-valuées

L'ambiguïté inhérente aux textes constitue un défi majeur pour les systèmes de traitement automatique du langage, qui doivent être capables de comprendre précisément le sens des énoncés. Lors de la phase d'extraction des entités, l'ambiguïté peut concerner le type (par exemple *Paris* peut se référer à une ville ou à une personne) ou la valeur (par exemple, plusieurs personnes dans la base de données ont le même nom). En général, on cherche à éliminer cette ambiguïté en ne conservant que la solution la plus probable. Une telle solution peut introduire des contradictions par rapport à la sémantique du texte. Dans nos travaux, nous choisissons de représenter explicitement l'ambiguïté : une même entité peut alors posséder plusieurs valeurs et une valeur peut être associée à différents types. Nous gardons trace des interprétations multiples au cours des étapes d'extraction. Durant certaines étapes du traitement, cette ambiguïté pourra être résolue : si l'on détermine le type exact d'une entité, il est alors possible de supprimer les valeurs incohérentes. De plus, la détection de la coréférence peut aussi permettre de résoudre l'ambiguïté sur les valeurs si on suppose que les valeurs sont alors l'intersection de l'ensemble des références. Si l'ambiguïté ne peut être résolue, c'est l'utilisateur qui sera chargé d'effectuer cette tâche. Nous verrons aussi dans la section 5.6 que, selon l'usage, l'ambiguïté n'a pas nécessairement besoin d'être résolue.

Definition 5.3.1 (Entité multi-valuée). Une entité est représentée par un tuple $E = (\mathcal{V}, \mathcal{T}, m)$ où $\mathcal{V}$ est l'ensemble des valeurs de l'entité, $\mathcal{T}$ l'ensemble des types et m est une fonction surjective telle que $\forall v \in \mathcal{V}, \exists T \subseteq \mathcal{T}, m(v) \rightarrow T$

Cette représentation permet aussi la représentation de la conjonction de coordination « ou ». Deux entités $E_1 = (\mathcal{V}_1, \mathcal{T}_1, m_1)$ et $E_2 = (\mathcal{V}_2, \mathcal{T}_2, m_2)$ liées par une conjonction « ou », peuvent alors être représentées comme une seule entité $E = (\mathcal{V}_1 \cup \mathcal{V}_2, \mathcal{T}_1 \cup \mathcal{T}_2, m_1 \circ m_2)$. Les entités sont extraites en utilisant différentes stratégies de façon indépendante. En particulier, si plusieurs approches identifient des entités se chevauchant (mais pas nécessairement avec les mêmes bornes), nous ne conservons que l'entité résultant de l'union de ces entités pour représenter l'ambiguïté causée par l'extraction. On peut aussi utiliser le même phénomène pour accentuer la précision en gardant alors l'intersection des entités.

5.3.2 Extraction par lexiques

Les lexiques sont un ensemble de termes, de synonymes, d'acronymes, d'expressions et de concepts propres à un domaine ou à une profession spécifique. Ils sont souvent utilisés pour faciliter la communication et la compréhension au sein d'une communauté ou d'un secteur d'activité particulier. Les lexiques permettent aux professionnels d'articuler leur discours de manière précise, de prévenir les éventuels malentendus, et d'assurer une compréhension commune des sujets et des enjeux relatifs à leur domaine d'expertise. Ils constituent une ressource riche pour l'extraction d'entités.

Un lexique se compose d'un ensemble d'entrées chacune composée d'un identifiant unique et d'une liste de lexèmes représentant cette entrée. Les lexiques peuvent être construits à partir de thésaurus, de terminologies ou de bases de connaissances qui représentent des ressources riches qui s'enrichissent avec le temps (souvent semi-automatiquement à partir de corpus annotés) et qui sont très souvent surveillées par une autorité qui vérifie les informations et se charge du nettoyage des entrées. Nous pouvons prendre comme exemple la base GeoNames qui contient un ensemble d'informations sur les lieux géographiques. À partir de cette ressource, un lexique pourrait être constitué d'une collection d'identifiants uniques pour chaque lieu auquel on associe l'ensemble de ses noms possibles. La ville de Paris¹ se voit donc associée l'ensemble de lexèmes suivant : Paris, Île-de-France, Paname, ...

Lexiques médicaux

Le domaine médical ne fait pas exception et fait peut-être partie des plus représentés, notamment avec des institutions comme la U.S. National Library of Medicine (NLM) qui a regroupé, dans le méta-thésaurus Unified Medical Language System (UMLS), un grand nombre de ressources² pouvant être très utiles pour le traitement du langage naturel. Parmi les ressources de l'UMLS on peut notamment citer Medical Subject Headings (MeSH), Medical Dictionary for Regulatory Activities (MedDRA), SNOMED Clinical Terms (SNOMED) et RxNORM.

1. <http://www.geonames.org/2988507/paris.html>

2. <https://www.nlm.nih.gov/research/umls/sourcereleasedocs/index.html>

MeSH Le Medical Subject Headings (MeSH) [Lip00] est un thésaurus du domaine biomédical, à l'origine en anglais, qui est géré par la NLM. Il permet entre autre d'indexer et d'interroger des bases de données comme MEDLINE/PubMed. Une traduction du MeSH en français, mise à jour chaque année, a été réalisée par l'Institut National de la Santé Et de la Recherche Médicale (INSERM). La version bilingue anglais-français peut être interrogée depuis une interface en ligne (<http://mesh.inserm.fr/FrenchMesh/>), et il est également possible de télécharger le thésaurus au format XML. Le MeSH est organisé en 16 catégories thématiques :

A Anatomie	I Anthropologie, enseignement, sociologie et phénomènes sociaux
B Organismes	J Technologie, industrie et agriculture
C Maladies	K Sciences humaines
D Produits chimiques et pharmaceutiques	L Sciences de l'information
E Techniques et équipements analytiques, diagnostiques et thérapeutiques	M Individus
F Psychiatrie et psychologie	N Santé
G Phénomènes et processus	V Caractéristiques d'une publication
H Disciplines et professions	Z Lieux géographiques

TABLE 5.3 – Liste des catégories thématiques du MeSH

MedDRA Medical Dictionary for Regulatory Activities (MedDRA)³ [BWW99] est un dictionnaire terminologique médical développé par le Conseil International d'Harmonisation des exigences techniques pour l'enregistrement des médicaments à usage humain (CIH) et utilisé par les autorités réglementaires et l'industrie bio-pharmaceutique. MedDRA est disponible en plusieurs langues, dont le français. Il contient aussi bien des termes référant à des symptômes, des examens ou encore des traitements, structurés en 5 niveaux qui regroupent les termes par étiologie, site de manifestation, etc. Le niveau le plus haut étant une classification par discipline médicale. Il existe 26 classes, par exemple *Affections vasculaires*, *Affections du rein et des voies urinaires* ou *Affections du système immunitaire*.

Médicaments et substances : Résumés des Caractéristiques du Produit (RCP) Chaque médicament soumis à une procédure d'autorisation de mise sur le marché au niveau européen est accompagné d'un RCP fourni par l'Agence Nationale de Sécurité du Médicament et des produits de santé (ANSM). Ces Résumés des Caractéristiques du Produit (RCP) sont accessibles sur le site de l'European Medicines Agency (EMA) et sont disponibles dans toutes les langues de l'Union Européenne. Ils constituent une source d'information riche pour la création de lexiques destinés à l'extraction des médicaments et des substances actives. Aux États-Unis, on trouve un équivalent dans le Physician's Desk Reference (PDR) (uniquement en anglais) qui est vérifié par l'agence de santé locale, la Food and Drug Administration (FDA).

DrugBank [WKG⁺08, WFG⁺18] (<https://go.drugbank.com/>) est une base de données unifiée en anglais disponible en ligne regroupant des informations détaillées sur les médicaments, y compris leurs propriétés, mécanismes d'action, indications et interactions. Fournit par l'université de l'Alberta au Canada, elle est accessible gratuitement en version publique, avec une version professionnelle disponible pour les chercheurs et les professionnels de la santé.

HeTOP Le projet Catalogue et Index des Sites Médicaux de langue Française (CISMeF) fournit le portail Health Terminology/Ontology Portal (HeTOP) pour accéder à une ontologie multilingue qui regroupe beaucoup de ressources comme l'UMLS.

Transducteurs finis

Les transducteurs finis sont une forme d'automate fini qui reconnaissent un langage, mais qui sont aussi capables de produire une sortie. Ils sont formellement définis comme des machines de Turing à

3. La marque MedDRA est enregistrée par l'IFPMA au nom du CIH.

deux rubans. À notre connaissance, [Gro89] est le premier à introduire l'utilisation des transducteurs finis dans le traitement automatique de la langue naturelle. Les transducteurs finis peuvent être utilisés pour de l'analyse syntaxique [BC02] mais aussi pour l'extraction d'entités [GM17]. [MM01] ont introduit un algorithme permettant de construire un transducteur minimal à partir d'une liste triée de mots reconnus par le langage avec leur sortie. Cet algorithme est celui implémenté dans pour la construction des Finite-State Transducer (FST) dans Lucene, un moteur d'indexation de texte notamment utilisé par SolR.

Définition 5.3.2 (FST). Un transducteur fini $T = (\Sigma^{in}, \Sigma^{out}, Q, I, F, \delta)$ est un automate fini qui reconnaît un langage $L = \{w_1, \dots, w_n\}$ sur un alphabet Σ^{in} où les transitions possèdent deux labels ($l^{in} \in \Sigma^{in}$ et $l^{out} \in \Sigma^{out}$). Le premier caractérise la transition et le second constitue la sortie de l'automate. Q est l'ensemble des états, I les états initiaux, F les états finaux, δ l'ensemble des transitions et ϵ est le mot vide. La sortie de l'automate (quand un mot w est reconnu, c.-à-d. $w \in L$) peut être une somme des labels de sortie, leur concaténation ou, comme ici, une unique valeur (la classe). Un transducteur n'est pas obligatoirement déterministe et peut donc, pour un même mot, retourner plusieurs valeurs de sortie.

Les transducteurs sont des structures plus optimisées en mémoire que d'autres structures comme les tables triées, mais au détriment d'un accès plus coûteux en ressources processeur. Ils sont par conséquent, très utiles pour traiter des langages de grande taille qui ne pourraient pas normalement tenir en mémoire tout en offrant un accès suffisamment rapide. Nous utilisons l'implémentation fournie dans SolR/Lucene par le projet OpenSextant⁴. Elle repose sur l'algorithme de [MM01] qui permet d'obtenir le transducteur minimal efficacement. Les transducteurs gardent aussi l'avantage d'être facilement mis à jour. Il est possible d'ajouter ou de supprimer de nouveaux mots dans le langage sans avoir à reconstruire l'automate entièrement.

Construction Nos lexiques sont définis comme une application surjective $\forall v_i \in V \exists X_i \subset X, Lex : v_i \rightarrow X_i$ où V est l'ensemble des valeurs du lexique et X est l'ensemble des lexèmes présents dans le lexique. X_i est l'ensemble des exemples de la valeur v_i . Afin de construire notre transducteur pour le lexique $Lex : V \rightarrow X$, nous définissons les alphabets $\Sigma^{in} = \{t_i \mid t_i \in token(x_j) \forall x_j \in X\} \cup \{\epsilon\}$ où $token$ est une fonction qui retourne l'ensemble des tokens t_i d'un lexème x_j et $\Sigma^{out} = V \cup \{\epsilon\}$. Notre langage L est alors naturellement défini comme l'ensemble des lexèmes X du lexique, c.-à-d. chaque lexème est un mot du langage.

La fonction *token* permet l'extraction des tokens utilisés pour la construction du transducteur. Cette fonction est aussi appliquée aux textes en entrée afin de les faire correspondre à l'alphabet Σ^{in} . Elle a pour rôle :

- Le découpage des lexèmes (mot ou suite de mots) en tokens. Le découpage est réalisé sur les caractères d'espacement, les ponctuations, les traits d'union et les chiffres accolés à du texte (ex : 50mg devient {50, mg});
- Le passage en minuscule de l'ensemble des tokens;
- Le filtrage des tokens correspondant à des mots vides (basé sur une liste);
- La transformation de tous les tokens non-ASCII par leur équivalent (suppression des accents);
- Le remplacement de chaque token par sa racine en utilisant l'algorithme Snowball [Por01].

Un prétraitement visant à transformer les mots en n -grammes a aussi été envisagé, mais a été jugé trop coûteux en ressources pour un gain de qualité trop faible.

Détection des valeurs Notre système transforme le texte en entrée en une liste de tokens avec l'aide de la fonction *token*. La liste de tokens est ensuite passée dans le transducteur afin d'extraire l'ensemble des valeurs du lexique. Si plusieurs valeurs sont trouvées pour un même mot de L , les multiples valeurs sont sélectionnées et on produit alors une entité multi-valuée. Cependant, si deux lexèmes différents se recoupent (ex : *aggravation transitoire des symptômes* et *symptômes cardiovasculaires*) seule les valeurs associées au plus grand lexème (ici, *aggravation transitoire des symptômes*) sont gardées. Les tokens restants (*cardiovasculaires*) sont remis en jeu (au cas où ils pourraient former un autre mot de L). Cette approche permet de sélectionner les plus grands lexèmes qui sont plus discriminants de par leur taille.

Exemple 5.3.1. Prenons comme exemple trois lexèmes du MeSH avec des classes arbitraires dans la table 5.4. À partir des exemples pour chaque classe, nous pouvons construire le transducteur Figure 5.2.

4. <https://github.com/OpenSextant/SolrTextTagger>

Classe	Lexème	Tokens
C1	Exacerbation transitoire des symptômes	{exacerb, transitoir, symptom}
C1	Aggravation transitoire des symptômes	{aggrav, transitoir, symptom}
C2	Aggravation passagère des symptômes	{aggrav, passager, symptom}

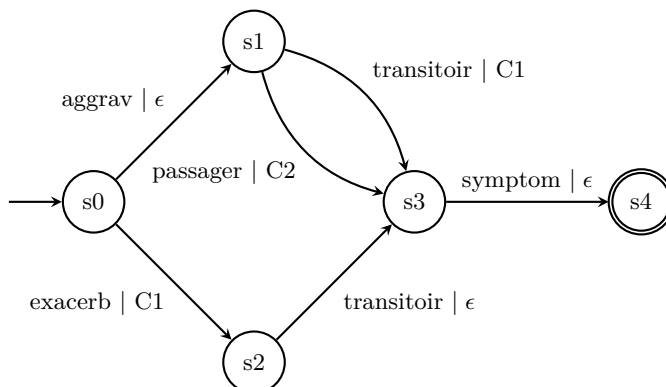
TABLE 5.4 – Liste de lexèmes, leur classe associée et la liste des tokens obtenus avec la fonction *token*

FIGURE 5.2 – Exemple d'un transducteur fini construit à partir de la table 5.4

5.3.3 Cascade de CRFs

Le corpus CAS permet de mettre en avant la complexité de certaines entités cliniques, dont la longueur rend difficile l'extraction précise des bornes de l'entité. En se basant sur le nombre significatif d'entités imbriquées (par exemple, les 1831 entités *SOSY* contiennent 1909 entités imbriquées), nous avons proposé dans [MRH⁺20] l'utilisation de cascades de Conditional Random Fields (CRFs) [LMP01, AHG07]. Une cascade de CRFs est une approche en apprentissage automatique qui consiste à utiliser plusieurs modèles CRF successivement en différents niveaux, chacun se concentrant sur une tâche spécifique, afin d'améliorer l'annotation. L'imbrication rend nécessaire cette approche, car les CRFs n'annotent qu'un seul label par token. De plus cela permet de tirer profit des labels extraits précédemment en les utilisant comme traits pour l'apprentissage des niveaux suivants. L'idée sous-jacente est de commencer par extraire des entités simples, puis d'utiliser ces annotations pour extraire des entités plus complexes composées de ces entités simples. Les entités de niveaux inférieurs peuvent être extraites à l'aide d'CRF, mais également par d'autres techniques telles que les lexiques ou les grammaires locales, comme expliqué dans l'introduction.

Dans l'article [MRH⁺20], uniquement des CRFs ont été employés. On propose de regrouper les entités qui partagent des similitudes sémantiques, morphologiques et en suivant les imbrications. Nous obtenons ainsi les niveaux suivants :

1. *Dose et Valeur*
2. *Durée et Fréquence*
3. *Date et Moment*
4. *Anatomie et Mode*
5. *Traitement et Examen*
6. *Substance*
7. *SOSY et Pathologie*

La table 5.5 donne un aperçu du fonctionnement de la cascade pour la phrase « La fonction rénale était normale et une rhabdomyolyse était éliminée, les CPK étant normales ». La première colonne montre la liste des tokens et les colonnes suivantes, la liste des étiquettes considérées pour chacun des niveaux correspondants au format Begin, Inside, Outside (BIO) [RM95]. La méthode proposée dans [MRH⁺20] a montrée une bonne précision mesurée à 0.839, avec un rappel plus faible de 0.613 pour une mesure F1 de 0.708.

DSL Les CRFs nécessitent une quantité conséquente de données annotées pour leur apprentissage. Quand il n'est pas possible de récupérer suffisamment de données, on utilise un Domain Specific Language (DSL) pour générer artificiellement des exemples. Un DSL est un langage de programmation spécifique à un domaine. Dans cette thèse, on utilise Chatette [Cha24] qui permet de construire une grammaire pour

	Niveau 1	Niveau 4	Niveau 5	Niveau 7
La	O	O	O	O
fonction	O	O	B-examen	B-sosy
rénale	O	B-anatomie	I-examen	I-sosy
était	O	O	O	I-sosy
normale	B-valeur	O	O	I-sosy
et	O	O	O	O
une	O	O	O	O
rhabdomyolyse	O	O	O	B-sosy
était	O	O	O	O
éliminée	O	O	O	O
,	O	O	O	O
les	O	O	O	O
CPK	O	O	B-examen	B-sosy
étant	O	O	O	I-sosy
normales	B-valeur	O	O	I-sosy
.	O	O	O	O

TABLE 5.5 – Exemple des différents niveaux d’annotation pour la cascade de CRFs

générer facilement une grande variété d’exemples synthétiques représentant différentes variations des entités attendues. Cela permet d’avoir des exemples pour les entités annotées qui sont utilisées comme attribut du CRF et permet de construire également des jeux de tests recouvrant l’ensemble des entités. Les exemples de valeurs pour les entités associés à des lexiques sont automatiquement générés.

5.4 Enrichissement et contextualisation

Une *entité enrichie* est une entité *simple* à laquelle on a ajouté de l’information. Il peut s’agir de préciser le type, la valeur ou de faire le lien avec un individu dans une base de données. L’enrichissement est opéré généralement en deux étapes, la première où l’on identifie les entités simples et les marqueurs de contextes et la seconde où l’on identifie le lien entre les entités et les marqueurs. On identifie les catégories d’entités suivantes :

Une entité de référence est celle qui est choisie pour être enrichie. Elle correspond à une valeur d’instance dans une base de données (nom de personnes, titre d’un documents, date, etc.)

Un opérateur ajoute des précisions sur la valeur de l’entité. Les opérateurs sont des marqueurs de comparaisons « commence par », « contient », etc ou de négation.

Un marqueur de contexte est une entité qui porte l’information de contexte.

Définition 5.4.1 (Entité enrichie). Une entité enrichie est définie comme un ensemble d’entités mono-valuées associées à un opérateur tel que : E_e est une entité enrichie de la forme $\{(v, \mathcal{T}, op)\}$ où v est une unique valeur d’entité, $\mathcal{T}$ est un ensemble de types associé à la valeur et op est un opérateur de comparaison sur la valeur (par défaut =). Une entité multi-valuée $E = (\mathcal{V}, \mathcal{T}, m)$ se traduit par une entité enrichie $E_e = \{(v, m(v), =) \mid \forall v \in \mathcal{V}\}$.

Pour une même valeur, l’ensemble des types associés s’appliquent. L’ambiguïté n’est représenté que par l’ensemble de tuples dans l’entité enrichie. Ainsi, l’entité $E_e = \{(:alice, \{Person\}, =), (:bob, \{Person\}, =)\}$ possède une ambiguïté sur sa valeur (*:alice* ou *bob*). Il est aussi possible de représenter l’ambiguïté sur le type, par exemple, l’entité $E_e = \{(:paris, \{Person\}, =), (:paris, \{City\}, =)\}$ à une ambiguïté sur le type de *Paris* qui peut être une personne ou une ville.

Dans la phrase « écrit par Alice », *Alice* pourrait représenter une entité de référence. Elle aurait le type *Personne* et pour valeur, un (ou plusieurs) identifiant dans une base de données. Ici, « écrit » pourrait être le marqueur du contexte *Auteur*. L’enrichissement consisterait alors à ajouter le type correspondant au contexte (ici, *Auteur*) à l’entité de référence (ici, *Alice*). Cette approche itérative permet de découper le processus d’extraction de l’information. Dans le cas où plusieurs valeurs sont possibles pour l’entité

Alice (par exemple si plusieurs individus de la base partagent le même prénom), l'identification de type plus précis peut permettre de réduire l'ambiguïté.

Un autre facteur important dans les textes sont les conjonctions de coordinations, en particulier les conjonctions « ou » et « et ». Par exemple, dans la phrase « écrit par Alice et Bob », la conjonction de coordination « et » est utilisée pour indiquer que le marqueur du contexte *Auteur* s'applique à la fois à l'entité *Alice* et à l'entité *Bob*. On obtient alors deux entités distinctes ayant le type *Auteur*. Si on part de la phrase « écrit par Alice ou Bob », la conjonction « ou » est vu comme un marqueur d'ambiguïté. On obtient alors une seule entité regroupant les valeurs de l'entité *Alice* et de l'entité *Bob*.

Pour réaliser la contextualisation des entités, nous proposons deux approches :

- Une première approche basée sur des règles ;
- Une seconde approche basée sur l'idée de la cascade de CRFs présenté dans la section 5.3.3.

Utilisation de règles

En analysant les textes avec lesquels on travaille, on remarque que le contexte est généralement porté par un mot précis et principalement le verbe. Intuitivement, l'approche basique consiste alors à identifier le marqueur de contexte comme une entité. Pour cela on utilise les mêmes approches que présenté précédemment, en particulier l'extraction par lexique. Une entité contexte est associée au type *Contexte* et a pour valeur, l'ensemble de types qu'il représente. Une fois le marqueur identifié, il faut déterminer sa portée, c'est-à-dire les entités qui sont impactées par ce dernier. Pour ce faire, on utilise l'arbre de dépendance et on identifie les situations suivantes :

1. Il existe un lien direct, vers la droite entre le contexte et l'entité de référence (par exemple si l'entité est l'objet du verbe) ;
2. Il y a un verbe entre le contexte et l'entité de référence. Dans ce cas, le contexte est le sujet et l'objet est l'entité de référence. Par exemple, dans la phrase « l'auteur est Alice ».
3. L'entité de référence est liée par une conjonction de coordination à un mot qui est lui-même dans la portée du contexte.
4. Si une entité de référence se trouve dans la portée d'un contexte, elle hérite aussi de tous les contextes conjoints.

On remarque avec les règles 3 et 4 que la portée n'est pas toujours triviale et on se rend compte que la détection de la portée est plus compliquée dès lors qu'on ajoute de multiples contextes, des conjonctions de coordinations. De plus, cet ensemble de règles ne couvre pas toutes les structures possibles. Par exemple, dans la phrase « écrit par Alice et signé ou relus par Bob avant le 10 mars », on s'attend à avoir les entités $E_1 = (:alice, \{Auteur\})$, $E_2 = (:bob, \{Signataire, Relecteur\})$ et $E_3 = (10/03, \{DateSignature, DateRelecture\})$. Cependant, la règle 3 ne permet pas d'associer les contextes *DateSignature* et *DateRelecture* à l'entité *date* à cause de sa distance avec le marqueur de contexte « signé ou relus ».

Cascade de CRF

Une autre approche s'inspire de la cascade de CRFs. Nous avons vu dans la section 5.3.3, que les CRFs peuvent être utilisés *successivement* en utilisant comme attribut des labels d'entités des étapes précédentes pour extraire des entités plus larges. Un CRF est utilisé pour annoter une portion du texte. L'identification de la portée d'un contexte correspond aussi à détecter la portion du texte impacté. L'idée est donc de construire un CRF utilisant comme attributs les entités présentes dans le texte pour identifier la portée des contextes. Une entité de référence est ensuite associée à un contexte si elle se situe dans la séquence annotée. Comme le contexte est majoritairement porté par un verbe, on ajoute comme attribut le lemme du premier verbe parent dans l'arbre de dépendance. Comme il n'est pas possible d'annoter un mot avec différent label quand on utilise un CRF, il est nécessaire de traiter chaque contexte de façon indépendante. Cependant, cela ne permet pas de traiter le cas de la conjonction « ou ».

5.5 Classification de cas cliniques

Dans la littérature, la classification de documents est souvent traitée comme un problème de classification traditionnelle. Étant donné des individus (documents) représentés sous forme vectorielle, l'objectif est de prédire la classe (un label) de chaque individu à partir d'un modèle qui a été entraîné sur un corpus d'exemples. Lorsque l'on traite des documents textuels, la représentation vectorielle ne paraît pas évidente. La méthode souvent retenue est celle du sac de mots, l'idée est de sélectionner un ensemble de mots de taille n qui représenteront les dimensions de notre espace vectoriel. Ainsi, chaque document est représenté par un vecteur de booléen (représentant l'apparition du mot ou non dans le texte) ou un vecteur d'entier représentant le nombre d'occurrences des termes dans le texte.

Pour travailler dans un espace à dimensions réduites, il est essentiel de sélectionner les mots les plus représentatifs de chaque classe. Pour ce faire, on trie chaque terme selon un critère spécifique, puis on choisit les m éléments ayant le score le plus élevé. Une mesure simple est le Term Frequency-Inverse Document Frequency (TF-IDF) [SJ72]. Intuitivement, cette mesure est un ratio entre la fréquence d'apparition du terme dans une classe donnée et sa fréquence dans l'ensemble du corpus qui repose sur la loi de Zipf [Zip35] (un terme a plus de chance d'être révélateur d'une classe s'il y est souvent présent ; au contraire, si un terme est trop fréquent dans le corpus, il n'est pas assez discriminant). Dans [WWM⁺17], les auteurs cherchent à extraire le domaine médical auquel appartient un ensemble de documents. Ils comparent une approche utilisant des réseaux de neurones et une approche utilisant des sacs de mots pondérés par TF-IDF avec un classifieur Support Vector Machine (SVM). Ils montrent des résultats similaires pour les deux approches avec un gain en explicabilité pour l'approche sac de mots. D'autres mesures de pondération existent comme l'index de Jaccard [Jac01]. [MT04] proposent une autre méthode de pondération des termes basée sur l'algorithme de PageRank [BP98] dans un graphe représentant les interactions entre les mots.

Cependant, dans un cadre où les données nécessaires à l'apprentissage de tels modèles ne sont pas toujours accessibles, des alternatives sont nécessaires. Les différents outils d'extraction d'informations décrits dans ce chapitre répondent à cette problématique et sont aussi efficaces pour la classification de cas cliniques. Cette section présente un système développé pour l'identification du profil clinique du patient [HMB21] lors d'une participation à DEFT 2021⁵ [GGI21] qui a obtenu les meilleurs résultats pour la tâche n°1. La tâche consistait à identifier les types de maladies d'un patient décrit dans un cas clinique et correspondant aux entrées génériques du chapitre C du MeSH, un thésaurus bilingue anglais-français du domaine médical fourni par la NLM. Chaque document décrivant le cas clinique d'un patient, l'extraction des classes du MeSH peut être vue comme un problème de classification (pour chaque classe du MeSH on cherche à savoir si la classe est représentée dans le cas ou non). Bien que certaines des remarques discutées dans cette section sont spécifiques à la tâche de DEFT et à la particularité des documents de cas cliniques, cette technique est applicable à d'autres types de documents médicaux, mais aussi d'autres domaines.

5.5.1 Système de classification basé sur l'extraction par lexiques

Le système, présenté dans la figure 5.3, utilise une méthode symbolique, basée sur l'extraction par lexique présentée section 5.3.2. Le lexique permet l'extraction des termes révélateurs de chaque classe et permet de remplacer un éventuel apprentissage d'un modèle TF-IDF. Une phase de post-traitement des annotations retournées par le transducteur est utilisée pour nettoyer les résultats obtenus. Elle se charge de gérer les négations (section 5.5.2) et de traiter l'ambiguïté engendrée par certains termes sur le genre.

Pour la construction du lexique, seule la catégorie C du MeSH (maladie) a été utilisée pour cette tâche. Elle contient 26 classes, qui correspondent aux types de maladie à identifier dans les cas cliniques. La table 5.6 donne les identifiants et les noms de chaque classe (trois classes ne sont pas mentionnées dans le tableau, car elles n'étaient pas utilisées dans la classification). Chaque classe est structurée en une arborescence de descripteurs, eux-mêmes constitués de concepts auxquels sont associés des termes. Nous avons extrait pour chaque classe du chapitre C tous les termes associés à des concepts. Au total 40 052 termes ont été extraits. Il est à noter qu'un même terme peut être associé à plusieurs classes. Par exemple "diabète gestationnel" est associé aux classes C13, C18 et C19.

5. <https://deft.lisn.upsaclay.fr/2021/>

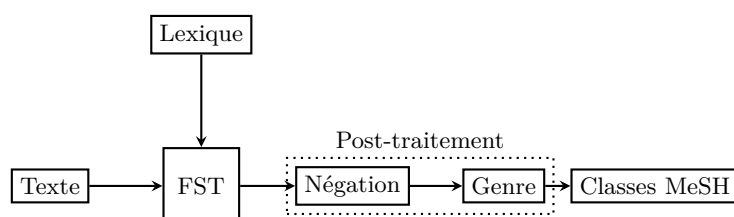


FIGURE 5.3 – Schéma du système d'identification du profil clinique du patient

C1 Infections bactériennes et mycoses	C13 Maladies de l'appareil urogénital féminin et complications de la grossesse
C2 Maladies virales	C14 Maladies cardiovasculaires
C3 Maladies parasitaires	C15 Hémopathies et maladies lymphatiques
C4 Tumeurs	C16 Malformations et maladies congénitales, héréditaires et néonatales
C5 Maladies ostéomusculaires	C17 Maladies de la peau et du tissu conjonctif
C6 Maladies de l'appareil digestif	C18 Maladies métaboliques et nutritionnelles
C7 Maladies du système stomatognathique	C19 Maladies endocriniennes
C8 Maladies de l'appareil respiratoire	C20 Maladies du système immunitaire
C9 Maladies oto-rhino-laryngologiques	C23 États, signes et symptômes pathologiques
C10 Maladies du système nerveux	C25 Troubles dus à des produits chimiques
C11 Maladies de l'œil	C26 Plaies et blessures
C12 Maladies urogénitales de l'homme	

TABLE 5.6 – Classes du chapitre C du MeSH

Bien que la ressource MeSH soit assez riche, le lexique peut être complété avec des lexèmes supplémentaires obtenus dans MedDRA grâce aux correspondances établies entre les classes dans l'UMLS. Au total, 13 classes de MedDRA ont été considérées, ajoutant 58 071 termes au lexique. Le MeSH contient un certains nombres de termes trop génériques (par exemple *maladie*) ou ambigus (par exemple *pris*). En suivant la loi de Zipf, ces termes peuvent être éliminés en utilisant un corpus de termes fréquents, extérieurs au domaine médical. Le corpus Wikipédia FR2008 [RED08] fournit une liste des termes avec leur fréquence dans l'ensemble du corpus. L'ensemble des lexèmes présent dans le corpus et qui sont suffisamment fréquents peuvent ainsi être supprimés pour nettoyer le lexique. Une évaluation a montré un meilleur rappel avec une fréquence maximum d'apparition de 5000 et une meilleure précision avec une fréquence de 100. La F-mesure maximale a été obtenue avec une fréquence de 1000.

5.5.2 Négation et incertitude

La négation est un phénomène linguistique complexe et très présent, elle est porteuse de sens comme l'information elle-même. Les cas cliniques font parfois référence à l'absence de symptômes, comme un résultat d'examen négatif. Dans le cas de la classification, ces termes sont reconnus par les lexiques, mais ne doivent pas être signalés comme marqueur d'une classe. La négation peut prendre différentes formes et être explicite ou implicite, mais est souvent portée par un mot (« non », « pas », ...), un groupe de mot (par exemple « ne [...] pas ») ou un affixe (« faire / dé-faire », « possible / im-possible », ...). De plus, elle n'est pas aussi stricte qu'en logique, la négation d'un fait n'est pas obligatoirement égale à son opposé dans le langage naturel. C'est le cas de la double négation comme « Il n'est pas impossible que [...] » qui ne signifie pas que c'est impossible.

[BM11] présente la complexité de la détection de la négation. Les auteurs étudient aussi la problématique de la portée de la négation. [MLD08] propose une approche basée sur l'apprentissage pour identifier la portée de la négation dans des textes médicaux. Dans le contexte de la classification, la tâche peut être simplifiée en raison de la recherche de marqueurs de classes. Il n'est alors pas nécessaire d'analyser

précisément la portée de la négation. Lorsqu’une classe est représentée, nous faisons l’hypothèse qu’il existe une quantité suffisante de marqueurs permettant de la reconnaître. En outre, l’élimination des faux négatifs est considérée comme ayant un impact négligeable sur le résultat final de la classification.

5.5.3 Évaluation

Les résultats de l’évaluation sont montrés dans la table 5.7. L’évaluation est réalisée à partir du corpus de tests utilisé dans la campagne DEFT 2021 et est réalisée sur différentes versions du système proposé. L’exécution ne contenant que le lexique MeSH, nous permet d’obtenir des bons résultats et supérieurs à la médiane de la tâche. La table 5.7 permet de mettre en évidence l’impact de la préparation des lexiques et des post-traitements sur les performances du système. Nous observons que le post-traitement lié à l’identification du genre du patient permet une amélioration importante des performances (+ 0.036 pour la F-mesure). Le filtrage par fréquence des mots trop génériques dans les lexiques permet une augmentation attendue de la précision. Un gain important est également observé avec l’utilisation des annotations du corpus d’entraînement pour enrichir le lexique. C’est cette version qui a obtenue la meilleure F-mesure à 0.814. Pour finir, nous avons également utilisé MedDRA pour augmenter le lexique de façon considérable (+58 071 termes). Cette augmentation permet d’augmenter un peu le rappel par rapport aux deux versions précédentes, mais fait chuter la précision à 0.679.

Les classes pour lesquelles nous avons obtenu les moins bons résultats sont les classes *blessures* (F-mesure 0.49 à 0.55), *chimiques* (F-mesure 0.36 à 0.54) et *virales* (1 classe sur 4 a été identifiée dans le corpus de test).

Configuration	Précision	Rappel	F1
MeSH	0.725	0.739	0.732
MeSH + négation	0.739	0.738	0.738
MeSH + genre	0.739	0.798	0.768
MeSH + négation + genre	0.816	0.738	0.775
MeSH + négation + genre + fréquence	0.873	0.713	0.785
MeSH + négation + genre + fréquence + annotation	0.888	0.750	0.814
MeSH + négation + genre + fréquence + annotation + MedDRA	0.686	0.769	0.725

TABLE 5.7 – Résultat d’expérimentations supplémentaires sur l’impact des post-traitements.

Une analyse d’erreur rapide nous a permis de mettre en évidence que la plupart des erreurs étaient dues à l’absence de certains termes dans le lexique (e.g. *kystes biliaires hépatiques*), ou au fait que certains termes présents ne sont pas reliés à la classe identifiée manuellement (e.g. *fistule* et *fistule cutanée* sont contenus dans la classe *etatsosy* et *peau* pour le deuxième, mais pas dans la classe *tumeur*, contrairement à ce qui a été annoté manuellement). Nous observons aussi quelques cas qui nécessitent un raisonnement, par exemple *violente chute* qui peut induire des blessures et donc être associée à la classe **C26**.

5.6 Interrogation de base de données en langue naturelle

Les systèmes d’interrogation de bases de données graphes visant les *experts du domaine* plutôt que les experts dans des bases de données, suscitent un intérêt grandissant, car ce sont ces utilisateurs auxquels profite le plus l’information et qui sont les plus aptes à la transformer en connaissance. Il est donc crucial de rendre ces systèmes plus conviviaux et accessibles aux utilisateurs sans compétences techniques avancées. Malgré la puissance des langages d’interrogation comme SQL, SPARQL ou encore Cypher, leur utilisation nécessite une compréhension de la structure des données et de leurs interactions. Afin d’améliorer l’accessibilité à l’information sauvegardée dans ces Systèmes de Gestion de Base de Données (SGBDs), la recherche sur les Interface en Langage Naturel (ILN) a suscité beaucoup d’intérêt ces dernières années [ZYZC18, Wan19, XMW23, VDE23, VCE⁺23]. L’intérêt fondamental des interfaces en langage naturel réside dans la capacité à permettre aux utilisateurs de se concentrer sur la signification de leurs requêtes, plutôt que sur les mécanismes sous-jacents de recherche d’information.

Certains travaux se concentrent sur l’augmentation du pouvoir d’expression des requêtes, tandis que d’autres se concentrent sur l’indépendance par rapport au domaine. Par exemple, dans [ZYZC18], les

auteurs proposent d'utiliser des modèles binaires plutôt que des analyses sémantiques pour mieux comprendre les requêtes complexes, tandis que [Wan19] propose un ILN inter-domaines basé sur un étiquetage générique des questions proposées. Plusieurs études [ZHW⁺14, FZE14, AKM15] s'intéressent aux systèmes de questions-réponses, ou Question Answering (QA), où les utilisateurs posent des questions en langage naturel sur une base de données RDF. Les requêtes d'agrégation sont examinées dans [HDYY18] : les auteurs proposent une méthode pour identifier automatiquement l'agrégation et la transformer en une instruction d'agrégation SPARQL. Les méthodes utilisées varient aussi considérablement. Dans [VDE23], les auteurs cherchent à construire des requêtes pour l'analyse de données à partir d'une requête formulée en langage naturel. [VCE⁺23] propose un système pour l'exploration de donnée pour des graphes de propriétés en mode conversationnel. Dans [ZNL18, ZYZC18, SAS19], les auteurs fondent leur approche sur des techniques de TAL avec l'extraction d'entités et des grammaires, tandis que dans [UWB⁺18, Wan19], ils utilisent des réseaux neuronaux.

Dans ce contexte, et en s'appuyant sur les méthodes de reconnaissances d'informations textuelles présentées dans les sections précédentes, on s'intéresse aux requêtes factuelles sur les instances d'une *classe* Resource Description Framework (RDF) avec deux objectifs principaux : (i) la facilité d'appliquer le système à divers domaines et (ii) la création de concepts d'interrogation plus complexes. L'objectif est de concevoir un système d'interrogation permettant aux utilisateurs d'effectuer des recherches par facettes. La recherche par facette consiste à filtrer une collection de données basée sur une unique collection d'individus par l'application de différents critères. Il s'agit d'une requête conjonctive sur une unique classe appelée *classe cible*.

Dans cette section, on présente la transformation d'une requête exprimée en langage naturel (nommée NL-query) en une requête de base de données (dénommée DB-query). Cette proposition, faite dans [AHH20], repose sur le formalisme établi dans la section 2.2.2, page 11 basé sur la logique du premier ordre (FOL). En conséquence, ces requêtes peuvent être facilement traduites vers une modélisation graphe ou relationnelle en utilisant les langages de requêtes tels que Structured Query Language (SQL), Simple Protocol and RDF Query Language (SPARQL), Cypher, etc. On définit une formule logique ϕ ayant pour atomes :

1. $P(t_1, \dots, t_n)$ où P est un symbole de prédicat d'arité n et $t_1, \dots, t_n$ sont des termes pouvant être des constantes ou des variables ;
2. $\top$ (Vrai) / $\perp$ (Faux) ;
3. $(t \text{ op } \alpha)$ où t est un terme, α est un terme ou un littéral et op est un opérateur de comparaison.

On rappelle qu'un fait est un atome $P(t_1, \dots, t_n)$ où l'ensemble des termes $t_1, \dots, t_n$ sont des constantes. Par exemple, en RDF, le fait $Book(Anatomy)$ signifie que *Anatomy* est une instance de la classe *Book*. De manière similaire, $writtenBy(Anatomy, Bob)$, exprime que *Anatomy* possède la valeur *Bob* pour la propriété *writtenBy*. Le schéma de la base de données est représenté par un ensemble de prédicats $\mathcal{S}$ et l'instance de la base de données est représenté par l'ensemble de faits $\mathcal{D}$

Définition 5.6.1 (NL-query). Une NL-query est une requête exprimée en langage naturel qui suit le format suivant : « Trouver des livres qui ... » (en établissant **Book** comme *classe cible*), « Quels médecins ... » (en utilisant **Doctor** comme *classe cible*), etc. Une requête valide sélectionne exclusivement des instances de *classe cible* (c'est-à-dire des nœuds du type donné) par l'intermédiaire de propriétés dont le domaine ou la portée est la *classe cible* (c.-à-d. les arcs sortant ou entrant).

Par exemple, en supposant que *Book* est une *classe cible*, une requête demandant des instances de « livres édités par le docteur Alice sur la cardiologie parut après 2018 » est une requête acceptable. Cependant, une requête demandant des exemples de « livres édités par le docteur Alice qui est cardiologue » n'est pas acceptable, car « est un cardiologue » n'est pas une propriété (ou une arête) de la *classe cible* sélectionnée. Si l'utilisateur souhaite identifier des médecins ayant rédigé un livre, il doit d'abord modifier la spécification de sa classe cible. Plus simplement, on ne considère que des requêtes simples qui sont traduites en requêtes conjonctives de base de données identifiant des instances d'une classe particulière, même si elles fournissent également des informations sur les propriétés de ces instances.

Définition 5.6.2 (DB-query). Une DB-query q est une requête conjonctive sur un schéma donné $\mathcal{S}$ de la forme $R_0(u_0) \leftarrow \phi$ où $\phi = R_1(u_1) \dots R_n(u_n), comp_1, \dots, comp_m$ où $n \geq 0$, R_i sont des symboles de prédicats avec $0 \leq i \leq n$, u_i est le tuple de termes de R_i de longueur égale à l'arité de R_i et $comp_j$ sont des formules de comparaison qui incluent des variables trouvées dans au moins un tuple de u_i avec

$0 \leq j \leq m$. $head(q)$ (respectivement $body(q)$) représente la partie gauche de la règle, dénommée tête (respectivement la partie droite, dénommée corps) de q . $\mathbb{I}_q$ représente l'ensemble de réponses de q .

Les réponses de q sont des instanciations du tuple u_0 où l'ensemble des termes $t \in u_0$ sont des constantes. C-à-d. qu'il existe, pour chaque instanciación $I \in \mathbb{I}_q$, une surjection h_I de u_0 vers I (qui associe des variables à des constantes et une constante à elle-même) tel que : $\{R_1(h_t(u_1)), \dots, R_n(h_t(u_n))\} \subseteq \mathcal{D}$, la conjonction de tous les $h_I(comp_j)$ est évaluée à vrai (selon la sémantique habituelle des opérateurs op) et $h_I(u_0) = I$.

Exemple 5.6.1. La requête NLQ_{run} en langage naturel suivante sera utilisée à titre d'exemple :

Livres intitulés « Principes de médecine », écrits par Alice et Bob et dont le prix est inférieur à 30 euros.

La figure 5.4 montre l'arbre de dépendance pour une version simplifiée de NLQ_{run} . Dans cette requête, « Livre » représente la *classe cible* : *Book*. Cette requête se traduit dans la DB-query Q_{run} suivante où *:alice* et *:bob* sont respectivement les identifiants en base d'Alice et Bob :

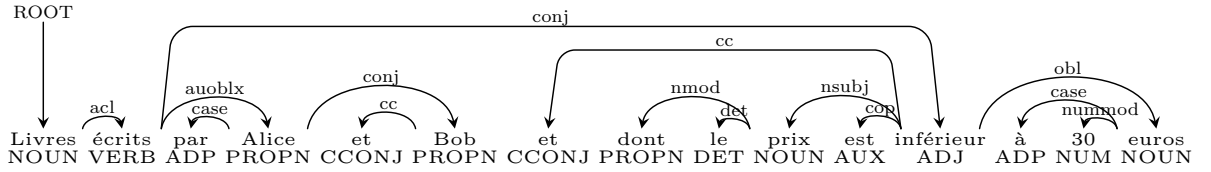
$$Q(x) \leftarrow Book(x), hasTitle(x, y_1), writtenBy(x, y_2), Person(y_2), writtenBy(x, y_3), Person(y_3), \\ hasPrice(x, y_4), (y_1 = \text{"Principes de médecine"}), (y_2 = :alice), (y_3 = :bob), (y_4 < 30)$$


FIGURE 5.4 – Arbre de dépendance d'une version simplifiée de NLQ_{run} obtenu avec SpaCy

5.6.1 Extraction d'entités

En s'appuyant sur les sections précédentes, l'ensemble des classes présentes dans la base de données peuvent être représentées par un lexique. Dans l'exemple de NLQ_{run} , on suppose que l'on a un lexique de *Personne* qui a pour valeur les identifiants des individus associés au nom, prénom, etc. de chaque personne. Ainsi, l'ensemble des individus de la base peuvent être reconnus et associés directement à la classe (correspondant au type de l'entité). Un lexique des classes cibles disponibles et un lexique des champs/propriétés dans la base sont aussi construits. Ces lexiques nécessitent généralement une intervention humaine pour compléter la liste des synonymes pour ne pas uniquement dépendre du nom de la classe. Pour identifier les dates et les nombres (ex : 30 euros), on utilise un ensemble de grammaire locale. Les *textes livres*, comme le titre dans NLQ_{run} , sont extraits à l'aide d'un CRF appris sur un jeu de donnée généré en utilisant un DSL. L'exemple 5.6.2 montre les entités extraites pour la requête NLQ_{run} .

Exemple 5.6.2. Ci-dessous nous présentons la requête NLQ_{run} avec les entités extraites dans des rectangles. Chaque entité est numérotée. Les entités E_1 , E_3 , E_4 et E_5 sont extraite par l'intermédiaire de lexiques. L'entité E_6 est obtenue en utilisant une grammaire locale et E_2 est extraite à l'aide d'un CRF.

Livres 1 intitulés « Principes de médecine » 2 écrits par Alice 3 et Bob 4 et dont le prix 5
est inférieur à 6 30 euros 7.

On a donc les entités suivantes :

$E_1 = (Book, Class)$	$E_5 = (hasPrice, Attribut)$
$E_2 = (Principes de médecine, Text)$	$E_6 = (<, Operator)$
$E_3 = (: alice, Person)$	$E_7 = (30, Number)$
$E_4 = (: bob, Person)$	

Pour des raisons de lecture, on ne représente pas la notation des ensembles de valeurs et de types. Ici, chaque entité n'a qu'une seule valeur et chaque valeur est associée à un unique type. Dans la pratique,

il serait possible d'avoir plusieurs valeurs ou types à cause de l'ambiguïté. Ainsi, si p_1 et p_2 sont deux personnes dans la base qui se prénomment Alice, alors on aurait :

$$E_3 = (\{p_1, p_2\}, \{Personne\}, \langle p_1 \mapsto \{Personne\}, p_2 \mapsto \{Personne\} \rangle)$$

L'extraction des associations *classe cible* – attributs (c.-à-d. des arêtes dans le graphe) se fait par contextualisation. Comme discuté dans la section 5.4, cette dernière peut s'opérer par extraction de marqueur de contexte (dans NLQ_{run} , on peut identifier les marqueurs « intitulés » et « écrits par ») ou en utilisant un CRF. Ici, on utilise des CRFs pour les attributs génériques et l'ensemble de règles donné dans la section 5.4 pour identifier le lien entre une entité correspondant à un attribut (ex : E_5 « prix ») et l'entité correspondant à la valeur de l'attribut (ex : E_6 « 30 euros »). Dans NLQ_{run} , on remarque la présence d'un opérateur « inférieur à ». Un opérateur est un modificateur (comme les contextes) qui vient enrichir la valeur (contrairement au type pour les contextes) d'une entité. Les opérateurs sont identifiés comme les contextes, on utilise un lexique pour le marqueur et la détection des liens avec une entité est effectuée par l'ensemble des règles sur l'arbre de dépendances (figure 5.4) utilisées pour la contextualisation et listée dans la section 5.4. Il est possible d'avoir des opérateurs sans valeurs associées et qui peuvent être liés à un attribut de la *classe cible*. Par exemple, c'est le cas pour les opérateurs « est vide » et « est rempli ».

Exemple 5.6.3. Pour NLQ_{run} , l'association entre les entités E_5 « prix » et E_6 « 30 euros » est réalisée en deux temps. On commence par associer l'opérateur « inférieur à » à l'entité E_7 . On obtient alors l'annotation suivante :

Livres ₁ intitulés « Principes de médecine » ₂ qui ont été écrits par Alice ₃ et Bob ₄ et dont le prix ₅ est inférieur à 30 euros ₇.

Ensuite, on identifie que l'entité E_7 (< 30) est associée à l'attribut pointé par l'entité E_5 (*hasPrice*). Seuls les types associés au schéma de la base sont gardés ainsi, les types *Text*, *Number* et *Date* sont filtrés. On obtient alors l'entité enrichie $E_7 = (30, \{Number, hasPrice\}, <)$.

$$\begin{aligned} E_{e1} &= \{(Book, \{Class\}, =)\} & E_{e4} &= \{(:bob, \{Person, Author\}, =)\} \\ E_{e2} &= \{("Principes de médecine", \{Title\}, =)\} \\ E_{e3} &= \{(:alice, \{Person, Author\}, =)\} & E_{e5} &= \{(30, \{hasPrice\}, <)\} \end{aligned}$$

5.6.2 Construction de DB-query à partir d'entités enrichies

Après analyse de la NL-query et l'extraction de l'ensemble $\mathcal{E}$ des entités enrichies, la DB-query est construite à l'aide de la procédure *entitiesToQueries*. L'algorithme commence par considérer l'entité E_{e1} (ligne 3) qui a un rôle spécial puisqu'elle spécifie la *classe cible* qui est interrogée. La requête est initialisée avec un corps composé d'un unique atome sur le prédicat associé à la classe cible. La procédure *buildAtom* est responsable de la construction d'un atome pour la requête en cours de construction. Le symbole du prédicat à utiliser dans la construction d'un atome est trouvé via la valeur de l'entité E_{e1} . Dans l'exemple 5.6.2, *Book* est la valeur de E_{e1} et le nom du prédicat unaire associé. Dans notre cas, l'atome A_0 est *Book*(x). Remarquons que l'algorithme *entitiesToQueries* ne construit que des requêtes dont les réponses sont des identifiants de livres (c'est-à-dire des instanciations de x). Dans cet exemple, la requête initiale est donc : $q(x) \leftarrow Book(x)$.

Les lignes 9 à 13 de la procédure *entitiesToQueries* traite les entités E_e enrichies. On commence par construire pour chaque valeur de l'entité une variable avec la fonction *getNewVar* et on construit la formule de comparaison correspondant à l'opérateur entre la variable et la valeur à l'aide de la fonction *buildAtomOP*. Pour chaque type associé à la valeur ($E(value)$, ligne 10), on construit les atomes correspondants. La fonction *buildAtom* est utilisée pour construire les atomes qui seront ajoutés à *body*(q). Notez que *buildAtom* associe à chaque type, le symbole de prédicat correspondant et construit l'atome en prenant en compte l'emplacement de la valeur de l'entité (variable y) dans l'atome. Dans les prédicats binaires, x est toujours l'autre variable. Si, dans E_e , il y a plus d'une valeur, alors l'entité est ambiguë, soit parce qu'elle a été extraite par un lexique et que le lexème est ambiguë soit c'est une entité obtenue après prise en compte de la conjonction de coordination *or*. On matérialise l'ambiguïté par une union de requête. Pour ce faire, la procédure *entitiesToQueries* divise les valeurs de E_e en sous partie de la requête. Chaque $l \in Parts$ est un ensemble d'atomes à ajouter au corps de la requête en cours de construction.

Procédure entitiesToQueries($\mathcal{E}$)

```

1  $Q \leftarrow \emptyset$  ;
2 pour chaque entité enrichie  $E \in \mathcal{E}$  faire
3   si  $E$  is  $E_{e1}$  alors
4      $\{(value, type, op)\} \leftarrow E$  ;
5      $A_0 \leftarrow \text{buildAtom}(value, x)$  ; // Construit le premier atome du corps de la requête
6      $Q \leftarrow \{Q(x) \leftarrow A_0\}$  ;
7   sinon
8      $Parts \leftarrow \emptyset$  ; // Ensemble des listes d'atomes. Chaque  $l \in Parts$  est une liste d'atomes dont la
        // conjonction doit être ajoutée au corps de la requête.
9     pour chaque  $(value, op) \in E$  faire
10      pour chaque  $type \in E(value)$  faire
11         $y \leftarrow \text{getNewVar}$  ;
12         $part \leftarrow \{\text{buildAtomOP}(value, y, opval), \text{buildAtom}(type, y)\}$  ;
13         $Parts \leftarrow Parts \cup \{part\}$  ;
14       $Q' \leftarrow \emptyset$  ;
15      pour chaque requête  $q \in Q$  faire
16        pour chaque  $part \in Parts$  faire
17           $q' = \text{buildNewQuery}(q, part)$  ;
18           $Q' \leftarrow Q' \cup \{q'\}$  ;
19       $Q \leftarrow Q'$  ;
20 retourner  $Q$  ;

```

Dans la boucle à la ligne 15 on calcule le produit cartésien entre les requêtes déjà construites et les parties correspondant à chaque valeur de l'entité E_e . C'est-à-dire que si E_e possède deux valeurs possibles, on aura le double de requêtes qui seront construites.

D'après l'exemple 5.6.3, E_{e3} a deux types : *Person* et *Author*. Ainsi, sur la ligne 13 on ajoute l'ensemble $\{Person(y_2), \text{writtenBy}(x, y_2), (y_2 = :alice)\}$ à *Parts* et à la ligne 17 la requête construite est :

$$q(x) \leftarrow \text{Book}(x), \text{Person}(y_2), \text{writtenBy}(x, y_2), (y_2 = :alice)$$

Le résultat obtenu avec l'entité E_{e4} est similaire. Pour E_{e2} et E_{e5} , un seul prédicat est construit, car les valeurs n'ont qu'un seul type associé. Par exemple, E_{e2} donne lieu à l'ensemble $\{\text{hasTitle}(x, y_1), (y_1 = \text{"Principes de médecine"})\}$. Le traitement de l'entité E_{e5} construit la comparaison $y_4 < 30$. Après avoir pris en compte toutes les entités, la procédure entitiesToQueries renvoie l'ensemble Q qui forme la DB-query suivante :

$$Q(x) \leftarrow \text{Book}(x), \text{hasTitle}(x, y_1), \text{writtenBy}(x, y_2), \text{Person}(y_2), \text{writtenBy}(x, y_3), \text{Person}(y_3), \\ \text{hasPrice}(x, y_4), (y_1 = \text{"Principes de médecine"}), (y_2 = :alice), (y_3 = :bob), (y_4 < 30)$$

Cependant, considérons qu'il existe une ambiguïté sur l'entité E_{e3} avec les valeurs *alice1* et *alice2* tel que $E_{e3} = \{(:alice1, \{Person, Author\}, =), (:alice2, \{Person, Author\}, =)\}$. La procédure entitiesToQueries (lignes 9 à 13) produit deux sous requêtes à partir de la même entité, à savoir : (i) $part_1 = \{Person(y_2), \text{writtenBy}(x, y_2), (y_2 = :alice1)\}$ et (ii) $part_2 = \{Person(y_2), \text{writtenBy}(x, y_2), (y_2 = :alice2)\}$. Ensuite, à la ligne 15, chaque sous requête est considérée séparément et la requête Q est remplacée par deux nouvelles requêtes. À la fin de la procédure, Q est une requête DB-query composée de deux règles :

$$Q(x) \leftarrow \text{Book}(x), \text{hasTitle}(x, y_1), \text{writtenBy}(x, y_2), \text{Person}(y_2), \text{writtenBy}(x, y_3), \text{Person}(y_3), \\ \text{hasPrice}(x, y_4), (y_1 = \text{"Principes de médecine"}), (y_2 = :alice1), (y_3 = :bob), (y_4 < 30) \\ Q(x) \leftarrow \text{Book}(x), \text{hasTitle}(x, y_1), \text{writtenBy}(x, y_2), \text{Person}(y_2), \text{writtenBy}(x, y_3), \text{Person}(y_3), \\ \text{hasPrice}(x, y_4), (y_1 = \text{"Principes de médecine"}), (y_2 = :alice2), (y_3 = :bob), (y_4 < 30)$$

Enfin, considérons la NL-query *livres édités ou écrits par Alice* comprenant l'entité enrichie $E_{e2} = \{(\text{:alice}, \{Person, Author\}, =), (\text{:alice}, \{Person, Editor\}, =)\}$. Les ensembles construits à la ligne 13 sont : (i) $part_1 = \{Person(y), writtenBy(x, y), (y = \text{:alice})\}$ et (ii) $part_2 = \{Person(y), editedBy(x, y), (y = \text{:alice})\}$. On obtient alors la DB-query suivante :

$$Q(x) \leftarrow Book(x), Person(y), writtenBy(x, y), (y = \text{:alice})$$

$$Q(x) \leftarrow Book(x), Person(y), editedBy(x, y), (y = \text{:alice})$$

5.6.3 Évaluation

Le système est évalué sur un cas concret impliquant une instance de la Gestion Electronique de Documents (GED) d'Ennov. Dans cette configuration, les métadonnées des documents sont regroupées et indexées automatiquement dans des lexiques. Après l'extraction des différentes entités, une phase de post-traitement est ajoutée pour éliminer les incohérences éventuelles. Par exemple, cette phase vérifie si une entité identifiée comme un champ personne est bien associé à un identifiant valide de personne. Si ce n'est pas le cas, la condition est supprimée. Cette étape de post-traitement permet ainsi de rectifier d'éventuelles erreurs survenues lors de l'extraction et de la contextualisation des entités.

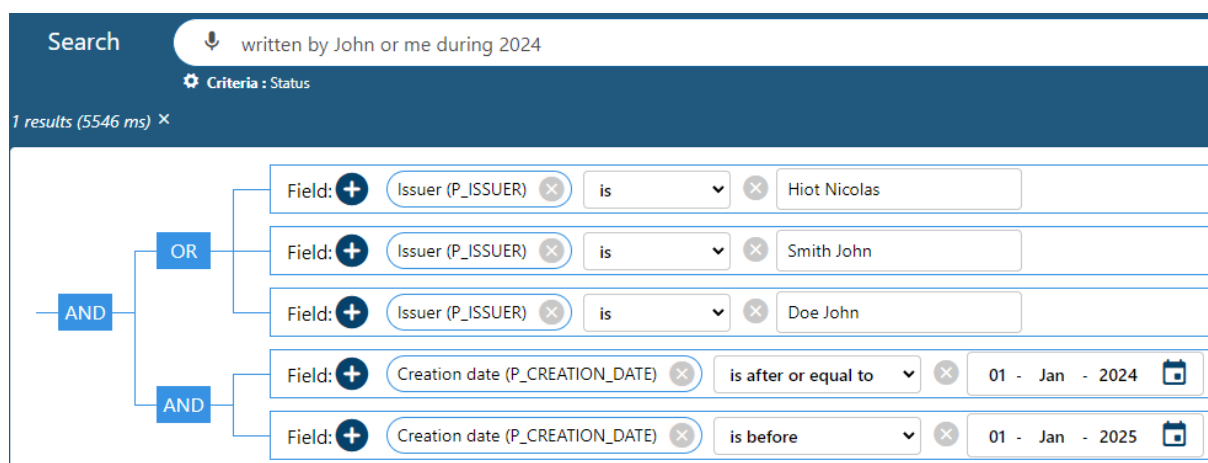


FIGURE 5.5 – Interface utilisateur et exemple de résultat pour la recherche en langage naturel

La figure 5.5 présente l'interface utilisateur ainsi qu'un exemple de requête en anglais : « written by John or me during 2024 ». Le résultat de la recherche n'est pas présent sur cette figure. Pour aider l'utilisateur à comprendre la requête qui a été formulée, celle-ci est représentée sous forme d'un arbre, tel que présenté sur la figure. Dans la requête, on observe par exemple l'utilisation de la conjonction « ou » entre *Nicolas* et *John* ainsi que l'ambiguïté du prénom *John*. De plus, la date exprimée par « during 2024 » est convertie en une plage de valeur, créant ainsi deux conditions : ≥ 2024 et < 2025 . Dans cette implémentation, nous utilisons la bibliothèque Duckling⁶ pour extraire les entités détectables par des grammaires locales telles que les dates, les nombres, les adresses e-mail, etc.

La table 5.8 présente les résultats de l'évaluation sur un corpus synthétique de test. Le corpus de test a été généré en utilisant le même DSL que celui utilisé pour les données d'entraînement des CRFs. La première section (de *object type* à *unit*) concerne les entités extraites à l'aide de lexiques. Les entités *time* sont extraites en utilisant des grammaires locales (via Duckling), tandis que les entités *title* sont extraites à l'aide d'un CRF. La section suivante (de *application date* à *signatories*) concerne les entités contextualisées, où les dates sont extraites à partir de l'entité *time* et les entités *issuer/signatories* sont obtenues à partir de l'entité *person*.

De manière générale, on observe que les lexiques donnent un très bon rappel (supérieur à 0.9), mais une précision plus faible, en particulier pour les entités *field* et *type*. Cela s'explique par le manque de filtrage sur les lexiques : dans cette version, nous ne filtrons pas les lexèmes trop génériques. Les entités *field* et *type* sont deux cas particuliers où les lexèmes associés à chaque valeur peuvent être longs. Pour

6. <https://github.com/facebook/duckling>

	Précision	Rappel	Mesure F1	Support
object type	0.600 764	0.990 551	0.747 919	635
person	0.654 574	0.734 513	0.692 244	1130
field	0.292 157	0.977 049	0.449 811	305
type	0.330 645	0.976 190	0.493 976	168
unit	0.638 254	0.987 138	0.775 253	311
status	0.575 368	0.970 543	0.722 447	645
time	0.941 496	0.916 618	0.928 890	1703
title	0.577 215	0.679 920	0.624 372	1006
application date	0.888 889	0.522 105	0.657 825	475
archive date	0.899 135	0.715 596	0.796 935	436
creation date	0.698 080	0.804 829	0.747 664	497
expiration date	0.814 655	0.675 000	0.738 281	280
issuer	0.724 954	0.597 598	0.655 144	666
signatories	0.708 738	0.478 166	0.571 056	458
Total	0.667 495	0.787 558	0.685 844	8715

TABLE 5.8 – Score obtenu pour l’extraction d’entités pour la recherche en langage naturel

l’entité *title*, il est difficile d’annoter correctement les bornes de l’entité en raison de la longueur de la séquence et de la grande variabilité dans les titres. L’entité *time* est la mieux annotée avec une mesure F1 de 0.93. En ce qui concerne la contextualisation, on remarque que les dates sont correctement classifiées à 70 %. En revanche, pour les entités *issuer* et *signatories*, le système a du mal à classifier correctement les personnes. Le CRF associé s’appuie sur les entités *person* extraites précédemment et dépend donc de leur qualité, qui est parmi les trois plus faibles, avec seulement 0.69 pour la mesure F1.

Une des problématiques majeures rencontrées dans ce type d’application est le temps de traitement. L’extraction est effectuée par un enchaînement séquentiel de composants exécuté avec l’écosystème Ray sur une machine équipée de 4 vCPU et 16 GB de mémoire primaire. Nous utilisons les modèles SpaCy en_core_web_md (version 3.7.1) pour l’anglais et fr_core_news_md (version 3.7.0) pour le français. Le temps moyen nécessaire pour l’analyse du texte, les vérifications de cohérence par rapport à la base pour le post-traitement et l’exécution de la recherche sur un core SolR est d’environ 2.5 s. Bien que ce temps d’exécution soit acceptable, il demeure élevé, en particulier en raison des vérifications qui génèrent plusieurs requêtes en base et sont traitées de manière séquentielle. En plus de traiter les vérifications de façon concurrente, il serait possible de considérer l’enchaînement des composants comme un graphe acyclique orienté ou Directed Acyclic Graph (DAG) en anglais plutôt que comme une séquence, afin de calculer certaines étapes en parallèle.

Une autre problématique de cette approche est que les différents composants du système sont entraînés individuellement. Il pourrait être intéressant d’envisager d’entraîner les CRFs à partir des annotations que les composants précédents sont capables d’extraire, plutôt que de les entraîner à partir de données correctement annotées.

5.6.4 Construction de graphique

Pour aller plus loin, nous proposons de permettre la création de graphiques à partir d’une requête en langage naturel. L’idée est de permettre à un utilisateur de construire simplement un graphique à partir des données présentes dans la base. Chaque graphique est construit à partir des champs d’une seule table. La figure 5.6 illustre le résultat de cette fonctionnalité dans l’interface utilisateur. Par exemple, dans cette illustration, nous recherchons l’évolution du nombre de cas de cosmétovigilance par année de réception de l’entreprise et par pays.

Pour cela, on commence par identifier le type de graphique (*chart type*), si on ne le trouve pas, on détermine le type automatiquement en fonction du nombre d’axes sélectionnés. Ensuite, on identifie les différents axes du graphique : *value axis* pour les ordonnées, *x axis* pour l’axe des abscisses et *y axis* pour le second axe utilisé pour construire plusieurs courbes dans l’exemple de la figure 5.6. Pour chaque axe, on identifie l’étiquette (*value label*, *x label* et *y label*) si elle est fournie, le champ de la table utilisé pour l’axe

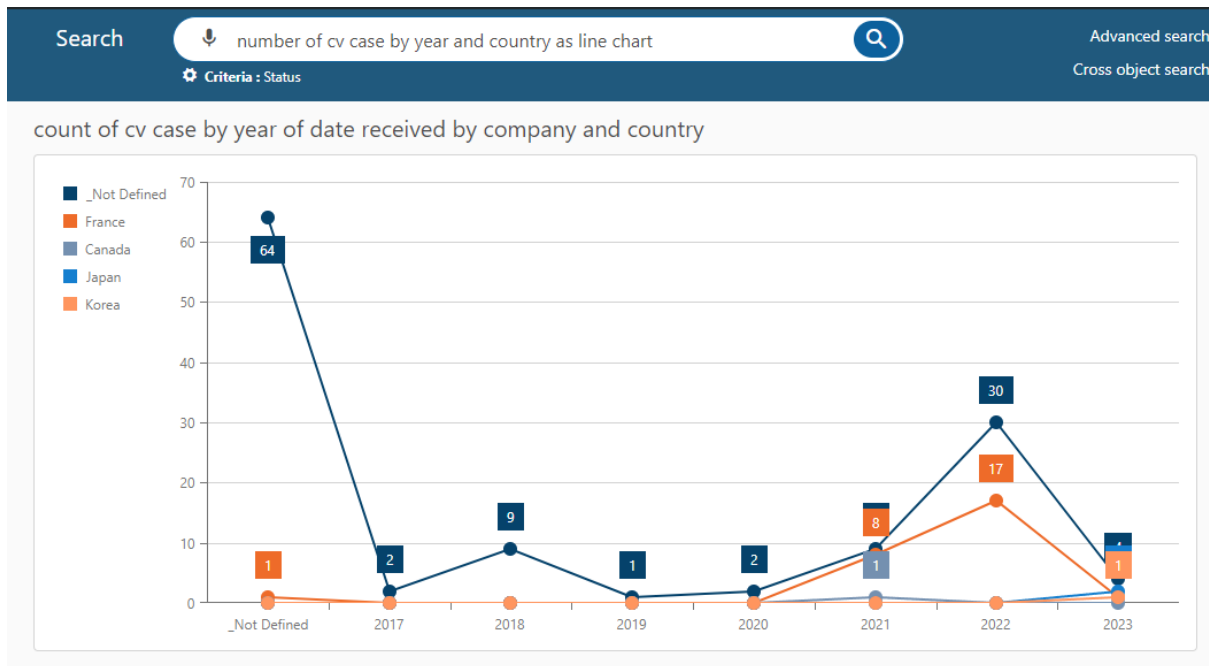


FIGURE 5.6 – Interface utilisateur et exemple de résultat pour la construction de graphique à partir d’une requête en langage naturel

(*field*) la fonction d’agrégation pour les valeurs (*axis aggregate*, moyenne, somme, etc) et le modificateur (*axis modifier*, par exemple, pour une date, on peut demander seulement le mois ou l’année). Pour finir, on cherche la table à utiliser, si elle n’est pas spécifiée dans la requête de l’utilisateur, on cherche la table la plus petite qui contient les champs identifiés. Ici on ne s’intéresse qu’à l’extraction des entités, mais un filtrage est effectué après l’extraction pour s’assurer que le graphique construit est cohérent, permettant de rejeter certaines ambiguïtés.

	Précision	Rappel	Mesure F1	Support
axis aggregate	0.910 714	0.980 769	0.944 444	156
axis modifier	0.432 500	0.994 253	0.602 787	174
chart type	0.936 747	1.000 000	0.967 341	311
table	0.691 983	0.716 157	0.703 863	229
field	0.292 157	0.977 049	0.449 811	305
value axis	0.940 461	0.990 560	0.964 860	1483
x axis	0.894 330	0.973 807	0.932 378	1069
y axis	0.822 669	0.813 743	0.818 182	553
value label	0.894 886	0.875 000	0.884 831	360
x label	0.915 541	1.000 000	0.955 908	271
y label	0.913 043	0.777 778	0.840 000	162
Total	0.785 912	0.918 128	0.824 037	5073

TABLE 5.9 – Score obtenu pour l’extraction d’entités pour la construction de graphiques à partir d’une requête en langage naturel

La table 5.9 présente les résultats obtenus. Le premier groupe d’entités (de *axis aggregate* à *tables*) est extrait à partir de lexiques et le second groupe (de *value axis* à *y label*) sont extraits à partir de CRFs. L’extraction des axes (*value axis*, *x axis*, *y axis*) se repose sur les entités : *axis aggregate*, *axis modifier*, *chart type*, *table* et *fields*. Les étiquettes des axes (*value label*, *x label*, *y label*) se basent sur l’extraction des entités : *value axis*, *x axis* et *y axis*.

On identifie, pour les lexiques *axis modifier* et *field*, une problématique similaire à celle évoquée pour la recherche en langage naturel où les lexèmes sont assez génériques. On obtient donc une bonne couverture

(rappel de 0.99) mais une précision assez faible (0.36). La contextualisation s'appuie ici sur un plus grand nombre d'entités, ce qui augmente la qualité générale (F1 de 0.69 pour la recherche et 0.90 pour la construction de graphique). De plus, les requêtes sont plus structurées avec un ordre clair des axes (*value* $\rightarrow x \rightarrow y$), n'ont pas de chevauchement possible (contrairement aux contextes de dates ou de personne) et sont composées d'un plus faible nombre d'entités.

5.7 Conclusion

Nous avons examiné deux applications directes des différentes méthodes utilisées dans cette thèse dans des contextes variés. Cela met en lumière l'efficacité de ces méthodes pour un vocabulaire contrôlé et avec peu de données d'apprentissage. Dans le cadre d'Ennov, il est essentiel de combiner des lexiques avec des méthodes d'apprentissage plus traditionnelles. En effet, même si deux clients peuvent manipuler les mêmes catégories d'objets dans leur base de données (telles que des personnes, des produits, etc.), les valeurs peuvent différer considérablement. L'utilisation de lexiques permet d'éviter d'entraîner un modèle pour chaque base de données et évite également d'avoir besoin d'accéder aux données des clients, puisque les lexiques sont construits automatiquement et localement. Notre approche montre que les lexiques constituent une ressource riche, facilement mise à jour et permettent d'obtenir une bonne qualité pour l'extraction d'entités et la classification de documents. Nous avons également vu que le filtrage automatique de termes trop génériques fonctionne bien pour du vocabulaire spécifique.

En ce qui concerne la classification, nous avons constaté que le système proposé offre de bons résultats et surpasse les approches basées sur l'apprentissage automatique proposées lors de DEFT 2021. Contrairement aux méthodes d'apprentissage, notre approche basée sur les lexiques ne nécessite pas d'apprentissage initial (à l'exception du raffinement), ni de grands corpus d'entraînement. De plus, elle est facilement mise à jour (il est possible d'ajouter ou de supprimer des valeurs sans nécessiter un réapprentissage du modèle) et elle est performante (la classification de 108 documents s'exécute en environ une minute avec notre système). L'impact en ressources utilisées est réduit et un tel système peut aussi être facilement déployé sur des systèmes embarqués.

Cela met en perspective l'utilisation quasi systématique de l'apprentissage automatique, notamment des réseaux de neurones et des modèles de langue (LLMs), dans l'industrie, malgré les coûts élevés et l'impact environnemental de ces solutions. Selon [SGM19], l'entraînement d'un modèle de réseau de neurones profond « à l'état de l'art » pour effectuer de la traduction équivaut à l'impact de la durée de vie de cinq voitures. De plus, les durées d'entraînement pour ces modèles peuvent varier de quelques jours à plusieurs semaines. Il est donc important de continuer à considérer les approches symboliques, qui demeurent efficaces pour certaines tâches.

Le traitement de la négation, tel qu'abordé dans la section 5.5.2, constitue une version simplifiée de la problématique. Dans le cadre de futurs travaux, il serait envisageable de mettre en place une méthode sémantique visant à mieux identifier la portée des marqueurs de négation. La détection de la négation peut aussi être considérée comme le problème de la détection de la portée de la contextualisation et pourrait être réalisée par une cascade de CRFs. De plus, il pourrait être intéressant d'explorer la détection de négations plus complexes, comme dans la phrase « Elle n'a présenté des nausées que durant la nuit et aucun vomissement ». Par ailleurs, il serait également pertinent de pouvoir extraire la temporalité et la causalité au sein des textes médicaux, car ces éléments représentent des informations capitales pour la compréhension des cas cliniques.

Pour aller plus loin dans l'extraction d'information, nous avons proposé dans [SSM⁺22] d'extraire un ensemble de relations médicales couramment rencontrées dans la littérature. Cette approche repose sur un ensemble de règles appliquées sur l'arbre de dépendances et exploite le fait que la relation entre deux entités est entièrement déterminée par les types de ces entités. Malgré les limites discutées dans la section 5.4 pour la contextualisation des entités dans des phrases complexes, les résultats obtenus dans [SSM⁺22] sont encourageants. En effet, avec un ensemble restreint de sept règles simples, le système proposé atteint un rappel de 0.84, une précision de 0.95 et une mesure F1 de 0.89.

Les outils présentés dans ce chapitre ne constituent qu'une première étape dans le processus d'extraction d'informations. La structuration automatique des informations, telle que présentée dans le chapitre 4, représente une seconde étape cruciale permettant de stocker ces informations dans des bases de données. Leur stockage facilite l'interrogation, l'exploitation et le raisonnement efficace sur ces données afin d'extraire de la connaissance.

Chapitre 6

Conclusion et perspectives

Sommaire

6.1 Contributions	113
6.1.1 Maintenance et cohérence d’une base de données incomplète	113
6.1.2 Intégration de données textuelles	114
6.2 Perspectives	114
6.2.1 Maintien de la cohérence de graphes attribués	114
6.2.2 Uniformisation de données pour l’application de processus	115
6.2.3 Base de données floues	116

6.1 Contributions

Nous avons présenté dans ce manuscrit un ensemble de travaux pour la construction automatique de bases de données pour le domaine médical. En particulier, nos recherches se focalisent sur les bases de données graphes pouvant contenir des données incomplètes et s’articulent autour de deux axes principaux :

- La maintenance et le maintien de la cohérence d’une base de données incomplète par rapport à un ensemble de règles définies.
- L’extraction et la structuration automatique d’informations à partir de textes en vue de leur intégration dans une base de données.

Dans la suite de cette section, nous détaillerons les contributions majeures réalisées sur ces deux axes.

6.1.1 Maintenance et cohérence d’une base de données incomplète

Maintenir une base de données cohérente est essentiel pour faciliter l’analyse des données. Cette tâche devient particulièrement complexe lorsque la base de données peut contenir des données incomplètes, matérialisées par des valeurs nulles liées. Dans la première partie de ce manuscrit, nous avons examiné une politique de maintenance qui accorde la priorité aux mises à jour. Concrètement, au lieu de rejeter des mises à jour susceptibles de rendre les données incohérentes, nous cherchons à effectuer les modifications nécessaires pour que ces mises à jour soient applicables. À cette fin, nous avons proposé une extension des travaux de [CHL20] basée sur la sémantique de [Rei86] et qui ne limite pas le pouvoir d’expressivité des contraintes. Contrairement à l’approche en mémoire présentée dans [CHL20], nos travaux se concentrent sur la maintenance en relation avec un Système de Gestion de Base de Données (SGBD).

Notre extension comporte les contributions suivantes :

- Une modélisation d’un graphe attribué visant à optimiser les différentes opérations requises pour assurer la cohérence de la base de données, en particulier pour la recherche des atomes liés, une opération coûteuse dans une modélisation classique dans une base de données relationnelle.
- Une procédure de mise à jour incrémentale, en mode connecté, reposant sur un ensemble de requêtes.

L'évaluation présentée dans la section 3.6 (page 34) met en lumière l'évolution significative des performances de notre approche incrémentale par rapport à l'approche complète proposée dans [CHL20], malgré l'interaction avec un SGBD. Cette étude des performances souligne également les qualités de notre modèle de graphe, qui permet de réduire de manière significative le coût de la recherche des atomes liés en construisant un graphe similaire à RDF, où les individus sont représentés par des nœuds. Toutefois, elle met également en évidence les limitations du modèle de graphe, où les performances des requêtes dépendent fortement de la modélisation. Dans notre cas, la fragmentation des données en un grand nombre de nœuds et de relations rend la recherche d'homomorphismes difficile.

6.1.2 Intégration de données textuelles

Ce second axe de recherche concerne l'exploitation des informations contenues dans les textes. L'approche proposée repose sur des techniques d'extraction d'information, en particulier l'extraction d'entités. Le cœur de ces travaux réside dans la structuration automatique des informations. Dans ce contexte, ce manuscrit propose :

- Un méta-modèle permettant d'organiser les informations de manière cohérente ;
- Une méta-grammaire capable de reconnaître l'ensemble des schémas valides du méta-modèle ;
- Une procédure itérative de structuration de l'information ;
- Une preuve de concept avec l'implémentation de la procédure, avec pour stratégie la minimisation de la grammaire cible.

Une évaluation préliminaire a démontré que la procédure proposée répond à la problématique et produit des résultats prometteur tant pour la structuration que pour l'intégration de données. L'évaluation souligne également l'importance de cette étape dans la construction automatique d'une base de données à partir de données textuelles.

Dans un second temps, nous proposons différentes techniques pour l'extraction d'entités permettant de répondre aux problématiques d'accès restreint aux données, d'efficacité et d'explicabilité. En particulier, nous proposons une approche itérative hybride reposant sur :

- Une représentation multi-valuée des entités nommées, permettant de prendre en compte l'ambiguïté et l'incertitude ;
- L'utilisation de lexiques par la construction d'un Finite-State Transducer (FST) pour l'annotation efficace d'entités nommées ;
- La notion de contextualisation et de spécification basée sur des marqueurs, avec une approche basée sur des règles ainsi qu'une approche reposant sur une cascade de Conditional Random Fields (CRFs).

Nous avons également présenté deux cas d'application concrets reposant sur cette approche, l'un pour la classification de documents et l'autre pour la construction de Interface en Langage Naturel (ILN) permettant l'interrogation et l'analyse de données. Ces deux cas d'application ont permis de démontrer les performances de notre approche en obtenant des résultats satisfaisants sans recourir à un apprentissage automatique nécessitant de vastes corpus d'entraînement. Cette approche permet ainsi de travailler efficacement avec des données sensibles ou difficilement accessibles. Ces deux cas d'usage témoignent également de la flexibilité et de la simplicité d'adaptation du processus itératif d'extraction.

6.2 Perspectives

Les travaux présentés dans cette thèse offrent des résultats importants et ouvrent de nombreuses perspectives pour la suite. En particulier, nous nous intéressons à trois d'entre elles que nous souhaitons développer davantage.

6.2.1 Maintien de la cohérence de graphes attribués

Dans le chapitre 3, nous avons présenté un modèle de graphe attribué conçu pour répondre à la problématique de la mise à jour cohérente. Cette représentation de la base de données vise à optimiser les

interactions nécessaires au maintien de la cohérence, telles que la recherche des nœuds liés et la recherche d'homomorphismes pour le calcul du cœur. Cette approche repose sur un modèle qui ne correspond pas nécessairement à une représentation intuitive des données, ce qui rend l'interrogation du graphe difficile, contrairement à l'approche proposée avec le modèle relationnel (voir la section 3.6.1, page 35). À l'usage, il est envisageable qu'une autre représentation des données sous forme de graphe soit utilisée pour l'interrogation par un utilisateur.

Pour répondre à cette problématique, nous envisageons d'explorer et de comparer deux approches :

1. Les deux graphes existent simultanément et notre modélisation du graphe est considérée comme un index utilisé pour le maintien de la cohérence. Cette approche implique un coût de maintenance supplémentaire pour maintenir les deux représentations à jour. Elle s'inscrit dans la perspective d'utiliser des bases de données hybrides [HKJ⁺18] permettant d'exploiter efficacement le modèle relationnel et le modèle de graphe attribué de manière simultanée.
2. Seule notre représentation est utilisée pour la sauvegarde en base, mais l'ensemble des requêtes émises lors de l'interrogation est automatiquement traduit vers notre modélisation. Cette approche pourrait simplifier la maintenance, mais elle pourrait également entraîner des coûts supplémentaires lors de la traduction des requêtes, surtout si elles sont complexes [Suj01, AA20].

Dans la section 3.6 (page 34), nous avons examiné les coûts engendrés par notre modélisation pour la recherche d'homomorphismes. Une perspective de ces travaux consiste à proposer l'application de notre politique de mise à jour sur un graphe attribué plus intuitif pour améliorer la recherche des homomorphismes. Bien que cette approche ait été envisagée dans nos travaux, elle présente les défis suivants :

1. Dans un graphe quelconque, l'incomplétude peut se situer au niveau des propriétés d'un nœud ou d'une relation, mais peut également concerner les nœuds ou les relations eux-mêmes. Cette approche complexifie grandement la recherche des valeurs nulles et la détection des ensembles liés.
2. Dans notre approche, nous proposons une vision RDF où les individus sont les termes de nos faits logiques. En revanche, dans le cadre des graphes de propriété, les individus sont des nœuds, et il est donc nécessaire de définir quand deux nœuds sont équivalents pour le calcul du Core. Contrairement au modèle relationnel, les graphes n'ont pas de notion de dépendances fonctionnelles ni de clés. Bien que des travaux tentent de répondre à cette question, comme nous l'avons vu avec les Graph Entity Dependencies (GED) [FL19], cette question n'est pas encore résolue.

6.2.2 Uniformisation de données pour l'application de processus

Dans le chapitre 4 (section 4.7.3, page 87), nous avons examiné comment notre approche de structuration automatique de l'information pourrait être appliquée à l'intégration de données. En particulier, on s'intéresse à développer l'idée d'unification des données. En effet, on remarque de plus en plus le besoin de construire des méta-modèles afin de standardiser la représentation des données et de les rendre interopérables, facilitant ainsi leur intégration et leur utilisation cohérente à travers différentes applications et systèmes.

Malgré l'existence de modèles unifiés comme les graphes Resource Description Framework (RDF) et l'implémentation de l'Open Linked Data, leur adoption reste limitée. De plus, les ontologies mises en place présentent souvent des chevauchements et les liens d'équivalence ne sont pas toujours spécifiés, ce qui requiert une vigilance accrue pour garantir une interopérabilité optimale entre les différentes sources de données [RMB16]. Par ailleurs, un grand nombre de données proviennent généralement de divers outils utilisant des formats variés, certains étant plus structurés que d'autres. Les bases de données sont régulièrement fragmentées avec des schémas variés, souvent pour des raisons historiques. Il est courant que chaque service au sein d'une même entreprise enregistre ses données de manière indépendante et non uniforme afin de répondre à des besoins spécifiques. Cette diversité de formats complique l'interrogation, l'analyse et le traitement des données.

L'adoption d'une approche uniforme pour la gestion de ces données offre des avantages significatifs dans des domaines tels que le journalisme de données en facilitant la corrélation des données provenant de différentes sources et services. En s'appuyant sur les travaux présentés dans le chapitre 4, notamment le méta-modèle proposé, une perspective de recherche de chercher à utiliser ce modèle pour pouvoir agréger de manière dynamique et automatique un ensemble de ressources, ainsi que l'ensemble des entrées et sorties des différents processus en place. Cette approche permettrait de rendre un processus appliqué à des données d'une ressource également applicable aux données d'autres ressources. Elle pourrait également

faciliter l'enchaînement de divers processus et leur application sur l'ensemble des ressources disponibles de manière simultanée.

Contrairement à l'approche proposée dans le chapitre 4, le méta-modèle servirait de représentation intermédiaire de l'ensemble des ressources. Le défi de cette approche réside dans la capacité à identifier les correspondances entre les différentes ressources. Une approche envisagée serait d'utiliser des mécanismes similaires à ceux présentés dans la structuration automatique pour l'identification des regroupements équivalents. La différence majeure est que l'on ne cherche pas à construire une base unifiée, mais à identifier les transformations nécessaires pour convertir dynamiquement une ressource vers un autre format. Plutôt que de reconstruire l'ensemble des ressources, l'idée est de fournir une couche supplémentaire d'interopérabilité, permettant ainsi une intégration dynamique et une utilisation cohérente des données provenant de différentes sources et formats.

6.2.3 Base de données floues

Dans le chapitre 4, nous avons examiné la structuration automatique d'informations basée sur l'extraction d'entités nommées, mais qui ne repose pas sur les valeurs spécifiques associées à ces entités. Dans le chapitre 5, nous avons proposé une représentation multi-valuée des entités nommées pour capturer l'ambiguïté inhérente aux textes. Une partie de cette ambiguïté peut être résolue par le contexte : par exemple, pour le mot « Paris », qui peut faire référence à une personne ou à la ville, si la valeur est détectée comme une adresse, alors la valeur associée à la personne peut être supprimée. Cependant, toutes les ambiguïtés ne peuvent pas être résolues de cette manière et l'insertion d'informations structurées dans une base de données implique nécessairement la résolution préalable de ces ambiguïtés.

Une perspective envisagée consiste à enregistrer les données sans résoudre leur ambiguïté, en s'appuyant sur les principes de la logique floue. Dans cette approche, chaque valeur serait associée à une valeur de vérité comprise entre 0 et 1. La valeur de vérité n'est pas nécessairement déterminée par le nombre de valeurs ambiguës, mais pourrait également refléter un certain niveau de confiance dans les données. L'idée sous-jacente est que l'ambiguïté pourrait être résolue progressivement au fil des mises à jour : en étant contraintes par un ensemble de règles, ces dernières pourraient contribuer à clarifier l'ambiguïté. De même, l'ambiguïté pourrait être résolue au moment de l'interrogation en croisant des résultats. Cette valeur de vérité pourrait ensuite être utilisée pour filtrer ou organiser les résultats.

Exemple 6.2.1. Par exemple, supposons qu'un événement ait pour adresse *Paris, France* et *Orléans, France*. Interroger la base pour obtenir l'adresse exacte pourrait renvoyer deux solutions possibles (*Paris, France* et *Orléans, France*), chacune avec une valeur de vérité de 0.5. En revanche, si l'utilisateur demande uniquement le pays, la réponse peut être unique (*France*) avec une valeur de vérité de 1 (100 % des entrées dans la base ont la valeur *France*).

L'introduction de contraintes supplémentaires lors de la requête, telles que la ville, pourrait conduire à une réponse unique, mais la valeur de vérité est plus incertaine : elle pourrait être de 0.5 si l'on prend en compte l'ensemble de la base de données ou de 1 si l'on ne considère que l'ensemble des résultats de la requête. Cette deuxième approche est justifiable si l'on suppose que les deux réponses sont vraies, mais pas simultanément. Ainsi, demander l'adresse en précisant la ville reviendrait à préciser le contexte d'interrogation. Ce dernier point révèle l'une des difficultés de la tâche, à savoir le calcul de cette valeur de vérité et pourrait nécessiter l'ajout de la notion de contexte au moment de l'interrogation.

Cette idée présente des similitudes avec les solutions proposées dans le modèle relationnel pour la gestion des données temporelles [KM12]. Un parallèle peut également être établi avec les travaux sur les bases de données contextuelles [SG02, Sta03]. Ces travaux s'attardent sur la représentation de données contextuelles en introduisant un modèle multidimensionnel qui se réduit à un modèle conventionnel pour un contexte donné. La mise en place de bases de données floues pourrait également représenter un cas d'application des moteurs de recommandation, où la valeur de vérité serait propre à chaque utilisateur et correspondrait au score de recommandation. Cette proposition soulève plusieurs questions : Comment rendre l'interrogation de données efficace dans ce contexte ? Comment calculer de manière cohérente cette valeur de vérité lors de l'interrogation, en particulier lors de l'agrégation ? Et comment garantir la cohérence de la base de données avec un ensemble de règles dans ce contexte ?

Troisième partie

Annexes

Annexe A

Méthode de calcul et preuves pour la recherche de l'homomorphisme le plus spécifique

To explain our method for computing the most specific homomorphism h_m we introduce the notion of P -homomorphism.

Definition A.0.1. Given I a set of instantiated atoms and N a null occurring in I , let $P = \text{LinkedNull}_{I,N}$. A P -homomorphism is a homomorphism h such that $h(I) \subseteq I$ and for every null N' in $\text{null}(I) \setminus \text{null}(P)$, $h(N') = N'$. I is said to be P -reducible if there exists a P -homomorphism h such that $h(I)$ is a strict subset of I .

In the following proposition, given a set I of instantiated atoms and a null N in ν_0 , we use the following notation:

- P denotes the set of atoms $\text{LinkedNull}_{I,N}$, and $\text{null}(P) = \{N_1, \dots, N_p\}$ denotes the set of nulls occurring in P ;
- $q_{core}(I)$ is the answer to q_{core} computed against I . That is, $q_{core}(I)$ is the set $\{h_1, \dots, h_q\}$ of all possible P -homomorphisms defined over $\text{null}(P)$. We suppose that h_1 is the identity, i.e. for every $j = 1, \dots, p$, $h_1(N_j) = N_j$;
- H_P denotes the table with p columns and q rows such that $H_P[i, j] = h_i(N_j)$.
- Given a set of atoms Q , we denote by $\text{cons_null}(Q)$ the set of all symbols σ such that σ is a constant or a null *not* in $\text{null}(Q)$.

We recall that given two homomorphisms h_1 and h_2 over the same set of symbols Σ , h_1 is said to be *less specific than* h_2 , denoted by $h_1 \preceq h_2$, if there exists a homomorphism h over Σ such that $h \circ h_1 = h_2$. Using these notations, the following proposition holds.

Proposition A.0.1. Given h_i and $h_{i'}$ in $q_{core}(I)$, $h_i \preceq h_{i'}$ holds if and only if, for every $j = 1, \dots, p$, we have:

1. If $H_P[i, j]$ is in $\text{cons_null}(P)$, then $H_P[i, j] = H_P[i', j]$;
2. If $H_P[i, j]$ is a null N in $\text{null}(P)$, then for every $j' \neq j$ such that $H_P[i, j] = H_P[i, j']$ it holds that $H_P[i', j] = H_P[i', j']$.

Proof. Let us first assume that $h_i \preceq h_{i'}$ holds. In this case, there exists h such that $h \circ h_i = h_{i'}$. If N_j is such that $h_i(N_j)$ is a constant or a null not in $\text{null}(P)$, then for every P -homomorphism h_P , $h_P(h_i(N_j)) = h_i(N_j)$. Hence, $h_{i'}(N_j) = h \circ h_i(N_j) = h_i(N_j)$, which shows item (1). If j and j' are such that $h_i(N_j) = h_i(N_{j'})$, then $h_{i'}(N_j) = h_{i'}(N_{j'})$ also holds, showing item (2).

Conversely, assume that for h_i and $h_{i'}$, items (1) and (2) hold. Let h be defined for every $j = 1, \dots, p$ as follows: if there exists N_k such that $h_i(N_k) = N_j$ then $h(N_j) = h_{i'}(N_k)$, otherwise $h(N_j) = N_j$. We first notice that h is well-defined. Indeed, if k and k' are such that $h_i(N_k) = h_i(N_{k'})$, then we have two expressions defining $h(N_j)$, namely $h(N_j) = h_{i'}(N_k)$ and $h(N_j) = h_{i'}(N_{k'})$. However, by

item (2) we have $h_{i'}(N_k) = h_{i'}(N_{k'})$, and thus, these two expressions yield the same value. We now prove that $h_i \preceq h_{i'}$, that is, that for every $k = 1, \dots, p$, then $h_{i'}(N_k) = h(h_i(N_k))$. If $h_i(N_k)$ is not in $null(P)$, then, we have $h_i(N_k) = h_{i'}(N_k) = N_k$, and by construction of h we also have $h(N_k) = N_k$. Therefore, $h_{i'}(N_k) = h(h_i(N_k)) = N_k$. On the other hand, if $h_i(N_k) = N_j$, by definition of h , we have $h_{i'}(N_k) = h(N_j)$. Hence, $h_{i'}(N_k) = h(N_j) = h(h_i(N_k))$. Since for every $k = 1, \dots, p$, we have $h_{i'}(N_k) = h(h_i(N_k))$, it follows that $h_i \preceq h_{i'}$, and the proof is complete. $\square$

Example A.0.1. Let $I = \{B(N_1, N_2), B(N_2, N_1), C(N_1, a), C(N_2, a), C(N_3, a)\}$ and $\nu_0 = \{N_1\}$.

In this case, $P = \{\{B(N_1, N_2), B(N_2, N_1), C(N_1, a), C(N_2, a)\}\}$ and thus $null(P) = \{N_1, N_2\}$ and $cons_null(P) = \{a, N_3\}$. This implies that P -homomorphisms should *not* change N_3 , or in other words, N_3 should be treated as constant. The query q_{core} is thus written as follows, and the table H_P representing the answer $q_{core}(I)$ is shown below :

$$q_{core} : ans(x_1, x_2) \leftarrow B(x_1, x_2), B(x_2, x_1), C(x_1, a), C(x_2, a)$$

H_P	x_1	x_2
1	N_1	N_2
2	N_2	N_1

H_P has 2 columns (because $null(P)$ contains two nulls), and 2 rows due to two answers in $q_{core}(I)$. It is easy to see that $h_1 \preceq h_2$, and $h_2 \preceq h_1$ meaning that there is no advantage in trying to simplify the database instance in this case. Indeed, we have $h_1(I) = I$, where h_1 is the identity.

We have $h_2(I) = I$ as well, although h_2 is not the identity. Remark that h_2 does not satisfy $h_2 = h_2 \circ h_2$ (i.e. h_2 is not idempotent) because $h_2(h_2(N_1)) = h_2(N_2) = N_1$, whereas $h_2(N_1) = N_2$. As will be seen shortly, detecting such homomorphisms allows for computational optimizations. $\square$

The following corollary shows how to find one most specific homomorphism, based on Proposition A.0.1. To state the corollary, we use the following notation for $i = 1, \dots, q$:

- γ_i is the number of nulls N in $null(P)$ such that $h_i(N)$ is in $cons_null(P)$;
- $\mu_i = \{k \in \{1, \dots, q\} \mid (\forall j = 1, \dots, p)(h_i(N_j) \in cons_null(P) \Rightarrow h_k(N_j) = h_i(N_j))\}$;
- π_i is the number of distinct nulls in $null(P)$ in the set $h_i(null(P))$.

Intuitively speaking, considering that H_P is the tableau, then γ_i is the number of columns that, at row i , contain a symbol in $cons_null(P)$. On the other hand, μ_i is the set of all rows in H_P containing the same symbols of $cons_null(P)$ in the same columns as row i does (i.e., if $h_i(N_j) = c$ then $h_k(N_j) = c$). Then π_i is the number of distinct nulls in $nulls(P)$ occurring in row i .

The corollary below formalizes the following informal remarks:

1. If $h_i \neq h_i \circ h_i$, then h_i cannot be one of the most specific homomorphisms, because in this case, $h_i \prec h_i \circ h_i$. For instance, in Example A.0.1, we have $h_2 \neq h_2 \circ h_2$.
2. Most specific homomorphisms are among the rows of H_P with the largest number of symbols in $cons_null(P)$.

Indeed, let h_i and h_j be such that row i contains strictly more symbols in $cons_null(P)$ than row j and $h_i \prec h_j$. Then, there exists h such that $h \circ h_i = h_j$, and so, if N in $null(P)$ is such that $h_i(N)$ is in $cons_null(P)$, we have $h(h_i(N)) = h_i(N)$, and so $h_i(N) = h_j(N)$.

Thus, row j has at least as many symbols in $cons_null(P)$ as row i , which implies a contradiction. Hence, for every N in $null(P)$, $h_i(N)$ is also in $null(P)$, in which case rows i and j have no symbols in $cons_null(P)$, which is another contradiction.

3. Considering one of the rows defined just above, say row i , among all rows having the same symbols in $cons_null(P)$ in the same columns as row i , we argue that a row with as few distinct nulls in $null(P)$ defines one most specific homomorphism.

Corollaire A.0.1. Given I and P as above, denoting by $\{h_1, \dots, h_q\}$ the set $q_{core}(I)$, the following holds:

1. If h_i is one of the most specific P -homomorphisms in $q_{core}(I)$ then h_i is idempotent, that is, $h_i \circ h_i = h_i$.
2. h_i is one of the most specific P -homomorphisms in $q_{core}(I)$ if : (a) $\gamma_i = \max_{1 \leq j \leq q}(\gamma_j)$, and (b) $\pi_i = \min_{k \in \mu_i}(\pi_k)$.

Proof. First, Proposition A.0.1 implies that $h_i \preceq h_i \circ h_i$ holds for every h_i . Moreover, as h_i is a P -homomorphism, we have $h_i(I) \subseteq I$. Thus, $h_i \circ h_i(I) \subseteq I$, which implies that $h_i \circ h_i$ is a P -homomorphism as well. The proof of item (1) is therefore complete.

Assume that h_i satisfies (2) and let h_k be a P -homomorphism such that $h_i \preceq h_k$. By Proposition A.0.1, if $h_i(N_j)$ is in $\text{cons_null}(P)$, then $h_i(N_j) = h_k(N_j)$. Therefore, $\gamma_i \leq \gamma_k$, and as $\gamma_k \leq \gamma_i$, this implies $\gamma_i = \gamma_k$. Thus, k is in μ_i , which implies that $h_i(N_j)$ is in $\text{null}(P)$ if and only if so is $h_k(N_j)$. By Proposition A.0.1, if j and j' are such that $h_i(N_j) = h_i(N_{j'})$ then we also have $h_k(N_j) = h_k(N_{j'})$. It therefore follows that fewer nulls in $\text{null}(P)$ occur for h_k , that is $\pi_k \leq \pi_i$. As $\pi_i \leq \pi_k$ must hold, we obtain that $\pi_i = \pi_k$, meaning that h_i and h_k are equal up to a null renaming. The proof is therefore complete. $\square$

Procédure ChooseMostSpecific(H_P)

```

1 row_max ← 1 ;
2 count_max ← 0 ;
3 i ← 2;
4 pour tous i = 2 à q faire
5   idemPot ← true;
6   count_curr ← 0 ; j ← 1;
7   tant que idemPot = true and j ≤ p faire
8     si  $H_P[i, j]$  is in cons_null( $P$ ) alors
9       count_curr ← count_curr + 1;
10    sinon
11      Let  $N_k = H_P[i, j]$  ; //  $N_k$  is in null( $P$ )
12      si  $H_P[i, k] \neq N_k$  alors
13        idemPot ← false ; //  $h_i$  is not idem-potent
14        Mark row  $H_P[i]$ ;
15    j ← j + 1;
16  si idemPot = true alors
17    si count_curr > count_max alors
18      row_max ← i;
19      count_max ← count_curr;
20 row_spec ← row_max;
21 Let count_min be the number of distinct nulls in null( $P$ ) occurring in  $H_P[\text{row\_max}]$ ;
22 pour i = 2 à q faire
23   si row  $H_P[i]$  is not marked and  $i \neq \text{row\_max}$  alors
24     match ← true ; j ← 1;
25     tant que match = true and j ≤ p faire
26       si  $H_P[\text{row\_max}, j]$  is in cons_null( $P$ ) and  $H_P[\text{row\_max}, j] \neq H_P[i, j]$  alors
27         match ← false;
28       j ← j + 1;
29     si match = true alors
30       Let count_null be the number of distinct nulls in null( $P$ ) occurring in  $H_P[i]$ ;
31       si count_null < count_min alors
32         row_spec ← i;
33         count_min ← count_null;
34 retourner  $h_m$ , the homomorphism defined by  $H_P[\text{row\_spec}]$  ;
```

As a consequence, finding a most specific P -homomorphism in $q_{\text{core}}(I)$ amounts to (i) discard any row not defining an idem-potent homomorphism and (ii) among the remaining rows, identify one homomor-

phism satisfying item 2 of Corollary A.0.1. Procedure ChooseMostSpecific shows how to compute such a most specific homomorphism, and we notice that this does *not* require data access. To end the section, we illustrate the procedure ChooseMostSpecific as follows.

Example A.0.2. We first consider the context of Example A.0.1, where $I = \{B(N_1, N_2), B(N_2, N_1), C(N_1, a), C(N_2, a), C(N_3, a)\}$ and $P = \{B(N_1, N_2), B(N_2, N_1), C(N_1, a), C(N_2, a)\}$.

In this case, $null(P) = \{N_1, N_2\}$, $cons_null(P) = \{N_3\}$, and the associated table H_P has been shown already. Applying the procedure ChooseMostSpecific based on the table H_P , the following computations are achieved.

The first loop line 4 aims at marking rows defining non-idempotent P -homomorphisms (that is, such that $h \circ h \neq h$) and mean-while to find one unmarked row with as many symbols in $cons_null(P)$ as possible, in reference to Corollary A.0.1(2). These computations return the following:

- When processing row 2 of H_P , we have $H_P[2, 1] = N_2$ where N_2 is in $null(P)$, and $H_P[i, 2] = N_1$. Since $N_1 \neq N_2$, *idemPot* is set to **false** and row 2 is marked on line 14.
- Since there is no other row to process, the loop line 4 returns *row_max* = 1 and *count_curr* = 0.

Hence, ChooseMostSpecific returns *row_spec* = 1 and so, h_m is defined by $h_m(N_1) = N_1$ and $h_m(N_2) = N_2$. In other words, I is not simplified, which is indeed the expected result.

We now illustrate further the procedure ChooseMostSpecific, using two more sophisticated cases. First, let $\nu_0 = \{N_1\}$ and $I_1 = \{B(N_1, N_2), B(a, N_2), B(a, N_3), B(N_4, N_3), C(N_2, N_2), C(N_3, N_3)\}$. In this case, $LinkedNull_{I, N_1} = \{N_1, N_2\}$ and thus, $PSet = \{P\}$ where $P = \{B(N_1, N_2), B(a, N_2), C(N_2, N_2)\}$. Moreover, the query q_{core} below, is generated and its answer against I_1 , $q_{core}(I_1)$, is defined in the following table H_P^1 :

$$q_{core} : ans(x_1, x_2) \leftarrow B(x_1, x_2), B(a, x_2), C(x_2, x_2)$$

H_P^1	x_1	x_2
1	N_1	N_2
2	a	N_2
3	a	N_3
4	N_4	N_3

H_P^1 has 2 columns and 4 rows due to four possible answers in $q_{core}(I_1)$. Moreover, $h_1 \preceq h_2$, $h_2 \preceq h_3$ and $h_2 \preceq h_4$. Notice that h_3 and h_4 are not comparable because a, N_3 and N_4 are in $cons_null(P)$. Applying the procedure ChooseMostSpecific based on the table H_P^1 , the first loop line 4 achieves the following:

- No row is marked as non-idempotent on line 14. This is so because for every $i = 1, \dots, 4$, and every $j = 1, 2$, if $H_P^1[i, j] = N_k$ where N_k is N_1 or N_2 , $H_P^1[i, k] = N_k$.
- Regarding the value of *count_curr*, the computed value is 0 for the first row, 1 for row 2, and 2 for rows 3 and 4 (because a, N_3 and N_4 are in $cons_null(P)$). Thus, applying the test line 17, *count_curr* to set to 2, and on line 18, *row_max* is set to 3. Indeed, although for row 4, we have *count_curr* = 2, the test line 17 fails, and thus the value of *row_max* is not changed. Then, *row_spec* is set to 3 on line 20 and *cont_min* is set to 0 on line 21.

Therefore, processing the loop line 22 yields no change and the procedure ChooseMostSpecific returns h_m defined by $h_m(N_1) = a$ and $h_m(N_2) = N_3$, in which case, $h_m(I_1) = \{B(a, N_3), B(N_4, N_3), C(N_3, N_3)\}$, which is not redundant, when considering N_3 and N_4 as particular ‘constants’.

As a more sophisticated illustration of ChooseMostSpecific, let $\nu_0 = \{N_1\}$ and $I_2 = \{B(N_1, N_2), B(a, N_2), C(N_2, N_2), C(N_2, N_3)\}$. Here, $LinkedNull_{I, N_1} = \{N_1, N_2, N_3\}$ and thus, $PSet = \{P\}$ where $P = \{B(N_1, N_2), B(a, N_2), C(N_2, N_2), C(N_2, N_3)\}$. Moreover, the query q_{core} below, is generated and $q_{core}(I_2)$, is defined in the following table H_P^2 :

$$q_{core} : ans(x_1, x_2, x_3) \leftarrow B(x_1, x_2), B(a, x_2), C(x_2, x_2), C(x_2, x_3)$$

H_P^2	x_1	x_2	x_3
1	N_1	N_2	N_3
2	N_1	N_2	N_2
3	a	N_2	N_3
4	a	N_2	N_2

H_P^2 has 3 columns and 4 rows due to four possible answers in $q_{core}(I_2)$. Moreover, $h_1 \prec h_2$, $h_1 \prec h_3$, $h_2 \prec h_4$ and $h_3 \prec h_4$. Applying the procedure ChooseMostSpecific based on the table H_P^2 , the loop line 4 achieves the following:

- As above, no row is marked as non-idempotent on line 14. This is so because for $i = 1, \dots, 4$, and $j = 1, 2, 3$, if $H_P^2[i, j] = N_k$ where N_k is N_1 , N_2 or N_3 , $H_P^2[i, k] = N_k$.
- As above, on line 18, *row_max* is set to 3 and thus, *row_spec* is set to 3 on line 20. Here, *count_min* is set to 2 on line 21 because N_2 and N_3 are in $null(P)$.

When processing the loop line 22, the only row to be considered is row 4, for which *match* is **true**, thus implying that the test on line 29 succeeds. Since for row 4, the value of *count_null* is 1 (because row 4 contains the only null N_2), the value of *row_spec* is set to 4, line 32. Hence, ChooseMostSpecific returns h_m defined by $h_m(N_1) = a$, $h_m(N_2) = N_2$ and $h_m(N_3) = N_2$. In this case, $h_m(I_2) = \{B(a, N_2), C(N_2, N_2)\}$, which is not redundant.

Annexe B

Description des jeux de données (mise à jour cohérente)

B.1 Movie

B.1.1 Description

La base de données *Movie* (<https://github.com/neo4j-graph-examples/movies>) est fournie par Neo4J et sert d'exemple pratique pour l'apprentissage de la syntaxe Cypher et l'exécution d'opérations courantes sur les bases de données graphes telles que l'interrogation, le filtrage et l'exploration des relations. La base de données contient des informations sur les films, les acteurs, les réalisateurs, les producteurs, les scénaristes, les critiques et leurs relations. Chaque entité *Person* ou *Movie* comprend plusieurs attributs. Les acteurs, réalisateurs, producteurs, scénaristes et critiques sont liés aux films par les relations : *ActedIn*, *Directed*, *Produced*, *Wrote* et *Reviewed*. Les relations *ActedIn* et *Reviewed* possèdent aussi des attributs.

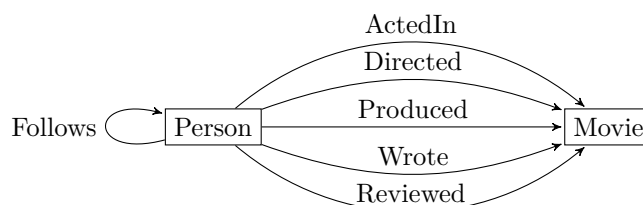


FIGURE B.1 – Schéma de la base *Movie*

B.1.2 Prédicats et contraintes

Prédicat	Arité	Termes	Description
<i>Movie</i>	4	<i>id, year, title, subtitle</i>	La description d'un film
<i>Person</i>	3	<i>id, birthYear, name</i>	La description d'une personne
<i>ActedIn</i>	3	<i>actorId, movieId, role</i>	Relie un acteur à un rôle dans un film
<i>Directed</i>	2	<i>directorId, movieId</i>	Relie un réalisateur à un film
<i>Produced</i>	2	<i>producerId, movieId</i>	Relie un producteur à un film
<i>Wrote</i>	2	<i>authorId, movieId</i>	Relie un scénariste à un film
<i>Reviewed</i>	4	<i>reviewerId, movieId, score, review</i>	Une critique sur un film, rattaché à l'auteur
<i>Follows</i>	2	<i>followerId, personId</i>	Décrit une relation de suivi entre les personnes

TABLE B.1 – Description de la base *Movie*

$$\begin{aligned}
& \text{Movie}(\text{movieId}, \text{year}, \text{title}, \text{subtitle}) \rightarrow \text{ActedIn}(\text{actorId}, \text{movieId}, \text{role}) \\
& \quad \rightarrow \text{Directed}(\text{directorId}, \text{movieId}) \\
& \quad \rightarrow \text{Produced}(\text{producerId}, \text{movieId}) \\
& \quad \rightarrow \text{Reviewed}(\text{reviewerId}, \text{movieId}, \text{score}, \text{review}) \\
& \quad \rightarrow \text{Wrote}(\text{authorId}, \text{movieId}) \\
& \text{ActedIn}(\text{actorId}, \text{movieId}, \text{role}) \rightarrow \text{Person}(\text{actorId}, \text{birthYear}, \text{name}) \\
& \quad \rightarrow \text{Movie}(\text{movieId}, \text{year}, \text{title}, \text{subtitle}) \\
& \text{Directed}(\text{directorId}, \text{movieId}) \rightarrow \text{Person}(\text{directorId}, \text{birthYear}, \text{name}) \\
& \quad \rightarrow \text{Movie}(\text{movieId}, \text{year}, \text{title}, \text{subtitle}) \\
& \text{Produced}(\text{producerId}, \text{movieId}) \rightarrow \text{Person}(\text{producerId}, \text{birthYear}, \text{name}) \\
& \quad \rightarrow \text{Movie}(\text{movieId}, \text{year}, \text{title}, \text{subtitle}) \\
& \text{Wrote}(\text{authorId}, \text{movieId}) \rightarrow \text{Person}(\text{authorId}, \text{birthYear}, \text{name}) \\
& \quad \rightarrow \text{Movie}(\text{movieId}, \text{year}, \text{title}, \text{subtitle}) \\
& \text{Reviewed}(\text{reviewerId}, \text{movieId}, \text{score}, \text{review}) \rightarrow \text{Person}(\text{reviewerId}, \text{birthYear}, \text{name}) \\
& \quad \rightarrow \text{Movie}(\text{movieId}, \text{year}, \text{title}, \text{subtitle}) \\
& \text{Follows}(\text{followerId}, \text{personId}) \rightarrow \text{Person}(\text{followerId}, \text{birthYear}, \text{name}) \\
& \quad \rightarrow \text{Person}(\text{personId}, \text{birthYear}, \text{name})
\end{aligned}$$
FIGURE B.2 – Liste des contraintes de la base *Movie*

B.2 GameOfThrone

B.2.1 Description

La base de données *GOT* « Game of Thrones » (<https://github.com/neo4j-graph-examples/graph-data-science>), fournie par Neo4J, est un exemple de démonstration pour leur bibliothèque *Graph Data Science*, axée sur la détection des communautés et l'analyse des réseaux basé sur les travaux de [BS16]. Elle modélise les relations au sein du monde fictif de Westeros, tiré de la série de livres « A Song of Ice and Fire » de George R.R. Martin. Le graphe contient des nœuds *Person*, représentant les personnages et des relations *Interacts*, représentant les interactions entre les personnages. Une interaction se produit chaque fois que les noms (ou surnoms) de deux personnages apparaissent à moins de 15 mots l'un de l'autre dans le texte du livre. Pour chaque personnage, nous avons sons status, la maison auquel il appartient et les livres ou il est mentionné. Une partie du graphe modélise les batailles avec ses différents acteurs et la zone géographique où elle prend place.

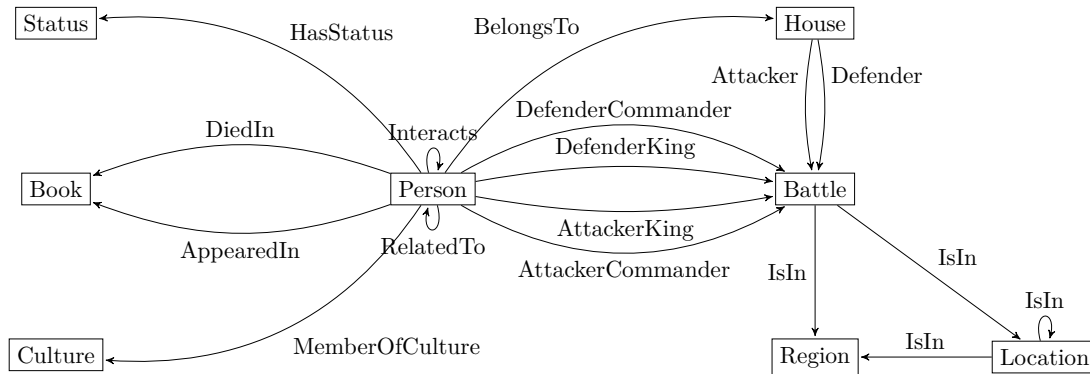


FIGURE B.3 – Schéma de la base

B.2.2 Prédicats et contraintes

Prédicat	Arité	Termes	Description
<i>Book</i>	3	<i>bookId, name, bookNumber</i>	La description d'un livre
<i>House</i>	2	<i>houseId, name</i>	La description d'une maison
<i>Person</i>	10	<i>personId, age, birthYear, deathChapter, introChapter, community, deathYear, gender, name, title</i>	La description d'une personne
<i>Battle</i>	10	<i>battleId, attackerSize, battleType, defenderSize, majorCapture, majorDeath, name, note, summer, year</i>	La description d'une bataille
<i>Status</i>	2	<i>statusId, name</i>	La description d'un statut
<i>Culture</i>	2	<i>cultureId, name</i>	La description d'une culture
<i>Region</i>	1	<i>regionName</i>	Une région
<i>Location</i>	1	<i>locationName</i>	Un lieu
<i>Defender</i>	3	<i>houseId, battleId, outcome</i>	Une maison défendant dans une bataille
<i>Attacker</i>	3	<i>houseId, battleId, outcome</i>	Une maison attaquant dans une bataille
<i>DefenderCommander</i>	2	<i>personId, battleId</i>	Le commandant défendant dans une bataille
<i>AttackerCommander</i>	2	<i>personId, battleId</i>	Le commandant attaquant dans une bataille
<i>DefenderKing</i>	2	<i>personId, battleId</i>	Le roi défendant dans une bataille
<i>AttackerKing</i>	2	<i>personId, battleId</i>	Le roi attaquant dans une bataille
<i>AppearedIn</i>	2	<i>personId, bookId</i>	Un personnage est apparu dans un livre
<i>DiedIn</i>	2	<i>personId, bookId</i>	Un personnage est mort dans un livre
<i>HasStatus</i>	2	<i>personId, statusId</i>	Un personnage a un certain statut
<i>BelongsTo</i>	2	<i>personId, houseId</i>	Un personnage appartient à une maison
<i>MemberOfCulture</i>	2	<i>personId, cultureId</i>	Un personnage est membre d'une culture
<i>Interacts</i>	4	<i>personId, otherId, bookId, weight</i>	Interaction entre deux personnages
<i>RelatedTo</i>	3	<i>personId, otherId, relationName</i>	Relation entre deux personnages
<i>IsIn</i>	2	<i>id, locationId</i>	Une bataille se déroule dans un lieu ou un lieu est dans un autre

TABLE B.2 – Description de la base *GOT*

$$\begin{aligned}
& \textit{AppearedIn}(\textit{personId}, \textit{bookId}) \rightarrow \textit{Person}(\textit{personId}, \dots) \\
& \quad \rightarrow \textit{Book}(\textit{bookId}, \textit{name}, \textit{bookNumber}) \\
& \textit{BelongsTo}(\textit{personId}, \textit{houseId}) \rightarrow \textit{Person}(\textit{personId}, \dots) \\
& \quad \rightarrow \textit{House}(\textit{houseId}, \textit{name}) \\
& \textit{Attacker}(\textit{houseId}, \textit{battleId}, \textit{outcome}) \rightarrow \textit{House}(\textit{houseId}, \textit{name}) \\
& \quad \rightarrow \textit{Battle}(\textit{battleId}, \dots) \\
& \textit{Defender}(\textit{houseId}, \textit{battleId}, \textit{outcome}) \rightarrow \textit{House}(\textit{houseId}, \textit{name}) \\
& \quad \rightarrow \textit{Battle}(\textit{battleId}, \dots) \\
& \textit{DiedIn}(\textit{personId}, \textit{bookId}) \rightarrow \textit{Person}(\textit{personId}, \dots) \\
& \quad \rightarrow \textit{Book}(\textit{bookId}, \textit{name}, \textit{bookNumber}) \\
& \textit{HasStatus}(\textit{personId}, \textit{statusId}) \rightarrow \textit{Person}(\textit{personId}, \dots) \\
& \quad \rightarrow \textit{Status}(\textit{statusId}, \textit{name}) \\
& \textit{Interacts}(\textit{personId}, \textit{otherId}, \textit{bookId}, \textit{weight}) \rightarrow \textit{Person}(\textit{personId}, \dots) \\
& \quad \rightarrow \textit{Person}(\textit{otherId}, \dots) \\
& \quad \rightarrow \textit{Book}(\textit{bookId}, \textit{name}, \textit{bookNumber}) \\
& \textit{MemberOfCulture}(\textit{personId}, \textit{cultureId}) \rightarrow \textit{Person}(\textit{otherId}, \dots) \\
& \quad \rightarrow \textit{Culture}(\textit{cultureId}, \textit{name}) \\
& \textit{RelatedTo}(\textit{personId}, \textit{otherId}, \textit{relationName}) \rightarrow \textit{Person}(\textit{personId}, \dots) \\
& \quad \rightarrow \textit{Person}(\textit{otherId}, \dots)
\end{aligned}$$
FIGURE B.4 – Liste des contraintes de la base *GOT*

$Culture(cultureId, name) \rightarrow MemberOfCulture(personId, cultureId)$
 $House(houseId, name) \rightarrow BelongsTo(personId, houseId)$
 $Battle(battleId, \dots) \rightarrow Attacker(houseId, battleId, outcome)$
 $\rightarrow Defender(houseId, battleId, outcome)$
 $\rightarrow IsIn(battleId, locationId)$
 $Person(personId, \dots) \rightarrow AppearedIn(personId, bookId)$
 $\rightarrow BelongsTo(personId, houseId)$
 $\rightarrow HasStatus(personId, statusId)$
 $\rightarrow MemberOfCulture(personId, cultureId)$

FIGURE B.4 – Liste des contraintes de la base *GOT*

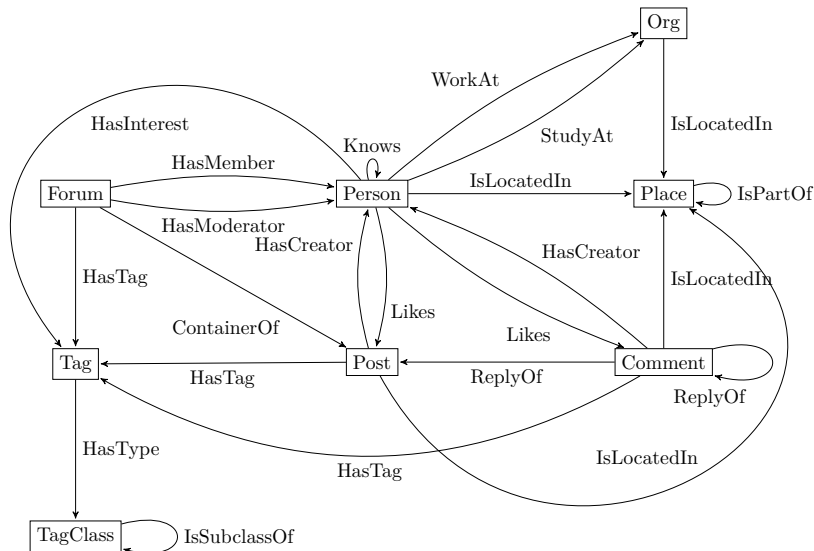
B.3 Social

B.3.1 Description

Social (<https://ldbcouncil.org/benchmarks/snb/>), ou LDBC Social Network Benchmark (LDBC SNB) est un graphe non attribué et généré artificiellement par le LDBC pour l'évaluation d'algorithmes sur les graphes. Ce graphe représente un réseau social avec des collections de postes, des commentaires, etc. Cette base est utilisée pour l'évaluation des systèmes de gestion de bases de données. Les benchmarks sur la base LDBC SNB regroupes deux catégories :

L'informatique décisionnelle qui se concentre sur des requêtes complexes, comprenant beaucoup d'agrégation et de jointure touchant une grande partie du graphe avec des micro-lots d'opérations d'insertion/suppression.

Une utilisation interactive qui capture un scénario de traitement transactionnel des graphes avec des requêtes de lecture complexes qui accèdent au voisinage d'un nœud donné dans le graphe et des opérations de mise à jour qui insèrent continuellement de nouvelles données dans le graphe.

FIGURE B.5 – Schéma de la base *Social*

B.3.2 Prédicats et contraintes

Prédicat	Arité	Termes	Description
<i>Forum</i>	1	<i>forum</i>	Un forum qui contient un ensemble de posts
<i>Post</i>	1	<i>post</i>	Un poste dans un forum
<i>Comment</i>	1	<i>comment</i>	Un commentaire à un poste
<i>Tag</i>	1	<i>tag</i>	Un label apposé sur un poste ou un forum
<i>TagClass</i>	1	<i>class</i>	Un type de label
<i>Person</i>	1	<i>person</i>	Une personne
<i>Organisation</i>	1	<i>org</i>	Une entreprise, un organisme, une université
<i>Place</i>	1	<i>place</i>	Un lieu ville/région/pays/continent
<i>HasCreator</i>	2	<i>commentOrPost, person</i>	Lie un commentaire ou un poste a son auteur
<i>ContainerOf</i>	2	<i>forum, post</i>	Lie un poste au forum dans lequel il est présent
<i>ReplyOf</i>	2	<i>comment, commentOrPost</i>	Lie un commentaire au poste/commentaire auquel il répond
<i>IsLocatedIn</i>	2	<i>x, place</i>	Donne la localisation d'un objet ou d'une personne
<i>Knows</i>	2	<i>person1, person2</i>	Deux personnes se connaissent
<i>HasInterest</i>	2	<i>person, tag</i>	Une personne qui est intéressée par un topique
<i>HasMember</i>	2	<i>forum, person</i>	Une personne est membre d'un forum
<i>WorkAt</i>	2	<i>person, org</i>	L'emploi d'une personne
<i>StudyAt</i>	2	<i>person, org</i>	Les études d'une personne
<i>IsSubclassOf</i>	2	<i>subClass, superClass</i>	Une relation hiérarchique entre les types de labels
<i>IsPartOf</i>	2	<i>subPlace, place</i>	Une relation d'inclusion entre les lieux
<i>HasType</i>	2	<i>tag, superClass</i>	Lie un label a son type

TABLE B.3 – Description de la base *Social*

$Comment(comment) \rightarrow HasCreator(comment, person)$
 $\rightarrow ReplyOf(comment, commentOrPost)$
 $\rightarrow IsLocatedIn(comment, place)$
 $ReplyOf(comment, commentOrPost) \rightarrow Comment(comment)$
 $Post(post) \rightarrow ContainerOf(forum, post)$
 $\rightarrow HasCreator(post, person)$
 $\rightarrow IsLocatedIn(post, place)$
 $ContainerOf(forum, post) \rightarrow Forum(forum)$
 $\rightarrow Post(post)$
 $HasMember(forum, person) \rightarrow Forum(forum)$
 $\rightarrow Person(person)$
 $HasModerator(forum, person) \rightarrow Forum(forum)$
 $\rightarrow Person(person)$
 $HasCreator(commentOrPost, person) \rightarrow Person(person)$
 $HasInterest(person, tag) \rightarrow Person(person)$
 $\rightarrow Tag(tag)$
 $Knows(person, other) \rightarrow Person(person)$
 $\rightarrow Person(other)$
 $\rightarrow Knows(other, person)$
 $WorkAt(person, org) \rightarrow Person(person)$
 $\rightarrow Organisation(org)$
 $Likes(person, commentOrPost) \rightarrow Person(person)$

FIGURE B.6 – Liste des contraintes de la base *Social*

$$\begin{aligned} & \text{StudyAt}(\text{person}, \text{org}) \rightarrow \text{Person}(\text{person}) \\ & \quad \rightarrow \text{Organisation}(\text{org}) \\ & \text{Organisation}(\text{org}) \rightarrow \text{IsLocatedIn}(\text{org}, \text{place}) \\ & \quad \text{TagClass}(\text{class}) \rightarrow \text{IsSubclassOf}(\text{class}, \text{'tagclass : 0'}) \\ & \text{IsSubclassOf}(\text{subClass}, \text{superClass}) \rightarrow \text{TagClass}(\text{subClass}) \\ & \quad \rightarrow \text{TagClass}(\text{superClass}) \\ & \text{HasTag}(\text{something}, \text{tag}) \rightarrow \text{Tag}(\text{tag}) \\ & \quad \text{Tag}(\text{tag}) \rightarrow \text{HasType}(\text{tag}, \text{'tagclass : 0'}) \\ & \text{HasType}(\text{tag}, \text{class}) \rightarrow \text{Tag}(\text{tag}) \\ & \quad \rightarrow \text{TagClass}(\text{class}) \\ & \text{IsLocatedIn}(\text{something}, \text{place}) \rightarrow \text{Place}(\text{place}) \\ & \quad \text{Place}(\text{place}) \rightarrow \text{IsPartOf}(\text{place}, \text{'place : 0'}) \\ & \text{IsPartOf}(\text{subPlace}, \text{place}) \rightarrow \text{Place}(\text{subPlace}) \\ & \quad \rightarrow \text{Place}(\text{place}) \\ & \text{Post}(\text{post}), \text{HasCreator}^-(\text{post}, \text{person}), \text{ContainerOf}(\text{forum}, \text{post}) \rightarrow \text{HasMember}(\text{forum}, \text{person}) \\ & \text{IsSubclassOf}^-(\text{subClass}, \text{class}), \text{IsSubclassOf}(\text{class}, \text{superClass}) \rightarrow \text{IsSubclassOf}(\text{subClass}, \text{superClass}) \\ & \text{HasType}(\text{tag}, \text{subClass})^-, \text{IsSubclassOf}(\text{subClass}, \text{superClass}) \rightarrow \text{HasType}(\text{tag}, \text{superClass}) \end{aligned}$$
FIGURE B.6 – Liste des contraintes de la base *Social*

Bibliographie

- [AA16] Basel Alshaikhdeeb and Kamsuriah Ahmad. Biomedical Named Entity Recognition : A Review. *International Journal on Advanced Science, Engineering and Information Technology*, 6(6) :889–895, December 2016. doi:10.18517/ijaseit.6.6.1367.
- [AA20] Muhammad Asfand-E-Yar and Ramis Ali. Semantic Integration of Heterogeneous Databases of Same Domain Using Ontology. *IEEE Access*, 8 :77903–77919, 2020. doi:10.1109/ACCESS.2020.2988685.
- [ABD⁺21] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savkovic, Michael Schmidt, Juan F. Sequeda, Slawek Staworko, and Dominik Tomaszuk. PG-Keys : Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2423–2436. ACM, 2021. doi:10.1145/3448016.3457561.
- [ABH⁺04] Maria Adriana Abrão, Béatrice Bouchou, Mirian Halfeld-Ferrari, Dominique Laurent, and Martin A. Musicante. Incremental Constraint Checking for XML Documents. In *Database and XML Technologies*, volume 3186 of *Lecture Notes in Computer Science*, pages 112–127, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. doi:10.1007/978-3-540-30081-6_9.
- [ABU79] Alfred V. Aho, Catriel Beeri, and Jeffrey D. Ullman. The theory of joins in relational databases. *ACM Transactions on Database Systems (TODS)*, 4(3) :297–314, 1979. doi:10.1145/320083.320091.
- [ACG20] Fatima N. Al-Aswadi, Huah Yong Chan, and Keng Hoon Gan. Automatic ontology construction from text : A review from shallow to deep learning trend. *Artificial Intelligence Review*, 53(6) :3901–3928, August 2020. doi:10.1007/s10462-019-09782-9.
- [ACP11] Waseem Akhtar, Álvaro Cortés-Calabuig, and Jan Paredaens. Constraints in RDF. In *Semantics in Data and Knowledge Bases*, volume 6834, pages 23–39, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. doi:10.1007/978-3-642-23441-5_2.
- [AG85] Serge Abiteboul and Gösta Grahne. Update Semantics for Incomplete Databases. In *Proceedings of the 11th International Conference on Very Large Data Bases - Volume 11, VLDB '85*, pages 1–12, Stockholm, Sweden, August 1985. Morgan Kaufmann. URL : <http://www.vldb.org/conf/1985/P001.PDF>.
- [AGOLOV20] Tareq Al-Moslmi, Marc Gallofré Ocaña, Andreas L. Opdahl, and Csaba Veres. Named Entity Extraction for Knowledge Graphs : A Literature Overview. *IEEE Access*, 8 :32862–32881, 2020. doi:10.1109/ACCESS.2020.2973928.
- [AHG07] Beatrice Alex, Barry Haddow, and Claire Grover. Recognising Nested Named Entities in Biomedical Text. In *Proceedings of the Workshop on BioNLP 2007 : Biological, Translational, and Clinical Language Processing*, BioNLP '07, pages 65–72, USA, June 2007. Association for Computational Linguistics. URL : <https://aclanthology.org/W07-1009/>.
- [AHH20] Joshua Amavi, Mirian Halfeld-Ferrari, and Nicolas Hiot. Natural Language Querying System Through Entity Enrichment. In *ADBIS, TPD and EDA 2020 Common Workshops and Doctoral Consortium*, volume 1260 of *Communications in Computer and Information Science*, pages 36–48, Lyon, France, August 2020. Springer International Publishing. doi:10.1007/978-3-030-55814-7_3.
- [AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*, volume 8. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1995.
- [AKM15] Yael Amsterdamer, Anna Kukliansky, and Tova Milo. A natural language interface for querying general and individual knowledge. *Proceedings of the VLDB Endowment*, 8(12) :1430–1441, August 2015. doi:10.14778/2824032.2824042.
- [All20] Allenai/scispace. AI2, January 2020. URL : <https://github.com/allenai/scispace>.
- [ALQ⁺21] Rana Alotaibi, Chuan Lei, Abdul Quamar, Vasilis Efthymiou, and Fatma Özcan. Property Graph Schema Optimization for Domain-Specific Knowledge Graphs. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 924–935, Chania, Greece, April 2021. IEEE Computer Society. doi:10.1109/ICDE51399.2021.00085.

- [APC02] A. Alves, F. Pereira, and A. Cardoso. Automatic reading and learning from text. In *Proceedings of the International Symposium on Artificial Intelligence*, 2002. URL : <https://old.cisuc.uc.pt/publication/show/624>.
- [ASU79] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Efficient optimization of a class of relational expressions. *ACM Transactions on Database Systems (TODS)*, 4(4) :435–454, 1979. doi:10.1145/320107.320112.
- [BB06] Loreto Bravo and Leopoldo Bertossi. Semantically Correct Query Answers in the Presence of Null Values. In *Current Trends in Database Technology – EDBT 2006*, Lecture Notes in Computer Science, pages 336–357, Berlin, Heidelberg, 2006. Springer. doi:10.1007/11896548_27.
- [BC02] Ted Briscoe and John Carroll. Robust Accurate Statistical Annotation of General Text. In *Proceedings of the Third International Conference on Language Resources and Evaluation (LREC’02)*, Las Palmas, Canary Islands - Spain, May 2002. European Language Resources Association (ELRA). URL : <http://www.lrec-conf.org/proceedings/lrec2002/pdf/250.pdf>.
- [BCC⁺20] Lucie Barque, Marie Candito, Matthieu Constant, Silvio Ricardo Cordeiro, Benoît Crabbé, Karën Fort, Bruno Guillaume, Pauline Haas, Richard Huyghe, Guy Perrier, Carlos Ramisch, Corentin Ribeyre, Agata Savary, Djamé Seddah, Vincent Segonne, Delphine Tribut, Eric Villemonte de la Clergerie, Yannick Parmentier, Caroline Pasquer, and Jean-Yves Antoine. Deep Sequoia corpus - PARSEME-FR corpus - FrSemCor, March 2020. URL : <https://deep-sequoia.inria.fr/>.
- [BDK⁺21] Ladjel Bellatreche, Marlon Dumas, Panagiotis Karras, Raimundas Matulevičius, Ahmed Awad, Matthias Weidlich, Mirjana Ivanović, and Olaf Hartig. *New Trends in Database and Information Systems : ADBIS 2021 Short Papers, Doctoral Consortium and Workshops : DOING, SIMPDA, MADEISD, MegaData, CAoNS, Tartu, Estonia, August 24-26, 2021, Proceedings*, volume 1450 of *Communications in Computer and Information Science*. Springer International Publishing, Cham, 2021. doi:10.1007/978-3-030-85082-1.
- [BEN22] Adrien Boiret, Cédric Eichler, and Benjamin Nguyen. Privacy Operators for Semantic Graph Databases as Graph Rewriting. In *New Trends in Database and Information Systems*, Communications in Computer and Information Science, pages 366–377, Cham, 2022. Springer International Publishing. doi:10.1007/978-3-031-15743-1_34.
- [BFV⁺18] Angela Bonifati, George Fletcher, Hannes Voigt, Nikolay Yakovets, and H. V. Jagadish. *Querying Graphs*, volume 10 of *Synthesis Lectures on Data Management*. Springer International Publishing, Cham, 2018. doi:10.1007/978-3-031-01864-0.
- [BH03] Béatrice Bouchou and Mirian Halfeld-Ferrari. Updates and Incremental Validation of XML Documents. In *The 9th International Workshop on Data Base Programming Languages (DBPL)*, Lecture Notes in Computer Science, pages 216–232, Berlin, Heidelberg, 2003. Springer. doi:10.1007/978-3-540-24607-7_14.
- [BKM⁺17] Michael Benedikt, Georgios Konstantinidis, Giansalvatore Mecca, Boris Motik, Paolo Papotti, Donatello Santoro, and Efthymia Tsamoura. Benchmarking the Chase. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS ’17, pages 37–52, New York, NY, USA, May 2017. Association for Computing Machinery. doi:10.1145/3034786.3034796.
- [BM11] Eduardo Blanco and Dan I. Moldovan. Some Issues on Detecting Negation from Text. In *Proceedings of the 24th International Florida Artificial Intelligence Research Society, FLAIRS - 24*, pages 228–233. AAAI Press, 2011. URL : <http://aaai.org/ocs/index.php/FLAIRS/FLAIRS11/paper/view/2629>.
- [BM15] Abel Browarnik and Oded Maimon. Ontology Learning from Text : Why the Ontology Learning Layer Cake is not Viable. *International Journal of Signs and Semiotic Systems (IJSSS)*, 4(2) :1–14, 2015. doi:10.4018/IJSSS.2015070101.
- [BMR⁺20] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya

- Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language Models are Few-Shot Learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., July 2020. [arXiv:2005.14165](https://arxiv.org/abs/2005.14165).
- [BMU22] Nelly Barret, Ioana Manolescu, and Prajna Upadhyay. Abstra : Toward Generic Abstractions for Data of Any Model. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management, CIKM '22*, pages 4803–4807, Atlanta GA USA, October 2022. Association for Computing Machinery. doi:10.1145/3511808.3557179.
- [BP98] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30(1-7) :107–117, April 1998. doi:10.1016/s0169-7552(98)00110-x.
- [BPV04] Andrey Balmin, Yannis Papakonstantinou, and Victor Vianu. Incremental validation of XML documents. *ACM Transactions on Database Systems*, 29(4) :710–751, December 2004. doi:10.1145/1042046.1042050.
- [BS16] Andrew Beveridge and Jie Shan. Network of Thrones. *Math Horizons*, 23(4) :18–22, April 2016. doi:10.4169/mathhorizons.23.4.18.
- [Bud99] Alexander Budanitsky. Lexical Semantic Relatedness and Its Application in Natural Language Processing. 1999. URL : <https://www.semanticscholar.org/paper/Lexical-Semantic-Relatedness-and-Its-Application-in-Budanitsky/c2d95e890ee904f70701fa27326d31980424d5dd>.
- [BWW99] Elliot G. Brown, Louise Wood, and Sue Wood. The Medical Dictionary for Regulatory Activities (MedDRA). *Drug Safety*, 20(2) :109–117, February 1999. doi:10.2165/00002018-199920020-00002.
- [CEH19] Jacques Chabin, Cédric Eichler, and Mirian Halfeld-Ferrari. Using a graph grammar to update a RDF/S document. 2019.
- [CEHH20] Jacques Chabin, Cédric Eichler, Mirian Halfeld-Ferrari, and Nicolas Hiot. Graph Rewriting System for Consistent Evolution of RDF/S databases. Research Report, LIFO, Université d’Orléans, INSA Centre Val de Loire, 2020. URL : <https://hal.science/hal-02560325>.
- [CEHH21] Jacques Chabin, Cédric Eichler, Mirian Halfeld-Ferrari, and Nicolas Hiot. Graph rewriting rules for RDF database evolution : Optimizing side-effect processing. *International Journal of Web Information Systems*, 17(6) :622–644, December 2021. doi:10.1108/IJWIS-03-2021-0033.
- [CGGH20] Rémi Cardon, Natalia Grabar, Cyril Grouin, and Thierry Hamon. Présentation de la campagne d’évaluation DEFT 2020 : similarité textuelle en domaine ouvert et extraction d’information précise dans des cas cliniques. In *6e conférence conjointe journées d’Études sur la parole (JEP, 33e édition), traitement automatique des langues naturelles (TALN, 27e édition), rencontre des étudiants chercheurs en informatique pour le traitement automatique des langues (RÉCITAL, 22e édition). Atelier DÉfi fouille de textes*, pages 1–13, Nancy, France, 2020. ATALA. URL : <https://hal.science/hal-02784737>.
- [CGLT20] Marco Console, Paolo Guagliardo, Leonid Libkin, and Etienne Toussaint. Coping with Incomplete Data : Recent Advances. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS’20*, pages 33–47, New York, NY, USA, June 2020. Association for Computing Machinery. doi:10.1145/3375395.3387970.
- [Cha24] Chatette : A dataset generator for Rasa NLU, March 2024. URL : <https://github.com/SimGus/Chatette>.
- [Che76] Peter Pin-Shan Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1) :9–36, March 1976. doi:10.1145/320434.320440.
- [CHH+23] Jacques Chabin, Mirian Halfeld-Ferrari, Nicolas Hiot, Dominique Laurent, and Lucas Moret-Bailly. DataFix, January 2023. URL : <https://gitlab.com/jacques-chabin/DataFix>.

- [CHH24] Jacques Chabin, Mirian Halfeld-Ferrari, and Nicolas Hiot. ArchiTXT, May 2024. URL : <https://github.com/Neplex/ArchiTXT>.
- [CHHL23] Jacques Chabin, Mirian Halfeld-Ferrari, Nicolas Hiot, and Dominique Laurent. Incremental consistent updating of incomplete databases (extended version - technical report). Technical report, LIFO, Université d'Orléans, INSA Centre Val de Loire, February 2023. URL : <https://hal.science/hal-03982841>, arXiv:2302.06246.
- [CHL20] Jacques Chabin, Mirian Halfeld-Ferrari, and Dominique Laurent. Consistent Updating of Databases with Marked Nulls. *Knowledge and Information Systems (KAIS)*, 62(4) :1571–1609, April 2020. doi:10.1007/s10115-019-01402-w.
- [CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, pages 77–90, New York, NY, USA, May 1977. Association for Computing Machinery. doi:10.1145/800105.803397.
- [CM10] Gunnar Carlsson and Facundo Mémoli. Characterization, Stability and Convergence of Hierarchical Clustering Methods. *Journal of Machine Learning Research*, 11(Apr) :1425–1470, August 2010. doi:10.5555/1756006.1859898.
- [CMO12] David Campos, Sérgio Matos, and José Luís Oliveira. Biomedical Named Entity Recognition : A Survey of Machine-Learning Tools. In *Theory and Applications for Advanced Text Mining*, volume 11, pages 175–195, Croatia, 2012. IntechOpen. doi:10.5772/51066.
- [Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6) :377–387, June 1970. doi:10.1145/362384.362685.
- [CS12] Marie Candito and Djamé Seddah. Le corpus Sequoia : annotation syntaxique et exploitation pour l'adaptation d'analyseur par pont lexical (The Sequoia Corpus : Syntactic Annotation and Use for a Parser Lexical Domain Adaptation Method) [in French]. In *Proceedings of the Joint Conference JEP-TALN-RECITAL 2012, volume 2 : TALN, Grenoble, France, June 4-8, 2012*, pages 321–334. ATALA/AFCP, 2012. URL : <https://hal.science/hal-00698938>.
- [CV05] Philipp Cimiano and Johanna Völker. Text2Onto. In *Natural Language Processing and Information Systems*, volume 3513, pages 227–238, Berlin, Heidelberg, June 2005. Springer Berlin Heidelberg. doi:10.1007/11428817_21.
- [DCLT19] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT : Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of NAACL-HLT*, pages 4171–4186. arXiv, May 2019. URL : <http://arxiv.org/abs/1810.04805>, arXiv:1810.04805.
- [DGLO⁺17] Giuseppe De Giacomo, Domenico Lembo, Xavier Oriol, Domenico Fabio Savo, and Ernest Teniente. Practical Update Management in Ontology-Based Data Access. In *The Semantic Web – ISWC 2017, Lecture Notes in Computer Science*, pages 225–242, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-68288-4_14.
- [DGLPR09] Giuseppe De Giacomo, Maurizio Lenzerini, Antonella Poggi, and Riccardo Rosati. Dealing with inconsistencies and incompleteness in database update (position paper). In *Inconsistency and Incompleteness in Databases*, page 97, August 2009. URL : https://www.academia.edu/61920314/Dealing_with_inconsistencies_and_incompleteness_in_database_update_position_paper_.
- [EB94] Martin G. Everett and Stephen P. Borgatti. Regular equivalence : General theory. *The Journal of Mathematical Sociology*, 19(1) :29–52, May 1994. doi:10.1080/0022250x.1994.9990134.
- [FFTD15] Wenfei Fan, Zhe Fan, Chao Tian, and Xin Luna Dong. Keys for graphs. *Proceedings of the VLDB Endowment*, 8(12) :1590–1601, August 2015. doi:10.14778/2824032.2824056.
- [FJR15] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1322–1333, New York, NY, USA, October 2015. Association for Computing Machinery. doi:10.1145/2810103.2813677.

- [FKAC13] Giorgos Flouris, George Konstantinidis, Grigoris Antoniou, and Vassilis Christophides. Formal foundations for RDF/S KB evolution. *Knowledge and Information Systems*, 35(1) :153–191, April 2013. doi:10.1007/s10115-012-0500-2.
- [FKP05] Ronald Fagin, Phokion G. Kolaitis, and Lucian Popa. Data exchange : Getting to the core. *ACM Transactions on Database Systems (TODS)*, 30(1) :174–210, March 2005. doi:10.1145/1061318.1061323.
- [FKUV86] Ronald Fagin, Gabriel M. Kuper, Jeffrey D. Ullman, and Moshe Y. Vardi. Updating Logical Databases. *Advances in Computing Research*, 3 :1–18, 1986. doi:10.21236/ada144937.
- [FL19] Wenfei Fan and Ping Lu. Dependencies for Graphs. *ACM Transactions on Database Systems*, 44(2) :1–40, June 2019. doi:10.1145/3287285.
- [FLP⁺24] Wenfei Fan, Ping Lu, Kehan Pang, Ruochun Jin, and Wenyuan Yu. Linking Entities across Relations and Graphs. In *ACM Transactions on Database Systems*, volume 49, pages 1–50, February 2024. doi:10.1145/3639363.
- [FTX⁺21] Wenfei Fan, Chao Tian, Ruiqi Xu, Qiang Yin, Wenyuan Yu, and Jingren Zhou. Incrementalizing Graph Algorithms. In *Proceedings of the 2021 International Conference on Management of Data, SIGMOD ’21*, pages 459–471, New York, NY, USA, June 2021. Association for Computing Machinery. doi:10.1145/3448016.3452796.
- [FUV83] Ronald Fagin, Jeffrey D. Ullman, and Moshe Y. Vardi. On the semantics of updates in databases. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, PODS ’83*, pages 352–365, New York, NY, USA, March 1983. Association for Computing Machinery. doi:10.1145/588058.588100.
- [FWX16] Wenfei Fan, Yinghui Wu, and Jingbo Xu. Functional Dependencies for Graphs. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD ’16*, pages 1843–1857, San Francisco California USA, June 2016. ACM. doi:10.1145/2882903.2915232.
- [FZE14] Anthony Fader, Luke Zettlemoyer, and Oren Etzioni. Open question answering over curated and extracted knowledge bases. In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD ’14*, pages 1156–1165, New York New York USA, August 2014. Association for Computing Machinery. doi:10.1145/2623330.2623677.
- [GCD18] Natalia Grabar, Vincent Claveau, and Clément Dalloux. CAS : French Corpus with Clinical Cases. In *Proceedings of the Ninth International Workshop on Health Text Mining and Information Analysis*, pages 122–128, Brussels, Belgium, October 2018. Association for Computational Linguistics. doi:10.18653/v1/w18-5614.
- [GGA⁺02] Pablo Gamallo, Marco González, Alexandre Agustini, Gabriel Lopes, and Vera Lúcia Strube de Lima. Mapping Syntactic Dependencies onto Semantic Relations. In *Proceedings of the ECAI Workshop on Machine Learning and Natural Language Processing for Ontology Engineering*, pages 15–22, 2002. URL : <https://www.semanticscholar.org/paper/Mapping-Syntactic-Dependencies-onto-Semantic-Gamallo-Gonz%C3%A1lez/45ee40319a8b57f2fe7ca3ecc33ee78b97697c92>.
- [GGI21] Cyril Grouin, Natalia Grabar, and Gabriel Illouz. Classification de cas cliniques et évaluation automatique de réponses d’étudiants : présentation de la campagne DEFT 2021 (Clinical cases classification and automatic evaluation of student answers : Presentation of the DEFT 2021 Challenge). In *Actes de la 28e Conférence sur le Traitement Automatique des Langues Naturelles. Atelier DÉfi Fouille de Textes (DEFT)*, pages 1–13, Lille, France, June 2021. ATALA. URL : <https://aclanthology.org/2021.jeptalnrecital-deft.1>.
- [GGK18] Archana Goyal, Vishal Gupta, and Manish Kumar. Recent Named Entity Recognition and Classification techniques : A systematic review. *Computer Science Review*, 29 :21–43, August 2018. doi:10.1016/j.cosrev.2018.06.001.
- [GL17] Paolo Guagliardo and Leonid Libkin. Correctness of SQL Queries on Databases with Nulls. *ACM SIGMOD Record*, 46(3) :5–16, October 2017. doi:10.1145/3156655.3156657.

- [GLDMRA19] Eduardo González López De Murillas, Hajo A. Reijers, and Wil M. Aalst. Connecting databases with process mining : A meta model and toolset. *Software and Systems Modeling (SoSyM)*, 18(2) :1209–1247, April 2019. doi:10.1007/s10270-018-0664-7.
- [GM17] Mauro Gaio and Ludovic Moncla. Extended named entity recognition using finite-state transducers : An application to place names. In *The Ninth International Conference on Advanced Geographic Information Systems, Applications, and Services (GEOProcessing 2017)*, Nice, France, March 2017. URL : <https://hal.science/hal-01492994>.
- [GMR13] François Goasdoué, Ioana Manolescu, and Alexandra Roatis. Efficient query answering against dynamic RDF databases. In *Proceedings of the 16th International Conference on Extending Database Technology, EDBT '13*, pages 299–310, New York, NY, USA, March 2013. Association for Computing Machinery. doi:10.1145/2452376.2452412.
- [GOP11] Georg Gottlob, Giorgio Orsi, and Andreas Pieris. Ontological queries : Rewriting and optimization. In *Proceedings of the 27th International Conference on Data Engineering, ICDE, Germany*, pages 2–13. IEEE, April 2011. doi:10.1109/ICDE.2011.5767965.
- [Got05] Georg Gottlob. Computing cores for data exchange : New algorithms and practical solutions. In *Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '05*, pages 148–159, New York, NY, USA, June 2005. Association for Computing Machinery. doi:10.1145/1065167.1065187.
- [Gra91] Gösta Grahne. *The Problem of Incomplete Information in Relational Databases*, volume 554 of *Lecture Notes in Computer Science*. Springer, Berlin, Heidelberg, November 1991. doi:10.1007/3-540-54919-6.
- [Gro89] Maurice Gross. The use of finite automata in the lexical representation of natural language. In *Electronic Dictionaries and Automata in Computational Linguistics*, Lecture Notes in Computer Science, pages 34–50, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. doi:10.1007/3-540-51465-1_3.
- [GZH10] Kavita Ganesan, ChengXiang Zhai, and Jiawei Han. Opinosis : A Graph Based Approach to Abstractive Summarization of Highly Redundant Opinions. In *Proceedings of the 23rd International Conference on Computational Linguistics, COLING '10*, pages 340–348, USA, August 2010. Association for Computational Linguistics. URL : <https://aclanthology.org/C10-1039/>.
- [HDYY18] Xin Hu, Depeng Dang, Yingting Yao, and Luting Ye. Natural language aggregate query over RDF data. *Information Sciences*, 454–455 :363–381, July 2018. doi:10.1016/j.ins.2018.04.042.
- [HER11] Maryam Hazman, Samhaa R. El-Beltagy, and Ahmed Rafea. A Survey of Ontology Learning Approaches. *International Journal of Computer Applications*, 22(9) :36–43, May 2011. doi:10.5120/2610-3642.
- [HHU17] Mirian Halfeld-Ferrari, Carmem S. Hara, and Flavio R. Uber. RDF Updates with Constraints. In *Knowledge Engineering and Semantic Web - 8th International Conference, KESW, Szczecin, Poland, Proceedings*, volume 786 of *Communications in Computer and Information Science*, pages 229–245, Szczecin, Poland, 2017. Springer International Publishing. doi:10.1007/978-3-319-69548-8_16.
- [HKJ⁺18] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. GRFusion : Graphs as First-Class Citizens in Main-Memory Relational Database Systems. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18*, pages 1789–1792, New York, NY, USA, May 2018. Association for Computing Machinery. doi:10.1145/3183713.3193541.
- [HL17] Mirian Halfeld-Ferrari and Dominique Laurent. Updating RDF/S Databases Under Constraints. In *Advances in Databases and Information Systems - 21st European Conference, ADBIS, Nicosia, Cyprus, Proceedings*, volume 10509, pages 357–371, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-66917-5_24.
- [HLS98] Mirian Halfeld-Ferrari, Dominique Laurent, and Nicolas Spyratos. Update Rules in Datalog Programs. *Journal of Logic and Computation*, 8(6) :745–775, December 1998. doi:10.1093/logcom/8.6.745.

- [HM17] Matthew Honnibal and Ines Montani. Spacy 2 : Natural language understanding with bloom embeddings, convolutional neural networks and incremental parsing. *Proceedings of the Association for Computational Linguistics (ACL)*, 7(1) :688–697, 2017.
- [HMB21] Nicolas Hiot, Anne-Lyse Minard, and Flora Badin. DOING@DEFT : utilisation de lexiques pour une classification efficace de cas cliniques. In *Traitement Automatique des Langues Naturelles*, pages 41–53, Lille, France, 2021. ATALA. URL : <https://hal.science/hal-03265924>.
- [HMYLB20] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy : Industrial-strength Natural Language Processing in Python. Explosion AI, 2020. doi:10.5281/zenodo.1212303.
- [HTZ⁺19] Shengze Hu, Zhen Tan, Weixin Zeng, Bin Ge, and Weidong Xiao. Entity Linking via Symmetrical Attention-Based Neural Network and Entity Structural Features. *Symmetry*, 11(4) :453, April 2019. doi:10.3390/sym11040453.
- [ILJ84] Tomasz Imielinski and Witold Lipski Jr. Incomplete information in relational databases. *Journal of the ACM (JACM)*, 31(4) :761–791, September 1984. doi:10.1145/1634.1886.
- [Jac01] Paul Jaccard. Distribution de la Flore Alpine dans le Bassin des Dranses et dans quelques régions voisines. *Bulletin de la Societe Vaudoise des Sciences Naturelles*, 37 :241–72, January 1901. doi:10.5169/seals-266440.
- [JM08] Dan Jurafsky and James H. Martin. *Speech and Language Processing : An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, volume 4. Prentice Hall, Upper Saddle River, N.J., 2nd edition edition, May 2008. URL : <https://books.google.fr/books?id=fZmj5UNK8AQC>.
- [KM12] Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL :2011. *ACM SIGMOD Record*, 41(3) :34–43, October 2012. doi:10.1145/2380776.2380786.
- [Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2) :127–145, June 1968. doi:10.1007/BF01692511.
- [Kon19] Agnieszka Konys. Knowledge Repository of Ontology Learning Tools from Text. *Procedia Computer Science*, 159 :1614–1628, January 2019. doi:10.1016/j.procs.2019.09.332.
- [KOTT03] J.-D. Kim, T. Ohta, Y. Tateisi, and J. Tsujii. GENIA corpus : A semantically annotated corpus for bio-textmining. *Bioinformatics*, 19(suppl_1) :i180–i182, July 2003. doi:10.1093/bioinformatics/btg1023.
- [LCL⁺18] Ming Liu, Lei Chen, Bingquan Liu, Guidong Zheng, and Xiaoming Zhang. DBpedia-Based Entity Linking via Greedy Search and Adjusted Monte Carlo Random Walk. *ACM Transactions on Information Systems*, 36(2) :1–34, April 2018. doi:10.1145/3086703.
- [Lea17] Learning multilingual named entity recognition from Wikipedia, October 2017. doi:10.6084/m9.figshare.5462500.v1.
- [Lib14] Leonid Libkin. Incomplete data : What went wrong, and how to fix it. In *Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS ’14, pages 1–13, New York, NY, USA, June 2014. Association for Computing Machinery. doi:10.1145/2594538.2594561.
- [Lib16] Leonid Libkin. SQL’s Three-Valued Logic and Certain Answers. *ACM Transactions on Database Systems (TODS)*, 41(1) :1–28, 2016. doi:10.1145/2877206.
- [Lip00] Carolyn E. Lipscomb. Medical Subject Headings (MeSH). *Bulletin of the Medical Library Association*, 88(3) :265–266, July 2000. URL : <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC35238/>.
- [LMI⁺20] Belinda Z. Li, Sewon Min, Srinivasan Iyer, Yashar Mehdad, and Wen-tau Yih. Efficient One-Pass End-to-End Entity Linking for Questions. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 6433–6441, Online, October 2020. Association for Computational Linguistics. arXiv:2010.02413, doi:10.18653/v1/2020.emnlp-main.522.

- [LMP01] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional Random Fields : Probabilistic Models for Segmenting and Labeling Sequence Data. In *Proceedings of the Eighteenth International Conference on Machine Learning (ICML 2001)*, Williams College, Williamstown, MA, USA, June 28 - July 1, 2001, number 2, pages 282–289. Morgan Kaufmann, 2001.
- [LMS08] Georg Lausen, Michael Meier, and Michael Schmidt. SPARQLing constraints for RDF. In *EDBT, 11th International Conference on Extending Database Technology, France, Proceedings*, EDBT '08, pages 499–509, New York, NY, USA, March 2008. ACM. doi:10.1145/1353343.1353404.
- [LS02] Sebastian Link and Klaus-Dieter Schewe. Towards an Arithmetic Theory of Consistency Enforcement based on Preservation of δ -constraints. *Electronic Notes in Theoretical Computer Science*, 61 :64–83, January 2002. doi:10.1016/S1571-0661(04)00306-8.
- [MAA⁺24] Jihane Mali, Shohreh Ahvar, Faten Atigui, Ahmed Azough, and Nicolas Travers. FACTDM : A Framework for Automated Cost-Based Data Model Transformation. In *Proceedings 27th International Conference on Extending Database Technology, EDBT 2024, Paestum, Italy, March 25 - March 28*, pages 822–825. OpenProceedings.org, 2024. doi:10.48786/EDBT.2024.79.
- [Mah15] Mariem Mahfoudh. *Adaptation d'ontologies avec les grammaires de graphes typés : évolution et fusion*. PhD thesis, Université de Haute Alsace-Mulhouse, May 2015. URL : <https://hal.science/tel-01528579>.
- [MC08] David McClosky and Eugene Charniak. Self-Training for Biomedical Parsing. In *Proceedings of ACL-08 : HLT, Short Papers*, pages 101–104, Columbus, Ohio, June 2008. Association for Computational Linguistics. URL : <https://aclanthology.org/P08-2026>.
- [MFTH15] Mariem Mahfoudh, Germain Forestier, Laurent Thiry, and Michel Hassenforder. Algebraic graph transformations for formalizing ontology changes and evolving ontologies. *Knowledge-Based Systems*, 73 :212–226, 2015. doi:10.1016/j.knosys.2014.10.007.
- [MLD08] Roser Morante, Anthony Liekens, and Walter Daelemans. Learning the scope of negation in biomedical texts. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing - EMNLP '08*, page 715, Honolulu, Hawaii, 2008. Association for Computational Linguistics. doi:10.3115/1613715.1613805.
- [MM01] Stoyan Mihov and Denis Maurel. Direct Construction of Minimal Acyclic Subsequential Transducers. In *Implementation and Application of Automata, 5th International Conference, CIAA 2000, London, Ontario, Canada, July 24-25, 2000, Revised Papers*, volume 2088 of *Lecture Notes in Computer Science*, pages 217–229. Springer, 2001. doi:10.1007/3-540-44674-5_18.
- [MMS79] David Maier, Alberto O. Mendelzon, and Yehoshua Sagiv. Testing implications of data dependencies. *ACM Transactions on Database Systems (TODS)*, 4(4) :455–469, December 1979. doi:10.1145/320107.320115.
- [Mor99] Emmanuel Morin. Automatic acquisition of semantic relations between terms from technical corpora. In *Proc. of the Fifth International Congress on Terminology and Knowledge Engineering-TKE'99*, August 1999.
- [MPS⁺19] Ishani Mondal, Sukannya Purkayastha, Sudeshna Sarkar, Pawan Goyal, Jitesh Pillai, Amitava Bhattacharyya, and Mahanandeeswar Gattu. Medical Entity Linking using Triplet Network. In *Proceedings of the 2nd Clinical Natural Language Processing Workshop*, pages 95–100, Minneapolis, Minnesota, USA, June 2019. Association for Computational Linguistics. arXiv:2012.11164, doi:10.18653/v1/W19-1912.
- [MRGL14] Pierre Maillot, Thomas Raimbault, David Genest, and Stéphane Loiseau. Consistency Evaluation of RDF Data : How Data and Updates are Relevant. In *Tenth International Conference on Signal-Image Technology and Internet-Based Systems, SITIS 2014, Marrakech, Morocco, November 23-27, 2014*, pages 187–193. IEEE, November 2014. doi:10.1109/SITIS.2014.39.
- [MRH⁺20] Anne-Lyse Minard, Andréane Roques, Nicolas Hiot, Mirian Halfeld-Ferrari, and Agata Savary. DOING@ DEFT : cascade de CRF pour l'annotation d'entités cliniques imbriquées.

- In *6e conférence conjointe Journées d'Études sur la Parole (JEP, 33e édition), Traitement Automatique des Langues Naturelles (TALN, 27e édition), Rencontre des Étudiants Chercheurs en Informatique pour le Traitement Automatique des Langues (RÉCITAL, 22e édition). Atelier DÉfi Fouille de Textes*, pages 66–78, Nancy, France, June 2020. ATALA et AFCP. URL : <https://hal.science/hal-02784743>.
- [MT04] Rada Mihalcea and Paul Tarau. TextRank : Bringing Order into Text. In *Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing*, pages 404–411, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL : <https://aclanthology.org/W04-3252>.
- [MV01] Alexander Maedche and Raphael Volz. The text-to-onto ontology extraction and maintenance environment. In *Proceedings of the ICDM-Workshop on Integrating Data Mining and Knowledge Management, San Jose, California, USA*, 2001.
- [Nav24] Roberto Navigli. Word Sense Disambiguation : A Survey. *ACM computing surveys (CSUR)*, 41(2) :1–69, 2024. doi:10/dhsjt2.
- [Niv04] Joakim Nivre. Incrementality in deterministic dependency parsing. In *Proceedings of the Workshop on Incremental Parsing : Bringing Engineering and Cognition Together, IncrementParsing '04*, pages 50–57, Barcelona, Spain, July 2004. Association for Computational Linguistics. doi:10.3115/1613148.1613156.
- [Niv15] Joakim Nivre. Towards a Universal Grammar for Natural Language Processing. In *Computational Linguistics and Intelligent Text Processing*, volume 9041 of *Lecture Notes in Computer Science*, pages 3–16, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-18111-0_1.
- [NJLC20] Raúl Navarro-Almanza, Reyes Juárez-Ramírez, Guillermo Licea, and Juan R. Castro. Automated Ontology Extraction from Unstructured Texts using Deep Learning. In *Intuitionistic and Type-2 Fuzzy Logic Enhancements in Neural and Optimization Algorithms : Theory and Applications*, volume 862, pages 727–755. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-35445-9_50.
- [NJM21] Zara Nasar, Syed Waqar Jaffry, and Muhammad Kamran Malik. Named Entity Recognition and Relation Extraction : State-of-the-Art. *ACM Computing Surveys*, 54(1) :1–39, February 2021. doi:10.1145/3445965.
- [NK16] Charalampos Nikolaou and Manolis Koubarakis. Querying incomplete information in RDF with SPARQL. *Artificial Intelligence*, 237 :138–171, August 2016. doi:10.1016/j.artint.2016.04.005.
- [NKBA19] Mark Neumann, Daniel King, Iz Beltagy, and Waleed Ammar. ScispaCy : Fast and Robust Models for Biomedical Natural Language Processing. In *Proceedings of the 18th BioNLP Workshop and Shared Task*, pages 319–327, Florence, Italy, August 2019. Association for Computational Linguistics. arXiv:1902.07669, doi:10.18653/v1/W19-5034.
- [NRR⁺13] Joel Nothman, Nicky Ringland, Will Radford, Tara Murphy, and James R. Curran. Learning multilingual named entity recognition from Wikipedia. *Artificial Intelligence*, 194 :151–175, January 2013. doi:10.1016/j.artint.2012.03.006.
- [One13] Adrian Onet. The Chase Procedure and its Applications in Data Exchange. In *Dagstuhl Follow-Ups*, volume 5, pages 1–37. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013. doi:10.4230/DFU.VOL5.10452.1.
- [PA21] Ankur A. Patel and Ajay Uppili Arasanipalai. *Applied Natural Language Processing in the Enterprise*. O'Reilly Media, Inc., May 2021. URL : <https://books.google.fr/books?id=5dktEAAAQBAJ>.
- [Pat15] Peter Patel-Schneider. Using Description Logics for RDF Constraint Checking and Closed-World Recognition. *Proceedings of the AAAI Conference on Artificial Intelligence*, 29(1) :247–253, February 2015. doi:10.1609/aaai.v29i1.9177.
- [PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking : Bringing Order to the Web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL : <http://ilpubs.stanford.edu:8090/422/>.

- [PEM⁺15] Thorsten Papenbrock, Jens Ehrlich, Jannik Marten, Tommy Neubert, Jan-Peer Rudolph, Martin Schönberg, Jakob Zwiener, and Felix Naumann. Functional dependency discovery : An experimental evaluation of seven algorithms. *Proceedings of the VLDB Endowment*, 8(10) :1082–1093, June 2015. doi:10.14778/2794367.2794377.
- [PKM14] Alexandre Passos, Vineet Kumar, and Andrew McCallum. Lexicon Infused Phrase Embeddings for Named Entity Resolution. In *Proceedings of the Eighteenth Conference on Computational Natural Language Learning*, pages 78–86. Citeseer, 2014. doi:10.3115/v1/W14-1609.
- [Pok15] Jaroslav Pokorný. Graph Databases : Their Power and Limitations. In *Computer Information Systems and Industrial Management : 14th IFIP TC 8 International Conference, CISIM 2015, Warsaw, Poland, September 24-26, 2015, Proceedings 14*, Lecture Notes in Computer Science, pages 58–69, Cham, 2015. Springer International Publishing. doi:10.1007/978-3-319-24369-6_5.
- [Por01] Martin F. Porter. Snowball : A language for stemming algorithms, October 2001. URL : <https://snowball.tartarus.org/texts/introduction>.
- [PS11] Reinhard Pichler and Sebastian Skritek. The complexity of evaluating tuple generating dependencies. In *Proceedings of the 14th International Conference on Database Theory, ICDT '11*, pages 244–255, New York, NY, USA, March 2011. Association for Computing Machinery. doi:10.1145/1938551.1938583.
- [Rau91] L. F. Rau. Extracting company names from text. In *The Seventh IEEE Conference on Artificial Intelligence Application*, volume 1, pages 29–32. IEEE, 1991. doi:10.1109/CAIA.1991.120841.
- [RBP⁺18] Joe Raad, Wouter BECK, Nathalie Pernelle, Fatiha Sais, and Frank Harmelen. Détection de liens d'identité erronés en utilisant la détection de communautés dans les graphes d'identité. *Ingénierie des systèmes d'information*, 23(3-4) :61–88, August 2018. doi:10.3166/isi.23.3-4.61-88.
- [RED08] REDAC : Corpus texte WikipédiaFR2008, June 2008. URL : <http://redac.univ-tlse2.fr/corpus/wikipedia.html>.
- [Rei86] Raymond Reiter. A sound and sometimes complete query evaluation algorithm for relational databases with null values. *Journal of the ACM (JACM)*, 33(2) :349–370, April 1986. doi:10.1145/5383.5388.
- [ReS] réSEaux Numériques de Données sémantiques : Utilité et vie Privée. URL : <https://anr.fr/Projet-ANR-18-CE23-0010>.
- [RM95] Lance A. Ramshaw and Mitchell P. Marcus. Text Chunking Using Transformation-Based Learning. In *Third Workshop on Very Large Corpora*, volume 11 of *Text, Speech and Language Technology*, pages 157–176, Dordrecht, May 1995. Springer Netherlands. arXiv:cmp-lg/9505040, doi:10.1007/978-94-017-2390-9_10.
- [RMB16] Anisa Rula, Andrea Maurino, and Carlo Batini. Data Quality Issues in Linked Open Data. In *Data and Information Quality*, pages 87–112, Cham, March 2016. Springer International Publishing. doi:10.1007/978-3-319-24106-7_4.
- [RNSS18] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving Language Understanding by Generative Pre-Training. OpenAI, June 2018. URL : <https://openai.com/index/language-unsupervised>.
- [RSG16] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. "Why Should I Trust You ?" : Explaining the Predictions of Any Classifier. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144, San Francisco California USA, August 2016. ACM. doi:10.1145/2939672.2939778.
- [SAS19] Nadine Steinmetz, Ann-Katrin Arning, and Kai-Uwe Sattler. From Natural Language Questions to SPARQL Queries : A Pattern-based Approach. In *BTW 2019*, pages 289–308. Gesellschaft für Informatik, Bonn, 2019. doi:10/ggdnr4.
- [SEN] SENDUP – SEmantic Network of Data : Utility and Privacy. URL : <https://www.univ-orleans.fr/lifo/evenements/sendup-project/>.

- [SG02] Yannis Stavarakas and Manolis Gergatsoulis. Multidimensional Semistructured Data : Representing Context-Dependent Information on the Web. In *Advanced Information Systems Engineering, 14th International Conference, CAiSE 2002, Toronto, Canada, May 27-31, 2002, Proceedings*, volume 2348 of *CAiSE '02*, pages 183–199, Berlin, Heidelberg, May 2002. Springer. doi:10.1007/3-540-47961-9_15.
- [SGM19] Emma Strubell, Ananya Ganesh, and Andrew McCallum. Energy and Policy Considerations for Deep Learning in NLP. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3645–3650, Florence, Italy, June 2019. Association for Computational Linguistics. arXiv:1906.02243, doi:10.18653/v1/P19-1355.
- [SHW14] Wei Shen, Jiawei Han, and Jianyong Wang. A probabilistic model for linking named entities in web text with heterogeneous information networks. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, pages 1199–1210, New York, NY, USA, June 2014. Association for Computing Machinery. doi:10.1145/2588555.2593676.
- [Sir14] Cristina Sirangelo. *Representing and Querying Incomplete Information : A Data Interoperability Perspective*. HDR, Ecole Normale Supérieure de Cachan, Cachan, December 2014. URL : <https://hal.science/tel-01092547>.
- [SJ72] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1) :11–21, January 1972. doi:10.1108/eb026526.
- [Spa] spaCy 101 : Everything you need to know · spaCy Usage Documentation. URL : <https://spacy.io/usage/spacy-101>.
- [SSM19] Liwei Song, Reza Shokri, and Prateek Mittal. Privacy Risks of Securing Machine Learning Models against Adversarial Examples. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS '19*, pages 241–257, New York, NY, USA, November 2019. Association for Computing Machinery. doi:10.1145/3319535.3354211.
- [SSM⁺22] Agata Savary, Alena Silvanovich, Anne-Lyse Minard, Nicolas Hiot, and Mirian Halfeld-Ferrari. Relation Extraction from Clinical Cases for a Knowledge Graph. In *New Trends in Database and Information Systems*, volume 1652 of *Communications in Computer and Information Science*, pages 353–365, Turin, Italy, August 2022. Springer International Publishing. doi:10.1007/978-3-031-15743-1_33.
- [SSPV20] Ahmad Sakor, Kuldeep Singh, Anery Patel, and Maria-Esther Vidal. Falcon 2.0 : An Entity and Relation Linking Tool over Wikidata. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management, CIKM '20*, pages 3141–3148, New York, NY, USA, October 2020. Association for Computing Machinery. arXiv:1912.11270, doi:10.1145/3340531.3412777.
- [ST98] Klaus-Dieter Schewe and Bernhard Thalheim. Limitations of rule triggering systems for integrity maintenance in the context of transition specifications. *Acta Cybernetica*, 13(3) :277–304, January 1998. doi:10.1007/3-540-63699-4_12.
- [Sta03] Yannis Stavarakas. *Multidimensional Semistructured Data : Representing Context-Dependent Information on the Web*. PhD thesis, June 2003.
- [Suj01] Walter Sujansky. Heterogeneous Database Integration in Biomedicine. *Journal of Biomedical Informatics*, 34(4) :285–298, August 2001. doi:10.1006/jbin.2001.1024.
- [UWB⁺18] Prasetya Utama, Nathaniel Weir, Fuat Basik, Carsten Binnig, Ugur Cetintemel, Benjamin Hättasch, Amir Ilkhechi, Shekar Ramaswamy, and Arif Usta. An End-to-end Neural Natural Language Interface for Databases. *arXiv preprint arXiv:1804.00401*, April 2018. arXiv:1804.00401.
- [VCE⁺23] Genoveva Vargas-Solar, Tania Cerquitelli, Javier A. Espinosa-Oviedo, François Cheval, Anthelme Buchaille, and Luca Polgar. Conversational Data Exploration : A Game-Changer for Designing Data Science Pipelines, November 2023. arXiv:2311.06695, doi:10.48550/arXiv.2311.06695.
- [VDE23] Genoveva Vargas-Solar, Karim Dao, and Javier Espinosa-Oviedo. Translating data science queries from natural language into graph analytics queries using NLDS-QL. In *Workshops*

- of the *EDBT/ICDT 2023 Joint Conference*, number 3379, Ioannina, Greece, March 2023. URL : <https://hal.science/hal-04272051>.
- [Wan19] Wenlu Wang. A cross-domain natural language interface to databases using adversarial text method. In *Database*, volume 1, page 4, 2019.
- [Wą20] Jarosław Wątróbski. Ontology learning methods from text-an extensive knowledge-based approach. *Procedia Computer Science*, 176 :3356–3368, January 2020. doi:10.1016/j.procs.2020.09.061.
- [WFG⁺18] David S. Wishart, Yannick D. Feunang, An C. Guo, Elvis J. Lo, Ana Marcu, Jason R. Grant, Tanvir Sajed, Daniel Johnson, Carin Li, and Zinat Sayeeda. DrugBank 5.0 : A major update to the DrugBank database for 2018. *Nucleic acids research*, 46(D1) :D1074–D1082, 2018. doi:10.1093/nar/gkx1037.
- [WHM⁺11] Ralph Weischedel, Eduard Hovy, Mitchell Marcus, Martha Palmer, Robert Belvin, Sameer Pradhan, Lance Ramshaw, and Nianwen Xue. OntoNotes : A large training corpus for enhanced processing. *Handbook of Natural Language Processing and Machine Translation*, page 59, 2011.
- [Win88] Marianne Winslett. A model-based approach to updating databases with incomplete information. *ACM Transactions on Database Systems (TODS)*, 13(2) :167–196, June 1988. doi:10.1145/42338.42386.
- [Win90] Marianne Southall Winslett. *Updating Logical Databases*. Number 9 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, Cambridge ; New York, 1990.
- [WKB18] Albert Weichselbraun, Philipp Kuntschik, and Adrian M.P. Braşoveanu. Mining and Leveraging Background Knowledge for Improving Named Entity Linking. In *Proceedings of the 8th International Conference on Web Intelligence, Mining and Semantics*, WIMS '18, pages 1–11, New York, NY, USA, June 2018. Association for Computing Machinery. doi:10.1145/3227609.3227670.
- [WKG⁺08] David S. Wishart, Craig Knox, An Chi Guo, Dean Cheng, Savita Shrivastava, Dan Tzur, Bijaya Gautam, and Murtaza Hassanali. DrugBank : A knowledgebase for drugs, drug actions and drug targets. *Nucleic acids research*, 36(suppl_1) :D901–D906, 2008. doi:10.1093/nar/gkm958.
- [WLB12] Wilson Wong, Wei Liu, and Mohammed Bennamoun. Ontology Learning from Text : A Look Back and into the Future. *ACM Computing Surveys*, 44(4) :1–36, August 2012. doi:10.1145/2333112.2333115.
- [WPM⁺13] Weischedel, Ralph, Palmer, Martha, Marcus, Mitchell, Hovy, Eduard, Pradhan, Sameer, Ramshaw, Lance, Xue, Nianwen, Taylor, Ann, Kaufman, Jeff, Franchini, Michelle, El-Bachouti, Mohammed, Belvin, Robert, and Houston, Ann. OntoNotes Release 5.0, October 2013. doi:10.35111/XMHB-2B84.
- [WR83] Douglas R. White and Karl P. Reitz. Graph and semigroup homomorphisms on networks of relations. *Social Networks*, 5(2) :193–234, June 1983. doi:10.1016/0378-8733(83)90025-4.
- [WWM⁺17] Wei-Hung Weng, Kavishwar B. Waghlikar, Alexa T. McCray, Peter Szolovits, and Henry C. Chueh. Medical subdomain classification of clinical notes using a machine learning-based natural language processing approach. *BMC Medical Informatics and Decision Making*, 17(1) :1–13, December 2017. doi:10.1186/s12911-017-0556-8.
- [XMW23] Canwen Xu, Julian McAuley, and Penghan Wang. Mirror : A Natural Language Interface for Data Querying, Summarization, and Visualization. In *Companion Proceedings of the ACM Web Conference 2023*, WWW '23 Companion, pages 49–52, New York, NY, USA, April 2023. Association for Computing Machinery. arXiv:2303.08697, doi:10.1145/3543873.3587309.
- [Zan84] Carlo Zaniolo. Database relations with null values. *Journal of Computer and System Sciences*, 28(1) :142–166, February 1984. doi:10.1016/0022-0000(84)90080-1.

- [ZHW⁺14] Lei Zou, Ruizhe Huang, Haixun Wang, Jeffrey Xu Yu, Wenqiang He, and Dongyan Zhao. Natural language question answering over RDF : A graph data driven approach. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 313–324, Snowbird Utah USA, June 2014. ACM. doi:10.1145/2588555.2610525.
- [Zip35] G. K. Zipf. *The Psycho-Biology of Language*. The Psycho-Biology of Language. Houghton, Mifflin, Oxford, England, 1935.
- [ZM93] John M. Zelle and Raymond J. Mooney. Learning semantic grammars with constructive inductive logic programming. In *Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI’93*, pages 817–822, Washington, D.C., July 1993. AAAI Press.
- [ZNL18] Hamid Zafar, Giulio Napolitano, and Jens Lehmann. Formal Query Generation for Question Answering over Knowledge Bases. In *European Semantic Web Conference*, Lecture Notes in Computer Science, pages 714–728, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-93417-4_46.
- [ZS89] Kaizhong Zhang and Dennis Shasha. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM Journal on Computing*, 18(6) :1245–1262, December 1989. doi:10.1137/0218082.
- [ZYZC18] Weiguo Zheng, Jeffrey Xu Yu, Lei Zou, and Hong Cheng. Question answering over knowledge graphs : Question understanding via template decomposition. *Proceedings of the VLDB Endowment*, 11(11) :1373–1386, July 2018. doi:10.14778/3236187.3236192.

Glossaire

Chase Algorithme de point fixe pour la saturation de contraintes dans une base de données. 20, 22, 28–33, 36, 37, 40

Core . 10, 11, 15, 20, 22, 24, 32–34, 37, 40, 115

CoreNLP <https://stanfordnlp.github.io/CoreNLP/>. 52, 81

Cypher Un langage de requetage conçu par Neo4J pour les bases de données graphes basée sur de la reconnaissance de motifs décrit en art ASCII. 12, 14, 18, 22, 25, 29–31, 103, 104, 125

Docker Docker est une plateforme permettant l'exécution dans des environnements reproductible et légers. <https://www.docker.com/>. 35

DOING Action de recherche s'intéressant aux problématiques autour de l'extraction, le stockage et le traitement de données médicales. <https://www.univ-orleans.fr/lifo/evenements/doing/>. 1

Duckling Duckling est une librairie open source développée par Meta qui permet d'extraire des informations structurées telles que les dates, les heures et les montants d'argent à partir de texte brut grâce à un ensemble de grammaire. <https://github.com/facebook/duckling>. 108

Ennov Editeur de logiciels concevant une plateforme unifiée de gestion du contenu et de l'information pour soutenir le développement des produits des sciences de la vie. <https://ennov.com/>. 1, 5, 91, 108, 111

Fondation Apache Une organisation à but non lucratif qui développe des logiciels open source. 145, 146

GeoNames Une base de données géographiques ouverte et gratuite couvrant tous les pays et contenant plus de onze millions de noms de lieux. <https://www.geonames.org/>. 95

Git Git est un système de gestion de versions décentralisé, libre et gratuit, créé par Linus Torvalds. Il est notamment utilisé pour gérer du code source comme le noyau Linux. <https://git-scm.com/>. 34, 35, 81

ISO L'organisation internationale de normalisation. 12

Linux Linux ou GNU/Linux est une famille de systèmes d'exploitation open source de type Unix fondés sur le noyau Linux. 145, 146

Lucene Une bibliothèque open source en Java et mise à disposition par la Fondation Apache pour l'indexation et la recherche de texte. 97, 146

Meta Meta Platforms ou Meta, anciennement connue sous le nom de Facebook, est une multinationale technologique américaine axée sur les plateformes sociales et la réalité virtuelle fondée en 2004 par Mark Zuckerberg.. 145

MySQL Un système de gestion de bases de données relationnelles. <https://www.mysql.com/>. 35–38

Neo4J Un système de gestion de base de données utilisant des graphes de propriétés. <https://neo4j.com/>. 12, 14, 21–23, 34–37, 39, 125, 126

- Python** Un langage de programmation interprété utilisant un typage dynamique. Il est principalement utilisé dans les sciences des données. <https://www.python.org/>. 81, 146
- Ray** Ray est une librairie Python open-source conçue pour faciliter le développement et le déploiement d'applications parallèles et distribuées à grande échelle.. 109
- Rocky Linux** Rocky Linux est une distribution Linux basée sur le code source du système d'exploitation Red Hat Enterprise Linux. <https://rockylinux.org/>. 35
- SolR** Une plateforme logicielle de moteur de recherche s'appuyant sur la bibliothèque de recherche Lucene, créée par la Fondation Apache. 97, 109
- SpaCy** SpaCy est une bibliothèque logicielle Python de traitement automatique des langues maintenue par ExplosionAI. <https://spacy.io/>. 52, 105, 109
- Universal Dependencies** Universal Dependencies (UD) est un guide d'annotation pour la construction cohérente de grammaires textuelle (parties du discours, caractéristiques morphologiques et dépendances syntaxiques) dans différentes langues humaines. L'UD est un effort communautaire ouvert avec plus de 500 contributeurs produisant plus de 200 banques d'arbres dans plus de 100 langues. <https://universaldependencies.org/>. 46, 52
- Wikipédia** Une encyclopédie universelle, multilangue, libre et collaborative. <https://wikipedia.org/>. 102

Acronymes

ANSM Agence Nationale de Sécurité du Médicament et des produits de santé. 96

BERT Bidirectional Encoder Representations from Transformers. 91, 94

BIO Begin, Inside, Outside. 98

BNF Forme de Backus-Naur. 56

CFG Grammaire hors contexte. 49–51, 65–67

CIH Conseil International d’Harmonisation des exigences techniques pour l’enregistrement des médicaments à usage humain. 96

CISMeF Catalogue et Index des Sites Médicaux de langue Francaise. 96

CNN Convolutional Neural Network. 94

CRF Conditional Random Fields. iv, ix, 91, 92, 98–100, 105, 106, 108–111, 114

CWA Closed World Assumption. 10

DAG Directed Acyclic Graph. 109

DEFT DÉfi Fouille de Textes. 92, 101, 103, 111

DSL Domain Specific Language. 98, 105, 108

E/A modèle Entité-Association. 48

EMA European Medicines Agency. 96

FDA Food and Drug Administration. 96

FOL logique du premier ordre. 10, 11, 15, 20, 29, 34, 35, 104

FST Finite-State Transducer. 97, 102, 114

GED Gestion Electronique de Documents. 1, 108

GED Graph Entity Dependencies. 15, 16, 18, 19, 115

GPT Generative Pre-trained Transformer. 91, 94

GQL Graph Query Language. 12

HeTOP Health Terminology/Ontology Portal. 96

IDMP Identification of Medicinal Products. 1

IE Information Extraction. 5, 46

IFPMA International Federation of Pharmaceutical Manufacturers and Associations. 96

ILN Interface en Langage Naturel. 103, 104, 114

INSERM Institut National de la Santé Et de la Recherche Médicale. 96

LDBC Linked Data Benchmark Council. 34, 128

LDBC SNB LDBC Social Network Benchmark. 128

- LHS** Left-Hand Side. 16, 53, 54, 60, 61, 64
- LLM** Large Language Model. 94, 111
- LPG** Labeled Property Graph. 12, 21, 34, 40
- MedDRA** . 95, 96, 102, 103
- MeSH** Un thésaurus de référence dans le domaine biomédical multilangue fournit par la NLM. <https://mesh.inserm.fr/FrenchMesh/index.htm>. ix, 95–97, 101–103
- NAC** Negative Application Conditions. 16
- NER** Named-Entity Recognition. 91
- NLM** U.S. National Library of Medicine. 95, 96, 101, 148
- OWA** Open World Assumption. 10, 15
- OWL** Web Ontology Language. 15
- PDR** Physician’s Desk Reference. 96
- PoS** Part of Speech. 52
- QA** Question Answering. 104
- RCP** Résumés des Caractéristiques du Produit. 96
- RDF** Resource Description Framework. 12, 14–17, 20, 47, 48, 104, 114, 115
- RDFS** Resource Description Framework Schema. 12, 15–17, 19
- RHS** Right-Hand Side. 16, 53, 54, 60, 61, 64
- RIM/RIMS** Regulatory Information Management System. 1
- RxNORM** . 95
- SENDUP** réSEaux Numériques de Données sémantiques : Utilité et vie Privée. 17, 19
- SETUP** Schema Evolution Through UPdates. 16, 17, 19
- SGBD** Système de Gestion de Base de Données. 2, 4, 5, 10, 12, 15, 21, 22, 35–37, 103, 113, 114
- SNOMED** Une collection organisée de termes médicaux fournissant des codes, des termes, des synonymes et des définitions utilisés dans la documentation et les rapports cliniques. <https://www.snomed.org/>. 95
- SPARQL** Simple Protocol and RDF Query Language. 12, 103, 104
- SQL** Structured Query Language. 14, 25, 35, 103, 104
- SVM** Support Vector Machine. 101
- TAL** Traitement Automatique des Langues. 3, 5, 45, 46, 91, 94, 104
- TF-IDF** Term Frequency-Inverse Document Frequency. 101
- TGD** Tuple-Generating Dependency. 10, 12, 15, 20, 28, 32, 40
- UMLS** Un thésaurus intégrant une terminologie clé, des normes de classification et de codage, ainsi que des ressources associées afin de promouvoir la création de systèmes et de services d’information biomédicale plus efficaces et interopérables, y compris les dossiers médicaux électroniques. <https://www.nlm.nih.gov/research/umls/index.html>. 95, 96, 102
- URI** Uniform Resource Identifier. 12, 47
- W3C** World Wide Web Consortium. 12
- XML** Extensible Markup Language. 40, 48, 96

Nicolas HIOT

Construction automatique de bases de données pour le domaine médical : Intégration de texte et maintien de la cohérence

Résumé La construction automatique de bases de données dans le domaine médical représente un défi majeur pour garantir une gestion efficace de l'information et faciliter les prises de décision. Ce projet de recherche se concentre sur l'utilisation des bases de données graphes, une approche qui offre une représentation dynamique et une interrogation efficace des données et en particulier de leur topologie. Notre projet explore la convergence entre les bases de données et le traitement automatique du langage, avec deux objectifs centraux. Tout d'abord, notre attention se porte sur le maintien de la cohérence au sein des bases de données graphes lors des mises à jour, en particulier avec des données incomplètes et des règles métiers spécifiques. Maintenir la cohérence lors des mises à jour permet de garantir un niveau de qualité de données uniforme pour tous les utilisateurs et de faciliter l'analyse. Dans un monde en constante évolution, nous donnons la priorité aux mises à jour, qui peuvent impliquer des modifications de l'instance pour accueillir de nouvelles informations. Mais comment gérer efficacement ces mises à jour successives au sein d'un système de gestion de base de données graphes ? Dans un second temps, nous nous concentrons sur la structuration d'informations extraites de documents textuels, une source de données majeure dans le domaine médical. En particulier, nous examinons les cas cliniques et de pharmacovigilance, un domaine crucial pour identifier les risques et les effets indésirables associés à l'utilisation des médicaments. Comment détecter l'information dans les textes ? Comment intégrer ces données non structurées de manière efficace dans une base de données graphe ? Comment les structurer automatiquement ? Et enfin, qu'est-ce qu'une structure valide dans ce contexte ? On s'intéresse en particulier à favoriser la recherche reproductible en adoptant une démarche transparente et documentée pour permettre la vérification et la validation indépendante de nos résultats.

Mots clés : base de données graphes, base de connaissances, pharmacovigilance, structuration automatique de données, données intelligentes, contraintes

Automated construction of databases for the medical domain : Integrating text and maintaining consistency

Abstract The automatic construction of databases in the medical field represents a major challenge for guaranteeing efficient information management and facilitating decision-making. This research project focuses on the use of graph databases, an approach that offers dynamic representation and efficient querying of data and its topology. Our project explores the convergence between databases and automatic language processing, with two central objectives. In one hand, our focus is on maintaining consistency within graph databases during updates, particularly with incomplete data and specific business rules. Maintaining consistency during updates ensures a uniform level of data quality for all users and facilitates analysis. In a world of constant change, we give priority to updates, which may involve modifying the instance to accommodate new information. But how can we effectively manage these successive updates within a graph database management system ? In a second hand, we focus on the structuration of information extracted from text documents, a major source of data in the medical field. In particular, we are looking at clinical cases and pharmacovigilance, a crucial area for identifying the risks and adverse effects associated with the use of drugs. But, how can we detect information in texts ? How can this unstructured data be efficiently integrated into a graph database ? How can it be structured automatically ? And finally, what is a valid structure in this context ? We are particularly interested in encouraging reproducible research by adopting a transparent and documented approach to enable independent verification and validation of our results.

Keywords : graph databases, knowledge base, pharmacovigilance, automatic data structuring, smart data, constraints