Techniques coopératives pour l'exploitation des bases de connaissances et passage à l'échelle

Louise Parkin

THÈSE

pour l'obtention du Grade de

Docteur de l'École Nationale Supérieure de Mécanique et d'Aérotechnique

(Diplôme National - Arrêté du 25 mai 2016)

École Doctorale : Sciences et Ingénierie des Systèmes, Mathématiques, Informatique
Secteur de Recherche : Informatique et Applications

Présentée par :

Louise Parkin

Techniques Coopératives pour l'Exploitation des Bases de Connaissances et Passage à l'Échelle

---

Sous la direction de **Brice CHARDIN**, **Allel HADJALI** et **Stéphane JEAN**

Soutenue le 9 décembre 2022 devant la Commission d'Examen

JURY

Présidente :

Anne LAURENT	Professeur, Université de Montpellier, Montpellier

Rapporteurs :

Guy DE TRÉ	Professeur, Ghent University, Ghent
Frédérique LAFOREST	Professeur, INSA Lyon, Lyon

Membres du jury :

Hala SKAF-MOLLI	Maître de Conférences HDR, Université de Nantes, Nantes
Brice CHARDIN	Maître de Conférences, ISAE - ENSMA, Poitiers
Allel HADJALI	Professeur, ISAE - ENSMA, Poitiers
Stéphane JEAN	Maître de Conférences HDR, Université de Poitiers, Poitiers

Remerciements

Au terme de ma thèse, je tiens à remercier toutes les personnes qui ont contribué à son bon déroulement.

— Je remercie **Guy De Tré** et **Frédérique Laforest** d'avoir accepté de rapporter ma thèse et pour les corrections et commentaires précieux qui m'ont permis d'améliorer mon travail. Je voudrais également remercier **Anne Laurent** et **Hala Skaf-Molli**, pour m'avoir fait l'honneur de faire partie de mon jury, et dont les questions ont permis d'enrichir mon travail.

— Un grand merci à mes encadrants de thèse **Brice Chardin**, **Stéphane Jean** et **Allel Hadjali** sans qui ce travail n'aurait pas été possible. Merci d'avoir eu confiance en moi et en mon travail, et d'avoir su trouver les mots pour m'encourager dans les moments difficiles. Merci pour votre disponibilité pour répondre à mes questions et votre engagement dans ma thèse.

— Je tiens à remercier **Mickaël Baron**, qui aurait pu être un quatrième encadrant, pour son aide et son accompagnement, notamment pour les aspects techniques du travail. Je remercie également **Bénédicte Boinot** pour la gestion parfaite des aspects administratifs. Merci à tous les deux également pour la convivialité que vous apportez au laboratoire.

— Merci au personnel de l'ENSMA et de l'école doctorale qui m'a permis de réaliser cette thèse dans de bonnes conditions.

— Merci à **Célia** et **Samy**, dont le travail de stage a contribué à cette thèse, et à **Géraud** et **Ibrahim** dont les travaux de thèse ont permis à la mienne d'exister.

— Merci à tous les membres du laboratoire et en particulier mes collègues d'enseignement : **Brice**, **Frédéric**, **Henri**, **Jorge**, **Michael** et **Stéphane** pour avoir été autant de modèles pour moi et pour m'avoir transmis leur enthousiasme pour l'enseignement. Merci également à **Ladjel** et **Emmanuel** pour m'avoir partagé leurs visions du monde de la recherche.

— Merci à mes ami·e·s doctorant·e·s pour leur accueil et pour les discussions que nous avons pu avoir pour se remotiver mutuellement : **Abir**, **Ayoub**, **Boumediene**, **Chayma**, **Chourouk**, **Houssameddine**, **Ibrahim**, **Ishaq**, **Issam**, **Jennie**, **Jorge**, **Khedidja**, **Luc**, **Matheus**, **Maxime G**, **Maxime P**, **Mohamed**, **Nesrine**, **Nour**, **Quentin**, **Réda**,**Richard**, **Sana**, **Soulimane** et **Thomas**.

— Merci à ma famille et mes amis pour leur soutien inconditionnel et à **Jorge**, **Matheus**, **Richard**, **Maxime** et **Julia** pour la bonne humeur au quotidien.

Table des matières

Introduction

Contexte

La création du Web sémantique dans les années 90 visait à permettre le partage de données entre différentes applications et groupes d'utilisateurs grâce à un modèle commun. Son créateur Tim Berners-Lee disait du Web Sémantique « J'ai fait un rêve pour le Web [dans lequel les ordinateurs] deviennent capables d'analyser toutes les données sur le Web — le contenu, liens, et les transactions entre les personnes et les ordinateurs » (Berners-Lee et Fischetti, 1999). Cette proposition s'est concrétisée avec le développement des langages du Web sémantique : RDF, RDFS, OWL, SPARQL. Elle s'est ensuite propagée à travers de nombreux domaines académiques comme industriels.

Au cœur du Web sémantique se trouvent les *ontologies* qui sont des représentations formelles, consensuelles et référençables des concepts partagés d'un domaine. Les *bases de connaissances (BC)* sont composées d'une ontologie et des données correspondantes. Ces BC peuvent être reliées entre elles, formant le Web des données ouvertes et liées *(Web of Linked Open Data)* (Hogan, 2020).

Aujourd'hui, il existe des bases de connaissances dans un grand nombre de domaines, allant de la culture (Hyvönen et al., 2006) à la santé (UniProt Consortium, 2015) en passant par la politique (Ronzhin et al., 2019). Des bases de connaissances publiques contenant une multitude de renseignements permettent d'extraire des informations concernant des personnes, des entreprises, etc. Les principales bases de connaissances publiques actuelles, comme DBpedia (Lehmann et al., 2015), Wikidata (Vrandečić et Krötzsch, 2014) et YAGO (Pellissier Tanon et al., 2020) contiennent des millions d'instances et des milliards de faits à propos de ces instances.

Dans l'industrie, avec la tendance grandissante d'accumulation de données est apparu le besoin pour des bases de connaissances massives. Toutes les grandes entreprises du Web possèdent désormais leurs propres BC, qui sont utilisées pour améliorer la performance de leurs services. Elles sont particulièrement utiles pour la recherche sur le Web, et sont utilisées dans la majorité des moteurs de recherche et systèmes de recommandation (Guo et al., 2020). Nous pouvons citer comme exemples Google (Dong et al., 2014), Amazon (Dong et al., 2020), ou

encore Alibaba (Luo et al., 2020). Les BC permettent également de faciliter la compréhension d'objets en fournissant des informations contextuelles. Elles sont ainsi utilisées pour le traitement du langage (Suchanek et Preda, 2014), pour l'intégration de données (Saïs, 2007), ou pour l'entrainement de modèles de machine learning dans le cadre du traitement d'images (Krishna et al., 2017).

Problématique

L'utilisabilité des bases de données est considérée comme un enjeu majeur, au même titre que leurs capacités de stockage de l'information (Jagadish et al., 2007). Cet enjeu apparaît majoritairement lors de l'interrogation des bases de données et se manifeste dès la formulation des requêtes. Une base de données doit proposer une interface qui soit un juste milieu en termes d'expressivité : avec trop peu de fonctionnalités, les utilisateurs ne peuvent pas formuler des requêtes complexes ; l'inverse requiert davantage d'expertise et masque les fonctionnalités utiles.

Une fois les requêtes formulées, l'utilisateur peut encore obtenir des réponses inattendues, ne correspondant pas à son besoin. Dans ce cas, le système de gestion de la base de données devrait être capable de fournir une explication de l'origine de ces résultats. Dans le contexte des bases de données, il existe des approches d'explication des réponses non satisfaisantes qui s'appuient sur les données ou sur la requête posée. Pour aller encore plus loin, le système peut proposer une correction, en interaction ou non avec l'utilisateur, afin de retourner des résultats plus appropriés.

Le problème de l'utilisabilité se pose également pour les bases de connaissances. L'utilisation grandissante des BC conduit de plus en plus d'utilisateurs à interagir avec elles pour en extraire des informations. Ces utilisateurs, même s'ils possèdent une expertise dans leur domaine, ne maîtrisent pas nécessairement le fonctionnement d'une BC. Trois obstacles viennent s'ajouter pour complexifier l'interrogation des bases de connaissances.

- Les BC sont structurées et interrogées avec des langages dont le formalisme peut échapper aux utilisateurs novices.
- Les BC peuvent être alimentées par plusieurs sources, et ont souvent une quantité d'informations importante, donc les utilisateurs ont une connaissance partielle du vocabulaire et du contenu de la BC.
- Les BC ne couvrent que partiellement un domaine et peuvent contenir des données erronées, incohérentes ou obsolètes.

Les deux premières difficultés sont liées à des problèmes dans la requête formulée par l'utilisateur, de par son manque de connaissance de la BC ou des outils nécessaires à sa manipulation. Le troisième rappelle que des réponses non satisfaisantes peuvent également être dues à des données erronées ou incomplètes. Dans ce cas corriger la requête ne permettra pas de fournir

les résultats attendus.

Le problème de réponses insatisfaisantes est un problème fréquent. L'étude d'un cas particulier, celui des réponses vides, à la fois dans les bases de connaissances et dans les bases de données relationnelles, a montré la frustration ressentie par des utilisateurs dont les requêtes ne produisent pas de réponses. Stadlera et al. (2022) affirment que le problème des réponses vides demeure un enjeu important : l'analyse de journaux de requêtes sur 27 BC généralistes ou dans le domaine de la biologie, montre que 20% des requêtes uniques syntaxiquement valides produisent des réponses vides. Pour la BC DBpedia, ce ratio s'élève à 41%.

L'absence de réponse n'est cependant pas la seule source possible d'insatisfaction. En effet, l'utilisateur peut obtenir un nombre non nul, mais insuffisant de réponses, et ressentir une frustration similaire. À l'inverse, une requête qui produirait un million de réponses noie l'utilisateur sous des informations non prévues dont il devra extraire les connaissances utiles avec difficulté. Enfin, un utilisateur peut être insatisfait non pas de la quantité de réponses fournies, mais de leur contenu. Une instance particulière qui est attendue peut ne pas figurer parmi les réponses, ou à l'inverse, une réponse inadaptée peut être présente.

Il est également intéressant de considérer des combinaisons de problèmes. Un utilisateur peut avoir d'emblée plusieurs attentes sur le résultat de sa requête. Par exemple, il attend un nombre minimal de résultats et connaît une réponse qui doit en faire partie. En corrigeant incrémentalement une requête, l'utilisateur peut également passer d'une source d'insatisfaction à une autre, par exemple en obtenant des réponses pléthoriques dès qu'il essaie de corriger son problème de réponses vides. Ici de nouveaux problèmes apparaissent à cause de la modification. Ainsi, le traitement d'un problème indépendamment des autres ne permet pas de répondre aux attentes des utilisateurs de façon optimale.

Motivation et contributions

Le problème des réponses vides a été largement étudié à la fois dans les bases de données relationnelles et dans le contexte des bases de connaissances. Les problèmes de présence ou d'absence d'un résultat particulier ont également fait l'objet de travaux dans différents contextes. En revanche, ce n'est pas le cas pour le problème des réponses insuffisantes, ni son complément, le problème des réponses pléthoriques. Dans la manière de les traiter, ces problèmes sont fondamentalement différents du problème des réponses vides : si une requête produit une réponse vide, toutes les requêtes la contenant produisent également des réponses vides, mais cette propriété n'est plus vérifiée lorsque l'on considère un nombre de réponses non nul.

Pour ces problèmes, les approches dominantes concernant davantage les données (classification des résultats, méthode top-k) que la reformulation des requêtes. Comme l'ont montré les travaux sur les réponses vides, l'identification de causes d'échec est une étape importante

pour reformuler les requêtes des utilisateurs qui ne produisent pas les résultats attendus. Une autre limitation des travaux existants est l'absence d'approche permettant de traiter de façon commune différentes sources d'insatisfaction. Un utilisateur identifiant plusieurs problèmes avec les résultats de sa requête devra corriger les problèmes un par un.

Pour le traitement des réponses non satisfaisantes dans les bases de connaissances, nous nous basons sur deux thèses : les travaux de Fokou (2016) concernant les réponses vides et les travaux de Dellal (2019) concernant les réponses pléthoriques. L'objectif de ces travaux est de comprendre pourquoi une requête échoue, ce qui a abouti à la définition de causes d'échec. Il s'agit des parties de la requête, qui, lorsqu'elles sont présentes, impliquent qu'une requête produit des réponses non satisfaisantes. Pour le problème des réponses vides, elles sont appelées MFS (Minimal Failing Subqueries), et pour les réponses pléthoriques, elles sont appelées MFIS (Minimal Failure Inducing Subqueries). Des requêtes alternatives, produites par suppression de contraintes de la requête initiale, sont également proposées : les XSS (maXimal Succeeding Subqueries).

Les travaux de Fokou (2016) introduisent les MFS et XSS pour le problème des réponses vides dans le contexte des bases de connaissances. Ils définissent des algorithmes pour les calculer et proposent la reformulation de requête basée sur les MFS. Les travaux de Dellal (2019) étendent cette approche aux bases de connaissances incertaines, et introduisent le traitement des réponses pléthoriques. Tous deux se limitent au traitement des requêtes conjonctives, qui, bien que majoritaires, excluent les requêtes plus complexes qui représentent au moins 45% des requêtes exécutées dans la réalité (Gallego et al., 2011).

Dans cette thèse, nous avons étendu les notions de causes d'échec ainsi que de requêtes alternatives avec les contributions suivantes.

— Nous avons conçu des algorithmes d'identification de causes d'échec et de requêtes alternatives pour le problème des réponses pléthoriques.
— Nous avons utilisé des propriétés sur les requêtes et sur les données RDF permettant de diminuer le nombre de requêtes à exécuter lors de la détermination des causes d'échec.
— Nous avons intégré les principaux opérateurs SPARQL. Notre méthode, initialement limitée aux requêtes conjonctives, a été étendue pour les requêtes contenant les opérateurs FILTER, UNION et OPTIONAL.
— Nous avons développé un outil qui étend un point d'accès SPARQL permettant d'afficher les causes d'échec et les réponses d'une requête alternative dans le cas de réponses pléthoriques.
— Nous avons proposé une stratégie pour traiter les problèmes impliquant diverses sources d'insatisfaction. Celle-ci se base sur le traitement commun des différents problèmes de réponses non satisfaisantes.

Ces contributions ont été validées par des expérimentations réalisées sur des données et des requêtes synthétiques du banc d'essai WatDiv (Aluç et al., 2014), ainsi que sur des données

réelles de DBpedia (Lehmann et al., 2015) avec des requêtes réelles provenant des journaux de requêtes du projet LSQ (Saleem et al., 2015).

Organisation du mémoire

Cette thèse s'articule en cinq chapitres.

Le premier chapitre présente les notions fondamentales liées aux bases de connaissances, avec leurs différentes utilisations et méthodes de construction. Nous y détaillons le formalisme des bases de connaissances, en décrivant les langages RDF, RDFS, OWL et SPARQL, ainsi que le formalisme spécifique employé dans cette thèse. Nous présentons la notion de raisonnement, permettant l'inférence de nouvelles informations, et terminons par une description des caractéristiques principales des systèmes de gestion des bases de connaissances, ou triplestores.

Dans le deuxième chapitre, nous formalisons les différents types de réponses insatisfaisantes pour des utilisateurs. Nous dressons ensuite un panorama des approches coopératives d'interrogation des données. Nous distinguons les méthodes selon la cause supposée d'insatisfaction des résultats : données ou requêtes. Nous passons en revue les méthodes de provenance, de classement et classification des résultats, d'aide à la construction de requêtes, de reformulation de requêtes et d'identification de causes d'échec. Nous présentons des techniques développées pour les bases de connaissances, mais aussi des techniques dans d'autres domaines liés, comme les bases de données relationnelles. Nous terminons par positionner nos travaux par rapport aux travaux existants : nous proposons une approche basée sur les requêtes, en considérant l'explication de l'échec et la reformulation de requête.

Notre première contribution est détaillée dans le troisième chapitre. Nous nous sommes intéressés au problème des réponses pléthoriques dans les BC. Nous nous basons sur le traitement des réponses vides pour identifier les MFIS et XSS venant expliquer un nombre de réponses trop important. Nous proposons dans ce chapitre des algorithmes de calcul des MFIS et XSS, en appliquant deux propriétés pour diminuer le nombre de requêtes à exécuter. Nous analysons le comportement de ces algorithmes à l'aide d'une étude expérimentale. Nous présentons enfin l'outil SHINY qui enrichit un point d'accès SPARQL et permet aux utilisateurs de manipuler des causes d'échec ou réponses alternatives pour des requêtes produisant des réponses pléthoriques.

Dans le quatrième chapitre, nous étendons l'approche proposée afin d'inclure des spécificités de RDF et SPARQL. Nous commençons par considérer les cardinalités de prédicats afin d'optimiser notre algorithme de recherche des MFIS et XSS. Nous envisageons ensuite l'utilisation des MFIS et XSS pour des requêtes contenant des opérateurs tels que UNION, FILTER, ou OPTIONAL. Chacune de ces extensions fait l'objet d'une évaluation expérimentale.

Le cinquième chapitre aborde les réponses insatisfaisantes dans leur ensemble. Nous dressons des similarités entre les différents problèmes, et montrons que les problèmes de contenu

peuvent être traités par le biais d'un problème de cardinalité. Nous étudions l'utilisation de propriétés de déduction d'échec pour ces problèmes. Nous comparons notre proposition, un algorithme capable de traiter n'importe quel type de réponses insatisfaisantes, avec des techniques existantes spécifiques à un problème. Nous considérons enfin les problèmes combinés où plusieurs types d'échec interviennent ensemble. Nous définissons et comparons deux méthodes pour déterminer des causes d'échec et requêtes alternatives dans ce contexte.

Cette thèse se termine par une conclusion générale où nous mettons en relief l'impact de nos travaux et exposons les perspectives ainsi ouvertes.

Ce mémoire comprend également une annexe, dans laquelle nous fournissons les requêtes utilisées pour nos expériences.

Les travaux présentés dans cette thèse ont fait l'objet des cinq publications suivantes.

Louise PARKIN, Ibrahim DELLAL, Brice CHARDIN, Stéphane JEAN, Allel HADJALI (2020, October). Traitement coopératif du problème des réponses pléthoriques dans les bases de connaissances RDF. In Proceedings of the BDA 2020 conference (pp. 24-25).

Louise PARKIN (2021, August). Cooperative Techniques for Dealing with Unsatisfactory Answers in RDF Knowledge Bases. In 47th International Conference on Very Large Data Bases (VLDB 2021), PhD Workshop, 2021, pp. 4.

Louise PARKIN, Brice CHARDIN, Stéphane JEAN, Allel HADJALI, Mickael BARON (2021, September). Dealing with Plethoric Answers of SPARQL Queries. In International Conference on Database and Expert Systems Applications (DEXA 2021), pp. 292-304. Springer, Cham. DOI : 10.1007/978-3-030-86472-9_27

Louise PARKIN, Brice CHARDIN, Stéphane JEAN, Allel HADJALI, Mickael BARON (2022). A cooperative treatment of the plethoric answers problem in RDF. Knowledge and Information Systems, 64(9) (KAIS), pp. 2481-2514. DOI : 10.1007/s10115-022-01710-8

Louise PARKIN, Brice CHARDIN, Stéphane JEAN, Allel HADJALI (2022, November). Explaining Unexpected Answers of SPARQL Queries. In International Conference on Web Information Systems Engineering (WISE 2022), pp.15. DOI : 10.1007/978-3-031-20891-1_11

Chapitre 1

Concepts fondamentaux des bases de connaissances

Sommaire

Résumé

Dans ce chapitre, nous présentons les notions fondamentales pour comprendre le fonctionnement des bases de connaissances et les formalismes associés, ainsi que la manière de les interroger. Il s'agit d'expliquer comment l'information est représentée dans une base de connaissances et comment un utilisateur peut l'extraire en soumettant des requêtes à un système de gestion des bases de connaissances. Nous présentons également les mécanismes de raisonnement permettant d'inférer des informations supplémentaires.

1.1 Introduction

Depuis l'apparition du Web sémantique dans les années 90, son utilisation s'est propagée à travers le monde académique ainsi que dans de nombreux domaines industriels. Cette utilisation grandissante des bases de connaissances (BC) conduit de plus en plus d'utilisateurs à interagir avec elles pour en extraire des connaissances. La problématique de l'interrogation se pose alors, puisque ces utilisateurs, même s'ils possèdent une expertise dans leur domaine, ne maîtrisent pas nécessairement le fonctionnement d'une BC. Dans nos travaux, nous traitons des problèmes de différence entre le résultat attendu par un utilisateur et le résultat réel produit par sa requête formulée sur une base de connaissances. Afin de présenter l'étude de ces problèmes dans les chapitres suivants, il est d'abord nécessaire de détailler les bases du fonctionnement des bases de connaissances et de leur interrogation.

Dans la section 1.2 nous définissons les concepts de bases de connaissances et d'ontologie, puis nous présentons dans la section 1.3 les outils permettant de les modéliser, en particulier RDF, RDFS et OWL. La section 1.4 présente le langage de requête SPARQL qui permet l'interrogation des BC et la section 1.5 décrit les mécanismes de raisonnement permis par l'utilisation des schémas. Enfin, la section 1.6 décrit le fonctionnement des systèmes de gestion des bases de connaissances.

1.2 Notion de base de connaissances

Les bases de connaissances étant basées sur des ontologies, nous commençons par définir cette notion. Nous donnerons ensuite des exemples d'utilisation des BC, et les différentes méthodes pour les construire.

1.2.1 Définition d'une base de connaissances

La définition usuelle d'une ontologie, proposée par Gruber (1995), est la conceptualisation explicite d'une spécification. Les ontologies permettent de garantir la cohérence entre acteurs d'un domaine qui partagent ainsi un même vocabulaire. D'après cette définition, une ontologie doit respecter des principes de clarté, de cohérence, d'extensibilité, de dépendance minimale

au système de représentation et de restriction aux notions nécessaires. La définition d'une ontologie a ensuite été précisée pour devenir une représentation des concepts partagés d'un domaine avec les propriétés suivantes (Jean, 2007).

— Formelle : une ontologie est définie dans un langage traitable par une machine, elle est basée sur une sémantique qui permet d'en vérifier la cohérence.
— Référençable : les ressources peuvent être identifiées de manière unique à partir de n'importe quel environnement.
— Consensuelle : une ontologie est basée sur des connaissances partagées par les acteurs d'une communauté et validées par des experts.

Les *bases de connaissances*, également appelées *bases de données sémantiques*, sont des sources de données contenant des ontologies, un ensemble de données, et des liens entre les données et les éléments des ontologies (Jean, 2007). Nous donnons maintenant des exemples académiques et industriels de bases de connaissances.

1.2.2 Utilisation des bases de connaissances

Les bases de connaissances sont utilisées dans de nombreux domaines. Il existe des BC massives et généralistes qui proposent une conceptualisation très large, mais peu approfondie, c'est-à-dire touchant de nombreux domaines avec un faible niveau de détail, ainsi que des BC de domaine très détaillées et utilisées par une communauté restreinte.

Dans le monde académique, plusieurs projets de BC se basent sur l'encyclopédie en ligne Wikipédia afin de produire des BC collaboratives et facilement accessibles. Parmi ces projets, nous trouvons DBpedia (Lehmann et al., 2015), YAGO (Suchanek et al., 2007), et WikiData (Vrandečić et Krötzsch, 2014). Plusieurs grandes entreprises du Web possèdent également leur propre graphe de connaissances, comme Google avec Knowledge Vault (Dong et al., 2014) et Amazon avec une BC sur tous les produits existants (Dong et al., 2020).

Il y a également des BC dans des domaines plus spécifiques tels que la culture, avec la BC Sampo de culture finlandaise (Hyvönen et al., 2006), ou le British Museum [1] qui met à disposition les données relatives à ses collections sous forme de BC. Dans le monde de la biologie, UniProt (UniProt Consortium, 2015) est une BC concernant les protéines maintenue par une centaine d'experts et BioPortal (Rubin et al., 2008) est une collection de plus de 300 ontologies de biologie et de médecine. Dans le domaine de la santé, il existe de nombreuses ontologies médicales, liées à différentes maladies et traitements (Gawich et al., 2012). Des états et des organisations utilisent également les bases de connaissances pour mettre à disposition des données publiques. C'est le cas notamment des Pays-Bas (Ronzhin et al., 2019) et de la Corée du Sud (Lee et Park, 2019).

Enfin, il existe des BC synthétiques qui sont créées afin de faire partie de bancs d'es-

1. british-museum.co.uk

sai permettant d'évaluer la performance de divers systèmes. Lehigh University Benchmark (LUBM) (Guo et al., 2005) et Waterloo SPARQL Diversity Test Suite (WatDiv) (Aluç et al., 2014) sont deux BC générées avec des informations autour du monde universitaire que nous avons utilisées dans nos expérimentations. Linked SPARQL Queries (LSQ) est une base de connaissances rassemblant des requêtes réelles soumises par des utilisateurs à des BC comme DBpedia (Saleem et al., 2015). Elle fournit des informations sur la structure des requêtes et sur leur exécution. Dans nos expérimentations, cette BC a servi pour identifier des requêtes permettant de tester nos algorithmes.

1.2.3 Construction des bases de connaissances

Une méthode de construction descendante est utilisée dans beaucoup de domaines. Cela implique de construire d'abord le schéma et les règles de l'ontologie, puis de peupler la BC avec des données. Cette approche est utilisée pour construire les ontologies médicales (Gawich et al., 2012), en faisant intervenir des experts du domaine tout au long de la construction de l'ontologie. La méthode de construction descendante est difficilement exploitable à l'échelle des BC sur le Web. En effet, l'utilisation des données par des communautés diverses entraîne la modification fréquente des données et méta-données. Des méthodes de construction en partant des données sont donc apparues (Auer et al., 2007).

La BC YAGO est construite à partir des données de Wikipédia, et utilise WordNet pour les structurer (Suchanek et al., 2007). Les dimensions formelles et référençables sont garanties par construction, et la dimension consensuelle est assurée par le caractère consensuel des sources de données initiales, Wikipédia et Wordnet. La précision de YAGO a été calculée à plus de 95%. De la même façon, DBpedia est construite à partir du prélèvement automatique des données des *infobox* de Wikipédia (Lehmann et al., 2015).

Il existe également des bases de connaissances construites par une communauté d'utilisateurs, comme Wikidata (Vrandečić et Krötzsch, 2014). Les contributeurs peuvent ajouter ou éditer manuellement des informations, mais également charger des données en lot. Cette facilité d'interaction a permis la création d'une large communauté de plus de 20 000 contributeurs actifs, et une base de connaissances riche, avec plus d'un milliard de triplets en septembre 2022. Par conséquent, Wikidata est davantage une collection d'informations, plutôt qu'un état de la connaissance partagée et approuvée par une communauté. Wikidata n'impose pas de contraintes sémantiques, et la structure de cette BC est complexe avec un grand nombre de classes et un héritage multiple.

Désormais, des nouvelles bases de connaissances sont créées en combinant le contenu de plusieurs BC existantes. La nouvelle version de YAGO (Pellissier Tanon et al., 2020), combine les avantages existants de YAGO avec le volume de données de Wikidata. Le contenu de Wikidata est extrait et structuré dans une hiérarchie de classes plus simple issu de `schema.org`

(Guha et al., 2016).

Les BC sont diverses tant dans leur utilisation que leur construction, mais partagent un formalisme commun. Dans la section suivante, nous présenterons la modélisation des bases de connaissances utilisée dans nos travaux.

1.3 Modélisation des bases de connaissances

Pour définir des bases de connaissances dans le contexte du Web Sémantique, différents langages ont été proposés et standardisés par le World Wide Web Consortium (W3C). Le langage *Ressource Description Framework* (RDF) (Manola et al., 2004), adopté comme standard par le W3C en 1999, permet la modélisation des données des BC. Il est étendu par les langages *RDF Schéma* (RDFS) (Brickley et Guha, 2014) et *Ontology Web Language* (OWL) (Bechhofer et al., 2004) pour la définition des ontologies.

1.3.1 Langage RDF

Le langage RDF permet de décrire des *ressources* grâce à des *triplets RDF*. La notion de ressource se veut générale et englobe à la fois des objets physiques, des éléments qui existent sur le Web, ainsi que des concepts abstraits. Les triplets RDF sont de la forme (sujet, prédicat, objet) où le sujet représente la ressource à décrire, le prédicat est une propriété qui s'applique au sujet et l'objet est la valeur du prédicat pour ce sujet. Par exemple, l'information que l'individu p_1 a 35 ans peut être stockée avec le triplet *(p_1 âge 35)*.

Les éléments d'un triplet peuvent être de trois formes.

— Une IRI *(Internationalized Ressource Identifier)*, qui est formée d'un espace de nom et d'un identifiant et permet de référencer de façon unique un élément.

— Un nœud anonyme, qui permet d'inclure des informations incomplètes dans la BC.

— Un littéral, qui est une valeur comme un entier, un booléen, un flottant, une chaîne de caractères et peut être complété par une indication de type et une balise de langue.

Dans l'exemple précédent, *35* est un littéral et p_1 et *âge* sont des représentations simplifiées d'IRI. Pour l'IRI *http://exemple.com/âge*, l'espace de nom est *http://exemple.com* et l'identifiant est *âge*. Les espaces de nom peuvent être raccourcis en préfixes, par exemple l'IRI précédente peut être notée *exemple:âge*. Par lisibilité, dans la suite des exemples, les préfixes seront généralement omis.

Les nœuds anonymes permettent de référencer des éléments sans IRI. Si nous voulons stocker l'information que p_1 est un patient d'un hôpital anonyme au sens des IRI mais labellisé, nous ajoutons les trois triplets (p_1 *patient* _:b_1), (_:b_1 *rdf:type Hôpital*), (_:b_1 *nom "CHU_ Tours"*). Ainsi, le patient est lié au nom de l'hôpital même si l'hôpital ne possède pas d'IRI. Ici _:b_1 est un nœud anonyme.

sujet	prédicat	objet
p_1	âge	35
p_1	nom	Alex
p_1	patient	_:b_1
_:b_1	rdf:type	Hôpital
_:b_1	nom	CHU_Tours
_:b_1	emploie	d_1
d_1	âge	42
d_2	employeur	_:b_1
d_2	nom	Jane
d_2	âge	60
d_2	rdf:type	Docteur
_:b_1	emploie	d_3

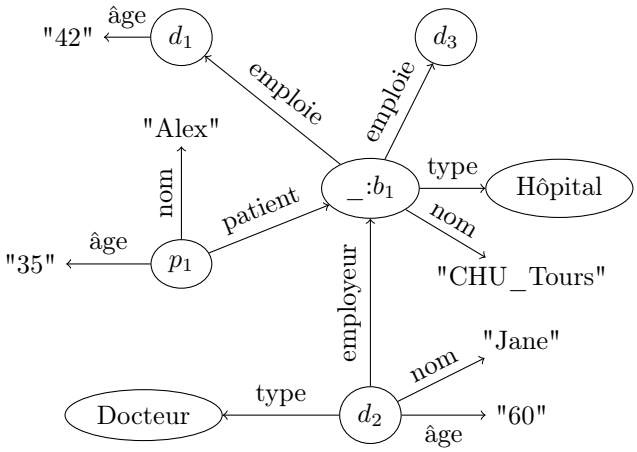


FIGURE 1.1 – Une base de connaissances, sous forme de table et de graphe RDF

Dans un triplet, le sujet peut être une IRI (p_1) ou un nœud anonyme (_:b_1). Le prédicat est nécessairement une IRI (*patient*). L'objet peut être une IRI (*Hôpital*), un nœud anonyme (_:b_1) ou un littéral (*"CHU_ Tours"*).

Définition 1.1 (triplet RDF)**.** *Nous notons I l'ensemble des IRI, B l'ensemble des nœuds anonymes, et L l'ensemble des littéraux. Ainsi, un triplet RDF est formellement défini comme un triplet $(s,p,o) \in (I \cup B) \times I \times (I \cup B \cup L)$. L'union $I \cup B \cup L$ est notée T.*

Comme le montre notre exemple, les prédicats et objets d'un triplet peuvent également être des ressources, ce qui permet récursivement de créer des triplets liés les uns aux autres. Un ensemble de triplets RDF est alors appelé *graphe RDF*. Dans la figure 1.1, nous donnons la représentation d'une base de connaissances contenant les triplets présentés précédemment, plus quelques autres, sous forme de table et sous forme de graphe. Dans ce graphe, les sujets et objets sont des nœuds et les prédicats sont des arêtes orientées du sujet vers l'objet.

Dans le langage RDF, *rdf:type* permet de typer les ressources de la BC. C'est le cas dans le triplet (_:b_1 *rdf:type Hôpital*) de notre exemple. Cependant, RDF ne permet pas de définir une hiérarchie de classes dans la BC. Pour cela, RDF peut être étendu avec un schéma.

1.3.2 Schéma RDFS et OWL

L'intégration d'un schéma dans une base de connaissances permet de structurer les informations présentes. RDF Schema (RDFS) est une extension du vocabulaire de base RDF qui apporte deux contributions : la définition de classes, et la description de propriétés (Brickley et Guha, 2014).

RDF permet de typer un élément avec le prédicat *rdf:type*. RDF Schéma ajoute un vocabulaire supplémentaire qui permet de définir plus précisément des classes avec *rdfs:Class*, *rdfs:Ressource*, *rdfs:Litteral*, et *rdfs:Datatype*. Si la BC contient le triplet (*s rdf:type C*), *s* est

une *instance* de C. Tous les éléments d'une BC RDF sont des ressources et sont des instances de la classe *rdfs:Ressource*.

Pour la description de propriétés, RDF définit les propriétés *rdfs:domain* et *rdfs:range* qui correspondent respectivement au domaine et au co-domaine d'un prédicat, c'est-à-dire à la classe d'appartenance du sujet et de l'objet de ce prédicat. Les propriétés *rdfs:subPropertyOf* et *rdfs:subClassOf* sont des propriétés transitives permettant de définir respectivement une hiérarchie entre propriétés et entre classes. Le triplet (P_2 *rdfs:subPropertyOf* P_1) indique que deux ressources liées par la propriété P_2 sont également liées par la propriété P_1. Le triplet (C_2 *rdfs:subClassOf* C_1) indique que toutes les ressources qui sont des instances de la classe C_2 sont également des instances de la classe C_1. Finalement, les propriétés *rdfs:label* et *rdfs:comment* permettent de fournir pour une ressource un nom et une description compréhensibles par un humain.

Exemple 1.1 (triplets RDF)**.** *Le triplet (patient rdf:type rdf:Property) signifie que patient est une propriété. Le triplet (Hôpital rdf:type rdf:Class) signifie qu'Hôpital est une classe. Les triplets (patient rdfs:domain Personne) et (patient rdfs:range Hôpital) indiquent que les sujets du prédicat patient sont des personnes, et que les objets du prédicat patient sont des hôpitaux. Cela est vérifié pour le triplet (p_1 patient _:b_1), car _:b_1 est une instance de la classe Hôpital, (_:b_1 type Hôpital). En ajoutant le triplet (Hôpital rdfs:subClassOf Lieu), nous indiquons que les hôpitaux sont des lieux.*

OWL (Web Ontology Language) permet d'enrichir le vocabulaire RDFS (Bechhofer et al., 2004). Entre autres, le langage OWL propose des propriétés supplémentaires pour caractériser les classes : *owl:intersectionOf* pour créer une classe à partir de deux classes existantes, *owl:disjointWith* pour indiquer que deux classes ne contiennent pas de ressources communes. Il ajoute également des propriétés concernant les prédicats : *owl:minCardinality* et *owl:maxCardinality* qui définissent respectivement un nombre minimal et maximal d'occurrences d'un prédicat pour une même ressource. Ces propriétés permettent par exemple de spécifier qu'une ressource peut avoir au maximum une valeur pour l'âge en indiquant une *maxCardinality* de 1. Les BC utilisées dans nos travaux utilisent principalement le langage RDFS, mais le vocabulaire de OWL, en particulier concernant les cardinalités, sera évoqué dans le chapitre 4.

1.4 Requêtes SPARQL

Afin d'obtenir des informations d'une base de connaissances, un utilisateur peut formuler des interrogations avec le langage de requêtes standard SPARQL. Le standard W3C définit quatre types de requête SPARQL :

— ASK : retourne un résultat booléen indiquant s'il existe ou non une réponse qui respecte un ensemble de contraintes ;
— CONSTRUCT : retourne les réponses qui respectent un ensemble de contraintes données sous forme d'un nouveau graphe RDF ;
— DESCRIBE : retourne le sous-graphe qui respecte les contraintes fixées dans la requête ;
— SELECT : retourne les réponses qui respectent les contraintes de la requête.

L'analyse de journaux de requête a montré que les requêtes SELECT sont de loin les plus utilisées en pratique (Bonifati et al., 2020). Elles représentent plus de 91% des requêtes distinctes valides, contre 5% de requêtes DESCRIBE, 2% de requêtes ASK et moins de 1% de requêtes CONSTRUCT. Cette répartition a été mesurée sur un corpus de plus de 350 millions de requêtes issues de sept sources de données : DBpedia (Lehmann et al., 2015), Semantic Web Dog Food (Möller et al., 2007), LinkedGeoData (Auer et al., 2009), BioPortal (Rubin et al., 2008), OpenBioMed, British Musuem et Wikidata (Vrandečić et Krötzsch, 2014). Dans la suite du manuscrit, nous considérons uniquement les requêtes SELECT. Nous commençons par présenter les requêtes les plus simples, les requêtes conjonctives, puis nous détaillerons les autres opérateurs SPARQL.

1.4.1 Requêtes conjonctives

Les requêtes SELECT sont de la forme SELECT V WHERE P, où V est un ensemble de *variables* et P est un *patron de graphe*. Pour les requêtes conjonctives, ce patron de graphe est une conjonction de *patrons de triplet*. La sémantique des requêtes SPARQL a été définie par Pérez et al. (2009). Nous donnons les définitions de chacune de ces notions, ainsi que des exemples dans la suite de cette partie.

1.4.1.1 Variables et mappings

Lors de l'exécution d'une requête, nous recherchons les valeurs possibles pour un ensemble de variables qui respectent certaines conditions. Une association de valeurs à un ensemble de variables est appelée *mapping*. Le résultat de l'exécution d'une requête SPARQL de type SELECT est un ensemble de mappings.

Définition 1.2 (mapping)**.** *Un mapping est une fonction partielle μ de V dans T, $\mu : V \to T$. Le domaine de μ, $dom(\mu)$, est le sous-ensemble de V où μ est défini. Pour chaque variable $v \in dom(\mu)$, $\mu(v)$ renvoie la valeur associée à la variable v. La restriction d'un mapping μ à un sous-ensemble de variables $var \subseteq dom(\mu)$ est noté $\mu_{|var}$ et est définie comme une fonction partielle $\mu_{|var} : var \to T$, où $\forall x \in var, \mu_{|var}(x) = \mu(x)$.*

Deux mappings μ_1 et μ_2 sont *égaux* si $\text{dom}(\mu_1) = \text{dom}(\mu_2)$ et $\forall x \in \text{dom}(\mu_1),\ \mu_1(x) = \mu_2(x)$. Deux mappings μ_1 et μ_2 sont *compatibles* si $\forall x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2),\ \mu_1(x) = \mu_2(x)$.

$$\mu_1 : \begin{cases} a \to p_1 \\ b \to age \end{cases} \qquad \mu_2 : \begin{cases} b \to age \\ c \to 35 \end{cases} \qquad \mu_3 : \begin{cases} c \to type \\ d \to Hopital \end{cases}$$

$$\mu_1 \cup \mu_2 : \begin{cases} a \to p_1 \\ b \to age \\ c \to 35 \end{cases} \qquad \mu_1 \cup \mu_3 : \begin{cases} a \to p_1 \\ b \to age \\ c \to type \\ d \to Hopital \end{cases}$$

FIGURE 1.2 – Exemples de mappings

Cela signifie que $\mu_1 \cup \mu_2$ est aussi un mapping. En particulier, deux mappings égaux sont compatibles et deux mappings avec des domaines disjoints sont compatibles. Un mapping μ_1 est *inclus* dans le mapping μ_2, noté $\mu_1 \subseteq \mu_2$, si $\text{dom}(\mu_1) \subseteq \text{dom}(\mu_2)$ et μ_1 et μ_2 sont compatibles. La non-inclusion est notée $\mu_1 \not\subseteq \mu_2$.

Exemple 1.2 (Liens entre mappings)**.** *Considérons un ensemble de variables $V = (a, b, c, d)$, et trois mappings μ_1, μ_2, et μ_3 décrits dans la figure 1.2. Nous avons $dom(\mu_1) = \{a, b\}$, $dom(\mu_2) = \{b, c\}$, $dom(\mu_3) = \{c, d\}$. En notant $B = \{b\}$, la restriction du mapping μ_1 à B est une fonction partielle $\mu_{1|B}$, dont le domaine est B, avec $\mu_{1|B}(b) = age$.*

Les mappings μ_1 et μ_2 sont compatibles, car $dom(\mu_1) \cap dom(\mu_2) = \{b\}$ et $\mu_1(b) = \mu_2(b) = age$. Les mappings μ_1 et μ_3 sont compatibles, car leurs domaines sont disjoints. Les mappings μ_2 et μ_3 ne sont pas compatibles car $dom(\mu_2) \cap dom(\mu_3) = c$ et $\mu_2(c) = 35 \neq \mu_3(c) = type$. Nous avons $\mu_{1|B} \subseteq \mu_2$, car $dom(\mu_{1|B}) = \{b\} \subseteq dom(\mu_2)$ et $\mu_{1|B}$ et μ_2 sont compatibles.

1.4.1.2 Patrons de triplet

Un patron de triplet correspond à un triplet RDF qui contient une ou plusieurs variables. Il servira de condition à respecter pour les résultats de la requête. Considérons les ensembles I, B, L de la définition 1.1 et un ensemble de variables V. Les variables sont identifiées par un nom commençant par " ?".

Définition 1.3 (patron de triplet)**.** *Un patron de triplet est un triplet t (sujet, prédicat, objet) $\in (I \cup B \cup V) \times (I \cup V) \times (I \cup L \cup B \cup V)$. Nous notons $s(t)$, $p(t)$, $o(t)$ et $var(t)$ respectivement le sujet, le prédicat, l'objet et l'ensemble des variables du patron de triplet t.*

L'*évaluation* d'un patron de triplet t sur une base de connaissances D correspond à l'identification des valeurs possibles pour les variables du patron de triplet afin que le triplet obtenu en remplaçant chaque variable par sa valeur appartienne à la BC. Par abus de notation, nous employons la notation suivante :

$$\mu(t) = \big(\mu_i(s(t)), \mu_i(p(t)), \mu_i(o(t))\big) \text{ où } \mu_i(x) : \begin{cases} \mu(x) \text{ si } x \in \text{dom}(\mu) \\ x \text{ sinon} \end{cases}$$

Le résultat de l'évaluation d'un patron de triplet t est alors formellement défini comme $[[t]]_D = \{\mu \mid \text{dom}(\mu) = \text{var}(t) \wedge \mu(t) \in D\}$. Cela revient à rechercher les mappings qui, lorsqu'ils sont appliqués aux variables d'un patron de triplet, lui associent un triplet présent dans la BC.

Exemple 1.3 (patron de triplet)**.** *Le patron de triplet (p_1 ?p ?o) a comme sujet une IRI, comme prédicat la variable p et comme objet la variable o. L'évaluation de ce patron de triplet retournera toutes les propriétés qui s'appliquent au sujet p_1 avec leurs valeurs. Avec notre exemple de la figure 1.1, il y a trois tels couples : (patient, _:b_1), (âge, 35) et (nom, Alex).*

1.4.1.3 Patrons de graphe

Un patron de graphe correspond à un ensemble de patrons de triplet. Dans le cadre des requêtes conjonctives, il s'agit précisément d'une conjonction de patrons de triplet. La conjonction des patrons de triplet t et t' peut être notée $t \wedge t'$, $t.t'$ ou encore tt'.

Les patrons de graphes des requêtes conjonctives peuvent avoir trois formes.

— En étoile : tous les patrons de triplet partagent le même sujet.

— En chaîne : le $n+1^e$ patron de triplet a comme sujet l'objet du n^e patron de triplet.

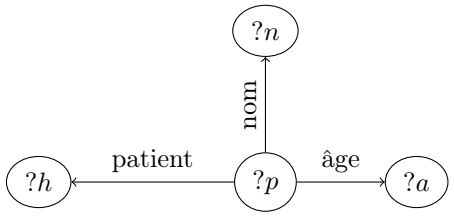
— Composite : toute autre forme de requête.

Dans le cas où un patron de graphe peut être séparé en plusieurs patrons de graphes qui n'ont aucune variable en commun, on parle de *produit cartésien*. Dans ce cas, l'évaluation du patron de graphe est réalisée en évaluant chaque sous-graphe et en réalisant un produit cartésien des réponses.

Exemple 1.4 (patron de graphe)**.** *La figure 1.3 donne un exemple pour chacun de ces cas. Nous identifions les patrons de triplet par t_1, t_2, etc. Le patron de graphe P_1 est en étoile, tous les patrons de triplet ont pour sujet la variable p. Le patron de graphe P_2 est en chaîne, le sujet de t_2 est l'objet de t_1 (variable h) et le sujet de t_3 est l'objet de t_2 (variable d). Le patron de graphe P_3 est composite puisqu'il ne respecte ni la condition des patrons de graphe en étoile ni celle des patrons de graphe en chaîne. Enfin, P_4 est un produit cartésien, puisqu'il peut être séparé en t_1 et t_2 d'une part (variables p, t, n) et t_3 d'autre part (variables h et d).*

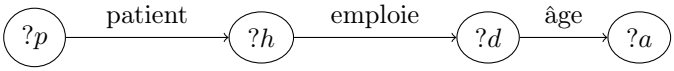
L'évaluation d'un patron de graphe consiste en la recherche de valeurs pour chacune des variables telles que le graphe RDF obtenu en remplaçant chaque variable par sa valeur soit un sous-graphe du graphe RDF. De façon équivalente, cela correspond à identifier des valeurs pour les variables afin que chaque triplet obtenu en remplaçant les variables par leur valeur appartienne à la BC. L'évaluation d'un patron de graphe $P = t_1 \cdots t_n$ sur D est $[[P]]_D = [[t_1]]_D \bowtie \cdots \bowtie [[t_n]]_D$. L'opérateur de conjonction ($\bowtie$) entre deux ensembles de mappings Ω_1 et Ω_2 est défini comme : $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1,\ \mu_2 \in \Omega_2 \text{ sont compatibles}\}$. Le résultat de l'évaluation de chacun des patrons de triplet présenté dans la figure 1.3 sur la BC de la figure 1.1 est donné dans la figure 1.4.

```
{
  ?p patient ?h .  #t1
  ?p nom ?n .      #t2
  ?p age ?a        #t3
}
```



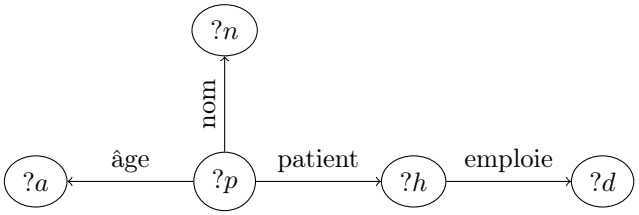
(a) Patron de graphe en étoile P_1

```
{
  ?p patient ?h .  #t1
  ?h emploie ?d .  #t2
  ?d age ?a        #t3
}
```



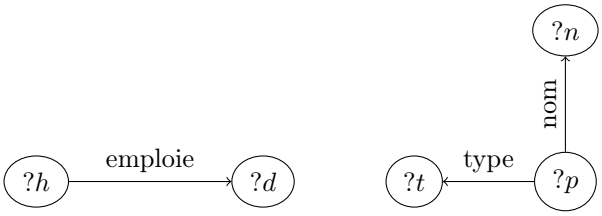
(b) Patron de graphe en chaîne P_2

```
{
  ?p age ?a .      #t1
  ?p nom ?n .      #t2
  ?p patient ?h .  #t3
  ?h emploie ?d    #t4
}
```



(c) Patron de graphe composite P_3

```
{
  ?p type ?t .   #t1
  ?p nom ?n .    #t2
  ?h emploie ?d  #t3
}
```



(d) Produit cartésien P_4

FIGURE 1.3 – Quatre patrons de graphe conjonctifs

?p	?n	?a	?h
p_1	Alex	35	_:b_1

(a) Évaluation de P_1

?p	?h	?d	?a
p_1	_:b_1	d_1	42

(c) Évaluation de P_2

?p	?n	?h	?d	?a
p_1	Alex	_:b_1	d_1	35
p_1	Alex	_:b_1	d_3	35

(b) Évaluation de P_3

?p	?t	?n	?h	?d
_:b_1	Hôpital	CHU_Tours	_:b_1	d_1
d_2	Docteur	Jane	_:b_1	d_1
_:b_1	Hôpital	CHU_Tours	_:b_1	d_3
d_2	Docteur	Jane	_:b_1	d_3

(d) Évaluation de P_4

FIGURE 1.4 – Résultats de l'exécution des patrons de graphe

```
{
  ?p patient ?h .    #t1
  ?p nom ?n .        #t2
  ?p age ?a  .       #t3
  FILTER (?a < 45)   #f1
}
```

(a) Patron de graphe avec FILTER P_5

```
{
  ?p patient ?h .   #t1
  ?h emploie ?d .   #t2
  OPTIONAL
  {?d age ?a}       #t3
}
```

(b) Patron de graphe avec OPTIONAL P_6

```
{{
  ?p patient ?h .   #t1
  ?h emploie ?d     #t2
  } UNION {
  ?p patient ?h .   #t3
  ?d employeur ?h   #t4
}}
```

(c) Patron de graphe avec UNION P_7

FIGURE 1.5 – Trois patrons de graphe avec opérateurs

1.4.2 Autres opérateurs

Les opérateurs proposés par le langage SPARQL sont de deux types. Certains interviennent au niveau des patrons de graphes : FILTER, OPTIONAL, UNION, d'autres sont des modificateurs de résultats qui interviennent en dehors des patrons de graphe.

FILTER permet d'ajouter une contrainte à la requête. La condition de filtre peut contenir des éléments de $V \cup I \cup L$, des constantes, des connecteurs logiques, des opérateurs d'égalité ou d'inégalité, ainsi que des propriétés unaires telles que *bound*. Il est nécessaire que les variables présentes dans la condition de filtre soient également dans le corps de la requête. La figure 1.5a présente la syntaxe de cet opérateur.

OPTIONAL permet de créer une jointure à gauche, c'est-à-dire d'ajouter des informations supplémentaires si elles existent. Dans l'exemple de la figure 1.5b, l'âge des employés est donné si cette information est disponible. Les employés dont l'âge n'est pas renseigné font tout de même partie du résultat et n'auront pas de valeur pour la variable a.

UNION permet de rassembler les résultats correspondant à deux patrons de graphe, qui peuvent ou non partager des variables. La figure 1.5c montre un exemple avec deux propriétés possibles pour désigner les employés : *employeur* et *emploie*.

Les patrons de graphes sont définis récursivement à partir de la définition des patrons de triplet (Pérez et al., 2009).

1. Un patron de triplet est un patron de graphe.
2. Si P_1 et P_2 sont des patrons de graphe, alors $(P_1 \wedge P_2)$, (P_1 OPTIONAL P_2), et (P_1 UNION P_2) sont des patrons de graphe.
3. Si P est un patron de graphe et R est une *condition de filtre* qui respecte la condition $\text{var}(R) \subseteq \text{var}(P)$, alors ($P$ FILTER R) est un patron de graphe.

L'évaluation de ces patrons de graphe est définie par :

— $[[P\ FILTER\ R]]_D = \{\mu \in [[P]]_D,\ \mu \text{ satisfait } R\}$;

— $[[P_1\ UNION\ P_2]]_D = [[P_1]]_D \cup [[P_2]]_D = \{\mu, \mu \in [[P_1]]_D \vee \mu \in [[P_2]]_D\}$;

— $[[P_1\ OPTIONAL\ P_2]]_D = [[P_1]]_D \bowtie [[P_2]]_D \cup [[P_1]]_D \backslash [[P_2]]_D$ où $A_1 \backslash A_2 = \{\mu_1 \in A_1, \forall \mu_2 \in A_2, \mu_1 \text{ et } \mu_2 \text{ ne sont pas compatibles}\}$.

?p	?n	?a
p_1	Alex	35

(a) Évaluation de P_5

?h	?p	?d	?a
_:b_1	p_1	d_1	42
_:b_1	p_1	d_3	

(b) Évaluation de P_6

?h	?p	?d
_:b_1	p_1	d_1
_:b_1	p_1	d_3
_:b_1	p_1	d_2

(c) Évaluation de P_7

FIGURE 1.6 – Résultats de l'évaluation de patrons de graphe avec opérateurs

Exemple 1.5 (évaluation de patrons de graphe avec opérateurs)**.** *La figure 1.6 donne les résultats de l'évaluation des patrons de graphe de la figure 1.5. La comparaison entre le résultat de P_5 et celui de P_1 montre que la deuxième réponse n'apparaît plus, car la contrainte d'âge présente dans le filtre n'est pas respectée. La comparaison entre le résultat de P_6 et celui de P_2 montre l'effet de l'opérateur OPTIONAL, puisqu'une nouvelle réponse apparaît, qui n'avait pas de valeur pour l'âge et a donc une valeur non renseignée pour la variable a. Dans l'évaluation de P_7 il y a deux réponses qui correspondent au premier patron de graphe de l'union et une qui correspond au deuxième patron de graphe de l'union.*

Les modificateurs de résultats ont un fonctionnement différent puisqu'ils n'interviennent pas au niveau des patrons de triplet, mais dans l'affichage des résultats. La forme la plus simple d'une requête est SELECT * WHERE P où P est un patron de graphe. Le symbole * dans la clause SELECT signifie que toutes les variables sont à afficher. Les modificateurs les plus souvent ajoutés sont DISTINCT, GROUP BY, LIMIT, OFFSET et ORDER BY.

- DISTINCT permet de supprimer les doublons dans un ensemble de résultats.
- GROUP BY regroupe les résultats selon la valeur d'une ou plusieurs variables.
- LIMIT permet de fixer un nombre maximal de réponses à afficher.
- OFFSET permet de ne pas afficher les premiers résultats.
- ORDER BY permet d'ordonner les réponses.

Exemple 1.6 (évaluation de requêtes avec modificateurs)**.** *La figure 1.7 montre l'effet de l'ajout de modificateurs sur les résultats d'une requête. Nous partons d'une requête élémentaire basée sur le patron de graphe P_4. La première modification est la sélection des variables (p, n, h). Cela supprime deux colonnes dans l'affichage des résultats (celles des variables t et d) mais ne supprime pas de résultats. Pour retirer les résultats en doublon, l'opérateur DISTINCT est ensuite ajouté. Enfin, l'opérateur ORDER BY classe les résultats par ordre lexicographique de la variable n, puis l'utilisation de l'opérateur LIMIT restreint le nombre de résultats à 1. La deuxième réponse n'est donc pas affichée.*

1.4.3 Formalisation adoptée

Quand nous parlons de requêtes conjonctives, nous considérons des requêtes de la forme Q=SELECT * WHERE P, où P est une conjonction de patrons de triplets : $P = t_1 t_2 ... t_n$.

```
SELECT * WHERE {
  ?p type ?t .  #t1
  ?p nom ?n .   #t2
  ?h emploie ?d  #t3
}
```

?p	?t	?n	?h	?d
_:b_1	Hôpital	CHU_Tours	_:b_1	d_1
d_2	Docteur	Jane	_:b_1	d_1
_:b_1	Hôpital	CHU_Tours	_:b_1	d_3
d_2	Docteur	Jane	_:b_1	d_3

```
SELECT ?p ?n ?h WHERE {
  ?p type ?t .  #t1
  ?p nom ?n .   #t2
  ?h emploie ?d  #t3
}
```

?p	?n	?h
_:b_1	CHU_Tours	_:b_1
_:b_1	CHU_Tours	_:b_1
d_2	Jane	_:b_1
d_2	Jane	_:b_1

```
SELECT DISTINCT ?p ?n ?h WHERE {
  ?p type ?t .  #t1
  ?p nom ?n .   #t2
  ?h emploie ?d  #t3
}
```

?p	?n	?h
_:b_1	CHU_Tours	_:b_1
d_2	Jane	_:b_1

```
SELECT DISTINCT ?p ?n ?h WHERE {
  ?p type ?t .  #t1
  ?p nom ?n .   #t2
  ?h emploie ?d  #t3
}  ORDER BY ?n LIMIT 2
```

?p	?n	?h
_:b_1	CHU_Tours	_:b_1

FIGURE 1.7 – Inclusion des modificateurs de résultats dans la requête et évolution des résultats

Pour simplifier les notations, nous utiliserons la même notation pour la requête associée $Q = t_1 t_2 \cdots t_n$. Nous notons que t est un patron de triplet qui apparaît dans Q comme $t \in Q$. Pour une requête $Q = t_1 \cdots t_n$ et un patron de triplet $t \notin Q$, l'ajout de t à Q est noté $Q \wedge t = t_1 \cdots t_n t$. Cette notation est étendue aux requêtes, donc $Q \wedge Q'$ correspond à la requête conjonctive composée de l'ensemble des patrons de triplets de Q et Q'. A l'inverse pour un patron de triplet $t \in Q$, le retrait de t à la requête Q est noté $Q - t$.

La notion de sous-requête sera importante dans notre travail. Nous les définissions ici pour les requêtes conjonctives, une étude des requêtes avec opérateurs sera réalisée au chapitre 4.

Définition 1.4 (sous-requête, super-requête)**.** *Pour deux requêtes conjonctives Q et Q', Q' est une sous-requête de Q, que nous notons $Q' \subseteq Q$, si et seulement si $\forall t \in Q', t \in Q$. Q est alors une super-requête de Q'. Si $Q' \subseteq Q$ et $Q' \neq Q$, Q' est une sous-requête stricte de Q (et Q est une super-requête stricte de Q'), noté $Q' \subset Q$. Q' est une sous-requête directe de Q (et Q est une super-requête directe de Q') si $Q' \subset Q$ et $\nexists Q'', Q' \subset Q'' \subset Q$.*

Une sous-requête Q' d'une requête $Q = t_1 \cdots t_n$ peut être décrite comme un n-uplet de booléens $(a_1, ..., a_n)$ où a_i est vrai si et seulement si $t_i \in Q'$. Cette modélisation nous permettra d'utiliser des propriétés des fonctions booléennes dans la suite de notre travail.

Pour simplifier les notations, nous ne conservons pas la référence à la BC D dans la notation

règle	triplet(s) existant(s)	triplet déduit
rdfs2	$(s\ p\ o)$ et $(p$ *rdfs:domain* $c)$	$(s$ *rdf:type* $c)$
rdfs3	$(s\ p\ o)$ et $(p$ *rdfs:range* $c)$	$(o$ *rdf:type* $c)$
rdfs7	$(s\ p_1\ o)$ et $(p_1$ *rdfs:subPropertyOf* $p_2)$	$(s\ p_2\ o)$
rdfs9	$(s$ *rdf:type* $c_1)$ et $(c_1$ *rdfs:subClassOf* $c_2)$	$(s$ *rdf:type* $c_2)$

TABLE 1.1 – Exemple de règles de raisonnement RDFS

de l'évaluation, sauf si plusieurs BC sont considérées. Ainsi $[[Q]]_D$ devient $[[Q]]$.

1.5 Raisonnement

Les langages du web sémantique sont bâtis sur deux hypothèses.

— Hypothèse du monde ouvert : tout fait non présent dans la BC peut être vrai.

— Hypothèse de non-unicité des noms : si deux ressources ont des identifiants différents, ce ne sont pas nécessairement deux ressources distinctes.

Sous ces hypothèses, différents raisonnements sont possibles. Ils permettent de déduire des nouveaux triplets à partir des triplets existant dans la base et dans le schéma. Ce raisonnement peut être effectué avant l'exécution des requêtes, c'est la *saturation des données*, ou à la volée au moment de l'interrogation, c'est la réécriture de requête.

1.5.1 Saturation des données

Les règles de raisonnement permettant de déduire de nouveaux triplets sont appelées *règles d'inférence*. RDFS en fournit treize (Hayes et Pater-Schneider, 2014). Nous en donnons quelques-unes dans la table 1.1.

Exemple 1.7 (saturation)**.** *En reprenant l'exemple de la figure 1.1 enrichi d'informations de schéma en bleu dans la figure 1.8, et en utilisant la règle rdfs2, comme la BC contient les triplets (p_1 patient _:b_1) et (patient rdfs:domain Personne), nous pouvons déduire que p_1 est une instance de la classe Personne, et inférer le triplet (p_1 rdf:type Personne). En utilisant la règle rdfs9, comme la BC contient les triplets (_:b_1 type Hôpital) et (Hôpital rdfs:subClassOf Lieu), nous pouvons inférer le triplet (_:b_1 type Lieu).*

Dans la situation de l'exemple, si un utilisateur souhaite obtenir les personnes renseignées dans la base (instances de la classe Personne), p_1 ne serait pas retourné à partir du contenu initial de la BC. L'utilisation de la règle d'inférence et l'ajout du triplet correspondant permet d'obtenir p_1 dans les résultats. L'opération qui complète le contenu de la BC en ajoutant les triplets issus des règles d'inférence s'appelle la saturation. Dans une BC saturée, aucune déduction supplémentaire ne peut être effectuée en utilisant les règles d'inférence. Nous reproduisons notre BC dans la figure 1.8 en y ajoutant les triplets du schéma (en bleu), et les

sujet	prédicat	objet
p_1	âge	35
p_1	nom	Alex
p_1	patient	_:b_1
_:b_1	rdf:type	Hôpital
_:b_1	nom	CHU_Tours
_:b_1	emploie	d_1
d_1	âge	42
d_2	employeur	_:b_1
d_2	nom	Jane
d_2	âge	60
d_2	rdf:type	Docteur
_:b_1	emploie	d_3
patient	rdfs:domain	Personne
patient	rdfs:range	Hôpital
Hôpital	rdfs:subClassOf	Lieu
Docteur	rdfs:subClassOf	Personne
p_1	rdf:type	Personne
d_2	rdf:type	Personne
_:b_1	rdf:type	Lieu

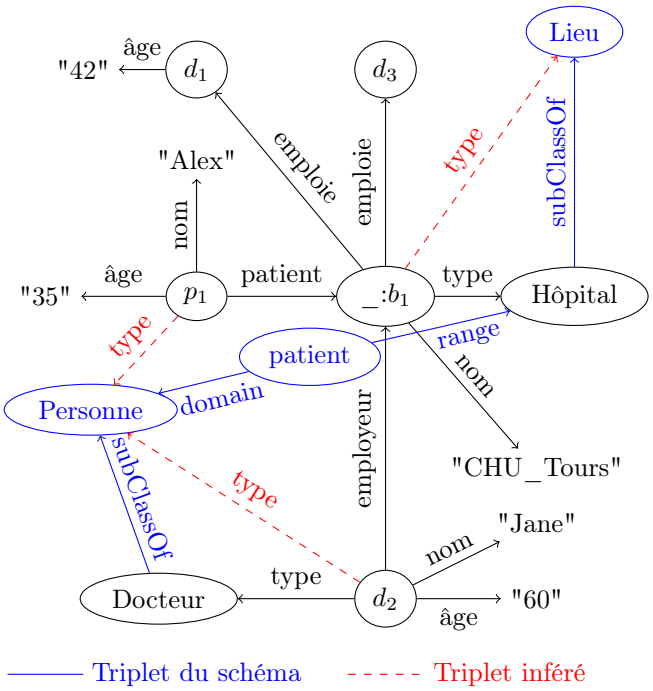


FIGURE 1.8 – Une base de connaissances saturée

triplets inférés (en rouge).

Pour les BC qui utilisent OWL, des règles d'inférence existent également. La notion de complexité du raisonnement décrit la possibilité de concevoir un algorithme qui détermine si une ontologie peut être construite à partir d'une autre en utilisant le raisonnement. Il existe trois versions de OWL qui offrent des expressivités de raisonnement et des degrés de complexité croissants (Horrocks et Patel-Schneider, 2003).

— OWL Lite a l'expressivité la plus faible, n'est pas compatible avec RDFS et a une complexité NEXPTIME.

— OWL DL est plus expressif, mais est toujours incompatible avec RDFS. Sa complexité est EXPTIME.

— OWL Full est le plus expressif et est compatible avec RDFS. Le raisonnement est non décidable, c'est-à-dire qu'il n'existe pas d'algorithme qui détermine si une ontologie peut être construite à partir d'une autre en temps fini.

Les propositions faites dans la suite de cette thèse peuvent s'appliquer aux bases de connaissances saturées ou non saturées.

1.5.2 Reformulation des requêtes

Nous avons décrit dans la section 1.5 les possibilités offertes par les schémas de données de déduire des informations supplémentaires à partir des données de la BC et du schéma. Nous avons illustré les techniques de saturation qui consistent à enrichir les BC avec les triplets

obtenus en utilisant les règles de déduction. Dans le cas où le processus de saturation d'une BC n'est pas réalisé, il est également possible de mettre en place le raisonnement au moment de l'exécution des requêtes (Goasdoué et al., 2012b). Cela implique la réécriture des requêtes afin de prendre en compte les triplets qui peuvent être inférés par le schéma mais qui ne font pas partie de la BC.

Exemple 1.8 (reformulation de requêtes)**.** *Reprenons l'exemple de la partie 3.3, où un utilisateur souhaite obtenir les personnes renseignées dans la base. La requête correspondante est : SELECT * WHERE ?s rdf:type Personne. En utilisant la BC saturée, le triplet (p_1 rdf:type Personne) a été inféré et ajouté à la BC, donc p_1 est retourné en réponse à cette requête. En soumettant cette requête à la BC non saturée, il n'y a pas de résultat.*

En revanche, nous pouvons utiliser la règle d'inférence rdfs2 pour réécrire la requête en : SELECT ?s WHERE { ?s rdf:type Personne} UNION { ?s ?p ?o . ?p rdfs:domain Personne}. Ainsi, nous obtenons à la fois les ressources qui sont des instances de la classe Personne d'après le contenu de la BC, mais aussi les ressources qui sont le sujet d'un prédicat dont le domaine est la classe Personne.

Répondre à une requête est plus rapide avec la méthode de saturation, puisqu'il suffit d'évaluer la requête sur la base déjà saturée. Cependant, la saturation a un coût lors du calcul initial et doit être répétée lors des mises à jour. Goasdoué et al. (2012a) ont calculé que le nombre d'exécutions d'une requête nécessaire pour compenser le coût de la saturation peut dépasser dix millions.

1.6 Systèmes de gestion des bases de connaissances

Les systèmes de gestion des bases de connaissances, aussi appelés *triplestores*, permettent de stocker une BC et à des utilisateurs d'interagir avec. Ces systèmes se distinguent par leur architecture, par leur stockage de l'information et par les techniques d'interrogation proposées. Nous présentons ici les principes fondamentaux du fonctionnement des triplestores, sans chercher à être exhaustif. Diverses études proposent des comparaisons plus approfondies, dont des panoramas généraux des triplestores existants (Ali et al., 2021 ; Özsu, 2016), une comparaison des triplestores distribués (Janke et Staab, 2018) et une étude sur les méthodes de stockage de l'information (Faye et al., 2012).

1.6.1 Architecture des systèmes

Un triplestore peut s'appuyer sur un système de gestion de données existant (par exemple, un système relationnel) pour le stockage des informations. Ces triplestores, dits *non natifs*, permettent de bénéficier des optimisations déjà mises en places dans les systèmes existants (Faye et al., 2012). Beaucoup de triplestores non natifs s'appuient sur des bases de données

relationnelles, c'est le cas pour Virtuoso (Erling et Mikhailov, 2010), basé sur le système de gestion de bases de données relationnelles (SGDBR) Virtuoso et 3store (Harris et Gibbins, 2003) basé sur le SGBDR MySQL par exemple. Il existe également des triplestores qui se basent sur des bases de données clé-valeur comme Trinity.RDF (Zeng et al., 2013). À l'inverse, les triplestores dits *natifs*, sont créés spécifiquement pour contenir des données RDF, ce qui permet de meilleures performances lors du chargement et de la mise à jour des données (Ma et al., 2008). RDF-3X (Neumann et Weikum, 2008) et Jena-TDB (The Apache Software Foundation, 2004) sont deux exemples de triplestores natifs.

Nous pouvons encore distinguer les triplestores *centralisés* et les triplestores *distribués*. Les triplestores distribués permettent de répartir le travail de l'évaluation des requêtes sur plusieurs nœuds et nécessitent de choisir une méthode de partitionnement des données. Dans les triplestores distribués mais non partitionnés comme Neptune (Bebee et al., 2018) et GraphDB (Kiryakov et al., 2005) (anciennement OWLIM), les données sont dupliquées sur chaque nœud. Lorsque le partitionnement des données considère les triplets individuellement, on parle de *partitionnement horizontal*. Celui-ci peut être aléatoire ou basé sur une clé de partitionnement qui correspond à un ou plusieurs éléments parmi sujet, prédicat et objet. Dans ce second cas, on distingue le partitionnement par plage de valeur (Blazegraph ; Systap 2008) et par hachage (4store ; Harris et al. 2009). Le partitionnement des données peut également considérer le graphe entier et appliquer des techniques de clustering pour former des sous-graphes disjoints deux-à-deux. Ce partitionnement est adopté dans les systèmes EAGRE (Zhang et al., 2013) et Trinity.RDF (Zeng et al., 2013).

1.6.2 Stockage de l'information

Le stockage de l'information est une caractéristique centrale des systèmes de gestion des bases de connaissances. Parmi les méthodes de stockage, nous distinguons les méthodes basées sur des tables et les méthodes basées sur des graphes.

Il existe trois méthodes de stockage basées sur des tables.

— Le stockage *vertical*, qui correspond à la représentation des BC dans les figures 1.1 et 1.8, utilise trois colonnes qui correspondent au sujet, au prédicat et à l'objet des triplets avec parfois une quatrième colonne pour indiquer la source de données d'appartenance du triplet. Le triplestore Virtuoso utilise le stockage vertical.

— Le stockage *binaire* correspond à un partitionnement vertical du stockage vertical. Cela implique de créer une table binaire pour chaque prédicat, avec une colonne pour le sujet et une colonne pour l'objet associé au prédicat. Ce système de stockage est utilisé par le triplestore RDF4J (anciennement Sesame) (Broekstra et al., 2002). La figure 1.9 montre la BC de l'exemple 1.1 sous forme binaire.

— Le stockage *horizontal* utilise la séparation en classes des entités et crée une table

âge

sujet	objet
p_1	35
d_1	42
d_2	60

nom

sujet	objet
p_1	Alex
$\_ : b_1$	CHU_Tours
d_2	Jane

type

sujet	objet
$\_ : b_1$	Hopital
d_2	Docteur

emploie

sujet	objet
$\_ : b_1$	d_1
$\_ : b_1$	d_3

patient

sujet	objet
p_1	$\_ : b_1$

employeur

sujet	objet
d_2	$\_ : b_1$

FIGURE 1.9 – Base de connaissances en représentation binaire

par classe avec une colonne pour les instances et une colonne pour chaque propriété applicable à la classe. Le triplestore OntoDB (Jean, 2007) utilise le stockage horizontal.

L'implémentation de méthodes de stockage sous forme de graphe peut être réalisée avec des listes d'adjacence, comme pour les triplestores Hexastore (Weiss et al., 2008) et gStore (Zou et al., 2011), ou plus récemment avec des matrices comme BMatrix (Brisaboa et al., 2020) et Tentris (Bigerl et al., 2020).

Il existe des méthodes de stockage qui ne font partie d'aucune de ces deux catégories. Il s'agit de systèmes non natifs dont la méthode de stockage est dictée par le système support. Par exemple, Stardog (Complexible Inc., 2010) est une BC construite à partir de la base de données clé-valeur RocksDb, et MarkLogic (MarkLogic, 2017) se base sur une base de stockage et d'interrogation de documents XML.

Pour tous les types de triplestores, l'utilisation de techniques d'indexation peut améliorer les performances lors de l'interrogation. Cependant, les index doivent être mis à jour lors de la modification des données, donc il est nécessaire de les choisir judicieusement. Certains triplestores non natifs se basent sur l'indexation proposée dans le système sous-jacent, par exemple 3store qui s'appuie sur MySQL utilise l'optimiseur de requêtes intégré. Il y a également des techniques d'indexation spécifiques pour les données RDF. Le stockage sous forme de table binaire, que nous avons présenté ci-dessus, correspond à une indexation par prédicat, et est adapté pour des requêtes avec des prédicats fixes. Pour s'adapter aux formes de requêtes avec des prédicats variables, la permutation de triplets permet de modifier l'ordre des triplets, habituellement sujet (S), prédicat (P) puis objet (O), et ainsi de réaliser des index dans chacune des permutations SPO, SOP, OSP, OPS, POS, PSO. L'indexation par permutation de triplet est utilisée dans de nombreux triplestores, tels que Virtuoso, Stardog, AllegroGraph (Franz Inc., 2004), Blazegraph ou Jena-TDB.

Dans la table 1.2, nous donnons les dix triplestores les plus populaires, d'après le classement de *db-engines*[2] en septembre 2022, avec leurs caractéristiques. Sur ces dix triplestores, il y a une répartition équilibrée entre les systèmes natifs et non natifs, ainsi qu'entre les systèmes

2. https://db-engines.com/en/ranking/rdf+store

Triplestore	Natif	Partitionnement	Stockage	Langage
MarkLogic	Non natif	Plage	Document	C++
Virtuoso	Non natif	Centralisé	Table verticale	C
Amazon Neptune	Non natif	Non partitionné	Graphe	SAAS
Apache Jena-TDB	Natif	Centralisé	Graphe	Java
GraphDB	Non natif	Non partitionné	Divers	Java
Stardog	Non natif	Centralisé	Clé-valeur	Java
AllegroGraph	Natif	Horizontal	Graphe	Lisp
Blazegraph	Natif	Plage	Graphe	Java
RDF4J	Natif	Centralisé	Table binaire	Java
4store	Natif	Hachage	Table verticale	C

TABLE 1.2 – Caractéristiques des triplestores les plus utilisés

centralisés et distribués. Les méthodes de stockage et de partitionnement sont principalement celles liées à des tables de triplets plutôt qu'aux graphes, les triplestores utilisant des outils liés aux graphes étant plus récents et moins développés. Dans notre travail, nous avons utilisé les triplestores Virtuoso et Jena-TDB, car ce sont les systèmes distribués sous des licences non commerciales les plus utilisées et ils permettent de comparer les performances obtenues avec un triplestore natif et un triplestore non natif.

1.6.3 Interrogation

Nous avons vu que l'interrogation d'une base de connaissances passe habituellement par l'écriture de requêtes SPARQL. La plupart des triplestores proposent une interface de requêtes SPARQL sous la forme d'un point d'accès SPARQL *(SPARQL endpoint)*, ou fournissent des API permettant d'interagir avec la BC depuis un programme. Un point d'accès SPARQL est une interface permettant de soumettre des requêtes SPARQL via le protocole HTTP. Le triplestore Jena-TDB fournit une API pour le langage JAVA, ainsi qu'un point d'accès nommé Fuseki qui accepte des requêtes SPARQL.

Certains triplestores non natifs, comme Virtuoso, proposent une interface de requêtes SPARQL. Ces requêtes SPARQL sont ensuite traduites en requêtes SQL afin d'être exécutées sur la base de données relationnelle sous-jacente. La BC DBpedia, que nous utilisons pour nos expérimentations, fournit un point d'accès SPARQL [3] en utilisant le triplestore Virtuoso.

1.7 Conclusion

Dans ce chapitre, nous avons présenté la modélisation des bases de connaissances avec les langages RDF, RDFS et OWL. Nous avons détaillé les requêtes SELECT du langage SPARQL

3. https://dbpedia.org/sparql

en reprenant la formalisation de Pérez et al. (2009) pour décrire d'abord les requêtes conjonctives, puis les autres opérateurs. Nous avons ensuite introduit une nouvelle formalisation pour les requêtes conjonctives, décrivant la décomposition en sous-requêtes que nous utiliserons par la suite. Finalement, nous avons présenté différents systèmes de gestion des bases de connaissances et leurs caractéristiques afin de justifier des choix d'implémentation que nous avons faits dans cette thèse.

En raison des contraintes de formalisation lors de l'écriture de requêtes SPARQL, associées à une connaissance incomplète des utilisateurs sur la représentation de l'information dans une base de connaissances, des utilisateurs novices peuvent rencontrer des difficultés pour exprimer correctement leur besoin lorsqu'ils interrogent une BC. Pour éviter ce problème, des systèmes coopératifs permettent d'aider les utilisateurs lors de l'écriture des requêtes ou pour en comprendre les résultats. Nous les présentons dans le chapitre suivant.

Chapitre 2

État de l'art sur le traitement des réponses insatisfaisantes

Sommaire

Résumé

Nos travaux concernent le traitement des réponses non satisfaisantes. Dans ce chapitre, nous définissons les cinq problèmes élémentaires de réponses insatisfaisantes, et leurs combinaisons. Nous présentons par la suite les méthodes d'interrogations coopératives qui ont été proposées dans la littérature, et qui permettent d'éviter ou de corriger les réponses non satisfaisantes. Ces méthodes sont séparées en méthodes basées sur les données et méthodes basées sur les requêtes. Nous analysons ces propositions pour montrer leurs limites qui ont motivé notre travail. Celui-ci s'inscrit dans la catégorie des approches orientées requêtes, plus particulièrement dans l'identification des causes d'échec.

2.1 Introduction

Avec la généralisation de l'utilisation des bases de connaissances (BC) dans de multiples domaines (par exemple, la santé, la politique, la culture, etc), des utilisateurs non experts peuvent être amenés à les interroger. En plus d'avoir une connaissance partielle du contenu d'une base de connaissances, les utilisateurs doivent respecter les contraintes de formalisation liées à l'écriture de requêtes SPARQL. Cela peut entrainer des requêtes mal formulées, ne correspondant pas au besoin de l'utilisateur. Le retour de réponses non satisfaisantes crée une frustration chez les utilisateurs, qui doivent modifier leurs requêtes à l'aveugle afin d'obtenir les réponses souhaitées. Différents types de réponses non satisfaisantes peuvent être à l'origine de cette frustration. Les réponses inattendues ont été identifiées comme un des enjeux pour l'utilisabilité des bases de données (Jagadish et al., 2007). Cette problématique a donné lieu à la création de systèmes coopératifs permettant de guider les utilisateurs dans l'interrogation des bases de données.

Il existe cinq raisons élémentaires qui peuvent être source d'insatisfaction chez des utilisateurs d'une base de connaissances.

— La requête ne produit aucune réponse. Ce problème est le premier à avoir été étudié, sous le nom de problème de *réponses vides*.

— La requête produit un nombre trop faible de réponses. C'est le problème des *réponses insuffisantes*.

— La requête produit un nombre trop important de réponses, rendant difficile pour l'utilisateur l'identification des informations utiles. On parle de problème de *réponses pléthoriques*.

— Un résultat attendu n'apparaît pas dans les réponses. C'est le problème de *réponse manquante*.

— Un résultat non désiré fait partie des réponses. C'est le problème de *réponse présente*.

Les techniques d'interrogation coopératives ont d'abord été utilisées pour l'interrogation des données en langage naturel (Kaplan, 1979) et ont, par la suite, été appliquées pour les systèmes de gestion de bases de données (Motro, 1984). Les techniques d'interrogation coopératives aident l'utilisateur à développer sa compréhension de la base de données et à formuler des requêtes qui correspondent à ses besoins. Pour des utilisateurs expérimentés, dont on peut supposer que les requêtes sont écrites correctement, des approches *basées sur les données* permettent de présenter les résultats de façon lisible, ou de détecter des anomalies dans les données disponibles. Au contraire, pour des utilisateurs novices, des approches *basées sur les requêtes* visent soit à aider l'utilisateur dès la formulation de la requête, soit à détecter des anomalies dans une requête soumise, à en identifier les causes et à les réparer. Les travaux existants ne permettent pas de traiter la totalité des problèmes de réponses insatisfaisantes avec une approche basée sur les requêtes. Nos travaux visent à combler cette limitation.

Dans ce chapitre, nous présenterons à la fois des méthodes introduites pour les bases de connaissances, mais également les solutions proposées pour l'interrogation coopérative dans d'autres contextes. En particulier, de nombreux travaux s'intéressent aux bases de données relationnelles et peuvent suggérer des pistes de travaux pour les BC.

Nous commençons par décrire dans la section 2.2 les différents types de réponses non satisfaisantes, en proposant une définition formelle de l'échec avec des fonctions booléennes. Nous présenterons ensuite, dans la section 2.3, les méthodes coopératives pour l'interaction entre un utilisateur et une base de connaissances. Dans la section 2.4, nous analyserons plus particulièrement les approches basées sur les données et dans la section 2.5 les approches orientées requêtes. Nous conclurons ce chapitre en synthétisant les limites de cet état de l'art qui ont motivé nos travaux.

2.2 Typologie des réponses non satisfaisantes

Il existe diverses raisons pour lesquelles un utilisateur peut être insatisfait des résultats d'une requête. Nous définissons des *conditions d'échec* qui sont des fonctions booléennes indiquant si une requête produit des résultats satisfaisants ou non satisfaisants.

Pour une requête Q, nous notons cette condition d'échec $\text{FAIL}(Q)$: $\text{FAIL}(Q) = 1 \iff Q$ échoue. Sa formalisation sous forme de fonction booléenne est notée $\text{FAIL}(x_1, ..., x_n)$, où n est le nombre de patrons de triplet de la requête initiale, et $\forall i,\ x_i = 1 \iff t_i \in Q$. Cette section fournit les définitions des conditions d'échec pour chaque problème identifié.

sujet	predicat	objet
p_1	âge	25
p_1	maladie	brasCassé
p_1	maladie	grippe
p_1	pôle	soinsIntensifs
p_1	état	décédé
p_2	âge	47
p_2	maladie	arrêtCardique
p_2	pôle	Urgences
p_3	âge	33
p_3	maladie	fièvre
p_3	pôle	soinsIntensifs
n_1	service	soinsIntensifs
n_1	soigne	p_3
n_2	service	Urgences
n_2	soigne	p_2
n_2	soigne	p_3
n_3	service	Urgence
n_3	soigne	p_1
n_3	soigne	p_2

(a) Base de connaissances D

```
SELECT * WHERE {
?p pôle soinsIntensifs .   #t1
?p état décédé .           #t2
?n soigne ?p .             #t3
?n pôle soinsIntensifs .   #t4
?p maladie ?i }            #t5
```
Réponses vides

(b) Requête $Q = t_1t_2t_3t_4t_5$

```
SELECT * WHERE {
?p pôle soinsIntensifs .   #t1
?p état décédé .           #t2
?n soigne ?p .             #t3
?n pôle soinsIntensifs }   #t4
```
Réponses vides

(c) Requête $Q' = t_1t_2t_3t_4$

```
SELECT * WHERE {
?p pôle soinsIntensifs .   #t1
?p état décédé .           #t2
?n pôle soinsIntensifs .   #t4
?p maladie ?i }            #t5
```
Réponses pléthoriques

(d) Requête $Q'' = t_1t_2t_3t_5$

```
SELECT * WHERE {
?p pôle soinsIntensifs .   #t1
?p état décédé .           #t2
?p maladie ?i }            #t5
```
Réponses satisfaisantes

(e) Requête $Q''' = t_1t_2t_5$

FIGURE 2.1 – Une base de connaissances et une suite de requêtes

2.2.1 Problèmes élémentaires

Il y a cinq conditions d'échec élémentaires. Elles sont basées sur le résultat de l'exécution d'une requête $[[Q]]$, et peuvent être séparées entre échec basé sur le nombre de résultats, on parle de problèmes de *cardinalité*, et échec basé sur le *contenu* des résultats.

2.2.1.1 Cardinalité

Les conditions d'échec basées sur la cardinalité étudient le nombre de réponses d'une requête. Ainsi, il n'est pas nécessaire de connaître le contenu des réponses pour déterminer si la requête réussit ou échoue. Les conditions d'échec dépendent d'un seuil K qui peut être défini par l'utilisateur ou fixé par le système. Les trois conditions d'échec basées sur les cardinalités sont :

— *Réponses pléthoriques* : $\text{FAIL}_{>K}(Q) = (|[[Q]]| > K)$ pour $K \geq 0$.

— *Réponses insuffisantes* : $\text{FAIL}_{<K}(Q) = (|[[Q]]| < K)$ pour $K > 0$.

— *Réponses vides* : $\text{FAIL}_{\emptyset}(Q) = (|[[Q]]| = 0)$. Il s'agit d'un cas particulier du problème des réponses insuffisantes avec K=1.

Exemple 2.1. *Nous reprenons l'exemple de la base de connaissances de l'hôpital et un utilisateur qui recherche des renseignements sur les patients dans le pôle de soins intensifs. Cette situation est illustrée dans la figure 2.1. Avec sa connaissance contextuelle, l'utilisateur s'attend à un certain nombre de réponses à sa requête. Pour cet exemple minimal, considérons que l'utilisateur s'attend à obtenir au moins deux réponses. Cette attente de l'utilisateur est décrite par la condition d'échec $FAIL_{<2}(Q)$. Nous constatons que les requêtes $t_1t_2t_3t_4t_5$ et t_2t_3*

échouent avec 0 et 1 réponse respectivement. t_4t_5 et $t_1t_2t_5$ réussissent avec 8 et 2 réponses respectivement.

2.2.1.2 Contenu

Les conditions d'échec basées sur le contenu correspondent aux cas où les utilisateurs souhaitent obtenir ou éviter une réponse particulière. Les deux conditions d'échec basées sur le contenu utilisent un mapping fourni par l'utilisateur, noté μ_w. Elles sont :

— *Réponses manquantes* : $\text{FAIL}_{\not\subseteq\mu_w}(Q) = (\forall\mu \in [[Q]] : \mu_w$ *et* μ ne sont pas compatibles)

— *Réponses présentes* : $\text{FAIL}_{\subseteq\mu_w}(Q) = (\exists\mu \in [[Q]] : \mu_w$ *et* μ sont compatibles)

Pour qu'une condition d'échec basée sur le contenu ait du sens, les variables du mapping doivent toutes être présentes dans la requête. Ainsi, $\text{FAIL}_{\not\subseteq\mu_w}(Q)$ et $\text{FAIL}_{\subseteq\mu_w}(Q)$ ne sont pas définies si $\text{dom}(\mu_w) \not\subseteq \text{var}(Q)$. Donc, en créant les sous-requêtes par suppression de patrons de triplet, nous devrons retirer de l'espace de recherche toutes les sous-requêtes ne respectant pas cette condition.

Exemple 2.2. *Dans cet exemple, nous considérons qu'un utilisateur souhaite obtenir des informations sur des patients (variable p) et des infirmiers (variable n). Dans un premier temps, nous considérons l'insatisfaction associée au fait de ne pas obtenir le patient p_2 parmi les valeurs pour les patients. Il s'agit d'un problème de réponse manquante. Dans un second temps, nous considérons l'insatisfaction associée à l'affichage de p_1, qui est un patient, parmi les valeurs pour les infirmiers. Il s'agit d'un problème de réponse présente.*

Pour un mapping μ_w, défini par $dom(\mu_w) = \{p\}$ et $\mu_w(p) = p_2$, $FAIL_{\not\subseteq\mu_w}(Q)$ est une condition d'échec indiquant que les requêtes, où aucune réponse n'a p_2 comme valeur de la variable p, échouent. Dans l'exemple de la figure 2.1, t_1t_3 et t_2t_5 échouent, et t_4t_5 réussit. Leurs ensembles de réponses sont donnés dans la figure 2.2. La requête t_4 n'est pas valide pour ce problème, car elle ne contient pas la variable p du mapping.

Pour un mapping μ_w, défini par $dom(\mu_w) = \{n\}$ et $\mu_w(n) = p_1$, $FAIL_{\subseteq\mu_w}(Q)$ est une condition d'échec indiquant que les requêtes, dont une réponse a p_1 comme valeur pour la variable n, échouent. Dans la figure 2.1, $t_1t_2t_3t_4t_5$ réussit et t_2t_4 échoue. Leurs ensembles de réponses sont donnés dans la figure 2.3. La requête t_1 n'est pas valide pour ce problème, car elle ne contient pas la variable n du mapping.

2.2.2 Déclinaison des problèmes

La problématique de conditions d'échec combinées apparaît lorsque l'utilisateur a plusieurs attentes pour les résultats d'une requête. Cela peut correspondre à un besoin de cardinalité spécifique, qui combine alors une condition d'échec pléthorique et une condition d'échec insuffisante. Par ailleurs, si un utilisateur connaît un ensemble de réponses souhaitées et non

?p	?n
p_1	n_3
p_3	n_1
p_3	n_2

(a) Évaluation de t_1t_3

?p	?i
p_1	brasCassé
p_1	grippe

(b) Évaluation de t_2t_5

?n	?p	?i
p_1	p_1	brasCassé
p_3	p_1	brasCassé
p_1	p_1	grippe
p_3	p_1	grippe
p_1	p_2	arretCardiaque
p_3	p_2	arretCardiaque
p_1	p_3	fièvre
p_3	p_3	fièvre

(c) Évaluation de t_4t_5

FIGURE 2.2 – Résultats de l'exécution des requêtes t_1t_3, t_2t_5 et t_4t_5

?p	?n	?i

(a) Évaluation de $t_1t_2t_3t_4t_5$

?p	?n
p_1	p_1
p_1	p_3

(b) Évaluation de t_2t_4

FIGURE 2.3 – Résultats de l'exécution des requêtes $t_1t_2t_3t_4t_5$ et t_2t_4

souhaitées, ce problème combine des conditions d'échec de réponses manquantes et présentes. Nous donnons un exemple où la correction d'un problème de réponses vides fait apparaître un problème de réponses pléthoriques.

Exemple 2.3. *Reprenons l'exemple de la figure 2.1, avec une succession de requêtes formulées par un utilisateur qui s'attend à obtenir un nombre précis de réponses (pour l'exemple, une ou deux réponses). La première requête (figure b) ne produit aucune réponse. L'utilisateur essaie de la corriger en retirant le patron de triplet t_5, ce qui ne fonctionne pas (figure c) car il y a toujours des réponses vides, puis en retirant le patron de triplet t_3 (figure d). Ici, l'utilisateur obtient des réponses considérées pléthoriques (quatre réponses). Il doit donc modifier de nouveau sa requête pour éviter à la fois les réponses vides et les réponses pléthoriques. Dans la figure 2.4, nous indiquons les sous-requêtes qui échouent pour l'une des deux conditions.*

Afin de traiter tous les problèmes pouvant être construits à partir des cinq conditions d'échec élémentaires et des opérations logiques, nous définissons dans les deux sections suivantes la négation, la conjonction et la disjonction des conditions d'échec. Nous introduisons quelques simplifications élémentaires pour restreindre le champ de problèmes à traiter.

2.2.2.1 Négation

Pour chacun des problèmes, la négation de la condition d'échec s'exprime avec une des autres conditions d'échec :

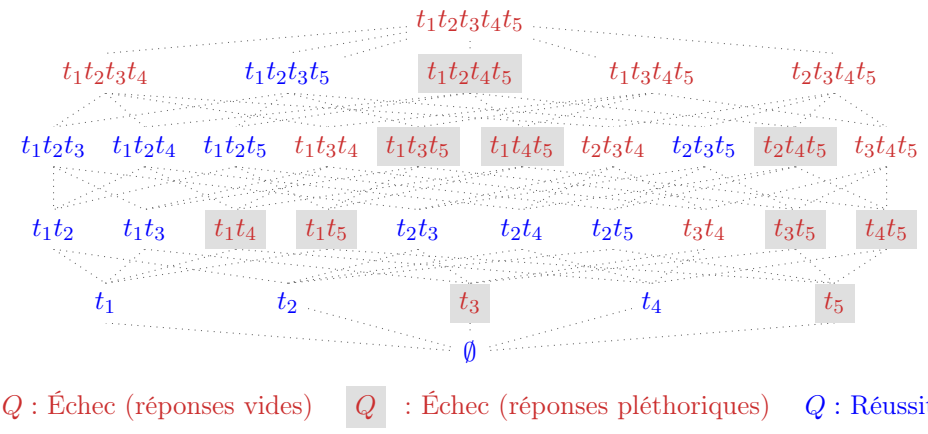


FIGURE 2.4 – Treillis des sous-requêtes de Q

— $\neg(\text{FAIL}_{>K}(Q)) = \text{FAIL}_{<K+1}(Q)$;
— $\neg(\text{FAIL}_{\subseteq\mu_w}(Q)) = \text{FAIL}_{\not\subseteq\mu_w}(Q)$;
— $\neg(\text{FAIL}_{\emptyset}(Q)) = \text{FAIL}_{>0}(Q)$.

Ainsi, pour considérer l'ensemble des conditions d'échec construites à partir des problèmes élémentaires, il ne sera pas nécessaire de traiter la négation individuellement.

2.2.2.2 Conjonction et disjonction

Nous définissons la conjonction et disjonction des conditions d'échec de la façon suivante :

— $(\text{FAIL}_1 \wedge \text{FAIL}_2)(Q) = \text{FAIL}_1(Q) \wedge \text{FAIL}_2(Q)$
— $(\text{FAIL}_1 \vee \text{FAIL}_2)(Q) = \text{FAIL}_1(Q) \vee \text{FAIL}_2(Q)$

La conjonction pour l'échec correspond ainsi à une disjonction pour la réussite, et inversement. Certaines combinaisons peuvent d'emblée être simplifiées. La combinaison de plusieurs problèmes de cardinalité s'exprime comme un problème de cardinalité.

Propriété 2.1. *Pour une requête Q et deux entiers K_1 et K_2 :*

— $(FAIL_{>K_1} \wedge FAIL_{>K_2})(Q) = FAIL_{>max(K_1,K_2)}(Q)$
— $(FAIL_{>K_1} \vee FAIL_{>K_2})(Q) = FAIL_{>min(K_1,K_2)}(Q)$
— $(FAIL_{<K_1} \wedge FAIL_{<K_2})(Q) = FAIL_{<min(K_1,K_2)}(Q)$
— $(FAIL_{<K_1} \vee FAIL_{<K_2})(Q) = FAIL_{<max(K_1,K_2)}(Q)$

Par conséquent, la combinaison de problèmes de cardinalité peut créer un problème trivial, selon les seuils choisis :

— $\text{FAIL}_{>K_1}(Q) \wedge \text{FAIL}_{<K_2}(Q)$ est faux si $K_2 < K_1$
— $\text{FAIL}_{>K_1}(Q) \vee \text{FAIL}_{<K_2}(Q)$ est vrai si $K_2 > K_1$

Les problèmes de conjonction et de disjonction de conditions d'échec n'ont, à notre connaissance, pas été abordés dans la littérature. Nous les étudierons dans le chapitre 5.

2.3 Méthodes coopératives

L'objectif des méthodes coopératives est de permettre à des utilisateurs d'interagir avec une base de connaissances comme ils interagiraient avec une personne, non comme avec une machine. Considérons une question ou requête posée par un utilisateur, qui implique des présuppositions incorrectes ou utilise un formalisme erroné. Un système coopératif devra être capable de fournir non seulement la réponse exacte à la question posée, mais également des réponses autour de cette question, ou une sollicitation à préciser ou corriger la question.

2.3.1 Notion de système coopératif

Les premiers travaux sur les méthodes coopératives sont ceux de Kaplan (1979). Le problème de système non coopératif est illustré par l'échange entre un tel système et un utilisateur.

- **Utilisateur :** Quels étudiants ont obtenu un score F dans la matière CIS500 au printemps 1977 ?
- **Système non coopératif :** ∅
- **Utilisateur :** Existe-t-il un étudiant qui a échoué dans la matière CIS500 au printemps 1977 ?
- **Système non coopératif :** Non
- **Utilisateur :** Combien d'étudiants ont réussi l'examen de CIS500 au printemps 1977 ?
- **Système non coopératif :** 0
- **Utilisateur :** La matière CIS500 était-elle enseignée au printemps 1977 ?
- **Système non coopératif :** Non

Cette interaction montre comment un utilisateur pourrait être induit en erreur par la réponse, pourtant correcte, d'un système non coopératif. Suite à la première question, l'utilisateur pourrait conclure d'un taux de réussite de 100% en 1977 dans la matière CIS500.

Une réponse coopérative que pourrait fournir une personne à qui la première question serait posée est « La matière CIS500 n'existait pas au printemps 1977 », ou bien « Aucun, la matière n'existait pas au printemps 1977 ». L'humain infère la supposition inhérente à la question et répond à cette supposition, en plus ou à la place de répondre à la question posée, tandis que le système non coopératif répond correctement à la question posée, mais cette réponse peut induire en erreur l'utilisateur. La réponse fournie par l'humain est une réponse *corrective*, c'est-à-dire qu'elle corrige une supposition erronée incluse dans la question. Les travaux concernant les méthodes d'interrogation coopératives visent à créer des systèmes qui sont capables de répondre à des questions à la manière d'une personne, en identifiant les suppositions sous-jacentes et en y répondant.

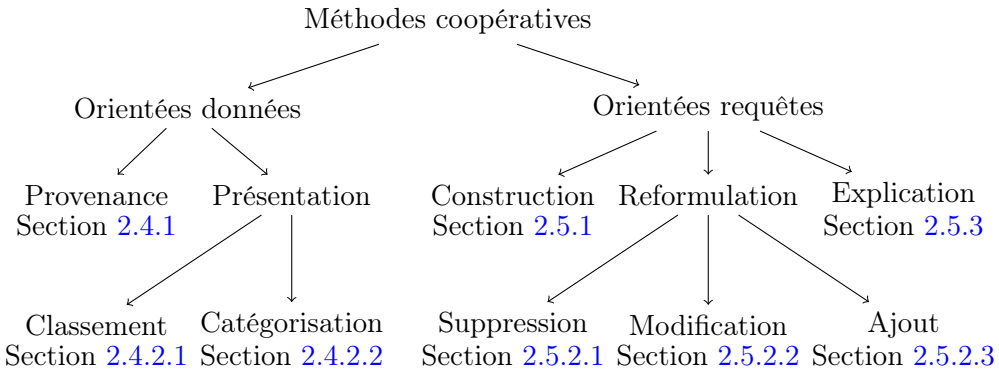


FIGURE 2.5 – Méthodes coopératives

2.3.2 Typologie des systèmes coopératifs

Nous distinguons deux catégories d'approches coopératives selon le type de problèmes relevés. La première fait l'hypothèse que la requête formulée par l'utilisateur correspond à son besoin, mais que la base de connaissances utilisée peut être incomplète, erronée ou massive. La seconde fait l'hypothèse que l'utilisateur peut rencontrer des difficultés pour formuler sa requête et est susceptible de faire des erreurs.

Dans le premier cas, l'utilisation de méthodes orientées données permet d'identifier les causes des résultats obtenus dans les données, on parle alors de *provenance*, ou de présenter les résultats de façon lisible pour l'utilisateur. Dans le second, les méthodes orientées requêtes identifient les raisons des réponses inattendues dans les requêtes. Il existe quelques travaux qui combinent une approche orientée donnée et une approche orientée requête pour créer une approche hybride (Herschel, 2015).

Les approches coopératives orientées requêtes ont également été séparées en deux catégories par Motro (1996).

— Les méthodes qui aident l'utilisateur lors de la formulation de la requête. Ces méthodes visent des utilisateurs qui ne connaissent pas précisément leur besoin ou ne savent pas l'exprimer dans le langage de requête.

— Les méthodes qui identifient des anomalies dans les requêtes formulées par des utilisateurs.

Nous résumons dans la figure 2.5 les différentes catégories de méthodes coopératives. Chacune de ces approches sera détaillée dans les sections suivantes.

2.4 Approches orientées données

Pour les approches orientées données, l'hypothèse est que la requête écrite correspond au besoin de l'utilisateur. Les différences entre les attentes de l'utilisateur et le contenu de la

réponse permettent alors de détecter des problèmes dans la base de connaissances qui peut être incomplète ou incorrecte.

2.4.1 Identification de la provenance des données

Les méthodes d'identification de la provenance des données visent à identifier les sources de données dont sont issues certaines réponses inattendues. Ces techniques peuvent être utilisées dans le cadre du problème des réponses présentes (Meliou et al., 2010 ; Wylot et al., 2014).

La provenance des données est la combinaison de l'origine des données et les transformations qu'elles ont subies (Vicknair, 2010). La problématique de la provenance des données a été étudiée pour des bases de données relationnelles (Buneman et al., 2001 ; Widom, 2004 ; Woodruff et Stonebraker, 1997), pour des entrepôts de données (Cui et al., 2000), ainsi que pour des bases de connaissances RDF (Wylot et al., 2014). Récemment Knorr et al. (2022) ont proposé le premier formalisme pour la provenance des données dans les bases de connaissances hétérogènes intégrant de multiples sources de données.

La notion de *why-provenance* (Cui et al., 2000) associe à un tuple d'une vue ou du résultat d'une requête les tuples présents dans les sources qui permettent l'existence de ce résultat. L'utilisation de polynômes de provenances, dans la méthode de *how-provenance* (Green et al., 2007) permet d'identifier non seulement les tuples dans les sources qui contribuent à une donnée, mais également le niveau de contribution de chacun de ces tuples. Enfin, le terme de *where-provenance* (Buneman et al., 2001) fait référence plus particulièrement aux sources de données dont sont issus les tuples qui contribuent à la présence d'un résultat.

Plusieurs systèmes permettent de stocker la provenance avec les données. Le système TRIO (Widom, 2004) ajoute une couche d'abstraction sur un système de gestion de bases de données relationnelles existant afin d'ajouter des méta-données concernant la précision des données et leur provenance. Ce système est interrogé avec le langage TriQL qui enrichit SQL et permet d'intégrer la provenance dans l'interrogation de la base de données. Le système TripleProv (Wylot et al., 2014) étend un triplestore RDF natif afin de stocker ensemble les instances et les données de provenance. Avec l'explosion du volume des données, le stockage de la provenance devient plus difficile. Ainsi, une solution de calcul paresseuse a été proposée (Woodruff et Stonebraker, 1997). Dans cette méthode, des informations de provenance sont calculées lors de l'exécution d'une requête, uniquement sur les données utilisées. Cette méthode ne permet pas une restitution parfaite de la provenance des données, mais utilise un processus de vérification pour garantir un niveau de qualité de la provenance générée.

La notion de provenance a également été proposée pour la résolution du problème des réponses manquantes dans les bases de données relationnelles (Huang et al., 2008). Dans ce cas, il s'agit d'identifier des données qui pourraient être ajoutées à la base ou modifiées et permettraient alors de faire apparaître les résultats manquants. Ces travaux établissent

une différence entre des sources de données de confiance, qui ne devront pas être éditées, et des sources de données qui peuvent être modifiées. Certaines réponses manquantes sont alors définies comme réponses impossibles, car les faire apparaître nécessiterait de modifier des sources de confiance. Les autres réponses manquantes sont des réponses potentielles et leur provenance est définie comme l'ensemble des tuples existants ou modifiés qui permettent de produire cette réponse.

Le système Artemis (Herschel et Hernández, 2010) étend cette proposition pour un ensemble de requêtes et un ensemble de réponses manquantes. Les utilisateurs peuvent également imposer des tables qui ne devront pas être modifiées, ou choisir de minimiser l'impact des données ajoutées sur les vues. Ce système traite les requêtes SPJUA, c'est-à-dire des requêtes contenant des opérateurs de sélection, projection, jointure, union et agrégation.

Afin d'utiliser la provenance des données pour répondre à des interrogations d'utilisateurs concernant la présence ou l'absence de réponses, Meliou et al. (2010) ont proposé d'identifier des causes aux réponses et aux non-réponses et de leur attribuer un degré de responsabilité dans la présence ou l'absence des réponses. Le degré de responsabilité d'un tuple t est inversement proportionnel au nombre minimal de tuples qui doivent être ajoutés ou retirés de la base de données pour que l'ajout ou le retrait de t entraîne l'apparition ou la disparition de la réponse considérée. Pour des requêtes conjonctives sur des bases de données relationnelles, la causalité peut être déterminée en temps polynomial. En revanche, le problème du calcul des degrés de responsabilité est un problème NP-complet. Cette méthode permet d'ordonner les informations de provenance, qui peuvent être trop nombreuses pour être traitées par les utilisateurs.

2.4.2 Présentation des résultats

Dans cette catégorie, nous trouvons des techniques de modification de la présentation des résultats qui visent à améliorer leur interprétation par les utilisateurs. Ces méthodes sont principalement utilisées pour traiter le problème des réponses pléthoriques. Nous pouvons distinguer les méthodes de classement et de classification. Les méthodes de classement ordonnent les résultats afin de placer les résultats les plus pertinents en premier. Les méthodes de classification regroupent les réponses selon leurs caractéristiques et peuvent présenter une réponse représentative pour le groupe. Dans chaque catégorie, il existe de multiples métriques permettant la classification ou le classement, qui peuvent impliquer plus ou moins l'utilisateur.

2.4.2.1 Classement

Les méthodes de classement sont également appelées méthodes *top-k* (Ilyas et al., 2008). Les résultats sont ordonnés selon un score défini en fonction de critères qui peuvent être fournis par les utilisateurs, basés sur des préférences définies au préalable, ou en exploitant l'historique des requêtes. Les k premiers résultats sont ensuite fournis aux utilisateurs.

Les premières méthodes top-k sont apparues pour les systèmes de recherche d'information, afin d'afficher les pages web obtenues suite à une recherche par ordre de pertinence. Dans ce contexte, trois modèles principaux sont utilisés pour le classement des pages web.

— Le modèle probabiliste implique une estimation de la probabilité qu'une page donnée soit pertinente pour répondre à une recherche (Maron et Kuhns, 1960).

— Le modèle vectoriel représente chaque page d'une collection de résultats sous forme de vecteur (Salton, 1975). Chaque dimension du vecteur correspond à l'importance d'un terme de la collection dans la page considérée avec la mesure TF-IDF (term frequency–inverse document frequency). Cette mesure indique l'importance d'un terme dans un document par rapport à son importance dans les autres documents d'une collection.

— Le modèle basé sur les liens considère l'ensemble des pages web sous forme de graphe et attribue un score à une page en fonction des liens autour de cette page (Brin et Page, 1998).

Ces méthodes top-k dans le domaine de la recherche d'information ont inspiré l'utilisation de méthodes similaires dans les bases de données relationnelles, puis pour les graphes RDF. Pour classer des réponses à des requêtes, la notion de fréquence de termes (TF) n'est pas applicable, car chaque tuple de la réponse contient toutes les valeurs spécifiées dans la requête. En revanche, les valeurs des attributs non présents dans la requête peuvent être utilisées pour ordonner les résultats en attribuant un score plus élevé à des résultats associés à des attributs supplémentaires jugés pertinents (Chaudhuri et al., 2004).

L'importance des attributs peut être déterminée par les utilisateurs. Dans le modèle de Kießling (2002), l'utilisation d'une extension SQL avec une clause PREFERRING permet à l'utilisateur de définir ses préférences dans la requête.

En l'absence de préférences définies par un utilisateur, une stratégie de minimisation du regret combine la méthode de top-k avec les méthodes de skyline (Xie et al., 2020). Les k réponses retournées maximisent la satisfaction minimale d'un utilisateur avec une fonction de préférence quelconque. L'utilisation de la *similarité IDF*, inspirée par TF-IDF dans le domaine de la recherche d'information, permet également d'attribuer un score aux attributs selon leur fréquence d'utilisation (Agrawal et al., 2003).

Le modèle probabiliste a été utilisé pour effectuer un classement automatique des réponses à des requêtes SQL (Chaudhuri et al., 2004). L'estimation de la probabilité de pertinence combine un score global qui retranscrit l'importance d'un attribut dans la base de données, avec un score conditionnel, qui décrit l'importance du lien entre cet attribut et les attributs présents dans la requête.

En parallèle de la notion de pertinence, la diversité au sein d'un ensemble de réponses a été identifiée comme un avantage pour leur utilité (Vieira et al., 2011). Des réponses pertinentes sont les plus similaires à la requête, tandis que les réponses diverses sont peu similaires entre elles. Ces deux notions sont calculées en considérant le recouvrement entre les ensembles de

réponses. Pertinence et diversité sont opposées, donc nécessitent de résoudre un problème d'optimisation avec contraintes (Tao et Zhai, 2006).

Dans le domaine des bases de connaissances, l'utilisation du schéma des données permet d'enrichir les fonctions de score. En étudiant la hiérarchie des classes dans ce schéma, les ressources les plus précises qui se situent en bas de la hiérarchie peuvent être affectée d'un score plus élevé (Ramakrishnan et al., 2005). En combinant le classement de réponses avec un processus de modification de requête basé sur le remplacement de constantes par des variables, Elbassuoni et al. (2009) ont introduit une approche pour traiter le problème des réponses pléthoriques et des réponses insuffisantes.

2.4.2.2 Catégorisation

Les méthodes de classification regroupent les résultats par catégories selon des caractéristiques communes. Les résultats sont alors présentés à l'utilisateur sous la forme d'un représentant par catégorie et l'utilisateur peut alors choisir les catégories les plus adaptées.

Ozawa et Yamada (1995) ont proposé de retourner aux utilisateurs une réponse en *intension*, plutôt qu'en *extension* comme la plupart des systèmes. Ainsi, au lieu d'obtenir une liste de résultats, l'utilisateur obtient une description du contenu des résultats basée sur une classification hiérarchique de classes qui sont étiquetées en langage naturel. En utilisant ce même principe, Chakrabarti et al. (2004) ont proposé une méthode de navigation automatique qui construit à la volée un arbre de navigation. Les résultats sont décomposés à travers l'arbre, dans des catégories imbriquées qui sont définies au fur et à mesure en distinguant les valeurs des attributs. Le choix de l'ordre des attributs et le découpage de leurs valeurs sont effectués avec un modèle de coût qui exploite les requêtes précédentes de l'utilisateur. Ce dernier peut alors parcourir l'arbre de résultats et sélectionner les catégories les plus intéressantes.

Afin d'adapter la catégorisation à un utilisateur particulier, dont les préférences peuvent être inconnues car il interroge la base de données pour la première fois, Chen et Li (2009) forment des groupes de requêtes qui caractérisent chacun un type de préférence utilisateur à partir des journaux de requêtes disponibles. Lorsqu'un nouvel utilisateur soumet une requête, elle est comparée aux groupes existants pour choisir le modèle de coût le plus adapté.

Les méthodes de classement et de classification peuvent être utilisées ensemble (Chen et al., 2012). Cela permet de regrouper des réponses similaires, puis de classer les réponses au sein de chaque groupe. L'outil PREFSKY intègre par ailleurs des techniques de skyline et d'apprentissage automatique pour définir les fonctions de classement et de catégorisation. Les choix de l'utilisateur en termes de sélection des catégories sont utilisés pour enrichir la fonction de classement.

Le tableau 2.1 récapitule les outils présentés dans cette section classés par ordre chronologique. Nous rappelons les méthodes utilisées et le type de problème traité. Les méthodes

Travaux	Type	Méthode	Réponses
Ozawa et Yamada (1995)	Cat.	Hiérarchie données	Pléthoriques
Cui et al. (2000)	Prov.	Tuple	Présente
Buneman et al. (2001)	Prov.	Tuple et source	Présente
Kießling (2002)	Class.	Défini par l'utilisateur	Pléthoriques
Agrawal et al. (2003)	Class.	Poids global	Pléthoriques
Chaudhuri et al. (2004)	Class.	Poids global/local	Pléthoriques
Chakrabarti et al. (2004)	Cat.	Historique utilisateur	Pléthoriques
Ramakrishnan et al. (2005)	Class.	Sémantique	Pléthoriques
Tao et Zhai (2006)	Class.	Pertinence et diversité	Pléthoriques
Huang et al. (2008)	Prov.	Confiance sources	Manquantes
Elbassuoni et al. (2009)	Class/Mod.	Divergence KL	Pléthoriques Insuffisantes
Chen et Li (2009)	Cat.	Historique requêtes	Pléthoriques
Herschel et Hernández (2010)	Prov.	Confiance choix utilisateur	Manquantes
Meliou et al. (2010)	Prov.	Tuple, degré responsabilité	Manquantes Présentes
Vieira et al. (2011)	Class.	Pertinence et diversité	Pléthoriques
Chen et al. (2012)	Cat/Class.	Comportement utilisateur	Pléthoriques
Xie et al. (2020)	Class.	Skyline	Pléthoriques

Cat. : Catégorisation, Prov. : Provenance, Class. : Classification, Mod. : Modification

TABLE 2.1 – Synthèse des approches orientées données

orientées données sont utiles pour corriger des données inexactes ou manquantes, ou expliquer des ensembles de résultats. Cependant, dans le cas où un utilisateur novice doit interroger une base de connaissances, l'hypothèse d'une requête initiale correcte n'est pas assurée. Ainsi, les méthodes orientées requêtes permettent d'accompagner l'utilisateur dans sa création d'une requête, ou de la modifier si les résultats ne correspondent pas à ses attentes.

2.5 Approches orientées requêtes

Les approches orientées requêtes supposent qu'un utilisateur a une connaissance partielle du contenu et du fonctionnement d'une BC. Ainsi, la requête formulée peut contenir des erreurs ou ne pas correspondre au besoin. Dans ce cas, l'écart entre la réponse attendue par l'utilisateur et la réponse obtenue est un indice que la requête n'a pas été formulée correctement. Les techniques d'interrogation coopératives peuvent anticiper ce problème en proposant une aide

à la construction des requêtes, ou intervenir une fois que l'utilisateur détecte une anomalie, afin de corriger la requête ou expliquer à l'utilisateur les résultats obtenus.

2.5.1 Aide à la construction

Les méthodes d'aide à la construction de requêtes permettent aux utilisateurs d'interroger plus simplement une base de connaissances. En s'affranchissant du formalisme SPARQL, ou en guidant l'utilisateur dans son utilisation, les erreurs dans la formulation de requêtes peuvent être limitées. Les méthodes d'aide à la construction incluent les interfaces graphiques de requête, la recherche par mots clés, et la recherche par l'exemple.

Les interfaces de construction de requête proposent des outils graphiques pour construire les requêtes SPARQL. Ces interfaces sont généralement basées sur des formulaires (Borsje et Embregts, 2006) ou des diagrammes (Groppe et al., 2011 ; Hogenboom et al., 2010 ; Russell et al., 2008). Avec le système NITELIGHHT (Russell et al., 2008), les utilisateurs peuvent formuler une requête avec une variante graphique de SPARQL, nommée vSPARQL, qui leur est présentée sous forme de diagramme. Par ailleurs, la requête formée, ainsi que l'ontologie de la base de connaissances, peuvent être visualisées pendant la construction de la requête. Haag et al. (2014) ont également proposé une interface de construction de requêtes purement graphique basée sur le modèle filter/flow. Ce modèle permet une représentation visuelle du processus de requête sous forme de graphe acyclique. Les arêtes du graphe représentent le flux de données et les noeuds définissent les critères pour filtrer les données. Ces interfaces de construction offrent une couche graphique au-dessus de SPARQL pour en masquer le formalisme.

L'interrogation des bases de connaissances par mots clés permet de formuler des requêtes sans préciser de structure. L'utilisation des mots clés a été ajoutée à un système d'interrogation sous forme de patrons de triplets (Arnaout et Elbassuoni, 2018). Ce système utilise également la relaxation de requêtes pour éviter des réponses vides. L'interrogation sans schéma et sans structure propose également une interrogation par mots clés et utilise l'apprentissage automatique pour choisir la requête la plus pertinente à partir de transformations des mots clés (Yang et al., 2014).

L'interrogation par l'exemple a été introduite par Zloof (1975) pour les bases de données relationnelles, puis adaptée à différents contextes, dont les donneés XML (Braga et al., 2005) et les graphes de connaissances (Jayaram et al., 2014). Ces méthodes sont adaptées pour des utilisateurs qui ne connaissent pas suffisamment les termes ou la structure d'une base de données pour formuler une requête, mais qui connaissent un exemple de réponse. Dans l'interrogation par l'exemple des graphes de connaissances, l'utilisateur fournit un tuple qui correspond à une réponse attendue. Le système commence par identifier le sous-graphe qui correspond à cette réponse, puis recherche des sous-graphes isomorphes, dont les nœuds sont proches sémantiquement des nœuds du sous-graphe exemple. Les résultats sont classés selon leur similarité

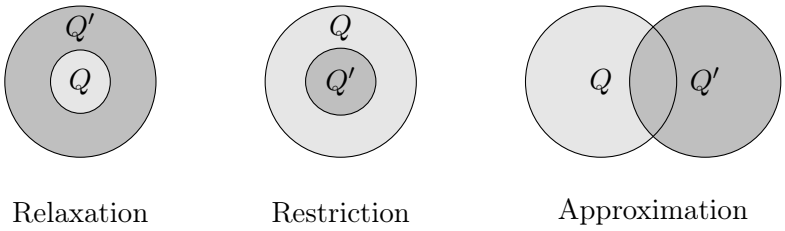


FIGURE 2.6 – Relation entre les réponses d'une requête initiale Q et d'une requête reformulée Q'

avec l'exemple fourni par l'utilisateur avant d'être présentés. Les *requêtes exemples* (Mottin et al., 2014) fonctionnent de façon similaire, avec un processus en deux étapes. D'abord, les éléments de la requête exemple sont identifiés dans le graphe, puis les structures similaires sont identifiées et classées avant d'être retournées comme résultats.

Les approches d'aide à la construction sont utiles pour les utilisateurs avec un faible niveau de maîtrise des bases de connaissances, car elles permettent de masquer le formalisme d'interrogation. Cependant, ces méthodes ne garantissent pas l'absence de réponses insatisfaisantes. Elles peuvent donc être combinées avec des méthodes d'explication de résultats ou de modification de requête en cas de réponses insatisfaisantes.

2.5.2 Reformulation de requête

Les techniques de *reformulation de requêtes* ou de *modification de requêtes* suggèrent des requêtes proches de celle proposée par l'utilisateur, mais dont les réponses peuvent être plus adaptées à son besoin. Elles sont catégorisées selon la relation entre l'ensemble de réponses de la requête de l'utilisateur et celui de la requête modifiée. Les trois configurations possibles sont illustrées dans la figure 2.6.

— Si l'ensemble des réponses de la requête initiale est inclus dans l'ensemble des réponses de la nouvelle requête, on parle de relaxation.
— Si l'ensemble de réponses de la nouvelle requête est inclus dans l'ensemble de réponses de la requête initiale, on parle de restriction.
— Si l'ensemble des réponses de la nouvelle requête inclut une partie des réponses de la requête initiale, ainsi que des nouvelles réponses qui ne font pas partie de l'ensemble de réponses de la requête initiale, on parle d'approximation.

Pour atteindre ces résultats, il y a trois manières de reformuler une requête :

— suppression de contraintes ;
— modification des contraintes présentes ;
— ajout de nouvelles contraintes.

Pour les requêtes SPARQL, cela correspond à la suppression, la modification ou l'ajout de patrons de triplets.

2.5.2.1 Suppression de contraintes

Pour la suppression de contraintes, l'espace de recherche est de taille fixe. Pour une requête SPARQL avec n patrons de triplets, il y a $2^n - 1$ façons de supprimer des contraintes pour former de nouvelles requêtes. Ainsi, pour cette approche, une recherche exhaustive est possible, mais implique un nombre d'exécutions exponentiel. Cette méthode est la première proposée dans le système CO-OP (Kaplan, 1979), où les fausses présuppositions, c'est-à-dire les conditions qui entraînent une réponse vide, sont supprimées.

La recherche exhaustive pour la suppression de patrons de triplets est adoptée pour traiter les problèmes des réponses vides et des réponses pléthoriques dans les bases de données graphe (Vasilyeva et al., 2016). Les algorithmes proposés identifient des sous-graphes connectés maximaux (MCCS) en retirant une à une les parties de la requête initiale afin de créer le plus grand graphe de requête qui réussit. Les sous-graphes maximaux sont retournés comme modification de requête, et leurs compléments sont identifiés comme les explications des réponses vides ou pléthoriques.

La suppression de contraintes est également utilisée pour déterminer les XSS (maXimal Succeeding Subquery), ou requête maximale qui réussit dans le cadre des réponses vides. Cette notion a été introduite dans le domaine des bases de données relationnelles (Godfrey, 1997), puis utilisée dans les bases de connaissances certaines (Fokou et al., 2015) et incertaines (Dellal et al., 2017) pour identifier les sous-requêtes maximales qui produisent des réponses. Les algorithmes proposés permettent d'identifier les XSS sans effectuer une recherche exhaustive. Pour cela, des propriétés de déduction d'échec pour les réponses vides sont exploitées, permettant d'affirmer l'échec d'une requête dont une sous-partie échoue sans devoir l'exécuter.

2.5.2.2 Modification de contraintes

Les méthodes de modification de contraintes ont un espace de recherche potentiellement infini. Dans la pratique, ces approches cherchent à obtenir des requêtes les plus proches possibles de la requête initiale de l'utilisateur, tout en corrigeant les réponses non satisfaisantes. Un enjeu de ces approches est donc de déterminer une notion de similarité entre requêtes afin de retourner les requêtes les plus proches. La similarité est généralement déterminée en fonction de la distance d'édition entre des requêtes (Lee et al., 1993), ou selon la différence entre les ensembles de réponses (Resnik, 1995).

Le système SEAVE (Motro, 1986) intègre un mécanisme de modification de contraintes afin d'éliminer les fausses présuppositions qui entraînent des réponses vides. Ce mécanisme s'applique dans le cadre des données relationnelles, où un treillis de relaxation est généré en considérant la relaxation de chaque élément de la requête. S'il s'agit d'une valeur numérique, elle est remplacée par une valeur plus permissive (plus grande si la condition est une condition d'infériorité et plus petite si c'est une condition de supériorité). S'il s'agit d'une valeur non

Type de relaxation	Requête initiale	Triplet existant	Requête relaxée
Type	$(a\ type\ b)$	$(b\ subType\ c)$	$(a\ type\ c)$
Prédicat	$(a\ p\ b)$	$(p\ subProperty\ q)$	$(a\ q\ b)$
Prédicat par son domaine	$(a\ p\ b)$	$(p\ domain\ c)$	$(a\ type\ c)$
Prédicat par son co-domaine	$(a\ p\ b)$	$(p\ range\ c)$	$(b\ type\ c)$

TABLE 2.2 – Règles de relaxation sémantique

numérique, la condition est supprimée.

Pour le problème des réponses manquantes dans les bases de données relationnelles, le système ConQuer (Tran et Chan, 2010) utilise d'abord une modification des opérateurs de sélection de la requête. Si cela ne permet pas d'obtenir les réponses manquantes, la structure de la requête peut être modifiée. Pour déterminer les requêtes modifiées les plus proches de la requête originale, ConQuer utilise deux métriques :

— métrique sémantique de calcul de la distance d'édition entre les requêtes ;

— métrique de précision qui mesure le nombre de résultats supplémentaires non souhaités.

Les requêtes dominantes, c'est-à-dire celles qui ont un meilleur score sémantique et un meilleur score de précision, sont conservées.

Dans le cadre des bases de connaissances, l'utilisation du schéma des données permet d'identifier des relaxations progressives des requêtes. Hurtado et al. (2006, 2008) ont proposé deux règles qui ne concernent pas la sémantique, et quatre règles issues des règles de raisonnement. La première relaxation est la suppression des constantes, ou plus exactement le remplacement de constantes par des variables. Retirer toutes les constantes d'un patron de triplet revient à retirer ce patron de triplet, on retrouve alors la méthode de suppression présentée dans la section précédente. La deuxième modification non sémantique est la suppression des jointures. Cela signifie que si une variable est partagée par plusieurs patrons de triplets, toutes les occurrences de cette variable sont remplacées par des variables différentes. Les relaxations sémantiques ou ontologiques utilisent les règles présentées dans la table 1.1, et sont détaillées dans la table 2.2.

Huang et al. (2012) utilisent ces règles de relaxation pour traiter le problème des réponses vides et définissent en outre une mesure de similarité pour classer les requêtes relaxées ainsi obtenues. La similarité entre deux classes ou deux propriétés est définie par le rapport logarithmique entre le nombre d'instances de chacune. Si une constante est relaxée en variable, la similarité est de 0. La similarité entre deux patrons de triplet est alors définie comme le tiers de la somme des similarités des sujets, prédicats et objets. La similarité entre deux requêtes est le produit des similarités de chaque patron de triplet. L'utilisateur peut choisir d'affecter une pondération entre les patrons de triplet. Les patrons de triplet jugés les plus importants affecteront davantage le score de similarité. L'algorithme *Best First Search* calcule itérativement les relaxations possibles des requêtes conjonctives et les classe selon leur score de similarité.

Ces requêtes relaxées sont ensuite exécutées dans l'ordre décroissant des scores de similarité, afin d'identifier les requêtes produisant des réponses non vides. Cette méthode a été adaptée par Fokou et al. (2016) pour y intégrer une étape d'explication d'échec.

Il existe des travaux sur la relaxation de requêtes pour d'autres types de données. Song et al. (2019) traitent les problèmes de réponses manquantes et de réponses présentes pour les données sous forme de graphe. Ils proposent des algorithmes exacts de relaxation et de raffinement des requêtes, ainsi que des heuristiques permettant de calculer des requêtes modifiées dans un temps plus court. Ces relaxations sont classées par distance d'édition et contenu des réponses. Les auteurs indiquent comment ces méthodes pourraient être étendues aux problèmes de réponses vides et de réponses pléthoriques.

Dans le domaine des requêtes floues, les contraintes des requêtes sont définies de manière flexible et les résultats sont accompagnés d'un niveau de satisfaction compris entre 0 et 1. Ainsi, le nombre de réponses satisfaisantes dépend du seuil de satisfaction choisi. Les problèmes de réponses vides ou pléthoriques se posent néanmoins lorsque aucune réponse n'apparait avec un niveau de satisfaction non nul ou lorsqu'un nombre trop important de réponses ont un niveau de satisfaction de 1. Des techniques de relaxation et d'intensification sont alors utilisées pour adapter les conditions des requêtes (Moises et Pereira, 2014).

2.5.2.3 Ajout de contraintes

La reformulation de requête par ajout de contraintes a fait l'objet de peu de travaux. L'intégration de nouvelles conditions peut faire diminuer le nombre de réponses d'une requête et ainsi contribuer à la résolution d'un problème de réponses pléthoriques ou de réponses présentes. La recherche de nouveaux prédicats à ajouter a été étudiée pour les requêtes conjonctives dans les bases de données relationnelles (Bosc et al., 2010), puis a été étendue aux requêtes floues (Smits, 2018). L'enjeu de ces approches est d'identifier des nouveaux prédicats à ajouter tout en étant sémantiquement proche de la requête initiale.

Dans l'approche de Smits, les prédicats à ajouter sont déterminés selon leur degré de corrélation avec la requête. La corrélation est comprise entre 0 et 1 et est définie comme le rapport du nombre d'éléments satisfaisant la requête et le nouveau prédicat avec le nombre d'éléments satisfaisant la requête ou le nouveau prédicat. Selon l'application, une valeur idéale γ pour le degré de corrélation est fixée, et les prédicats à ajouter sont ceux dont le degré de corrélation est le plus proche de γ. Plus le degré de corrélation fixé est élevé, plus les prédicats ajoutés seront efficaces pour faire diminuer le nombre de réponses, aux dépens d'un lien éloigné avec la requête existante. À l'inverse, plus le degré de corrélation est bas, plus le prédicat ajouté est proche de la requête initiale, mais plus sa capacité de réduction du nombre de réponses est faible.

Les méthodes de reformulation de requêtes visent à corriger les réponses non satisfaisantes

en proposant des requêtes proches de la requête de l'utilisateur. Cependant, la plupart de ces techniques ne garantissent pas que les requêtes reformulées ne produiront pas également des réponses insatisfaisantes. Ainsi, une étape d'explication de l'échec de la requête initiale permet de concentrer la modification sur les éléments problématiques de la requête.

2.5.3 Explication de l'échec

Les méthodes d'identification des causes d'échec ont pour objectif d'améliorer la connaissance de l'utilisateur de la base de données. En expliquant la présence ou l'absence de réponses, l'utilisateur peut corriger sa requête et progresser dans sa compréhension du fonctionnement de la base de données.

2.5.3.1 Causes d'échec

La première notion de cause d'échec est formalisée par Godfrey (1997) pour le problème des réponses vides dans les bases de données relationnelles. Elle se base sur la notion de fausse présupposition de Kaplan (1979). Les MFS (Minimal Failing Subqueries) sont les sous-requêtes, définies par suppression de patron de triplet de la requête initiale, qui échouent et qui sont minimales. C'est-à-dire qu'aucune de leurs propres sous-requêtes n'échoue. Ici, les requêtes qui échouent sont celles qui produisent des réponses vides. Toutes les requêtes qui contiennent une MFS échouent. Ainsi, pour corriger la requête afin d'obtenir des réponses, il sera nécessaire de corriger chacune des MFS. Si le nombre de MFS est potentiellement exponentiel par rapport à la taille de la requête, l'utilisation d'une propriété de déduction d'échec, concernant les réponses vides, permet d'identifier une MFS en exécutant au plus n requêtes, où n est le nombre de patrons de triplet de la requête initiale. La notion de MFS a été adaptée pour le problème des réponses vides dans les bases de connaissances certaines (Fokou et al., 2015) et incertaines (Dellal et al., 2017).

Dans le cadre du problème des réponses manquantes dans les bases de données relationnelles, des causes d'échec sont également proposées dans le système WHYNOT (Chapman et Jagadish, 2009) puis étendues dans le système NEDEXPLAIN (Bidoit et al., 2014). Ces causes d'échec sont déterminées en construisant un arbre d'exécution de la requête et en identifiant à quel niveau les réponses manquantes sont éliminées du résultat. Ainsi, l'opérateur responsable de l'absence des réponses souhaitées est identifié. Cette méthode est également appliquée dans le cadre des bases de connaissances (Wang et al., 2019). Ici, l'objectif est d'identifier le patron de triplet ou l'opérateur responsable de l'absence d'une réponse. La méthode de recherche des causes d'échec se décompose en trois phases. D'abord, la partie conjonctive de la requête est identifiée. Puis, les causes d'échec sont recherchées dans la partie conjonctive, en considérant l'ensemble de réponses et en ajoutant progressivement les patrons de triplets. À ce niveau, l'absence d'une réponse peut être due à un terme incorrect, c'est-à-dire qu'une

constante utilisée ne correspond pas aux attentes de l'utilisateur, ou à une direction incorrecte, c'est-à-dire que le sujet et l'objet du patron de triplet devraient être échangés. Finalement, si une cause d'échec n'est pas détectée dans la partie conjonctive, les opérateurs sont étudiés pour déterminer lequel entraîne l'absence de réponse.

2.5.3.2 Reformulation basée sur les explications

Les causes d'échec identifiées peuvent être utilisées pour améliorer le processus de reformulation des requêtes en orientant les modifications vers les parties de la requête qui sont à l'origine des réponses non satisfaisantes. Ainsi, les MFS ont été utilisées dans plusieurs systèmes de modification de requête pour éviter les réponses vides. Il existe des systèmes de reformulation automatique (Bosc et al., 2009 ; Jannach, 2006) ainsi que des interfaces de réécriture interactives où les utilisateurs peuvent sélectionner les parties de la requête à changer parmi les causes d'échec. Pour les requêtes RDF, Fokou et al. (2016) utilisent les MFS pour choisir les relaxations à effectuer. Si les MFS sont connues, certaines requêtes relaxées qui ne corrigent pas une MFS retourneront nécessairement des réponses vides. Ainsi, le coût de calcul des MFS est à comparer avec le coût d'exécution des requêtes modifiées que l'identification des MFS permet d'éviter. Une méthode hybride qui calcule les MFS de certaines requêtes au cours de la relaxation est proposée.

Pour les problèmes des réponses manquantes et des réponses présentes pour les requêtes SQL, l'explication de la présence ou de l'absence des réponses est utilisée pour déterminer l'intention de l'utilisateur (Islam et al., 2012). Ces explications sont ensuite utilisées pour modifier la requête afin d'exclure les réponses non désirées et inclure les réponses souhaitées.

2.5.4 Synthèse et positionnement de nos travaux

Le tableau 2.3 rappelle les approches orientées requêtes que nous avons présentées. Pour chacune, nous indiquons les problèmes traités et les outils utilisés. Dans nos travaux, nous considérons que la cause principale des réponses insatisfaisantes est la mauvaise formulation des requêtes par les utilisateurs. Nous nous intéressons donc plus particulièrement aux approches basées sur les requêtes.

La table 2.4 rappelle les approches orientées requêtes existantes pour traiter chacun des problèmes élémentaires. Les méthodes d'aide à la construction, n'étant pas liées à un problème particulier, ne sont pas présentées. S'ils existent, nous donnons un exemple des travaux pour les bases de données relationnelles, et un exemple pour les bases de connaissances (en italique). Les problèmes de réponses insuffisantes et réponses pléthoriques sont peu traités dans les bases de connaissances avec des méthodes orientées requêtes. Notre première contribution vise à proposer une approche orientée requêtes pour les réponses pléthoriques dans les bases de connaissances. Elle sera décrite dans les chapitres 3 et 4. Il existe peu de méthodes d'ex-

Travaux	Type	Méthode	Réponses
Zloof (1975)	Const.	Exemple	-
Kaplan (1979)	SC	Recherche exhaustive	Vides
Motro (1984)	MC	Valeurs numériques	Vides
Godfrey (1997)	SC/Exp.	Déduction	Vides
Jannach (2006)	MC	Explication	Vides
Russell et al. (2008)	Const.	Diagramme	-
Bosc et al. (2009)	MC	Explication	Vides
Chapman et Jagadish (2009)	Exp.	Opérateurs	Manquantes
Tran et Chan (2010)	MC	Sémantique et contenu	Manquantes
Bosc et al. (2010)	AC	Corrélation	Pléthoriques
Islam et al. (2012)	MC/Exp.	Explication	Manquantes Présentes
Huang et al. (2012)	MC	Sémantique et contenu	Vides
Haag et al. (2014)	Const.	filter/flow	-
Yang et al. (2014)	Const.	Mots clés	-
Jayaram et al. (2014)	Const.	Exemple	-
Moises et Pereira (2014)	MC	Sémantique et contenu	Pléthoriques Vides
Bidoit et al. (2014)	Exp.	Opérateurs	Manquantes
Fokou et al. (2015)	SC/Exp.	Déductions	Vides
Fokou et al. (2016)	MC	Explications	Vides
Mottin et al. (2014)	Const.	Exemple	-
Vasilyeva et al. (2016)	SC	Recherche exhaustive	Pléthoriques Vides
Dellal et al. (2017)	SC/Exp.	Déduction	Vides
Arnaout et Elbassuoni (2018)	Const./MC	Mots clés et relaxation	Vides
Smits (2018)	AC	Corrélation	Pléthoriques
Wang et al. (2019)	Exp.	Triplet ou opérateur	Manquantes
Song et al. (2019)	MC	Sémantique et contenu	Manquantes Présentes

Const. : Aide à la construction, SC : Suppression de contraintes, MC : Modification de contraintes, AC : Ajout de contraintes, Exp. : Explication de l'échec

TABLE 2.3 – Synthèse des approches orientées requêtes

Problème	Reformulation de requête	Explication de l'échec
Réponses vides	Godfrey (1997) *Fokou et al. (2016)*	Godfrey (1997) *Fokou et al. (2015)*
Réponses insuffisantes	-	-
Réponses manquantes	Tran et Chan (2010) *Song et al. (2019)*	Bidoit et al. (2014) *Wang et al. (2019)*
Réponses présentes	Islam et al. (2012) *Song et al. (2019)*	Islam et al. (2012)
Réponses pléthoriques	Moises et Pereira (2014) *Vasilyeva et al. (2016)*	-

TABLE 2.4 – Répartition des travaux existants pour les différents problèmes

plication d'échec dans le cadre des bases de connaissances et les solutions dans la littérature ne permettent pas de traiter conjointement différents problèmes. Ainsi, le chapitre 5 de cette thèse visera à proposer une technique unifiée d'explication des réponses non satisfaisantes pour des problèmes combinés.

2.6 Conclusion

L'utilisation des bases de données par un public non expérimenté, couplé au formalisme d'interrogation complexe, est à l'origine de divers problèmes de réponses non satisfaisantes pour les utilisateurs. Dans ce chapitre, nous avons dressé la liste des raisons possibles d'insatisfaction pour des utilisateurs de bases de connaissances, et présenté les méthodes coopératives existantes permettant de les éviter. Nous avons montré que ces outils peuvent fournir des réponses concernant les données ou concernant les requêtes afin d'expliquer ou de corriger les réponses non satisfaisantes.

Nos travaux s'inscrivent dans les méthodes d'explication de l'échec et de reformulation de requête par suppression de contrainte dans la catégorie des approches basées sur les requêtes. Nous nous intéressons à ces méthodes car leur intérêt a été montré pour le problème des réponses vides, mais ces techniques sont peu exploitées pour d'autres types de réponses insatisfaisantes, en particulier dans le contexte des bases de connaissances. Dans le chapitre suivant, nous proposons une méthode d'explication d'échec et reformulation de requête pour le problème des réponses pléthoriques.

Chapitre 3

Traitement des réponses pléthoriques

Sommaire

Résumé

Le problème des réponses pléthoriques a été peu étudié en s'intéressant aux requêtes. Or, les travaux sur le problème des réponses vides ont montré que des erreurs dans l'écriture des requêtes peuvent être à l'origine de réponses non satisfaisantes. Nous abordons

donc le problème des réponses pléthoriques sous l'angle des requêtes. En nous basant sur les notions existantes de causes d'échec, nous proposons une définition adaptée pour ce nouveau problème et identifions des algorithmes de calcul pour les énumérer. Dans ce chapitre, nous détaillons également les choix d'implémentation qui nous permettent de calculer les causes d'échec en temps raisonnable. Nous montrons à travers une évaluation expérimentale que celles-ci peuvent être obtenues en moins d'une seconde pour des bases de données de tailles usuelles et des requêtes variées.

3.1 Introduction

Nous avons vu dans le chapitre précédent les différentes raisons pour lesquelles un utilisateur peut être non satisfait par la réponse à sa requête. Dans ce chapitre, nous nous intéressons spécifiquement au problème des réponses pléthoriques. Dans ce cadre, une requête échoue si son nombre de réponses dépasse un seuil K, qui peut être fixé par l'utilisateur.

Les méthodes les plus utilisées pour ce type de problèmes sont des méthodes de classement ou de regroupement des résultats, telles que le top-k (Ilyas et al., 2008). Ces méthodes permettent à l'utilisateur de parcourir un sous-ensemble des résultats obtenus. Un défaut de ces méthodes est qu'elles ne considèrent pas la requête de l'utilisateur. Ainsi, pour un utilisateur inexpérimenté qui commet des erreurs en écrivant sa requête, ces méthodes ne permettent pas de résoudre le problème sous-jacent. Il conviendrait de modifier la requête afin de retourner les résultats attendus.

Nous voulons donc proposer une approche orientée requêtes basée sur des causes d'échec. Pour le problème des réponses vides, les MFS (Minimal Failing Subqueries) ont été utilisées comme causes d'échec dans plusieurs travaux (Fokou et al., 2016 ; Godfrey, 1997 ; Jannach, 2006 ; McSherry, 2004). Ces travaux ont montré l'intérêt des MFS et de leur complément, les XSS (maXimal Succeeding Subqueries) à la fois pour expliquer l'absence de réponses et pour guider la modification des requêtes. Dans la suite de ce chapitre, nous montrerons que les MFS ne peuvent pas directement être appliquées au problème des réponses pléthoriques, et nous proposerons une nouvelle définition adaptée à notre problème.

Pour des utilisateurs novices qui ne maîtrisent pas complètement le langage d'interrogation SPARQL ou le contenu de la base de connaissances, plusieurs facteurs peuvent se combiner pour produire des réponses pléthoriques.

— Présence d'un produit cartésien : l'utilisateur oublie un prédicat, ou utilise deux variables distinctes plutôt qu'une variable commune, ce qui produit une requête qui peut être séparée en morceaux ne partageant pas de variable.

— Utilisation de prédicats multi-valués : l'utilisateur pense qu'une propriété produit une valeur unique, alors qu'elle peut en produire plusieurs. Par exemple, le lieu de naissance pourrait avoir une valeur unique (ville de naissance) ou être multi-valué pour indiquer

le pays, la région, la ville, etc Un enchaînement de prédicats multi-valués entraînera une multiplication du nombre de réponses.

— Usage de filtres insuffisamment restrictifs : l'utilisateur inclut des patrons de triplets qui ont un grand nombre de valeurs, et les contraintes appliquées par les filtres ne réduisent pas suffisamment le domaine de recherche.

Nous chercherons à identifier ces types de problèmes afin de guider l'utilisateur dans la reformulation de sa requête.

L'objectif de ce chapitre est de proposer une approche permettant d'identifier et de calculer les causes d'échec d'une requête retournant des réponses pléthoriques. Dans la section 3.2 nous proposons et illustrons les définitions des notions de causes d'échec et de requêtes alternatives pour ce problème. Nous introduisons ensuite, dans la section 3.3, trois algorithmes pour les énumérer en intégrant des optimisations visant à minimiser le nombre de requêtes à exécuter. La section 3.4 est dédiée aux choix d'implémentation pour optimiser le temps d'exécution des algorithmes. Enfin, nous montrons l'intérêt de nos propositions dans la section 3.5 en présentant une évaluation expérimentale sur des données synthétiques et réelles.

3.2 Causes d'échec pour les réponses pléthoriques

Afin d'expliquer les réponses pléthoriques, nous cherchons à identifier des causes d'échec, c'est-à-dire des parties de la requête initiale pour lesquelles toute requête qui les contient produit nécessairement des réponses pléthoriques. Nous commençons par illustrer la notion équivalente pour le problème des réponses vides avant de montrer qu'elle ne s'applique pas aux réponses pléthoriques. Nous proposons donc une nouvelle définition de cause d'échec pour ce problème.

3.2.1 MFS et XSS pour les réponses vides

La figure 3.1a donne un extrait d'une base de connaissances. Considérons un utilisateur souhaitant trouver des informations sur les médecins, infirmiers et patients d'un hôpital, et qui soumet ainsi la requête présentée dans la figure 3.1b. Ici, l'utilisateur souhaite retrouver les médecins (variable d) avec leur nombre d'années d'expérience (e), les patients qu'ils traitent (p), ainsi que les infirmiers qui travaillent avec eux (n) et les patients de ces infirmiers (pt). Cette requête ne retourne aucune réponse. Pour déterminer quelle partie de la requête est à l'origine de l'absence de réponses, nous nous appuyons sur le treillis des sous-requêtes présenté en figure 3.1c. Sur ce treillis, les requêtes produisant des réponses vides sont indiquées en rouge, et celles qui produisent des réponses sont en bleu.

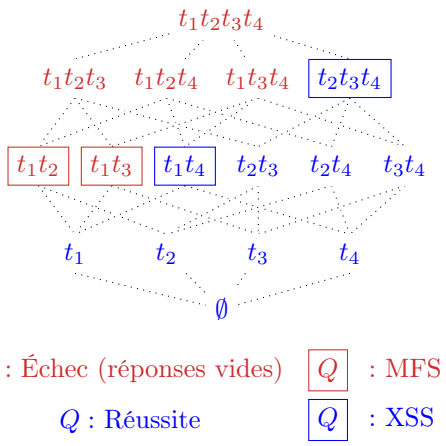
La notion de MFS *(Minimal Failing Subquery)* a été introduite pour le problème des réponses vides. Une MFS est une requête qui échoue dont toutes les sous-requêtes réussissent.

sujet	prédicat	objet
d_1	expérience	14
d_1	collègue	n_3
d_1	collègue	n_2
d_1	traite	p_1
d_1	traite	p_2
d_2	expérience	25
d_2	collègue	n_2
d_2	traite	p_3
d_3	collègue	n_1
n_1	type	InfirmierBloc
n_1	soigne	p_1
n_1	roleOpératoire	Instrumentiste
n_2	type	InfirmierUrgence
n_2	soigne	p_2
n_2	expérience	8
n_3	type	InfirmierUrgence
n_3	soigne	p_2
n_3	soigne	p_3
n_3	soigne	p_4

(a) Base de connaissances D

```
SELECT * WHERE {
  ?d soigne ?p .       # t1
  ?d experience ?e .   # t2
  ?d collegue ?n .     # t3
  ?n soigne ?pt }      # t4
```

(b) Requête $Q_0 = t_1t_2t_3t_4$



(c) Treillis des sous-requêtes de Q_0

FIGURE 3.1 – Une base de connaissances, une requête SPARQL et son treillis de sous-requêtes

Ici l'échec correspond au fait de produire des réponses vides. D'après une propriété sur les réponses vides (Godfrey, 1997), toutes les super-requêtes des MFS échouent, car une super-requête d'une requête qui produit des réponses vides produit nécessairement des réponses vides elle aussi. De façon complémentaire, une XSS *(maXimum Succeding Subquery)* est une requête qui réussit dont toutes les super-requêtes échouent.

L'exemple précédent comprend deux MFS : t_1t_2 et t_1t_3 et deux XSS : t_1t_4 et $t_2t_3t_4$. Les MFS sont les plus petites requêtes qui échouent, c'est-à-dire dans ce cas qui produisent des réponses vides, et les XSS sont les plus grandes requêtes qui réussissent. Les MFS s'expliquent car `soigne` est un prédicat qui s'applique aux infirmiers alors que `expérience` et `collègue` sont des prédicats qui s'appliquent aux médecins. Ainsi, il n'existe aucun élément dans la base de connaissances qui puisse être sujet de t_1 et de t_2 ou t_3. De la même façon, il suffit de retirer le patron de triplet t_1 pour obtenir une requête qui produit des réponses non vides, d'où la XSS $t_2t_3t_4$.

3.2.2 Passage aux MFIS

La définition du problème des réponses pléthoriques se base sur un seuil $K \geq 0$. Une requête qui échoue est une requête qui produit plus de K réponses et une requête qui réussit est une requête qui retourne au plus K réponses. Ainsi, l'échec d'une requête est défini par : $\text{FAIL}_K(Q) = |[[Q]]| > K$. Puisque l'évaluation de la requête vide $\emptyset$ renvoie un unique résultat,

on a $\text{FAIL}_K(\emptyset) = false$ pour tout $K > 0$.

Exemple 3.1 (réponses pléthoriques)**.** *Reprenons l'extrait de la base de connaissances présentée en figure 3.1 avec la requête donnée en figure 3.2 ainsi que ses résultats. Pour l'exemple, supposons que l'utilisateur définisse un seuil de nombre de réponses acceptables $K = 3$. La requête présentée échoue donc, car elle produit 8 réponses, ce qui dépasse le seuil fixé. Nous utilisons de nouveau le treillis des sous-requêtes, pour donner le nombre de réponses de chacune d'entre elles et montrer les requêtes qui réussissent et celles qui échouent.*

Dans le problème des réponses pléthoriques, l'échec d'une requête n'implique pas l'échec de ses super-requêtes, comme le montre l'exemple de la figure 3.2. En effet, on voit que t_2t_5 échoue tout en ayant une super-requête qui réussit : $t_2t_3t_5$. Pour cette raison, la notion de MFS introduite précédemment ne convient plus pour désigner une cause d'échec, car une requête contenant une MFS peut réussir. Nous introduisons donc une nouvelle définition de cause d'échec pour le problème des réponses pléthoriques.

Définition 3.1 (FIS)**.** *Une FIS (Failure Inducing Subquery) d'une requête Q est une de ses sous-requêtes qui échoue et dont toutes les super-requêtes échouent. L'ensemble des FIS d'une requête Q est donné par :*

$$fis^K(Q) = \{Q^* \mid Q^* \subseteq Q \wedge FAIL_K(Q^*) \wedge \forall Q' \subseteq Q, Q^* \subset Q' \Rightarrow FAIL_K(Q')\}$$

Les FIS sont donc des causes d'échec. Dans notre exemple, les FIS sont t_4, t_1t_3, t_1t_4, t_1t_5, t_2t_4, t_3t_4, t_4t_5, $t_1t_2t_3$, $t_1t_2t_4$, $t_1t_2t_5$, $t_1t_3t_4$, $t_1t_3t_5$, $t_1t_4t_5$, $t_2t_3t_4$, $t_2t_4t_5$, $t_3t_4t_5$, $t_1t_2t_3t_4$, $t_1t_2t_3t_5$, $t_1t_2t_4t_5$, $t_1t_3t_4t_5$, $t_2t_3t_4t_5$ et $t_1t_2t_3t_4t_5$. Afin de fournir une information concise à l'utilisateur nous nous intéressons plus précisément aux causes d'échec minimales, ou MFIS.

Définition 3.2 (MFIS)**.** *Une MFIS (Minimal Failure Inducing Subquery) d'une requête Q est une de ses FIS, telle qu'aucune de ses sous-requêtes n'est une FIS. L'ensemble des MFIS d'une requête est défini par :*

$$mfis^K(Q) = \{Q^* \mid Q^* \in fis^K(Q) \wedge \nexists Q' \subset Q^*,\ Q' \in fis^K(Q)\}$$

Exemple 3.2 (MFIS)**.** *Dans notre exemple, les MFIS sont t_4, t_1t_3 et t_1t_5. Chacune permet de donner une information à l'utilisateur pour corriger la requête.*

- *Les MFIS t_1t_3 et t_1t_5 suggèrent que demander simultanément des informations sur les patients (variable* `p`*) et sur les infirmiers (variable* `n`*) entraîne des réponses pléthoriques.*
- *La MFIS t_4 indique que lister les patients (variable* `pt`*) que soignent les infirmiers produit des réponses pléthoriques.*

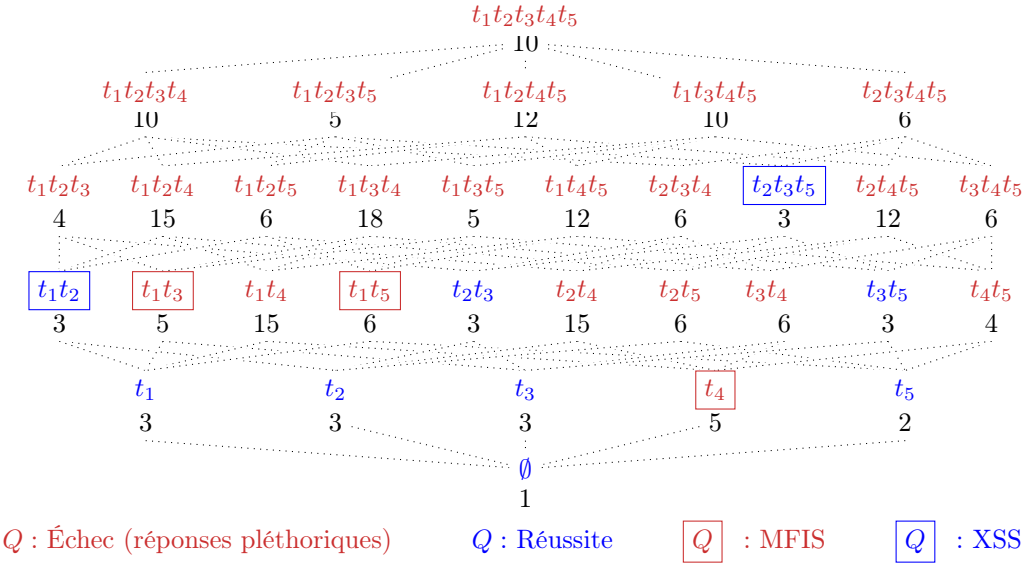
La notion de XSS est également utilisée dans le problème des réponses vides pour désigner

```
SELECT * WHERE {
  ?d traite ?p .                  # t1
  ?d experience ?e .              # t2
  ?d collegue ?n .                # t3
  ?n soigne ?pt .                 # t4
  ?n type InfirmierUrgence }      # t5
```

(a) Requête $Q = t_1t_2t_3t_4t_5$

?d	?p	?e	?n	?pt
d_1	p_1	14	n_3	p_3
d_1	p_2	14	n_3	p_3
d_1	p_1	14	n_3	p_2
d_1	p_2	14	n_3	p_2
d_1	p_1	14	n_2	p_2
d_1	p_2	14	n_2	p_2
d_1	p_1	14	n_3	p_4
d_1	p_2	14	n_3	p_4
d_2	p_3	25	n_2	p_2
d_2	p_3	25	n_3	p_4

(b) Évaluation de Q sur D



(c) Nombre de réponses pour chaque sous-requête de Q

FIGURE 3.2 – Une requête SPARQL, ses résultats et son treillis de sous-requêtes

les requêtes maximales qui réussissent. Cette notion reste applicable sans modification au problème des réponses pléthoriques.

Définition 3.3 (XSS ; Godfrey 1997)**.** *Une XSS (maXimal Succeeding Subquery) d'une requête Q est une de ses sous-requêtes qui réussit dont les super-requêtes sont toutes des FIS. L'ensemble des XSS d'une requête Q est donné par :*

$$xss^K(Q) = \{Q^* \mid Q^* \subseteq Q \wedge \neg FAIL_K(Q^*) \wedge \forall Q' \mid Q^* \subset Q',\ Q' \in fis^K(Q)\}$$

Les XSS sont les requêtes qui réussissent qui sont les plus proches, sémantiquement parlant, de la requête originale. L'utilisateur peut donc utiliser l'une d'elles comme une requête alter-

native.

Exemple 3.3 (XSS). *Dans notre exemple, les XSS sont $t_2t_3t_5$ et t_1t_2. Chacune correspond à une requête proche du besoin de l'utilisateur qui ne retourne pas des réponses pléthoriques.*

- *La XSS $t_2t_3t_5$ donne la liste des médecins avec leurs années d'expérience et les infirmiers qui sont leurs collègues.*
- *La XSS t_1t_2 donne la liste des médecins avec leurs années d'expérience et les patients qu'ils traitent.*

Dans nos travaux, nous ne nous sommes pas intéressés à la manière de présenter les MFIS et XSS à l'utilisateur et ainsi l'accompagner pour résoudre les problèmes de sa requête. Cela pourra faire l'objet de travaux futurs. Pour que les MFIS et XSS soient utilisables en pratique, elles doivent pouvoir être calculées en un temps raisonnable. Aussi, nous nous intéressons, dans la suite de ce chapitre, aux méthodes pour les énumérer.

3.3 Énumération des causes d'échec

Dans cette partie, nous voulons calculer $mfis^K(Q)$ et $xss^K(Q)$ pour une requête RDF conjonctive Q et un seuil fixé K dans une base de connaissances D. Pour cela, nous aurons besoin d'exécuter plusieurs sous-requêtes du treillis. Nous faisons l'hypothèse que c'est l'exécution des requêtes qui domine le temps d'exécution et donc que le nombre de requêtes exécutées est une métrique pertinente pour comparer les algorithmes. Nous commençons par montrer la complexité du problème, puis nous présentons un algorithme basique, que nous améliorons ensuite en appliquant diverses propriétés.

3.3.1 Complexité du problème

Dans un premier temps, nous montrons que le nombre de MFIS et XSS peut être exponentiel par rapport au nombre de patrons de triplets de la requête initiale. Nous montrerons ensuite que l'énumération de n MFIS est également un problème NP-difficile.

Propriété 3.1 (nombre de MFIS et XSS). *Le nombre maximum de MFIS (ou XSS) d'une requête avec n patrons de triplets est $\binom{n}{\lfloor n/2 \rfloor}$, et est atteint lorsque toutes les sous-requêtes de taille $n/2$ sont des MFIS.*

Nous montrons cette propriété en utilisant le théorème de Sperner.

Théorème 3.1 (antichaînes ; Sperner 1928). *Soit un ensemble X à n éléments. Une antichaîne est une famille d'ensembles telle qu'aucun ensemble ne soit strictement contenu dans un autre. Pour toute antichaîne A, $|A| \leq \binom{n}{\lfloor n/2 \rfloor}$. Cette borne est atteinte quand A est l'ensemble de parties de X à k éléments où $k = \frac{n}{2}$ si n est pair et $k = \frac{n-1}{2}$ si n est impair.*

Démonstration. Pour appliquer ce théorème, nous devons montrer que les ensembles de MFIS et de XSS sont des antichaînes. D'après les définitions de MFIS et XSS, à cause de la minimalité (resp. de la maximalité), aucune MFIS (resp. XSS) ne peut être contenue dans une autre MFIS (resp. XSS). Il s'agit donc bien d'antichaînes. D'après le théorème de Sperner, la taille maximale d'une antichaîne est bornée par $\binom{n}{\lfloor n/2 \rfloor}$, où n est le nombre d'éléments de l'ensemble. La taille maximale est atteinte lorsque l'antichaîne contient tous les éléments de taille $\lfloor n/2 \rfloor$. □

Nous venons de montrer un majorant du nombre de MFIS ou de XSS et que ce majorant peut être atteint. En utilisant l'approximation de Sterling pour les factorielles, le nombre maximal de MFIS est approximé par $\binom{n}{\lfloor n/2 \rfloor} \approx \sqrt{\frac{2}{\pi n}} * 2^n$.

Nous pouvons maintenant montrer que l'énumération de n MFIS d'une requête avec n patrons de triplets est NP-difficile. Notre intuition est que notre nouvelle définition de MFIS englobe la notion de MFS précédemment définie. La complexité de l'énumération des MFIS peut donc est déterminée à partir de la complexité de l'énumération des MFS. Pour cela, nous proposons une réduction polynomiale depuis le problème de l'énumération de n MFS, qui est un problème lui aussi NP-difficile (Godfrey, 1997).

Propriété 3.2 (complexité de l'énumération)**.** *L'énumération de n MFIS d'une requête avec n patrons de triplets est NP-difficile.*

Démonstration. Nous utilisons l'abstraction de Godfrey, qui utilise un ensemble $S = \{e_1, ...e_n\}$, et une machine de Turing T avec les propriétés suivantes.

— T est définie sur 2^S, les parties de S.

— T termine et retourne oui ou non pour chaque entrée dans son domaine.

— T s'exécute en temps polynomial pour chaque entrée dans son domaine.

Par ailleurs, si pour une entrée A, T renvoie oui, alors cela implique que pour tout $B \subseteq S$ tel que $A \subseteq B$, T renvoie oui pour l'entrée B ; T est dit fermée supérieurement.
L'ensemble des éléments minimaux d'un treillis est défini par Godfrey comme $MEL = \{A \subseteq S \mid T \text{ renvoie oui pour } A \wedge \forall B \subset A\,(T \text{ renvoie non pour } B)\}$ pour une machine de Turing fermée supérieurement. Cette abstraction correspond aux MFS d'une requête. Pour des machines de Turing non fermée supérieurement, nous proposons une définition généralisée :

$$GMEL = \{A \subseteq S \mid \forall B\,(A \subseteq B \Rightarrow T \text{ renvoie oui pour } B) \wedge \forall C \subset A\,(\exists D\,(C \subseteq D \Rightarrow T \text{ renvoie non pour } D))\}$$

Cette abstraction correspond aux MFIS d'une requête. Godfrey a montré que l'énumération de n éléments de *MEL* est un problème NP-difficile. Nous montrons que l'énumération de n éléments de *GMEL* est aussi NP-difficile par réduction de *MEL* vers *GMEL*.
Considérons un problème d'énumération de *MEL* qui implique un ensemble S et une machine

de Turing fermée supérieurement T. Le problème d'énumération de *GMEL* associé utilise également S et T, donc la transformation (identité) est polynomiale. Il reste à montrer que pour une machine de Turing fermée supérieurement $A \in MEL \iff A \in GMEL$.
Montrons d'abord $A \in MEL \Rightarrow A \in GMEL$. Si $A \in MEL$, alors T renvoie oui pour A. Considérons $B, A \subseteq B$, T renvoie oui pour B à cause de la fermeture supérieure de T. Considérons $C \subset A$, d'après la définition de *MEL*, T renvoie non pour C, donc $\exists D, C \subseteq D$ (c'est-à-dire que $C = D$) où T renvoie non pour D. Ainsi $A \in GMEL$ et donc $A \in MEL \Rightarrow A \in GMEL$.
Montrons ensuite $A \in GMEL \Rightarrow A \in MEL$. Si $A \in GMEL$, alors T renvoie oui pour A. Considérons $B \subset A$, si T renvoie oui pour B alors d'après la fermeture supérieure de T, $\forall C, B \subseteq C$ T renvoie oui pour C, ce qui contredit $A \in GMEL$. Donc T renvoie non pour B et $A \in MEL$. Ainsi, $A \in GMEL \Rightarrow A \in MEL$
Nous venons de faire la réduction de MEL vers GMEL, nous pouvons donc en conclure que l'énumération de n éléments de *GMEL* est NP-difficile. □

La complexité du problème ne nous permet pas d'envisager d'être exhaustifs pour des requêtes arbitrairement grandes. Cependant, l'exhaustivité est nécessaire pour corriger correctement une requête. En pratique, nous supposons que les requêtes écrites par les utilisateurs sont de taille raisonnable. Cette hypothèse est corroborée par une étude des journaux de requêtes réelles (Gallego et al., 2011).

Nous proposons maintenant plusieurs algorithmes de calcul des MFIS et XSS. D'abord, nous présentons un algorithme naïf qui servira de base de comparaison, puis nous introduisons des propriétés permettant de diminuer le nombre d'exécutions de requêtes.

3.3.2 Méthode naïve

Une méthode naïve pour calculer toutes les MFIS et XSS est simplement d'exécuter toutes les sous-requêtes de la requête initiale. Nous appliquons une recherche en largeur, c'est-à-dire que nous commençons par exécuter la requête avec le plus de patrons de triplets. Ainsi, lors de l'évaluation d'une requête, l'état de toutes ses super-requêtes sera connu. L'ordre d'évaluation des requêtes sur le treillis de l'exemple est : $t_1t_2t_3t_4t_5$, $t_1t_2t_3t_4$, $t_1t_2t_3t_5$, $t_1t_2t_4t_5$, $t_1t_3t_4t_5$, $t_2t_3t_4t_5$, $t_1t_2t_3$, ...

Cet algorithme, nommé Base, est décrit dans l'algorithme 3.1 et fournit les réponses correspond au treillis de la figure 3.2c, c'est-à-dire que le statut de chaque sous-requête est déterminé. Dans cet algorithme, les structures de données principales sont une liste (list) de sous-requêtes à évaluer et un dictionnaire (queryStatus) qui stocke le résultat des évaluations : échec ou réussite (lignes 5-13). Les requêtes de la liste sont considérées par ordre de parcours en largeur (lignes 5). Si la requête Q' échoue et que ses super-requêtes directes sont des FIS, les ensembles fis et mfis sont mis à jour (lignes 14-16). La sous-requête étudiée Q' remplace ses super-requêtes directes dans l'ensemble des mfis car elles ne peuvent plus être minimales

(lignes 15-16). Si la requête Q' réussit, elle est ajoutée à l'ensemble xss (ligne 19). Pour une requête initiale avec n patrons de triplets, BASE nécessite $2^n - 1$ exécutions de requêtes (la requête vide $\emptyset$ n'est pas exécutée), ce qui sera coûteux pour des requêtes avec beaucoup de patrons de triplets. Dans notre exemple, BASE exécute 31 requêtes.

Algorithme 3.1: Algorithme naïf pour l'énumération des MFIS et XSS de Q

```
1  Base(Q, K)
       entrées: Une requête Q = t_1 ∧ ... ∧ t_n ;
                Un seuil K
       sorties : MFIS et XSS de Q
2      mfis ← ∅, xss ← ∅, fis ← ∅, queryStatus ← ∅;
3      list ← lattice(Q) ; // génération du treillis des sous-requêtes
4      while list ≠ ∅ do
5          Q' ← première requête de list en parcours en largeur;
6          list ← list − {Q'};
7          queryStatus [Q'] ← FAIL_K(Q') ; // exécuter la requête et enregistrer
               son échec ou réussite
8          parents_fis ← true ;
9          superqueries ← direct_superQueries(Q);
10         foreach Q'' ∈ superqueries do
11             parents_fis ← parents_fis ∧ ((Q'') ∈ fis) ; // vérification que les
                   super-requêtes directes sont des FIS
12         if queryStatus [Q']=FAILS then // Q' échoue
13             if parents_fis then // Q' est une FIS
14                 fis ← fis ∪ {Q'} ;
15                 mfis ← mfis − superqueries ;
16                 mfis ← mfis ∪ {Q'} ;
17         else // Q' réussit
18             if parents_fis then
19                 xss ← xss ∪ {Q'};
20     return mfis, xss ;
```

3.3.3 Optimisation par élagage

Pour améliorer l'algorithme BASE, nous commençons par élaguer l'espace de recherche afin d'éviter des requêtes qui ne seront ni MFIS ni XSS. Nous utilisons la propriété suivante qui est directement déduite de la définition de MFIS et XSS.

Propriété 3.3 (réussite de requête)**.** *Si une sous-requête Q' réussit et Q" $\subset Q'$, alors Q" n'est ni une MFIS ni une XSS.*

Démonstration. Considérons deux requêtes : Q' et Q'' telles que Q' est une sous-requête qui réussit, et Q'' est une sous-requête de Q'. Nous allons montrer par contradiction que Q'' n'est ni une MFIS ni une XSS.
Supposons d'abord que Q'' soit une MFIS. Ainsi Q'' est également une FIS. D'après la définition d'une FIS, toutes les super-requêtes de Q'' échouent. En particulier, Q' échoue. Cela contredit notre hypothèse, donc Q'' n'est pas une MFIS.
Supposons maintenant que Q'' est une XSS. D'après la définition d'une XSS, toutes les super-requêtes strictes de Q'' sont des FIS. En particulier Q' est une FIS donc Q' échoue. Cela contredit notre hypothèse, donc Q'' n'est pas une XSS. □

Nous proposons ainsi une première amélioration de l'algorithme, appelée BFS, où les sous-requêtes de requêtes qui réussissent ne sont pas étudiées. Par rapport à l'algorithme Base (algorithme 3.1), cela implique d'inverser l'ordre des vérifications réalisées aux lignes 12 et 13. Dans la section suivante, nous intégrons une seconde amélioration en introduisant une propriété permettant de connaître l'échec d'une requête sans l'exécuter.

3.3.4 Optimisation par analyse de la requête

Une deuxième amélioration concerne la structure de la requête. Comme nous effectuons un parcours du treillis en largeur, nous pouvons éventuellement nous baser sur le résultat d'une requête pour déduire le résultat d'une de ses sous-requêtes. Nous considérons des cas où retirer un patron de triplet ne fait pas diminuer le nombre de variables. Considérons le patron de triplet t_5 : $?s$ *type* $InfirmierUrgence$ de notre exemple. Si ce patron de triplet est ajouté à une requête qui contient déjà la variable s, il impose une contrainte sur s et donc ne peut que faire baisser le nombre de réponses. L'intuition est que si, en retirant un patron de triplet, le nombre de variables ne diminue pas, alors on ne fait que retirer une contrainte et donc le nombre de réponses ne peut pas diminuer.

Propriété 3.4 (variables)**.** *Pour une requête Q et un patron de triplet t, si $var(Q \wedge t) = var(Q)$ alors $FAIL_K(Q \wedge t) \Rightarrow FAIL_K(Q)$.*

Démonstration. Nous allons montrer que $\forall \mu \in [[Q \wedge t]]$, $\mu \in [[Q]]$. Soit $\mu \in [[Q \wedge t]]$, et $\mu' = \mu_{|\text{var}(Q)}$. Montrons que $\mu' \in [[Q]]$. Supposons par l'absurde que $\mu' \notin [[Q]]$. Soit $\text{dom}(\mu') \neq \text{var}(Q)$ ou $\exists v \in \text{var}(Q), \mu'(v) \notin D$. Par définition de la restriction, $\text{dom}(\mu') = \text{var}(Q)$. Soit $v \in \text{var}(Q), \mu'(v) \notin D$. Par définition, $\mu'(v) = \mu(v)$. Donc $\mu(v) \notin D$, ce qui contredit $\mu \in [[Q \wedge t]]$. Donc $\mu' \in [[Q]]$, et comme $\text{var}(Q \wedge t) = \text{var}(Q)$, alors $\mu' = \mu$. Donc $[[Q \wedge t]] \subseteq [[Q]]$ et $|[[Q \wedge t]]| \leq |[[Q]]|$. Comme $\text{FAIL}_K(Q \wedge t)$, $|[[Q \wedge t]]| > K$ donc $|[[Q]]| \geq |[[Q \wedge t]]| > K$ c'est-à-dire $\text{FAIL}_K(Q)$. □

Cette propriété peut être utilisée dans les cas suivants :

Algorithme 3.2: Énumération des MFIS et XSS d'une requête Q

```
1  VAR(Q, K)
   entrées: Une requête Q = t_1 ∧ ... ∧ t_n
            Un seuil K
   sorties : MFIS et XSS de Q
2  mfis ← ∅, xss ← ∅, fis ← ∅, queryStatus ← ∅
3  list ← lattice(Q)
4  while list ≠ ∅ do
5      Q' ← première requête de list en parcours en largeur
6      list ← list − {Q'}
7      parents_fis ← true
8      superqueries ← direct_superQueries(Q)
9      foreach Q'' ∈ superqueries do
10         parents_fis ← parents_fis ∧ ((Q'') ∈ fis)
11     if parents_fis then
12         if Q' ∉ queryStatus then
13             queryStatus [Q'] ← FAIL_K(Q')
14         if queryStatus [Q']=FAILS then // si Q' échoue
15             fis ← fis ∪ {Q'}
16             mfis ← mfis − superqueries
17             mfis ← mfis ∪ {Q'}
18             foreach Q'' ∈ subQueries(Q') do
19                 if var(Q'') = var(Q') then
20                     queryStatus [Q''] ← true
21         else // Q' réussit et est donc une XSS
22             xss ← xss ∪ {Q'}
23 return mfis, xss
```

— lorsqu'une paire de variables est liée par plusieurs prédicats, par exemple ?d *traite* ?n . ?d *collegue* ?n ;

— lorsqu'un patron de triplet ne contient qu'une variable déjà présente dans la requête, par exemple ?n *soigne* ?pt . ?n *type* $InfirmierUrgence$;

— si retirer un patron de triplet crée un produit cartésien (une requête qui peut être séparée en deux parties qui ne partagent aucune variable), par exemple ?d *experience* ?e . ?d *collegue* ?n . ?n *soigne* ?pt.

Nous proposons l'algorithme VAR qui se base sur cette propriété.

L'algorithme VAR est donné dans l'algorithme 3.2. La première amélioration par rapport à la méthode de base est l'utilisation de la propriété 3.3, (ligne 11) en vérifiant que chaque super-requête de Q' est une FIS avant de l'exécuter. Cela correspond à l'algorithme BFS. La

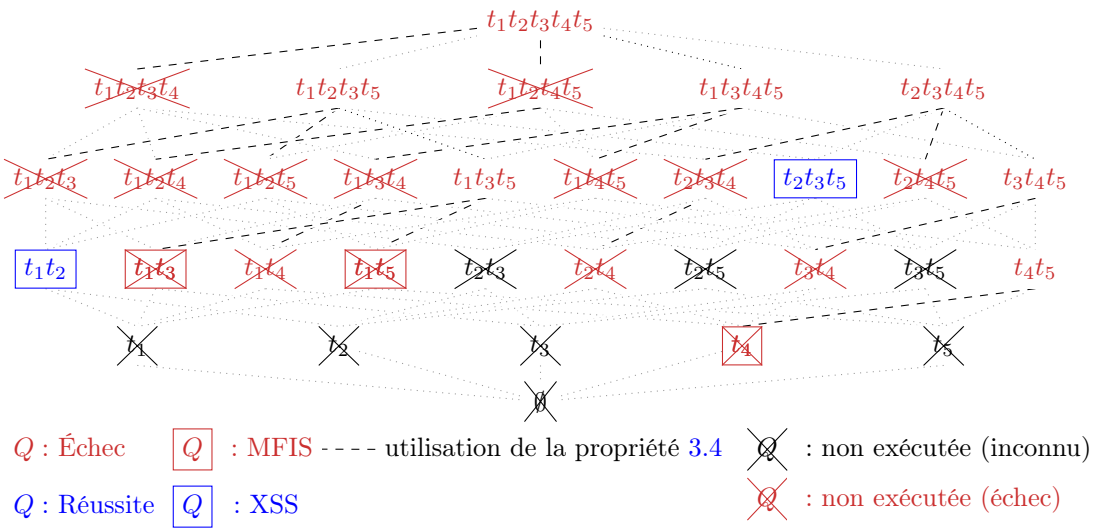


FIGURE 3.3 – Treillis des sous-requêtes de Q pour l'algorithme VAR

deuxième amélioration intervient après la mise à jour des FIS et MFIS. Nous étudions toutes les sous-requêtes directes de Q' (ligne 18-20), en vérifiant si la propriété 3.4 est applicable (ligne 19) afin de prédire son statut sans l'évaluer.

Nous illustrons dans la figure 3.3 le treillis des sous-requêtes de la requête Q de l'exemple et montrons les requêtes exécutées par l'algorithme VAR. Les requêtes dont l'exécution est évitée car elles ont une super-requête qui réussit ont un état inconnu. Ici, les requêtes t_2t_3, t_2t_5, t_3t_5, t_2, t_3 et t_5 ne sont pas exécutées, car $t_2t_3t_5$ réussit. Les requêtes dont l'exécution est évitée en utilisant la propriété sur les variables échouent. Par exemple, $t_1t_2t_3t_5$ échoue et a les mêmes variables que $t_1t_2t_3$, donc $t_1t_2t_3$ échoue d'après la propriété 3.4. Globalement, l'algorithme VAR exécute 9 requêtes, ce qui représente une nette amélioration par rapport à l'algorithme BASE (31 requêtes).

Nous définissons la complexité de notre algorithme en fonction du nombre d'exécutions de requêtes nécessaires, en utilisant n le nombre de patrons de triplets de la requête initiale. Pour les algorithmes BFS et VAR, le pire cas correspond à un cas où les propriétés ne peuvent pas être appliquées : pour BFS si tout le treillis échoue et pour VAR si chaque patron de triplet retiré diminue le nombre de variables. Leur complexité pire cas est donc la même que pour l'algorithme BASE ($2^n - 1$). Cependant, nous pouvons définir des situations où l'amélioration de nos algorithmes peut être mesurée.

Pour la propriété 3.3, les exécutions évitées sont les sous-requêtes des XSS. Ainsi, le nombre de requêtes exécutées dépend du nombre de XSS de la requête et du nombre de patrons de triplets que chacun contient. Plus les XSS sont grandes, plus le nombre de requêtes exécutées sera petit. Comme les XSS ne sont pas connues avant l'exécution de l'algorithme, nous ne pouvons pas déterminer à l'avance le nombre d'exécutions de requêtes qui seront évitées en

utilisant la propriété 3.3.

En revanche, pour la propriété 3.4, la structure de la requête peut être exploitée afin de déterminer à l'avance le nombre de requêtes exécutées. Nous nous sommes intéressés aux requêtes en étoile, pour lesquelles tous les patrons de triplets partagent le même sujet, car ce sont les requêtes les plus utilisées en pratique (Gallego et al., 2011). Nous donnons l'expression du nombre de requêtes exécutées pour des requêtes en étoile, car leur structure simple permet d'étudier les patrons de triplets un à un. Nous montrons que chaque patron de triplet sur lequel on peut appliquer la propriété 3.4 permet de diviser le nombre de requêtes exécutées par deux.

Propriété 3.5 (complexité de VAR pour les requêtes en étoile)**.** *Pour une requête en étoile avec n patrons de triplets, où k est le nombre de patrons de triplets sur lesquels s'applique la propriété 3.4 (avec $k \geq 1$), le nombre d'exécutions de requêtes évitées par l'utilisation de la propriété 3.4 est donné par :*

$$\left(\sum_{i=1}^{k} 2^{n-i}\right) - 1$$

Démonstration. Montrons la formule par récurrence.

Pour k=1, on considère un unique patron de triplet t, auquel on peut appliquer la propriété 3.4. Pour toute sous-requête Q qui ne contienne pas t, $\text{FAIL}_K(Q \wedge t)$ implique $\text{FAIL}_K(Q)$. Si $Q \wedge t$ réussit, alors Q n'est pas exécutée d'après la propriété 3.3. Ainsi, quel que soit l'état de $Q \wedge t$ l'exécution de toutes les requêtes Q peut être évitée. Le nombre de ces requêtes correspond au nombre d'éléments dans un treillis de taille $n-1$ sans la requête vide, soit $2^{n-1} - 1$.

Supposons que la propriété soit vérifiée pour une valeur de k et montrons qu'elle l'est également pour $k+1$. On note $t_1, ..., t_k$ les k premiers patrons de triplets, et t_{k+1} le nouveau patron de triplet considéré. D'après notre hypothèse, le nombre d'exécutions de requêtes évitées par $t_1, ..., t_k$ est $\sum_{i=1}^{k} 2^i - 1$. Les requêtes évitées en utilisant la propriété sur t_{k+1}, qui n'étaient pas évitées en considérant uniquement $t_1, ..., t_k$, sont les requêtes contenant tous les patrons de triplets $t_1, ..., t_k$, mais pas t_{k+1}. Cela revient à choisir entre 0 et $n-k-1$ patrons de triplets à ajouter à $t_1, ..., t_k$ pour construire la requête. Le nombre de telles requêtes est : $\sum_{i=0}^{n-k-1} \binom{n-k-1}{i} = 2^{n-k-1} = 2^{n-(k+1)}$. En tout, le nombre de requêtes évitées est donc : $\sum_{i=1}^{k+1} 2^i - 1$. □

La forme en étoile de la requête initiale est utile ici, car elle permet d'affirmer que la propriété 3.4 s'appliquera toujours à un patron de triplet t sur tout le treillis, ou ne s'appliquera pas du tout. Pour des requêtes avec une structure plus complexe, il est également possible de déterminer à l'avance le nombre d'exécutions de requêtes évitées, mais comme cela dépend de la structure de la requête, une formulation générale n'est pas donnée. Nous utilisons la requête

Q de notre exemple pour illustrer l'absence de formule générale pour des requêtes de forme quelconque. Q contient deux patrons de triplets sur lesquels la propriété 3.4 peut s'appliquer : t_3 et t_5. Cependant, pour appliquer la propriété 3.4 sur une requête $Q_a \wedge t_5$, Q_a doit contenir la variable n. Ainsi, cette propriété ne peut pas être utilisée avec la requête $t_1t_2t_5$ pour éviter l'exécution de t_1t_2. En tout, 22 exécutions de requêtes peuvent être évitées en appliquant la propriété 3.4 contre les 23 attendues s'il s'agissait d'une requête en étoile.

3.4 Choix d'implémentation

Dans la section précédente, nous avons présenté deux propriétés permettant de diminuer le nombre de requêtes à exécuter lors de l'énumération des MFIS et XSS. Dans cette section, nous étudions comment faire baisser le temps d'exécution de chaque requête afin d'améliorer la performance des algorithmes. Nous commençons par détailler le protocole expérimental qui sera utilisé pour valider les choix d'implémentation et qui sera également repris dans la section 3.5. Par la suite, nous étudions la véracité de l'hypothèse selon laquelle le temps d'exécution des requêtes est dominant par rapport au temps de calcul des algorithmes.

3.4.1 Protocole expérimental

Nos algorithmes sont implémentés en Java 1.8 et sont exécutés sur un serveur Ubuntu 16.04 LTS avec un processeur Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz et 32 GB RAM. Les temps d'exécution présentés dans les figures sont une moyenne de cinq exécutions consécutives. Avant la première mesure du temps d'exécution, les algorithmes sont exécutés une fois. Les requêtes sont exécutées sur les triplestores Jena TDB, Jena Fuseki et Virtuoso, qui ont été choisis car ils font partie des triplestores les plus utilisés en pratique. Nos expériences sont disponibles sur `https://forge.lias-lab.fr/projects/tma4kb` avec un tutoriel pour les reproduire.

Afin de mesurer la performance de nos algorithmes, nous avons dû choisir les données support et les requêtes initiales. Comme le problème des réponses pléthoriques a été peu étudié depuis l'angle des requêtes, nous ne disposons pas d'un banc d'essai sur lequel nous baser. Pour avoir une évaluation la plus complète possible, nous nous sommes basées sur une source de données synthétique et une réelle. Pour le choix des requêtes, nous avons utilisé des requêtes existantes pour chaque source de donnée, et nous avons repris l'étude de Gallego et al. (2011) sur les requêtes SPARQL afin d'avoir des caractéristiques représentatives des requêtes utilisées en pratique.

La première source de données utilisée est synthétique et provient du banc d'essai WatDiv (Aluç et al., 2014) qui permet de générer des jeux de données de tailles diverses. Nous avons utilisé des jeux de données de 100K à 88M triplets. Ce banc d'essai contient également des

	Requête	MFIS	XSS

Q4 (7TP)	SELECT * WHERE {
	?v0 userId "6347025" . #t1
	?v0 friendOf ?v1 . #t2
	?v0 likes ?v2 . #t3
	?v0 makesPurchase ?v3 . #t4
	?v0 gender ?v4 . #t5
	?v0 email ?v5 . #t6
	?v0 likes ?v6 #t7
	}
		2 MFIS :
		t2
		t3t4t7
			2 XSS :
			t1t3t5t6t7
			t1t3t4t5t6
			t1t4t5t6t7

Q5 (6TP)	SELECT * WHERE {
	?subject type Film . #t1
	?subject starring Paul_Newman . #t2
	?subject starring ?actors . #t3
	?subject comment ?abstract . #t4
	?subject label ?label . #t5
	?subject releaseDate ?released #t6
	}
		1 MFIS :
		t3
			1 XSS :
			t1t2t4t5t6

TABLE 3.1 – Exemples de requêtes pour WatDiv et DBpedia

requêtes, que nous avons légèrement modifiées pour créer des requêtes à réponses pléthoriques. Nous avons choisi d'utiliser 21 requêtes, dont 7 requêtes en étoile, 7 requêtes en chaîne et 7 requêtes composites. Ces requêtes ont entre 4 et 12 patrons de triplets. Les requêtes en chaîne sont issues de la requête IL-1-10 du banc d'essai, et les requêtes composites F1, F2, F4 et C2 ainsi que la requête en étoile C3 ont été utilisées (respectivement les requêtes Q15, Q16, Q18, Q19 et Q3). Nous avons également créé de nouvelles requêtes en étoile et composites afin d'avoir des caractéristiques variées (nombre de patrons de triplets, nombre et taille des MFIS et XSS).

La seconde source de données est réelle. Il s'agit de la base de connaissances DBpedia (la version 3.9 en anglais) qui contient 812 millions de triplets. Les requêtes utilisées sont des requêtes réelles soumises par des utilisateurs entre les mois d'avril et de juin 2010. Elles sont issues de journaux de requêtes publiés par DBpedia et traitées par le projet Linked SPARQL Queries Dataset (Saleem et al., 2015). Ce projet recense plus de 500 000 requêtes d'utilisateurs ainsi que des paramètres tels que leurs temps d'exécution ou leurs nombres de réponses. Les requêtes sont fournies sous forme d'une base de connaissances qui peut être interrogée à l'aide d'un endpoint SPARQL. Parmi les requêtes de ce projet, nous avons sélectionné des requêtes conjonctives retournant les plus grands nombres de réponses. Ces requêtes sont en étoile ou composites et contiennent de 4 à 10 patrons de triplets. Nous avons dû modifier légèrement certaines requêtes, car elles faisaient référence à la version 3.5.1 de DBpedia et utilisaient des URI qui n'existent plus dans la base.

Des exemples de requêtes pour WatDiv (Q4) et DBpedia (Q5) sont donnés en table 3.1. Pour la clarté d'affichage, ces requêtes n'incluent pas les préfixes des URI. Les MFIS et XSS de ces requêtes sont également fournies. La liste complète des requêtes utilisées pour les expériences est disponible dans l'annexe A.

```
SELECT * WHERE {
  ?d traite ?p .      # t1
  ?d experience ?e .  # t2
  ?n soigne ?pt .     # t4
  ?n type ERNurse }   # t5
```

(a) Requête $t_1t_2t_4t_5$

```
SELECT * WHERE {
  ?d traite ?p .      # t1
  ?d experience ?e }  # t2
```

(b) Requête t_1t_2

```
SELECT * WHERE {
  ?n soigne ?pt .     # t4
  ?n type ERNurse }   # t5
```

(c) Requête t_4t_5

FIGURE 3.4 – Produit cartésien $t_1t_2t_4t_5$ et sa décomposition

Comme nos requêtes ne sont pas associées à des nombres de réponses attendus par des utilisateurs, nous avons fixé un seuil par défaut pour les réponses pléthoriques. Nous avons choisi la valeur K=100, car il s'agit de la valeur par défaut de l'opérateur LIMIT dans l'end-point de DBpedia. Par la suite, nous faisons varier ce seuil afin de quantifier son impact sur les résultats.

3.4.2 Traitement des produits cartésiens

Un produit cartésien est une requête qui peut être séparée en plusieurs parties qui ne partagent pas de variable. Le nombre de réponses d'un produit cartésien est égal au produit des nombres de réponses de chacune des parties.

Exemple 3.4 (produit cartésien)**.** *La figure 3.4 montre un produit cartésien et sa décomposition. Dans notre exemple, $t_1t_2t_4t_5$ a 12 réponses, ce qui représente bien le produit du nombre de réponses de t_1t_2 (3 réponses) et t_4t_5 (4 réponses).*

L'exécution de produits cartésiens dans un triplestore est coûteux. Ainsi, plutôt que de les exécuter dans nos algorithmes, nous les décomposons en parties connectées. Nous exécutons ensuite chacune de ces parties et faisons le produit des nombres de réponses. Cette opération diminue les temps d'exécution des requêtes, mais a également un impact sur le nombre de requêtes exécutées. En effet, au lieu d'exécuter une seule requête, le produit cartésien, nous exécutons une requête pour chaque partie connectée. Cela fait baisser le nombre de requêtes exécutées par l'algorithme BASE (car les sous-parties seraient de toute façon exécutées par l'algorithme), mais peut faire augmenter le nombre de requêtes exécutées par les algorithmes BFS et VAR car les parties connectées pourraient ne pas être exécutées si elles ont une super-requête qui réussit (propriété 3.3).

3.4.3 Méthodes d'exécution des requêtes dans un triplestore

Pour déterminer si une requête réussit ou échoue, la méthode de base consiste à :

1. exécuter la requête sur le triplestore qui retourne toutes les réponses ;
2. compter ses réponses ;
3. comparer ce nombre de réponses au seuil fixé.

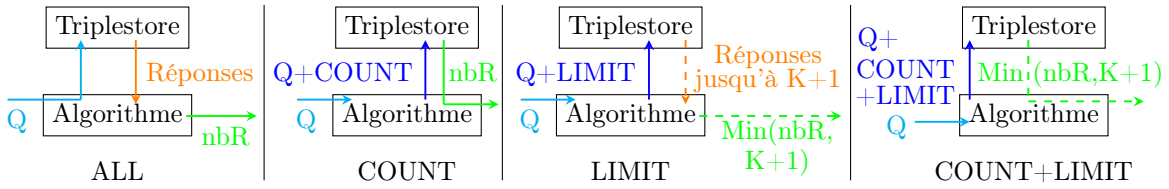
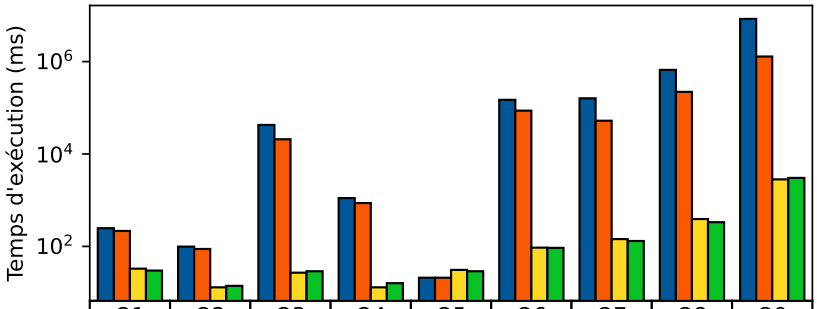


FIGURE 3.5 – Les quatre méthodes d'exécution



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
all	248	99	4.3E4	1,112	21	1.5E5	1.6E5	6.6E5	8.4E6
count	216	88	2.1E4	867	21	8.7E4	5.3E4	2.2E5	1.3E6
limit	33	13	27	13	31	94	144	391	2,825
countlimit	30	14	29	16	29	93	131	334	3,039

FIGURE 3.6 – Temps d'exécution pour l'algorithme VAR pour des requêtes DBpedia

Afin d'accélérer ce processus, nous proposons d'utiliser les opérateurs COUNT et LIMIT du langage SPARQL dans l'objectif de déléguer une partie du traitement au triplestore. Nous allons comparer quatre méthodes d'exécution qui sont illustrées en figure 3.5.

— ALL : méthode de base, on soumet la requête telle quelle au triplestore qui renvoie l'ensemble des réponses, puis l'algorithme compte le nombre de réponses (nbR). La requête envoyée au triplestore est de la forme `SELECT * WHERE {...}`.

— COUNT : ajout de l'opérateur COUNT afin de compter le nombre de réponses dans le triplestore. La requête est envoyée au triplestore sous la forme `SELECT COUNT(*) WHERE {...}`.

— LIMIT : ajout de l'opérateur LIMIT pour que le triplestore retourne au maximum K+1 réponses. La requête envoyée au triplestore est de la forme `SELECT * WHERE {...} LIMIT (K+1)`.

— COUNT+LIMIT : ajout des deux opérateurs COUNT et LIMIT. La requête envoyée au triplestore est de la forme `SELECT COUNT(*) WHERE {SELECT * WHERE {...} LIMIT (K+1)}`.

Lorsque l'on utilise l'opérateur LIMIT, on peut perdre l'information du nombre exact de réponses si celui-ci dépasse K+1. Pour le problème des réponses pléthoriques, cela n'est pas gênant puisqu'on souhaite uniquement savoir si le nombre de réponses dépasse le seuil K.

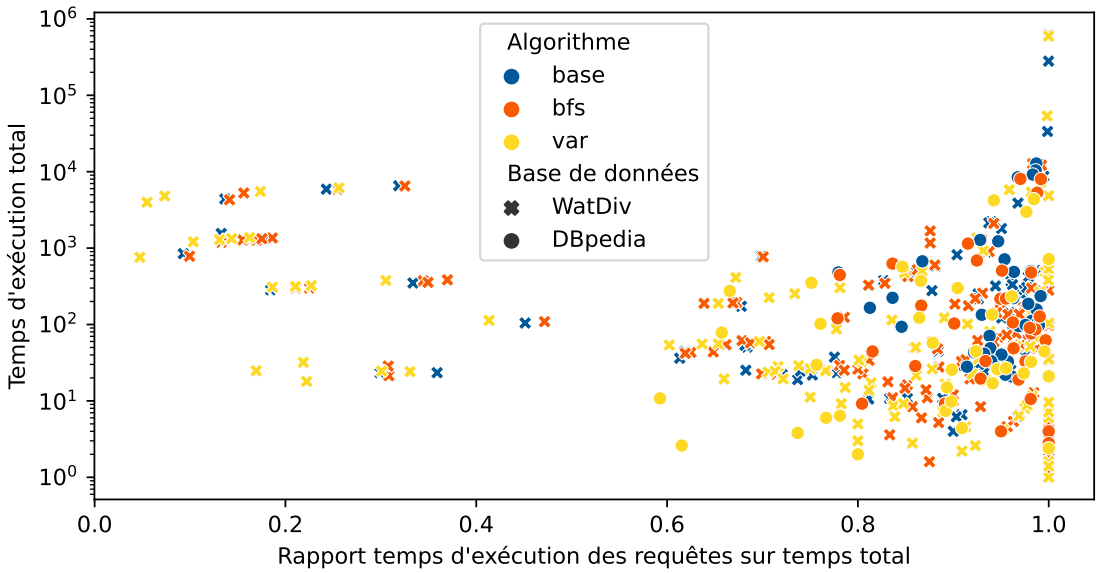


FIGURE 3.7 – Temps d'exécution total et rapport entre temps d'exécution des requêtes et temps d'exécution total

La figure 3.6 montre les temps d'exécution de chaque méthode pour neuf requêtes sur la base de connaissances DBpedia avec le triplestore JenaTDB et un seuil de réponses pléthoriques K=100. Pour cette étude, nous avons utilisé l'algorithme VAR. Les méthodes qui retournent toutes les réponses sont significativement plus lentes que celles qui utilisent l'opérateur LIMIT. Pour les requêtes Q6 et Q9, il y a trois ordres de grandeur entre les méthodes ALL et LIMIT. La méthode la plus rapide dans l'ensemble est COUNT+LIMIT. C'est donc cette méthode d'exécution des requêtes que nous adoptons.

3.4.4 Décomposition du temps d'exécution

Nous avons précédemment fait l'hypothèse que le temps d'exécution des algorithmes est majoritairement dû au temps d'exécution des requêtes. Nous allons vérifier cette hypothèse expérimentalement. La figure 3.7 montre le rapport entre le temps d'exécution total et le temps d'exécution des requêtes en fonction du temps total. Chaque point correspond à une exécution d'algorithme dans le cadre des expériences présentées dans la section suivante.

Pour chaque algorithme, le temps d'exécution des requêtes représente généralement entre 60% et 100% du temps total, c'est le cas pour 99% des exécutions sur DBpedia, et 83% des exécutions sur WatDiv. On peut aussi noter que plus le temps total est long, plus le temps d'exécution des requêtes représente une part importante du temps total. Pour les exécutions sur DBpedia, le rapport moyen entre le temps d'exécution des requêtes et le temps total est de 92% (94% pour BASE, 93% pour BFS et 87% pour VAR). Pour les exécutions sur WatDiv, il est de 77% (78% pour BASE, 80% pour BFS, et 74% pour VAR). Notre hypothèse est ainsi validée.

3.4.5 Utilisation de la méthode dans un endpoint SPARQL

Notre implémentation initiale, qui est utilisée dans l'évaluation des algorithmes pour montrer la faisabilité de l'approche, est exécutée depuis un terminal de commande. Pour déployer ces algorithmes, plusieurs solutions sont envisageables, mais chacune a des inconvénients.

— Déploiement local : les algorithmes sont disponibles via une interface Web spécifique sur un serveur du laboratoire, ce qui nécessite un coût de développement supplémentaire pour l'interface et une API REST.

— Déploiement distant : les algorithmes sont hébergés sur le serveur lié à l'endpoint SPARQL, ce qui nécessite de contacter un agent de chaque base de connaissances que l'on souhaite pouvoir utiliser et génère un coût de suivi et de maintenabilité.

— Téléchargement des données : les algorithmes sont exécutés localement sur une copie des données, ce qui implique une synchronisation avec les données distantes et un espace de stockage important.

Pour rendre notre solution plus adaptée à des utilisateurs finaux, nous voulons intégrer nos algorithmes de recherche des MFIS et XSS directement dans les endpoints SPARQL utilisés pour interroger des bases de connaissances. Ainsi, nous avons développé un module pour Chrome et Firefox qui enrichit les interfaces Web existantes. Notre outil nommé Sparql HINts for whY-questions (SHINY) utilise les technologies WebAssembly et le langage Go afin d'exécuter les algorithmes directement dans le navigateur. La figure 3.4.5 montre l'interface utilisateur, qui consiste à ajouter des éléments sur un endpoint SPARQL, ici celui de DBpedia.

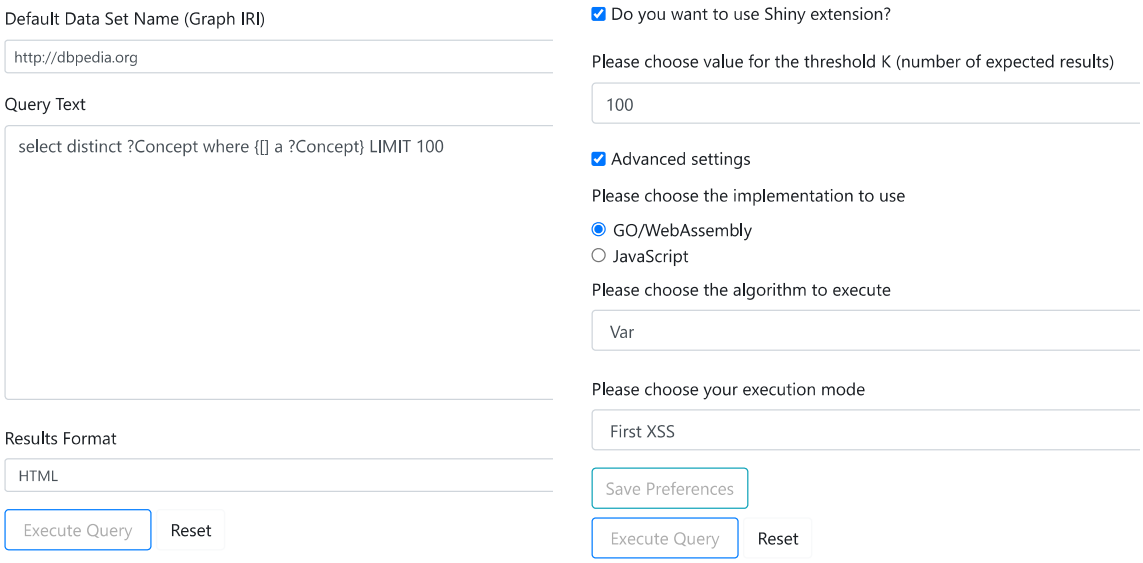
Un fonctionnement par défaut permet à l'utilisateur de renseigner le seuil des réponses pléthoriques et exécute l'algorithme Var. Un mode avancé permet à l'utilisateur de choisir un autre algorithme. Trois affichages sont possibles selon le mode choisi :

1. afficher toutes les MFIS et XSS de la requête ;
2. exécuter une XSS (fonctionnement par défaut) ;
3. modifier la requête pour corriger les MFIS.

Le premier mode donnera simplement la liste des XSS et MFIS de la requête et permet à l'utilisateur de choisir l'une des XSS à exécuter ou de modifier sa requête par la suite.

Le second mode exécute une des XSS et retourne ses résultats, car on sait que le nombre de résultats est en dessous du seuil et que la nouvelle requête est proche de celle de l'utilisateur. Dans cette première version, la XSS est choisie arbitrairement. La figure 3.9 montre l'affichage produit. Celui-ci indique à l'utilisateur que sa requête a produit trop de réponses et précise la requête qui a été exécutée à la place. L'utilisateur peut également choisir d'exécuter une des autres XSS.

Le dernier mode d'exécution utilise les MFIS. Il s'agit d'une interface permettant à l'utilisateur d'éditer sa requête tout en l'informant des MFIS qui ont été traitées ou non. La figure 3.10 montre l'affichage obtenu. La partie supérieure donne les MFIS et l'icône à droite de



(a) Avant activation de l'extension (b) Avec le module SHINY

FIGURE 3.8 – Endpoint enrichi par le module SHINY

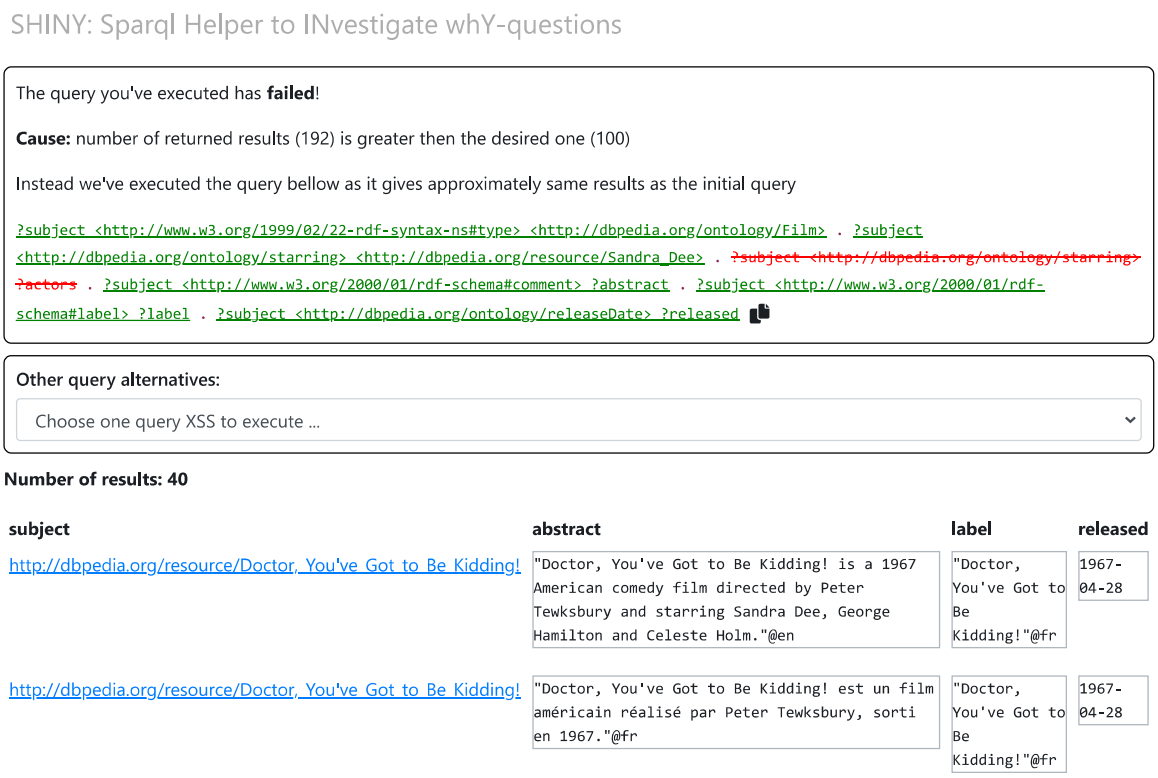


FIGURE 3.9 – Exécution d'une XSS avec SHINY

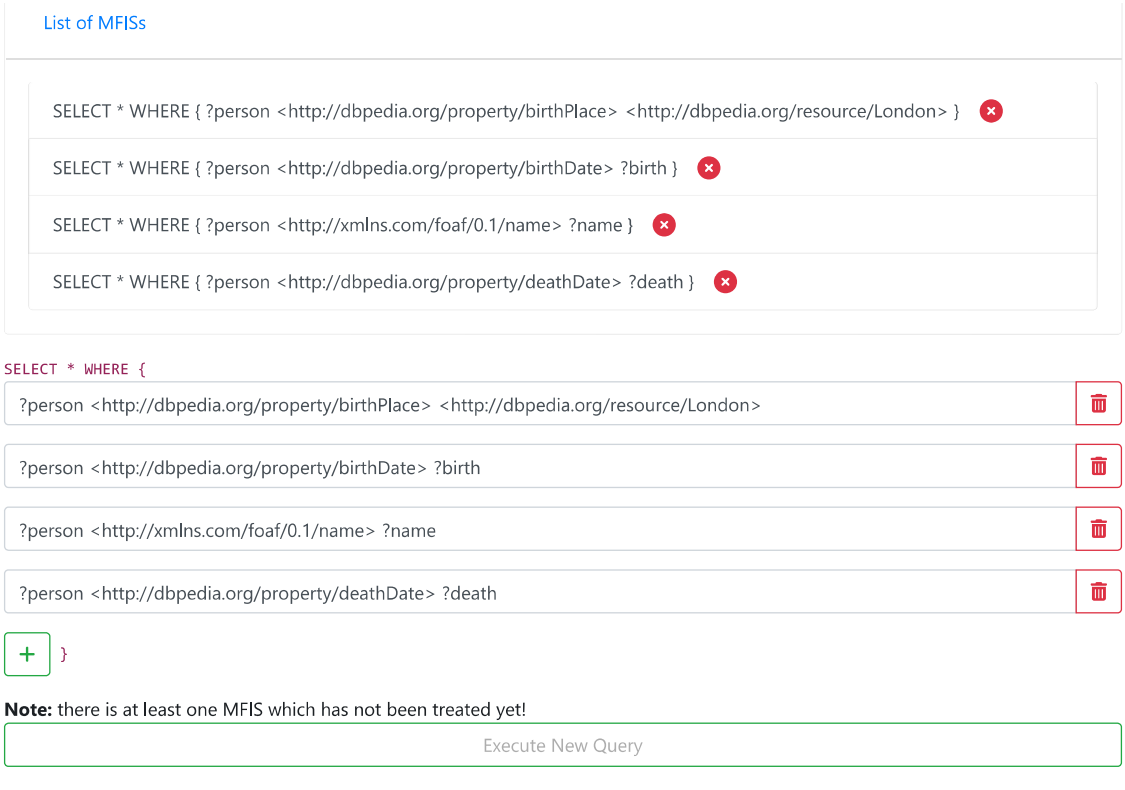


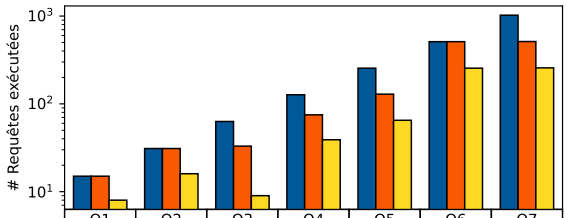
FIGURE 3.10 – Correction des MFIS avec SHINY

chacune indique si la MFIS a été corrigée. L'utilisateur peut retirer des patrons de triplets ou les modifier pour corriger les MFIS identifiées dans la partie inférieure de l'écran, puis exécuter la nouvelle requête.

En utilisant les choix d'implémentation décrits dans cette section, nous avons réalisé une évaluation expérimentale de nos algorithmes qui est présentée dans la section suivante. Il s'agit de mesurer les temps de calcul des différents algorithmes en fonction de la taille des requêtes et de la base de connaissances. Comme perspective de travaux futurs, nous souhaitons réaliser une étude avec des utilisateurs réels pour évaluer la qualité des retours basés sur les MFIS et XSS en utilisant l'outil SHINY.

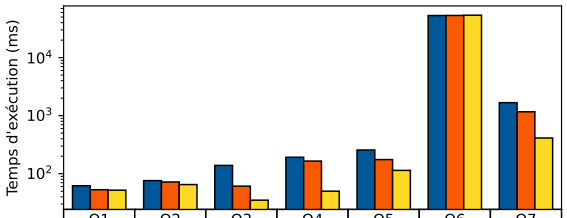
3.5 Évaluation des algorithmes

Nous allons comparer les performances des trois algorithmes BASE, BFS et VAR afin de quantifier l'apport des propriétés.



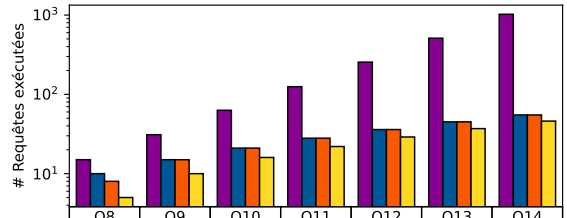
	Q1	Q2	Q3	Q4	Q5	Q6	Q7
base	15	31	63	127	255	511	1,023
bfs	15	31	33	75	129	511	513
var	8	16	9	39	65	255	257

FIGURE 3.11 – # Requêtes exécutées (requêtes en étoile) Watdiv 11M triplets



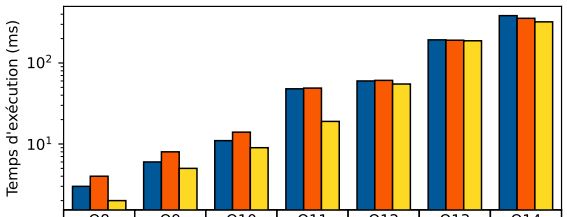
	Q1	Q2	Q3	Q4	Q5	Q6	Q7
base	62	76	139	192	256	5.3E4	1,674
bfs	53	72	61	165	175	5.3E4	1,165
var	52	65	35	50	114	5.4E4	412

FIGURE 3.12 – Temps d'exécution (requêtes en étoile) Watdiv 11M triplets



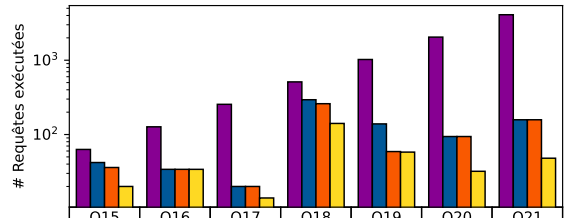
	Q8	Q9	Q10	Q11	Q12	Q13	Q14
théorie	15	31	63	125	255	511	1023
base	10	15	21	28	36	45	55
bfs	8	15	21	28	36	45	55
var	5	10	16	22	29	37	46

FIGURE 3.13 – # Requêtes exécutées (requêtes en chaîne) Watdiv 11M triplets



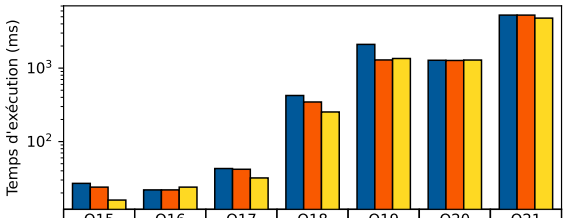
	Q8	Q9	Q10	Q11	Q12	Q13	Q14
base	3	6	11	48	60	193	384
bfs	4	8	14	49	61	191	356
var	2	5	9	19	55	188	322

FIGURE 3.14 – Temps d'exécution (requêtes en chaîne) Watdiv 11M triplets



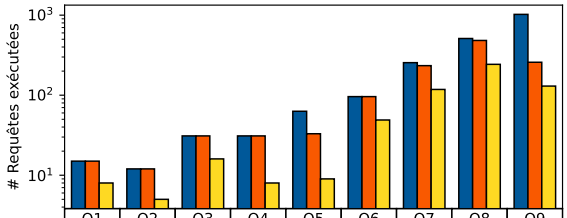
	Q15	Q16	Q17	Q18	Q19	Q20	Q21
théorie	63	127	255	511	1023	2047	4095
base	42	34	20	294	139	94	158
bfs	36	34	20	260	59	94	158
var	20	34	14	141	58	32	48

FIGURE 3.15 – # Requêtes exécutées (requêtes composites) Watdiv 11M triplets



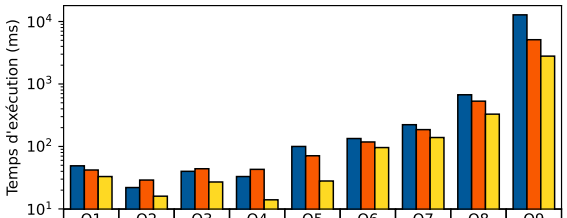
	Q15	Q16	Q17	Q18	Q19	Q20	Q21
base	27	22	43	424	2,104	1,279	5,268
bfs	24	22	42	346	1,292	1,271	5,266
var	16	24	32	253	1,352	1,288	4,793

FIGURE 3.16 – Temps d'exécution (requêtes composites) Watdiv 11M triplets



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
base	15	12	31	31	63	96	255	511	1,023
bfs	15	12	31	31	33	96	234	483	258
var	8	5	16	8	9	49	118	243	130

FIGURE 3.17 – # Requêtes exécutées DBpedia



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
base	49	22	40	33	100	134	223	672	1.3E4
bfs	42	29	44	43	71	118	186	531	5,113
var	33	16	27	14	28	96	139	329	2,792

FIGURE 3.18 – Temps d'exécution DBpedia

3.5.1 Comparaison des algorithmes

Pour les trois algorithmes, nous utilisons la méthode d'exécution COUNT+LIMIT. Ces expérimentations sont menées sur le triplestore JenaTDB. Nous commençons par comparer les performances des algorithmes sur des données synthétiques. Nous utilisons une base de données WatDiv générée de 11 millions de triplets. Les figures 3.11, 3.13 et 3.15 donnent le nombre de requêtes exécutées pour chaque algorithme pour les requêtes en étoile, en chaîne et composite respectivement. Les figures 3.12, 3.14 and 3.16 donnent les temps d'exécution.

Pour les requêtes en étoile (figures 3.11 et 3.12), même lorsque Bfs exécute la moitié des requêtes exécutées par l'algorithme Base (requêtes Q5 et Q7), les temps d'exécution sont similaires. En effet, Bfs évite l'exécution de requêtes qui ont une super-requête qui réussit. Ce sont souvent des requêtes qui réussissent qui ont peu de réponses et donc des temps d'exécution courts. À l'inverse, Var évite aussi l'exécution de requêtes qui échouent. Puisque celles-ci ont des réponses pléthoriques, elles ont un temps d'exécution plus long.

Pour les requêtes en chaîne (figures 3.13 et 3.14), la décomposition des produits cartésiens fait que Base exécute moins de $2^n - 1$ requêtes, la valeur théorique calculée est précisée dans les barres *théorie*. Ainsi, Base exécute souvent le même nombre de requêtes que Bfs, et les deux algorithmes ont des temps d'exécution très proches.

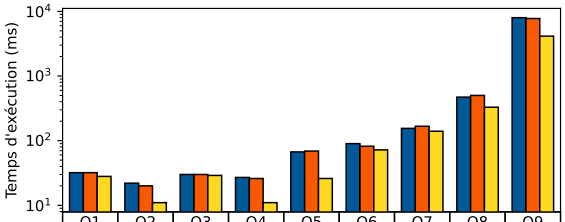
Pour les requêtes composites, les performances se situent entre celles des requêtes en étoiles et en chaîne. Globalement, l'algorithme Bfs fait baisser de 6% le temps d'exécution par rapport à Base et Var le fait diminuer de 29%.

Nous reproduisons les expériences précédentes en utilisant les données et requêtes réelles de DBpedia. Le nombre de requêtes exécutées est donné en figure 3.17 et le temps d'exécution en figure 3.18. Les tendances observées dans les expériences sur des données synthétiques sont confirmées sur des données réelles. Cette fois Bfs diminue le temps d'exécution de 9% par rapport à Base et Var le fait baisser de 46% en moyenne.

Sur les données réelles comme synthétiques, nous avons donc montré que l'utilisation des propriétés permet bien de diminuer le nombre de requêtes à exécuter pour identifier les MFIS et XSS. L'algorithme Var est systématiquement plus efficace que Base et Bfs, sauf pour 4 requêtes pour lesquelles la complexité des traitements peut faire augmenter le temps total jusqu'à 10%. Nous allons maintenant faire varier le seuil des réponses pléthoriques afin de vérifier si ces tendances se maintiennent.

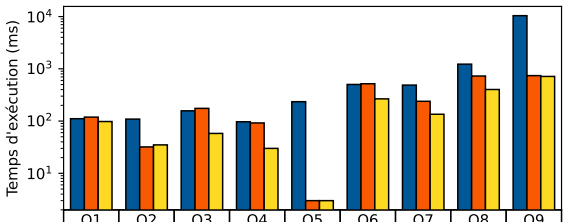
3.5.2 Effet du choix du seuil

Nous avons essayé de déterminer si le choix du seuil a un impact sur les résultats obtenus. Dans l'expérience précédente, le seuil a été fixé à K=100. Nous répétons l'expérience avec la base de connaissances DBpedia et de nouveaux seuils K=10 et K=1000. Les temps d'exécution sont donnés dans les figures 3.19 et 3.20. Avec l'augmentation du seuil, le temps d'exécution



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
base	32	22	30	27	67	90	155	472	7,988
bfs	32	20	30	26	69	82	167	501	7,754
var	28	11	29	11	26	72	140	329	4,139

FIGURE 3.19 – Temps d'exécution K=10



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
base	111	109	157	97	235	502	488	1,228	1.0E4
bfs	119	32	175	92	3	518	239	728	741
var	98	35	58	30	3	266	135	402	715

FIGURE 3.20 – Temps d'exécution K=1000

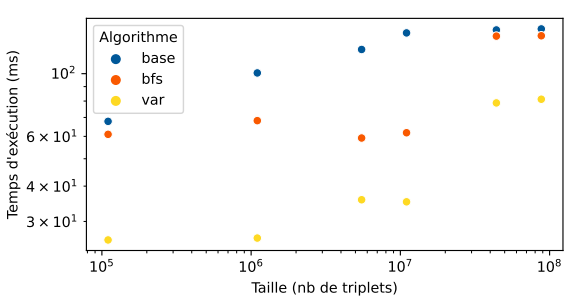
de chaque requête augmente, donc l'amélioration entre les performances de VAR et BASE augmente. Pour les requêtes Q2, Q5, et Q9, le temps d'exécution est faible pour K=1000, car la requête initiale réussit (elle a moins de 1000 réponses) donc seule la requête initiale est exécutée par les algorithmes BFS et VAR. En mettant de côté ces requêtes pour K=1000, BFS fait baisser le temps d'exécution de 1% en moyenne pour K=10, et 13% pour K=1000 (contre 9% pour K=100). VAR fait baisser le temps d'exécution de 28% pour K=10 et 55% pour K=1000 (contre 46% pour K=100). Le changement de seuil du nombre de réponses pléthoriques a pour effet d'accentuer les différences de performances entre les algorithmes, mais la tendance demeure. L'algorithme VAR conserve des temps d'exécution raisonnables pour chaque seuil avec une unique requête dont le temps d'exécution dépasse une seconde.

3.5.3 Effet de la base de connaissances

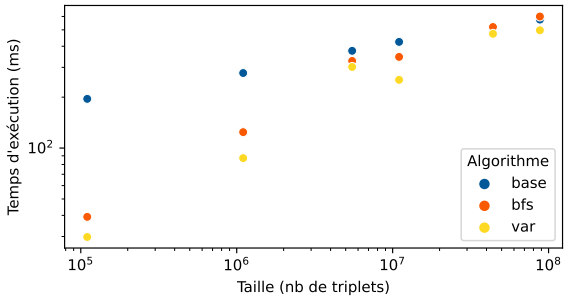
Les expériences précédentes ont été réalisées sur des jeux de données de taille fixe et sur le seul triplestore Jena TDB. Dans cette section, nous proposons des expérimentations faisant variant la taille des données et sur un autre triplestore : Virtuoso.

Pour étudier l'influence de la taille de la base de connaissances, nous avons reproduit les expériences utilisant les données synthétiques avec six bases de connaissances générées comprenant entre cent-mille et quatre-vingt-huit millions de triplets. La figure 3.21 donne l'évolution des temps d'exécution pour deux requêtes typiques. Comme nous pouvons le constater, le temps d'exécution évolue linéairement avec la taille de la base de connaissances. Ceci s'explique, car, comme nous l'avons vu précédemment, le temps d'exécution des algorithmes est directement lié au temps d'exécution des requêtes. Or, ce dernier évolue linéairement selon la taille des données, donc la même tendance est obtenue pour les algorithmes. Le volume des données a un impact important sur les performances des algorithmes. Pour la quantité de données utilisées ici, les durées d'exécution restent raisonnables, mais pour des données de plusieurs milliards de triplets, nous nous attendons à voir une dégradation des performances avec des durées d'exécution qui dépassent la seconde.

Nous comparons à présent les performances des algorithmes sur différents triplestores pour

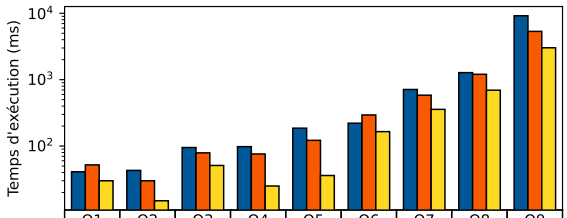


Q3



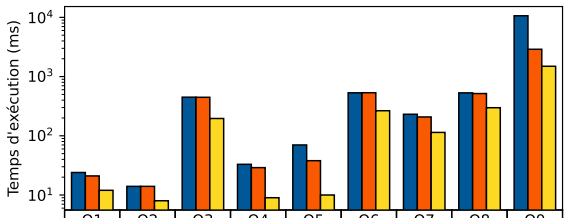
Q18

FIGURE 3.21 – Temps d'exécution en fonction du nombre de triplets dans la base WatDiv



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
base	41	43	95	98	186	221	712	1,277	9,171
bfs	52	30	79	76	122	294	584	1,205	5,340
var	30	15	51	25	36	165	357	694	3,023

FIGURE 3.22 – Temps d'exécution Fuseki



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
base	24	14	449	33	70	532	231	531	1.1E4
bfs	21	14	447	29	38	534	208	517	2,892
var	12	8	196	9	10	265	114	298	1,493

FIGURE 3.23 – Temps d'exécution Virtuoso

étudier l'influence de ces derniers. Les figures 3.22 and 3.23 donnent respectivement les temps d'exécutions des algorithmes exécutés sur Fuseki et Virtuoso avec les données de DBpedia. Ces résultats peuvent être comparés à la figure 3.18 qui présente les performances pour JenaTDB. Malgré le changement de triplestore, les tendances sont confirmées, et avec les mêmes ordres de grandeur. Il reste des différences mineures, par exemple, le fait que Virtuoso soit plus lent pour des requêtes avec un très grand nombre de réponses (Q3 et Q6). Globalement, même pour des requêtes avec beaucoup de réponses, l'algorithme VAR permet de calculer les MFIS et XSS en moins d'une seconde, quel que soit le triplestore utilisé.

3.6 Conclusion

Dans ce chapitre, nous avons adapté la notion de cause d'échec au problème des réponses pléthoriques dans les bases de connaissances. En fournissant les causes d'échec (MFIS) et requêtes alternatives (XSS) aux utilisateurs, nous les aidons à comprendre pourquoi leur requête a produit un nombre trop important de réponses. Ils peuvent ensuite modifier leur requête afin de satisfaire leur besoin.

Nous avons proposé une première méthode naïve pour l'énumération des MFIS et XSS puis avons démontré deux propriétés permettant de réduire le nombre de requêtes à exécuter. En parallèle, des solutions d'optimisation lors de l'implémentation permettent de réduire le

temps d'exécution de chaque requête. En ajoutant les opérateurs COUNT et LIMIT lors de l'exécution, le temps d'exécution diminue de 82%. Notre algorithme Bfs permet de faire diminuer le temps d'exécution de 7% en moyenne et l'algorithme Var de 35%. Nous avons mené des expériences en utilisant des données synthétiques et réelles avec des requêtes conjonctives de formes diverses comportant jusqu'à douze patrons de triplets. Nous avons ainsi montré que les MFIS et XSS peuvent être calculées dans une durée de l'ordre de la seconde, et ce pour de bases de connaissances de taille diverses.

La réalisation d'un outil permettant d'afficher et utiliser les MFIS et XSS depuis un endpoint SPARQL est un premier pas vers un futur travail sur l'explication des MFIS et XSS à l'utilisateur. Il faudra idéalement pouvoir retranscrire l'erreur de façon compréhensible pour l'utilisateur en langue naturelle.

Jusqu'ici, nous avons considéré uniquement les requêtes SPARQL les plus simples, c'est-à-dire les requêtes conjonctives. Dans le prochain chapitre, nous exploitons davantage les spécificités du langage SPARQL avec un objectif double : optimiser à nouveau le calcul des MFIS et XSS en exploitant les cardinalités implicites des prédicats RDF et étendre l'approche proposée aux requêtes contenant les opérateurs principaux de SPARQL : FILTER, UNION et OPTIONAL.

Chapitre 4

Extension aux spécificités de SPARQL

Sommaire

Résumé

Les notions de MFIS et XSS présentées dans le chapitre précédent ont été spécifiquement définies pour les requêtes conjonctives, ce qui est limitant par rapport à ce que permet le langage SPARQL. Dans ce chapitre, nous proposons une formalisation pour appliquer ces notions à des requêtes utilisant les opérateurs FILTER, OPTIONAL et UNION. Par ailleurs, les algorithmes d'énumération présentés précédemment restent généraux et n'exploitent que la forme de la requête. Dans ce chapitre, nous verrons également comment tirer parti d'une information sur les données – les cardinalités – pour optimiser davantage l'énumération des MFIS et XSS. Nous comparons expérimentalement différentes façons de les intégrer.

4.1 Introduction

Nous avons précédemment introduit les notions de MFIS et XSS afin d'expliquer à un utilisateur pourquoi la requête qu'il avait écrite a produit plus de réponses qu'il ne l'attendait. Jusqu'à présent, notre méthode utilise RDF et SPARQL sans prendre en compte leurs spécificités. Ainsi, nous nous sommes limités aux requêtes conjonctives et avons utilisé la forme de la requête pour réduire le nombre de requêtes à exécuter. Dans ce chapitre, nous étudions toujours le problème des réponses pléthoriques, mais introduisons des éléments spécifiques à RDF et SPARQL. L'intérêt est, d'une part, de permettre à un utilisateur d'utiliser les principaux opérateurs de SPARQL. D'autre part, ces spécificités permettent d'optimiser les traitements pour identifier les MFIS et XSS de la requête utilisateur.

Comme spécificité de RDF, nous considérons le fait que les prédicats en RDF sont par défaut multivalués, mais qu'ils peuvent avoir, en pratique, des *cardinalités* spécifiques. Les cardinalités de prédicats sont une façon de compter le nombre d'occurrences d'un prédicat par sujet dans une base de connaissances. Sur le même modèle que la propriété de déduction basée sur les variables du chapitre précédent, nous allons exploiter les cardinalités des prédicats pour déduire l'échec de requêtes sans avoir à les exécuter.

Comme spécificités de SPARQL, nous nous sommes intéressés aux types de requêtes exécutées par les utilisateurs en pratique. Si l'étude de requêtes SPARQL réelles montre que les requêtes conjonctives sont les plus utilisées, environ 45% des requêtes contiennent l'opérateur OPTIONAL, environ 40% contiennent l'opérateur FILTER et environ 25% l'opérateur UNION (Gallego et al., 2011). La plupart des travaux qui s'intéressent aux réponses non satisfaisantes ne considèrent que les requêtes conjonctives (Fokou et al., 2016 ; Godfrey, 1997). À notre connaissance, les seuls travaux considérants des opérateurs spécifiques de SPARQL sont ceux de Wang et al. (2019), qui ont proposé une approche pour identifier si l'échec provient de la partie conjonctive d'une requête ou de l'un de ses opérateurs. Cependant, ces travaux ont été faits dans le contexte du sous-problème *why-not* (absence d'un résultat attendu par l'utilisateur). Notre objectif est de le faire pour le problème des réponses pléthoriques. Nous souhaitons donc enrichir les notions du chapitre précédent afin d'intégrer des requêtes SPARQL complexes.

L'objectif de ce chapitre est de proposer deux extensions à la méthode précédente, en exploitant les cardinalités des prédicats RDF et en intégrant les opérateurs SPARQL. Dans la section 4.2 de ce chapitre, nous étudions l'utilisation des cardinalités des prédicats pour optimiser la recherche des MFIS et XSS. La section 4.3 est ensuite dédiée au support des requêtes non conjonctives. Nous présenterons enfin une analyse expérimentale dans la section 4.4.

4.2 Utilisation des cardinalités des prédicats

Pour limiter le nombre de requêtes exécutées pour identifier les MFIS et XSS, nous nous intéressons aux patrons de triplets dont la contribution est d'ajouter une information à chaque réponse, mais sans changer le nombre total de réponses. Considérons le patron de triplet t_2 : *?d expérience ?e* de l'exemple 3.1 (page 58). Si ce patron de triplet est retiré de la requête, comme chaque personne a au plus une valeur pour son nombre d'années d'expérience, chaque réponse perdra une partie de l'information, mais deux réponses ne deviendront pas identiques. Donc le nombre de réponses ne peut qu'augmenter si ce patron de triplet est retiré. Pour formaliser cette notion, nous introduisons la définition des cardinalités de prédicats (Dellal, 2019). Nous définissons trois types de cardinalité : globale, de classe et de Characteristic Set, puis présentons les propriétés qui utilisent les cardinalités pour optimiser les algorithmes d'énumération des MFIS et XSS.

4.2.1 Définitions

La cardinalité est une mesure du nombre d'occurrences d'un prédicat par sujet dans une base de données. Les définitions de cardinalités minimales et maximales d'un prédicat p pour un ensemble de sujets S sont :

$$\text{cardinalite}_{\min}(p, S) = \min_{s \in S} \text{count}(s, p) = \min_{s \in S} |\{(s\ p\ o) \mid (s\ p\ o) \in D\}|$$
$$\text{cardinalite}_{\max}(p, S) = \max_{s \in S} \text{count}(s, p) = \max_{s \in S} |\{(s\ p\ o) \mid (s\ p\ o) \in D\}|$$

Dans la suite, nous détaillerons trois méthodes pour calculer les cardinalités.

4.2.1.1 Cardinalités globales

La forme la plus simple de cardinalité est la cardinalité globale, qui est calculée sur l'ensemble des sujets d'une base de connaissances D : $S = \text{subject}(D) = \{s \mid \exists\, p, o : (s\ p\ o) \in D\}$.

Définition 4.1 (cardinalités globales)**.** *Les cardinalités globales d'un prédicat p sont :*

$$\text{cardinalite\_globale}_{\min}(p) = \min_{s \in \text{subject}(D)} \text{count}(s, p)$$

$$\text{cardinalite\_globale}_{\max}(p) = \max_{s \in \text{subject}(D)} \text{count}(s, p)$$

Exemple 4.1 (cardinalités globales)**.** *En utilisant l'exemple de la figure 3.1, les cardinalités globales sont données dans la table 4.1.*

prédicat	cardinalite_globale$_{\min}$	cardinalite_globale$_{\max}$
expérience	0	1
collègue	0	2
traite	0	2
type	0	1
soigne	0	3
roleOpératoire	0	1

TABLE 4.1 – Cardinalités globales

Un désavantage de la cardinalité globale est son manque de précision. La plupart des prédicats ont une cardinalité minimale de 0, car il est rare qu'un prédicat soit présent pour tous les sujets d'une base de connaissances. En regroupant les sujets par classe, nous pouvons calculer des cardinalités sur chacune des classes et ainsi affiner les valeurs de cardinalité.

4.2.1.2 Cardinalités de classe

Lorsqu'un schéma est associé à des données RDF, des classes permettent de grouper les sujets d'une base de connaissances. Les classes peuvent être utilisées pour affecter un type aux sujets d'une base de connaissances. Les instances d'une classe C dans une base de connaissances D sont définies par $\text{instances}(C) = \{s \mid \exists(s, \textit{rdf:type}, C) \in D\}$.

Définition 4.2 (cardinalités de classe)**.** *Les cardinalités de classes sont définies comme :*

$$\text{cardinalite\_classe}_{\min}(p, C) = \min_{s \in \text{instances}(C)} \text{count}(s, p)$$

$$\text{cardinalite\_classe}_{\max}(p, C) = \max_{s \in \text{instances}(C)} \text{count}(s, p)$$

Exemple 4.2 (cardinalités de classe)**.** *Dans notre exemple, sachant que InfirmierUrgence et InfirmierBloc sont des sous-classes de Infirmier :*

- cardinalite_classe$_{\min}$(*soigne, InfirmierBloc*) = 1 ;
- cardinalite_classe$_{\min}$(*soigne, Infirmier*) = 1 ;
- cardinalite_classe$_{\max}$(*soigne, InfirmierBloc*) = 1 ;
- cardinalite_classe$_{\max}$(*soigne, Infirmier*) = 3 ;

Contrairement aux cardinalités globales, les cardinalités de classe ne sont pas universelles. Elles doivent être déterminées en fonction de la requête considérée. Pour utiliser les cardinalités

```
SELECT * WHERE {
  ?p collegue ?s .          # t1
  ?s soigne ?c .            # t2
  ?n soigne ?c .            # t3
  ?s roleOperatoire ?g }    # t4
```

(a) Requête $Q_1 = t_1t_2t_3t_4$

```
SELECT * WHERE {
  ?p collegue ?s .      # t1
  ?s experience ?e .    # t2
  ?s soigne ?c }        # t3
```

(b) Requête $Q_2 = t_1t_2t_3$

FIGURE 4.1 – Deux requêtes SPARQL

de classe, il faut donc déterminer la ou les classes adaptées. Considérons un patron de triplet *t=(?s p ?o)* avec un sujet variable s et un prédicat fixé p. Pour connaître la cardinalité du prédicat p qui s'applique dans ce cas, il faut restreindre les classes d'appartenance possible des valeurs de la variable s. Pour notre application, nous cherchons la cardinalité d'un prédicat dans le cadre d'une requête. Nous pouvons donc utiliser les prédicats des autres patrons de triplets pour déterminer les classes concernées. La propriété domaine de RDFS *(rdfs:domain)* s'applique à un prédicat et indique la ou les classes d'appartenance du sujet associé.

$$\text{domaine}(p) = \bigcap_{C|\exists(p \ \textit{rdfs:domain} \ C)\in D} C$$

Considérons une requête Q et notons *t=(?s p ?o)* le patron de triplet pour lequel nous cherchons la cardinalité de classe du prédicat p. Il faut déterminer la ou les classes d'appartenance du mapping de s. Ce dernier appartient à tous les domaines des prédicats avec lesquels il partage un patron de triplet. Nous définissons donc une fonction pour déterminer les classes utiles pour le calcul des cardinalités :

$$\text{classes}(p, s, Q) = \bigcap_{t'\in Q \wedge s(t')=s} \text{domaine}(p(t'))$$

Exemple 4.3 (classes pertinentes pour un prédicat)**.** *Reprenons notre exemple et considérons les requêtes Q_1 et Q_2 de la figure 4.1. Les domaines des propriétés impliquées sont :*

- domaine(*roleOpératoire*) = *InfirmierBloc ;*
- domaine(*soigne*) = *Infirmier ;*
- domaine(*expérience*) = *Personne.*

Nous avons donc :

- classes($soigne, s, Q_1$) = $InfirmierBloc \cap Infirmier = InfirmierBloc$;
- classes($soigne, n, Q_1$) = $Infirmier$;
- classes($soigne, s, Q_2$) = $Infirmier \cap Personne = Infirmier$.

Une fois les classes définies, nous pouvons finalement définir la cardinalité maximale de classe d'un prédicat p, associé au sujet s dans une requête Q. Comme le mapping de s ap-

partient à chacune des classes de classes(p, s, Q), la cardinalité maximale est inférieure à la cardinalité maximale dans chacune des classes. De la même façon, la cardinalité minimale est supérieure à la cardinalité minimale dans chacune des classes.

Définition 4.3 (cardinalités de classes dans une requête)**.** *Nous avons :*

$$\text{cardinalite\_classe}_{\min}(p, s, Q) = \max_{C \in \text{classes}(p,s,Q)} \text{cardinalite\_classe}_{\min}(p, C)$$

$$\text{cardinalite\_classe}_{\max}(p, s, Q) = \min_{C \in \text{classes}(p,s,Q)} \text{cardinalite\_classe}_{\max}(p, C)$$

Exemple 4.4 (cardinalités de classes dans une requête)**.** *Dans l'exemple précédent, les cardinalités de classe sont les suivantes :*

- cardinalite_classe$_{\min}(soigne, s, Q_1) = 1$;
- cardinalite_classe$_{\min}(soigne, n, Q_1) = 1$;
- cardinalite_classe$_{\min}(soigne, s, Q_2) = 1$;
- cardinalite_classe$_{\max}(soigne, s, Q_1) = 1$;
- cardinalite_classe$_{\max}(soigne, n, Q_1) = 3$;
- cardinalite_classe$_{\max}(soigne, s, Q_2) = 3$.

Toutes les bases de connaissance n'utilisent pas le formalisme RDFS pour décrire les classes. Dans ce cas, une autre façon de regrouper les sujets a été proposée. Nous l'abordons dans la section suivante.

4.2.1.3 Cardinalités de Characteristic Set

Un Characteristic Set (CS) est défini par Neumann et Moerkotte (2011) comme l'ensemble des prédicats d'un sujet. Pour un élément s dans une base de connaissances D, le Characteristic Set de s est : $S_C(s) = \{p \mid \exists o, (s\ p\ o) \in D\}$.

L'ensemble de tous les CS d'une base de connaissances D est défini par $S_C(D) = \{S_C(s) \mid \exists p, o : (s\ p\ o) \in D\}$. La notion de CS permet d'identifier la structure des données RDF (Pham et al., 2015). Les entités qui partagent les mêmes CS ont tendance à être similaires. Les CS peuvent être annotés par le nombre d'entités qui partagent ce CS et le nombre d'occurrences de chaque prédicat pour les entités qui partagent ce CS.

Exemple 4.5 (characteristic set)**.** *Dans notre exemple, il y a cinq CS :*

- $S_1 = \{experience,\ collegue,\ traite\}$;
- $S_2 = \{collegue\}$;
- $S_3 = \{type,\ soigne\}$;
- $S_4 = \{type,\ soigne,\ roleOperatoire\}$;
- $S_5 = \{type,\ soigne,\ experience\}$.

Puisque chaque sujet a exactement un CS, nous pouvons regrouper les sujets qui ont le même CS et calculer les cardinalités des prédicats au sein de ces groupes. Pour un ensemble de prédicats P et un prédicat p dont nous voulons connaître la cardinalité :

$$\text{cardinalite\_CS}_{\min}(p, P) = \min_{s, S_C(s)=P} \text{count}(s, p)$$

$$\text{cardinalite\_CS}_{\max}(p, P) = \max_{s, S_C(s)=P} \text{count}(s, p)$$

Exemple 4.6 (cardinalités de CS)**.** *En utilisant notre exemple et les CS définis ci-dessus :*

- $\text{cardinalite\_CS}_{\max}(soigne, S_3) = 3$;
- $\text{cardinalite\_CS}_{\max}(soigne, S_4) = 1$;
- $\text{cardinalite\_CS}_{\max}(soigne, S_5) = 1$.

Comme pour les cardinalités de classe, lorsque l'on considère une requête Q et un prédicat t dont nous voulons connaître la cardinalité, nous devons connaître les CS concernés. En particulier, nous cherchons à déterminer aussi précisément que possible tous les CS potentiels du mapping du sujet s associé à ce prédicat. Le CS d'un mapping de s contient nécessairement tous les prédicats contenus dans des patrons de triplet de la requête Q pour lesquels le sujet est s.

Définition 4.4 (CS pertinents pour une requête)**.** *Pour un prédicat p, un sujet s et une requête Q, les CS à considérer sont définis par :*

$$S_C(p, s, Q) = \bigcup_{P \in S_C(D), \forall t' \in Q(s(t')=s \Rightarrow p(t') \in P)} P$$

Exemple 4.7 (CS pertinents pour une requête)**.** *En utilisant notre exemple et les requêtes de l'exemple 4.3 :*

- $S_C(soigne, s, Q_1) = \{S_4\}$;
- $S_C(soigne, s, Q_2) = \{S_5\}$;
- $S_C(soigne, n, Q_1) = \{S_3, S_4, S_5\}$.

Contrairement à la définition pour les cardinalités de classe, qui fait intervenir une intersection de classes, ici le mapping de s a pour CS l'un des CS de $S_C(p, s, Q)$. Pour borner la cardinalité, il faut donc retenir le minimum des cardinalités minimales et le maximum des cardinalités maximales.

Définition 4.5 (cardinalités CS pour une requête)**.** *Pour une requête Q, un sujet s et un prédicat p,*

$$\text{cardinalite\_CS}_{\min}(p, s, Q) = \min_{P \in S_C(p,s,Q)} \text{cardinalite\_CS}_{\min}(p, P)$$

$$\text{cardinalite\_CS}_{\max}(p, s, Q) = \max_{P \in S_C(p,s,Q)} \text{cardinalite\_CS}_{\max}(p, P)$$

Exemple 4.8 (cardinalités CS pour une requête)**.** *En utilisant notre exemple et les requêtes de l'exemple 4.3 :*

- cardinalite_CS$_{\min}(soigne, s, Q_1) = 1$;
- cardinalite_CS$_{\min}(soigne, s, Q_2) = 1$;
- cardinalite_CS$_{\min}(soigne, n, Q_1) = 1$;
- cardinalite_CS$_{\max}(soigne, s, Q_1) = 1$;
- cardinalite_CS$_{\max}(soigne, s, Q_2) = 1$;
- cardinalite_CS$_{\max}(soigne, n, Q_1) = 3$.

Les cardinalités sont déterminées en exécutant des requêtes sur la base de connaissances et peuvent ensuite être stockées comme méta-données. Elles sont susceptibles d'évoluer lors de l'ajout de nouvelles données dans la base. Elles doivent donc être maintenues à jour. Par ailleurs, les cardinalités de classe peuvent uniquement être calculées en présence d'un schéma RDFS. Le volume de données liées aux cardinalités dépend du nombre de prédicats dans la base de connaissances. Pour les cardinalités globales, il y a deux valeurs (min et max) pour chaque prédicat. Le nombre d'informations sera plus élevé pour les cardinalités de classe, car elles impliquent deux cardinalités par prédicat et par classe de son domaine. Pour les cardinalités de CS, il y aura deux cardinalités par prédicat pour chaque CS contenant ce prédicat.

Comme nous l'avons vu, contrairement aux cardinalités globales, les cardinalités de classe et de CS sont spécifiques à une requête. Cela signifie que pour utiliser les cardinalités de classe et de CS sur un treillis de requêtes, elles ne peuvent pas être déterminées une fois sur la requête initiale (comme peuvent l'être les cardinalités globales), mais doivent être recalculées pour chaque requête du treillis. Les cardinalités de classe et de CS ont cependant l'avantage d'être toujours au moins aussi précises que les cardinalités globales. La table 4.2 l'illustre en récapitulant les différentes cardinalités du prédicat *soigne*.

sujet	**requête**	**globale**		**classe**		**CS**	
		min	**max**	**min**	**max**	**min**	**max**
s	Q_1	0	3	1	1	1	1
n	Q_1	0	3	1	3	1	3
s	Q_2	0	3	1	3	1	1

TABLE 4.2 – Cardinalités du prédicat *soigne*

4.2.2 Propriétés

Les cardinalités peuvent être utilisées pour déterminer le succès ou l'échec d'une requête à partir de celui d'une autre requête. Nous commençons par montrer comment la cardinalité

maximale peut être utilisée pour relier les nombres de résultats de différentes requêtes.

Propriété 4.1 (cardinalité maximale)**.** *Soient une requête Q et un patron de triplet t avec un prédicat constant $p(t)$ de cardinalité maximale non nulle et $s(t) \in var(Q)$. Nous avons* $\text{cardinalite}_{\max}(p(t), s(t), Q \wedge t).|[[Q]]| \geq |[[Q \wedge t]]|$.

Démonstration. Pour démontrer la propriété, nous définissons la fonction suivante. Elle correspond à l'ensemble des mappings du résultat de l'évaluation de $Q \wedge t$ avec leurs domaines restreints à $\text{var}(Q)$:

$$f_{Q,t} : \begin{cases} [[Q \wedge t]] \rightarrow [[Q]] \\ \qquad \mu \mapsto \mu_{|\text{var}(Q)} \end{cases}$$

Nous définissons également la fonction $a_{Q,t}(\mu)$, qui correspond au nombre d'images réciproques d'un élément μ de $[[Q]]$ par $f_{Q,t}$, c'est-à-dire, $a_{Q,t}(\mu) = |\{\mu' \mid f_{Q,t}(\mu') = \mu\}|$. Notons également que si $\text{var}(Q) = \text{var}(Q \wedge t)$, la propriété 3.4 peut être utilisée pour affirmer que $|[[Q]]| \geq [[Q \wedge t]]|$. La propriété est donc vérifiée ($\text{cardinalite}_{\max}(p(t), s(t), Q \wedge t) \geq 1$). Nous allons montrer :

1. $\text{cardinalite}_{\max}(p(t), s(t), Q \wedge t) \geq \max_{s \in \{\mu(s(t)), \mu \in [[Q]]\}} \text{count}(s, p(t))$ pour les trois définitions des cardinalités ;
2. $a_{Q,t}(\mu) = count(\mu(s(t)), p(t))$.

(1) Notons $S = \{\mu(s(t)), \mu \in [[Q]]\}$.

Pour les cardinalités globales, $\text{cardinalite\_globale}_{\max}(p(t)) = \max_{s \in \text{subject}(D)} \text{count}(s, p)$ et $S \subseteq \text{subject}(D)$. Ainsi $\text{cardinalite\_globale}_{\max}(p(t)) \geq \max_{s \in S} \text{count}(s, p(t))$.

Pour les cardinalités de classe, considérons $s \in S$ et une classe C telle que $\exists t_1 \in Q, s(t_1) = s(t) \wedge C \subseteq \text{domain}(p(t_1))$. D'après la définition de s, $\exists o \mid (s\ p(t_1)\ o) \in D$, donc s appartient à toutes les classes qui sont des domaines de $p(t_1)$, en particulier, $s \in \text{instances}(C)$. Donc $\text{count}(s, p) \leq \max_{s \in \text{instances}(C)} \text{count}(s, p) = \text{cardinalite\_classe}_{\max}(p(t), C)$. Comme cette propriété est vérifiée pour toutes les telles classes C, $\text{count}(s, p) \leq \text{cardinalite\_classe}_{\max}(p(t), s(t), Q \wedge t)$, pour tout $s \in S$. Donc $\text{cardinalite\_classe}_{\max}(p(t), s(t), Q \wedge t) \geq max_{s \in S}\ \text{count}(s, p(t))$.

Pour les cardinalités CS, considérons $s \in S$. Notons P le CS de s : $S_C(s) = P$. Nous avons $\text{count}(s, p(t)) \leq \max_{s', S_C(s') = P} \text{count}(s', p(t)) = \text{cardinalite\_CS}_{\max}(p(t), P)$. P contient les prédicats de chaque triplet $t' \in Q$ tel que $s(t) = s(t')$, donc $\forall t' \in Q\,(s(t') = s(t) \Rightarrow p(t') \in P)$. Ainsi, $\exists P$ tel que $\forall t' \in Q\,(s(t') = s(t) \Rightarrow p(t') \in P) \wedge \text{count}(s, p(t)) \leq \text{cardinalite\_CS}_{\max}(p(t), P)$, c'est-à-dire que $\text{count}(s, p(t)) \leq \text{cardinalite\_CS}_{\max}(p(t), s(t), Q \wedge t)$ pour tout $s \in S$, et $\text{cardinalite\_CS}_{\max}(p(t), s(t), Q \wedge t) \geq \max_{s \in S}\ \text{count}(s, p(t))$.

(2) Considérons $\mu \in [[Q]]$, et μ_1 une image réciproque de μ par $f_{Q,t}$. Le mapping μ_1 est entièrement défini par les valeurs de $\mu_1(v)$ pour $v \in \text{var}(Q \wedge t)$. Comme $p(t)$ n'est pas une variable et $s(t) \in \text{var}(Q)$, $\text{var}(Q \wedge t) = \text{var}(Q) \cup o(t)$. Par ailleurs $\forall v \in \text{var}(Q),\ \mu_1(v) = \mu(v)$, donc μ_1 est entièrement fixé par $\mu_1(o(t))$. Nous avons donc $a_{Q,t}(\mu) = |\{(\mu(s(t))\ p(t)\ o(t)) \in$

$D\}| = \text{count}(\mu(s(t)), p(t))$.
En combinant (1) et (2), $\forall \mu \in [[Q]]$, $\text{count}(\mu(s(t)), p(t)) \leq \max_{s \in S} \text{count}(s, p(t))$ donc $\forall \mu \in [[Q]], a_{Q,t}(\mu) \leq \text{cardinalite}_{\max}(p(t), s(t), Q \wedge t)$.
Enfin, par définition de $f_{Q,t}$ et $a_{Q,t}$, $|[[Q \wedge t]]| = \sum_{\mu \in [[Q]]} a_{Q,t}(\mu)$, puis $\sum_{\mu \in [[Q]]} a_{Q,t}(\mu) \leq \sum_{\mu \in [[Q]]} \text{cardinalite}_{\max}(p(t), s(t), Q \wedge t) = |[[Q]]|.\, \text{cardinalite}_{\max}(p(t), s(t), Q \wedge t)$.

□

Corollaire 4.1 (cardinalité maximale égale à 1)**.** *Pour une requête Q et un patron de triplet t avec un prédicat constant $p(t)$ et $s(t) \in var(Q)$. Si* $\text{cardinalite}_{\max}(p(t)) = 1$ *alors* $FAIL_K(Q \wedge t) \Rightarrow FAIL_K(Q)$.

Démonstration. Si $\text{FAIL}_K(Q \wedge t)$ alors $|[[Q \wedge t]]| > K$. Par ailleurs, si $\text{cardinalite}_{\max}(p(t)) = 1$, alors nous pouvons réécrire la propriété 4.1 $\text{cardinalite}_{\max}(p(t)).|[[Q]]| = |[[Q]]| \geq |[[Q \wedge t]]| > K$ donc $\text{FAIL}_K(Q)$. □

Corollaire 4.2 (cardinalité maximale quelconque)**.** *Soient une requête Q et un patron de triplet t avec un prédicat constant $p(t)$, si $s(t) \in var(Q)$ et* $|[[Q \wedge t]]| / \text{cardinalite}_{\max}(p(t), Q) > K$ *alors* $FAIL_K(Q)$.

Démonstration. Avec la propriété 4.1, nous avons $\text{cardinalite}_{\max}(p(t)).|[[Q]]| \geq |[[Q \wedge t]]|$ donc $|[[Q]]| \geq |[[Q \wedge t]]| / \text{cardinalite}_{\max}(p(t), Q)$. Donc si $|[[Q \wedge t]]| / \text{cardinalite}_{\max}(p(t), Q) > K$, $|[[Q]]| > K$ et $\text{FAIL}_K(Q)$. □

En appliquant l'une de ces propriétés dans nos algorithmes, nous pouvons déduire l'échec d'une requête sans l'exécuter. Pour utiliser la seconde propriété, si la cardinalité maximale n'est pas 1, il n'est plus suffisant de savoir si $Q \wedge t$ échoue, il faut connaître précisément le nombre de réponses.

4.2.3 Algorithme Full

Notre algorithme Full exploite les propriétés présentées dans le chapitre 3 ainsi qu'une propriété basée sur la cardinalité (globale, de classe ou de CS). Il est présenté dans l'algorithme 4.1. Cet algorithme reprend la structure de l'algorithme Var mais ajoute l'utilisation de la propriété basée sur les cardinalités (lignes 21–22). La figure 4.2 illustre son utilisation pour notre exemple. Par rapport à l'algorithme Var, trois exécutions de requêtes supplémentaires sont évitées. Par exemple, comme $t_1t_2t_3t_4t_5$ échoue et que t_2 a une cardinalité maximale globale de 1 (chaque sujet a au maximum une valeur pour le nombre d'années d'expérience), nous en déduisons que $t_1t_3t_4t_5$ échoue. Au total, l'algorithme Full exécute ici seulement cinq requêtes, contre 31 pour l'algorithme naïf Base.

Contrairement aux propriétés introduites dans le chapitre 3 qui dépendent uniquement de l'information contenue dans la requête initiale et sont donc applicables à n'importe quelle

Algorithme 4.1: Énumération des MFIS et XSS d'une requête Q

```
1  FULL(Q, K)
       entrées: Une requête Q = t_1 ∧ ... ∧ t_n
                Un seuil K
       sorties : MFIS et XSS de Q
2      mfis ← ∅, xss ← ∅, fis ← ∅, queryStatus ← ∅;
3      list ← {lattice(Q)};
4      while list ≠ ∅ do
5          Q' ← première requête de list en parcours en largeur;
6          list ← list − {Q'};
7          parents_fis ← true;
8          superqueries ← superQueries(Q);
9          foreach Q'' ∈ superqueries do
10             parents_fis ← parents_fis ∧ ((Q'') ∈ fis);
11         if parents_fis then
12             if Q' ∉ queryStatus then
13                 queryStatus [Q'] ← FAIL_K(Q');
14             if queryStatus [Q'] then // si Q' échoue
15                 fis ← fis ∪ {Q'};
16                 mfis ← mfis − superqueries ;
17                 mfis ← mfis ∪ {Q'} ;
18                 foreach t ∈ triplePatterns(Q') do
19                     if var(Q' − t) = var(Q') then
20                         queryStatus [Q' − t] ← true
21                     else if cardinalite_max(p(t)) = 1 ∧ s(t) ∈ var(Q' − t) then
22                         queryStatus [Q' − t] ← true
23             else // Q' réussit et est donc une XSS
24                 xss ← xss ∪ {Q'};
25     return mfis, xss;
```

requête dans toute base de connaissances, les propriétés basées sur la cardinalité nécessitent de connaître les cardinalités de tous les prédicats.

Pour déterminer la complexité de l'algorithme, nous pouvons réutiliser la propriété du chapitre 3 pour l'étude de la complexité avec les requêtes en étoile (3.5). De la même façon, pour une requête en étoile avec n patrons de triplet, où k est le nombre de patrons de triplet sur lesquels s'applique le corollaire 4.1 avec les cardinalités globales (avec $k \geq 1$) le nombre d'exécutions de requêtes évitées par l'utilisation du corollaire 4.1 est donné par :

$$\left(\sum_{i=1}^{k} 2^{n-i}\right) - 1$$

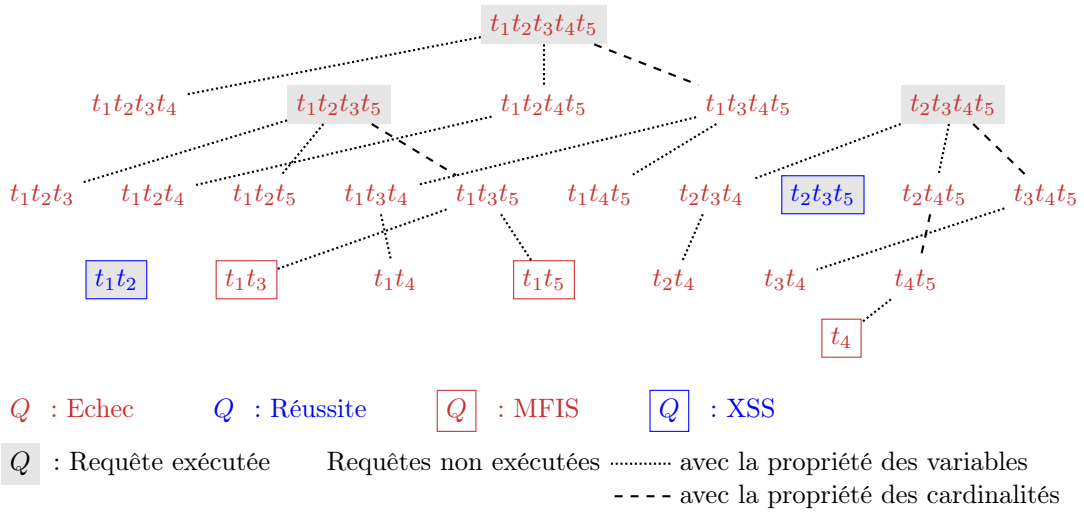


FIGURE 4.2 – Treillis des sous-requêtes de Q

Cette formule est valable pour les cardinalités globales, car la cardinalité est la même dans tout le treillis. Pour chaque triplet auquel s'applique la propriété basée sur la cardinalité, le nombre de requêtes exécutées est divisé par deux. L'utilisation des cardinalités de classe et de CS permet d'éviter au minimum autant de requêtes que l'utilisation des cardinalités globales. Ainsi, cette formule correspond à une borne inférieure du nombre d'exécutions de requêtes évitées par l'utilisation de ces cardinalités. Un calcul exact pour les cardinalités de classe ou de CS impliquerait d'étudier sur quelles requêtes du treillis la propriété de cardinalité peut être appliquée à un patron de triplet. Nous montrerons expérimentalement l'impact de l'utilisation des cardinalités et les différences entre les différents types de cardinalités en fin de chapitre, dans la section 4.4.1.

4.3 Effets des opérateurs SPARQL

Jusqu'à présent, les requêtes considérées dans nos travaux sont des requêtes conjonctives. Dans les deux prochaines sections, nous allons étudier l'application des MFIS et XSS pour des requêtes SPARQL quelconques. Ainsi, nous verrons comment l'intégration des opérateurs FILTER, OPTIONAL et UNION affecte la méthode précédemment définie. Dans un premier temps, nous devons redéfinir les sous-requêtes dans ce cadre afin de formaliser la relation d'ordre entre les requêtes. Par la suite, nous verrons quelles propriétés nous pouvons appliquer pour l'énumération des MFIS et XSS.

4.3.1 Décomposition en sous-requêtes

Pour les requêtes conjonctives, les sous-requêtes sont obtenues en retirant un par un les patrons de triplet. Nous devons maintenant définir le treillis de sous-requêtes en présence d'opérateurs. Nous commençons par rappeler la sémantique de chaque opérateur, puis nous définirons les sous-requêtes dans chaque cas.

4.3.1.1 FILTER

L'opérateur FILTER est défini par une condition d'égalité ou d'inégalité entre variables, constantes et IRI qui est appelée condition de filtre. Pour un patron de graphe P et une condition de filtre R : $[[P\ FILTER\ R]] = \{\mu \in [[P]], \mu\ satisfait\ R\}$.

Pour les requêtes avec un opérateur FILTER, les sous-requêtes sont créées sur le même modèle que les requêtes conjonctives, c'est-à-dire que l'on supprime un à un les patrons de triplets ou les conditions de filtre. Afin de respecter les contraintes du langage SPARQL, les variables présentes dans une condition de filtre doivent également faire partie des patrons de triplet de la requête. Ainsi, en construisant le treillis des sous-requêtes, nous devons vérifier cette condition. Les sous-requêtes qui ne respectent pas cette condition sont considérées invalides et sont donc écartées.

Exemple 4.9 (requête avec FILTER)**.** *En utilisant la requête Q de notre exemple, nous ajoutons $FILTER\ ?e \geq 20$, noté f_1, pour créer la requête $Q' = t_1t_2t_3t_4t_5f_1$. Considérons la sous-requête $Q" = t_1t_3t_4t_5f_1$. Puisque la variable e apparaît dans f_1, mais pas dans $t_1t_3t_4t_5$, $Q"$ n'est pas une requête SPARQL valide et ne doit donc pas être intégrée dans le treillis. Dans cet exemple, toute sous-requête qui contient f_1 doit également contenir t_2.*

4.3.1.2 OPTIONAL

Pour deux patrons de graphe P_1 et P_2, la définition de l'opérateur OPTIONAL est : $[[P_1\ OPT\ P_2]] = [[P_1]] \bowtie [[P_2]] \cup [[P_1]] \backslash [[P_2]]$ où $\Omega_1 \backslash \Omega_2 = \{\mu \in \Omega_1 | \forall \mu' \in \Omega_2, \mu$ et μ' ne sont pas compatibles$\}$.

Pour former les sous-requêtes, nous partons également du modèle des requêtes conjonctives, c'est-à-dire que l'on retire un à un les patrons de triplet. Nous obtenons alors, parmi les sous-requêtes, des requêtes où la clause optionnelle ne contient aucun patron de triplet et à l'inverse des requêtes où seule la clause optionnelle contient des patrons de triplet. Dans les deux cas, nous pouvons simplifier ces expressions.

Propriété 4.2 (OPTIONAL et requête vide)**.** *Pour un patron de graphe P, $[[\emptyset\ OPT\ P]] = [[P]]$ et $[[P\ OPT\ \emptyset]] = [[P]]$*

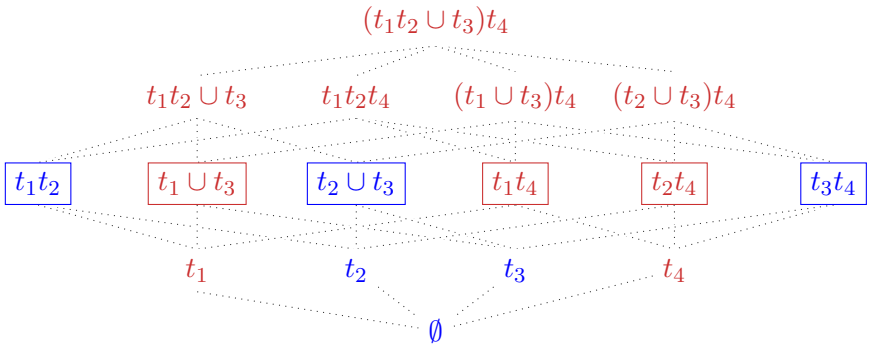
Démonstration. $[[\emptyset\ OPT\ P]] = [[\emptyset]] \bowtie [[P]] \cup [[\emptyset]] \backslash [[P]] = [[\emptyset]] \bowtie [[P]] = [[P]]$
$[[P\ OPT\ \emptyset]] = [[P]] \bowtie [[\emptyset]] \cup [[P]] \backslash [[\emptyset]] = [[P]] \cup [[\emptyset]] = [[P]]$ □

```
SELECT * WHERE {
  { { ?d traite ?p .              # t1
      ?d type Chirurgien }        # t2
    UNION
    { ?d type Cardiologue } } .   # t3
      ?d collegue ?n }            # t4
```

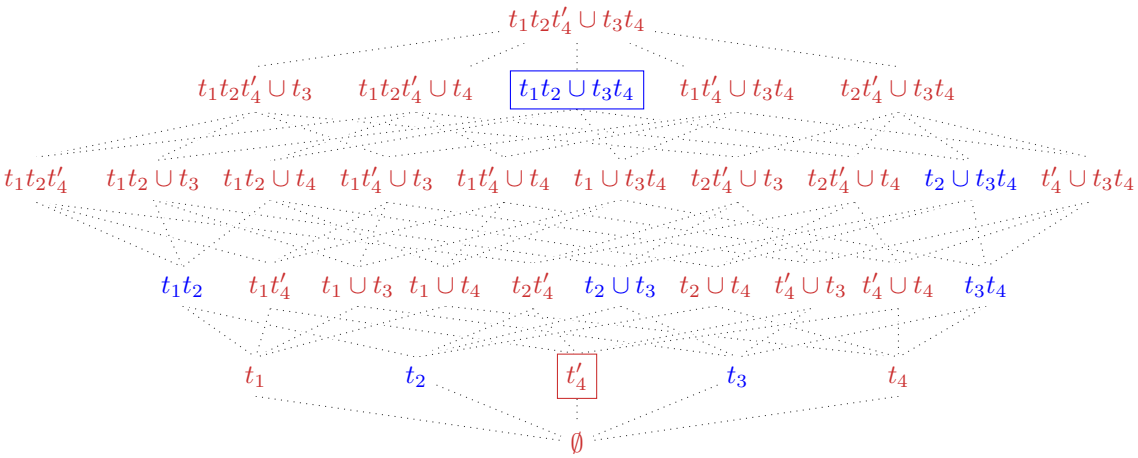
(a) Q_3 sous forme factorisée

```
SELECT * WHERE {
  { ?d traite ?p .              # t1
    ?d type Chirurgien .        # t2
    ?d collegue ?n }            # t4
  UNION
  { ?d type Cardiologue .       # t3
    ?d collegue ?n } }          # t4'
```

(b) Q_3 sous forme développée



(c) forme factorisée



(d) forme développée

Q : Echec Q : Réussite Q : MFIS Q : XSS

FIGURE 4.3 – Deux décompositions possibles pour une requête $Q_3 = (t_1t_2 \cup t_3)t_4 = t_1t_2t_4 \cup t_3t'_4$

4.3.1.3 UNION

Pour deux patrons de graphe P_1 et P_2, l'opérateur UNION est défini par : $[[P_1\ \mathit{UNION}\ P_2]] = [[P_1]] \cup [[P_2]]$. Les sous-requêtes directes de A UNION B sont les requêtes A UNION B' et A' UNION B, où A' et B' sont des sous-requêtes directes de A et B.

Nous avons utilisé les patrons de triplets comme base de la décomposition des requêtes.

Une difficulté supplémentaire à considérer quand nous manipulons des requêtes SPARQL non conjonctives est qu'une requête peut être écrite de deux façons différentes en termes de patrons de triplets, mais en conservant la même signification.

En effet, l'opérateur UNION dans SPARQL est distributif par rapport aux opérateurs de conjonction, FILTER et OPTIONAL. Cela signifie qu'une requête contenant des opérateurs UNION peut être écrite de différentes manières. Pour que notre méthode soit cohérente, nous ne pouvons pas permettre que deux requêtes équivalentes conduisent à différents calculs de MFIS et XSS.

Nous appellerons *forme factorisée* une requête qui ne contient pas plusieurs fois le même patron de triplet. Dans le cas contraire, nous parlons de *forme développée*. Pour des requêtes sous forme développée, lors de la création du treillis des sous-requêtes, nous considérons séparément les différentes occurrences d'un même patron de triplet. Nous l'illustrons avec un exemple basé sur la base de connaissances du chapitre 3.

Exemple 4.10 (forme factorisée, forme développée)**.** *Soit une requête* $Q_3 = (t_1 t_2 \cup t_3) t_4$*, détaillée dans la figure 4.3 sous forme factorisée (4.3a) et sous forme développée (4.3b). Pour la forme développée, nous notons* t_4 *et* t'_4 *les deux occurrences du patron de triplet (?d collegue ?n) afin de les distinguer lors de la décomposition. Dans cet exemple, nous avons repris un seuil K=3 pour les réponses pléthoriques et avons tracé les treillis des sous-requêtes dans les figures 4.3c et 4.3d. À partir de ces treillis, nous pouvons déterminer les MFIS et XSS. Non seulement les requêtes qui composent le treillis sont différentes, mais les MFIS et XSS le sont aussi. Dans la forme développée, la MFIS* t'_4 *pose de plus un problème d'interprétation. En effet,* t'_4 *est une MFIS, mais* t_4 *représente le même patron de triplet et* $t_3 t_4$ *réussit.*

Dans le traitement des requêtes sous forme développée, un même patron de triplet est considéré de plusieurs façons différentes, ce qui pose des problèmes d'interprétation des MFIS. Comme nous souhaitons que la présence d'une MFIS dans une requête informe sur l'échec de la requête, nous choisissons donc de traiter les requêtes sous forme factorisée. Comme nous pouvons appliquer les règles de distributivité des opérateurs, cela ne limite pas les requêtes supportées par notre approche.

4.3.1.4 Décomposition complète

En prenant en compte tous les opérateurs, les sous-requêtes d'une requête Q donnée sous forme factorisée disjonctive sont définies récursivement (Parkin et al., 2022).

1. La requête vide $\emptyset$ est une sous-requête de toutes les requêtes.
2. La seule sous-requête non vide d'un patron de triplet t est t.
3. Si $Q = Q_1 \ \wedge \ Q_2$, les sous-requêtes de Q sont $Q'_1 \ \wedge \ Q'_2$ où $Q'_1 \subseteq Q_1$ et $Q'_2 \subseteq Q_2$.

4. Si $Q = Q_1$ *FILTER*(R), les sous-requêtes de Q sont les requêtes Q'_1 et Q'_1 *FILTER*(R) où $Q'_1 \subseteq Q_1$ à condition que la requête obtenue soit une requête SPARQL valide (c'est-à-dire que les variables présentes dans les filtres sont aussi présentes dans la partie conjonctive de la requête).
5. Si $Q = Q_1$ *UNION* Q_2, les sous-requêtes non vides de Q sont les requêtes Q'_1, Q'_2 et Q'_1 *UNION* Q'_2 où $Q'_1 \subseteq Q_1$, $Q'_2 \subseteq Q_2$, $Q'_1 \neq \emptyset$ et $Q'_2 \neq \emptyset$.
6. Si $Q = Q_1$ *OPT* Q_2, les sous-requêtes non vides de Q sont les requêtes Q'_1, Q'_2 et Q'_1 *OPT* Q'_2 où $Q'_1 \subseteq Q_1$, $Q'_2 \subseteq Q_2$, $Q'_1 \neq \emptyset$ et $Q'_2 \neq \emptyset$.

4.3.2 Identification des MFIS et XSS

L'effet de l'opérateur FILTER est de sélectionner uniquement les réponses respectant certaines conditions. Comme retirer une condition de filtre à une requête ne peut pas retirer de variables (sinon la condition que toutes les variables des conditions de filtre doivent apparaître dans les patrons de triplet ne serait pas respectée), la propriété basée sur les variables détaillées dans le chapitre 3 nous permet d'affirmer que retirer une condition de filtre d'une requête ne peut pas diminuer le nombre de réponses. Ainsi, identifier les MFIS et XSS d'une requête contenant une condition de filtre nécessite au plus le même nombre d'exécutions de requêtes que la même requête sans la condition de filtre.

Quand nous exécutons les requêtes pour chercher les MFIS et XSS, nous pouvons encore appliquer les propriétés du chapitre 3 et de la partie 4.2 pour les requêtes conjonctives du treillis. En présence d'un opérateur UNION, nous pouvons également utiliser la relation $[[A\ UNION\ B]] = [[A]] + [[B]]$ afin de déterminer les MFIS et XSS d'une requête $A \cup B$ en fonction de celles de A et B :

Propriété 4.3 (MFIS avec UNION)**.** *Les MFIS d'une requête A sont des MFIS de la requête $A \cup B$.*

Démonstration. Considérons C, une MFIS de A, et montrons que C est une MFIS de $A \cup B$. D'après la définition d'une MFIS, nous savons que C échoue. Considérons C' une super-requête de C. C' est de la forme $A' \cup B'$, où A' est une super-requête de C et une sous-requête de A et B' est une sous-requête de B (potentiellement vide).
Comme C est une MFIS de A et que nous avons $C \subseteq A' \subseteq A$, $\text{FAIL}_K(A')$ est vrai. Comme $[[C']] = [[A']] + [[B']]$ et $[[A']] > K$ alors $[[C']] > K$ c'est-à-dire $\text{FAIL}_K(C')$. Donc toutes les super-requêtes de C échouent et C est une FIS de $A \cup B$.
Supposons que C ne soit pas une MFIS de $A \cup B$, c'est-à-dire qu'il existe une FIS C'' telle que $C'' \subset C \subset A$. Cela contredit l'hypothèse que C est une MFIS de A. Ainsi, C est une MFIS de $A \cup B$. □

Propriété 4.4 (XSS avec UNION)**.** *Toutes les sous-requêtes de $A \cup B$ qui réussissent ont une super-requête de la forme $A^* \cup B^*$ où A^* est une XSS de A et B^* est une XSS de B.*

Démonstration. Considérons une requête qui réussit $A' \cup B' \subseteq A \cup B$, où $A' \subseteq A$ et $B' \subseteq B$. Comme $|[[P_1 \cup P_2]]| = |[[P_1]]| + |[[P_2]]|$, la réussite de $A' \cup B'$ implique la réussite de A' et celle de B'. D'après la définition d'une XSS, cela signifie que soit A' est une XSS de A, ou c'est une sous-requête d'une XSS de A et idem pour B'. □

Si les MFIS et XSS de A et B sont connues, les requêtes supplémentaires à étudier sont donc limitées aux sous-requêtes d'une union entre une XSS de A et une XSS de B. Contrairement à l'opérateur UNION, pour l'opérateur OPTIONAL, il n'y a pas de relation directe entre les nombres de réponses de A, B et $A\ OPT\ B$. Il est cependant possible de déduire l'échec de $A\ OPT\ B$ à partir de l'échec de A.

Propriété 4.5 (nombre de réponses avec OPTIONAL)**.** $|[[A\ OPT\ B]]| \geq |[[A]]|$

Démonstration. Nous allons montrer que deux réponses distinctes de A sont associées à deux réponses distinctes de $A\ OPT\ B$.
Considérons $\mu_1 \neq \mu_2 \in [[A]]$. Si $\exists \mu_3 \in [[B]]$ tel que μ_3 et μ_1 sont compatibles, alors $\mu_1 \cup \mu_3 \in [[A]] \bowtie [[B]]$ donc $\mu_1 \cup \mu_3 \in [[A\ OPT\ B]]$. Sinon, $\mu_1 \in [[A]] \backslash [[B]]$ donc $\mu_1 \in [[A\ OPT\ B]]$. Notons m_1 le mapping qui appartient à $[[A\ OPT\ B]]$. De la même façon, soit $\exists \mu_4 \in [[B]]$ tel que $\mu_2 \cup \mu_4 \in [[A\ OPT\ B]]$ ou $\mu_2 \in [[A\ OPT\ B]]$. Notons m_2 ce mapping.
Comme $\mu_1 \neq \mu_2, \exists v \in \text{var}(A), \mu_1(v) \neq \mu_2(v)$. D'après la définition de m_1 et m_2, $m_1(v) = \mu_1(v)$ et $m_2(v) = \mu_2(v)$. Nous avons donc $v \in \text{var}(A\ OPT\ B)$ et $m_1(v) \neq m_2(v)$. Il y a donc deux réponses distinctes $m_1 \neq m_2 \in [[A\ OPT\ B]]$. □

Nous avons montré que si A échoue, alors $A\ OPT\ B$ échoue également. Nous pouvons en déduire deux propriétés basées sur les MFIS et XSS de A.

Propriété 4.6 (MFIS avec OPTIONAL)**.** *Les MFIS de A sont des MFIS de $A\ OPT\ B$.*

Démonstration. Soit C, une MFIS de A, et montrons que C est une MFIS de $A\ OPT\ B$.
D'après la définition de MFIS, nous savons que C échoue. Considérons C' une super-requête de C. C' est de la forme A' ou $A'\ OPT\ B'$ avec $C \subseteq A' \subseteq A$ et $B' \subseteq B$.
Comme C est une MFIS de A et que $C \subseteq A' \subseteq A$, A' échoue. Puisque $|[[C']]| \geq |[[A']]|$ d'après la propriété 4.5 et $|[[A']]| > K$ alors $|[[C']]| > K$ c'est-à-dire que C' échoue. Donc C est une FIS de $A\ OPT\ B$.
Supposons que C n'est pas une MFIS de $A\ OPT\ B$, c'est-à-dire qu'il existe D une FIS de $A\ OPT\ B$, telle que $D \subset C \subseteq A$. Donc D est aussi une FIS de A, ce qui contredit l'hypothèse que C est une MFIS de A. Ainsi C est une MFIS de $A\ OPT\ B$. □

```
SELECT * WHERE {
    { ?d traite ?p .              # t1
      ?d experience ?e .          # t2
      ?d collegue ?n .            # t3
      ?n soigne ?pt .             # t4
      ?n type InfirmierUrgence } # t5
      OPTIONAL
    { ?d diplome ?u .             # t6
      ?d specialite ?s } }        # t7
```

FIGURE 4.4 – $Q_4 = t_1t_2t_3t_4t_5\ OPT\ t_6t_7$

Propriété 4.7 (XSS avec OPTIONAL)**.** *Une sous-requête de $A\ OPT\ B$ qui réussit a une super-requête de la forme $A^*\ OPT\ B$ où A^* est une XSS de A.*

Démonstration. Considérons une sous-requête de $A\ OPT\ B$ qui réussit. Elle peut être de la forme A' ou $A'\ OPT\ B'$, avec $A' \subseteq A$ et $B' \subseteq B$. D'après la propriété 4.5, si $A\ OPT\ B$ réussit, alors A' réussit. D'après la définition de XSS, A' est soit une XSS de A ou la sous-requête d'une XSS de A. □

En utilisant ces propriétés, si les MFIS et XSS de A sont connues, plusieurs requêtes du treillis des sous-requêtes de $A\ OPT\ B$ n'ont pas besoin d'être exécutées.

Exemple 4.11 (MFIS et XSS avec OPTIONAL)**.** *Considérons la requête Q_4 donnée en figure 4.4. Nous savons déjà que les MFIS de $Q = t_1t_2t_3t_4t_5$ sont t_1t_3, t_1t_5 et t_4. Ce sont donc aussi des MFIS de Q_4. Les XSS de Q sont t_1t_2 et $t_2t_3t_5$. Ainsi, pour trouver toutes les MFIS et XSS de Q_4 il ne reste qu'à considérer les sous-requêtes de $t_1t_2\ OPT\ t_6t_7$ et $t_2t_3t_5\ OPT\ t_6t_7$.*

En regroupant les propriétés de cette section, nous proposons un algorithme pour identifier les MFIS et XSS d'une requête qui contient l'un des opérateurs FILTER, UNION, OPTIONAL. Pour cela, nous identifions la *partie conjonctive* d'une requête. Avec un opérateur OPTIONAL, la partie conjonctive est composée des patrons de triplets en dehors de la clause optionnelle. Avec un opérateur UNION, chaque partie de l'union peut être considérée comme une partie conjonctive. Pour la suite, cette appellation désignera la partie avec le plus grand nombre de patrons de triplets. Ainsi, dans les exemples précédents, la partie conjonctive de Q_3 est $t_1t_2t_4$, et la partie conjonctive de Q_4 est $t_1t_2t_3t_4t_5$. D'après les propriétés précédentes, les MFIS et XSS de la partie conjonctive d'une requête peuvent être utilisées pour la recherche des MFIS et XSS de la requête. La section suivante détaille comment ces propriétés sont intégrées dans notre algorithme.

4.3.3 Algorithme OPERATOR

L'algorithme OPERATOR, adapté à des requêtes contenant des opérateurs FILTER, UNION ou OPTIONAL est donné dans l'algorithme 4.2. Reprenons l'exemple 4.11 pour illustrer son fonctionnement. La première étape est d'identifier la partie conjonctive de notre requête, ici

Algorithme 4.2: Énumération des MFIS et XSS d'une requête Q

```
1  OPERATOR(Q, K)
       entrées: Une requête Q
                Un seuil K
       sorties : MFIS et XSS de Q
2      mfis ← ∅, xss ← ∅, fis ← ∅, list ← ∅, queryStatus ← ∅ ;
3      A ← getConjunctive(Q) ;// partie conjonctive de la requête
4      mfis, XSS ← VAR(A, K) ;// calcul des mfis et xss de A
5      foreach x :XSS do
6          list ← list ∪ replace(Q, A, x) ;// ajouter la sous-requête de Q où la partie
             conjonctive est remplacée par x
7      while list ≠ ∅ do
8          Q′ ← première requête de list en parcours en largeur
9          list ← list − {Q′} ;
10         parents_fis ← true ;
11         superqueries ← superQueries(Q) ;
12         foreach Q″ ∈ superqueries do
13             parents_fis ← parents_fis ∧ ((Q″) ∈ fis) ;
14         if parents_fis then
15             if Q′ ∉ queryStatus then
16                 queryStatus [Q′] ← FAIL_K(Q′) ;
17             if queryStatus [Q′] then // si Q′ échoue
18                 fis ← fis ∪ {Q′} ;
19                 mfis ← mfis − superqueries ;
20                 mfis ← mfis ∪ {Q′} ;
21                 foreach t ∈ triplePatterns(Q′) do
22                     if (Q′ − t) ∉list then
23                         list ← list ∪(Q′ − t) ;
24                         if var(Q′ − t) = var(Q′) then
25                             queryStatus [Q′ − t] ← true ;
26                         else if cardinalite_max(t) = 1 ∧ s(t) ∈ var(Q′ − t) then
27                             queryStatus [Q′ − t] ← true ;
28             else // Q′ réussit et est donc une XSS
29                 xss ← xss ∪ {Q′} ;
30     return mfis, xss
```

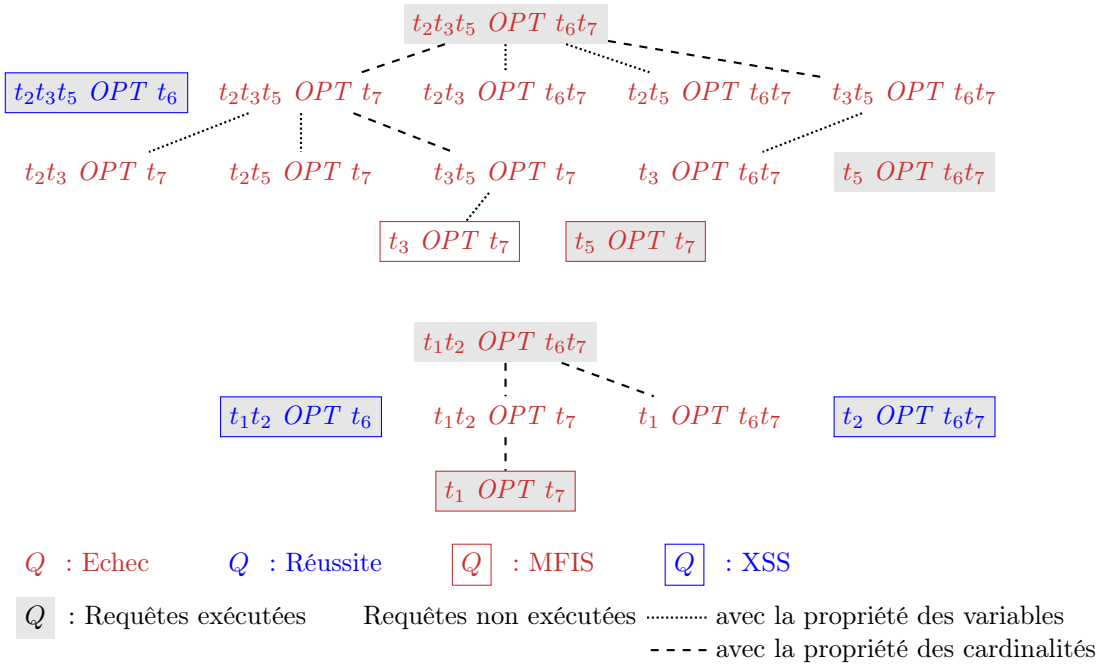


FIGURE 4.5 – Treillis des sous-requêtes de Q_4

$t_1t_2t_3t_4t_5$ (ligne 3). L'algorithme VAR est ensuite exécuté sur cette partie conjonctive, et les MFIS de la partie conjonctive sont ajoutées à la liste des MFIS (t_1t_3, t_1t_5 et t_4). Les XSS de la partie conjonctive (t_1t_2 et $t_2t_3t_5$) sont utilisées pour définir les requêtes suivantes à exécuter. Les sous-requêtes de la requête initiale obtenues en remplaçant la partie conjonctive par chaque XSS ($t_1t_2\ OPT\ t_6t_7$ et $t_2t_3t_5\ OPT\ t_6t_7$) sont donc ajoutées à la liste (ligne 6). L'algorithme se poursuit ensuite comme l'algorithme FULL. La figure 4.5 montre les requêtes exécutées à partir de cette étape. Finalement, cette requête a trois XSS : $t2t3t5\ OPT\ t6$, $t1t2\ OPT\ t6$, $t2\ OPT\ t6t7$, et six MFIS : t_1t_3, t_1t_5, t_4, $t_3\ OPT\ t_7$, $t_5\ OPT\ t_7$ et $t_1\ OPT\ t_7$. Pour cet exemple, l'utilisation d'un algorithme naïf nécessiterait 124 exécutions de requêtes. Avec l'algorithme OPERATOR, il ne reste que treize requêtes à exécuter : cinq pour trouver les MFIS et XSS de la partie conjonctive, puis huit pour en déduire toutes les MFIS et XSS de la requête avec les opérateurs.

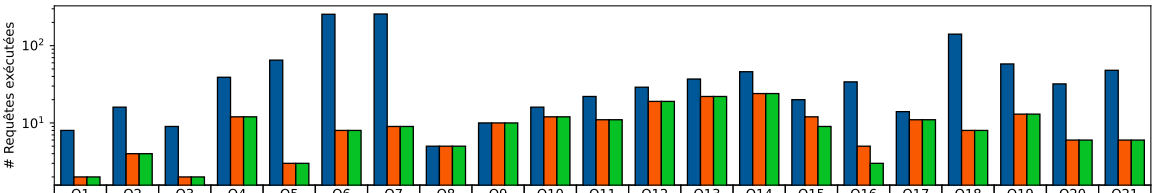
4.4 Expérimentation

4.4.1 Impact de l'utilisation des cardinalités

Dans cette section, nous évaluons les algorithmes utilisant les cardinalités par rapport à l'algorithme VAR du chapitre 3. Nous reprenons les paramètres expérimentaux et les requêtes présentées dans le chapitre 3 sur les bases de connaissance WatDiv et DBpedia. Les méthodes d'optimisation détaillées dans le chapitre 3 sont également utilisées (décomposition des pro-

BC	globale	classe	CS
WatDiv	86	-	13110
DBpedia	53696	494586	1445512

TABLE 4.3 – Nombre de lignes renseignées pour chaque BC et type de cardinalités



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21
var	8	16	9	39	65	255	257	5	10	16	22	29	37	46	20	34	14	141	58	32	48
global	2	4	2	12	3	8	9	5	10	12	11	19	22	24	12	5	11	8	13	6	6
cs	2	4	2	12	3	8	9	5	10	12	11	19	22	24	9	3	11	8	13	6	6

FIGURE 4.6 – # Requêtes exécutées WatDiv 11M triplets

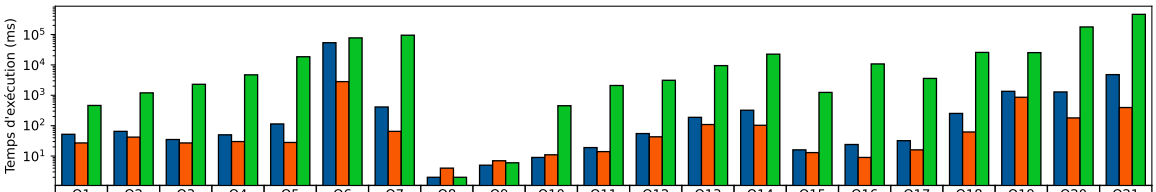
duits cartésiens et méthode d'exécution COUNT+LIMIT). Les cardinalités sont pré-calculées en utilisant les fichiers *.nt* des bases de connaissance. La table 4.3 fournit la taille de chaque fichier de cardinalités. Les requêtes utilisées sont fournies dans l'annexe A.

4.4.1.1 Utilisation des différents types de cardinalités

Pour les données synthétiques, nous comparons l'algorithme FULL en utilisant les cardinalités globales et les cardinalités de characteristic set, avec l'algorithme VAR. Puisque WatDiv ne fournit pas de schéma, nous ne pouvons pas utiliser la méthode basée sur les cardinalités de classe ici. La figure 4.6 donne le nombre de requêtes exécutées par chaque algorithme et la figure 4.7 fournit les temps d'exécution.

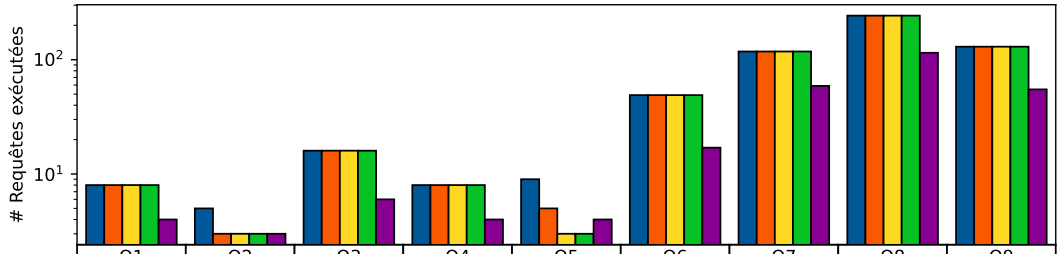
L'utilisation de la propriété basée sur les cardinalités diminue le nombre de requêtes exécutées d'environ 60%. Le temps d'exécution, lui, diminue de 39%. Dans la majorité des cas, l'utilisation des cardinalités plus précises de CS ne permet pas d'éviter davantage d'exécutions de requête. En revanche, le temps d'exécution est beaucoup plus élevé quand les cardinalités de CS sont utilisées. Dans les cas les plus extrêmes, il y a 2 ordres de grandeur entre l'exécution qui utilise les cardinalités globales et celle qui utilise les cardinalités de CS. Ce surcoût s'explique par la nature des cardinalités de CS qui nécessitent de calculer le maximum des cardinalités sur un nombre important de CS, tandis que les cardinalités globales sont obtenues une seule fois. Avec les CS, notre hypothèse que l'exécution des requêtes représente la majorité du temps d'exécution devient fausse. En raison de ce coût supplémentaire, les cardinalités de CS ne permettent pas d'améliorer les performances avec notre implémentation, malgré leur intérêt théorique.

En utilisant les données réelles de DBpedia, nous pouvons ajouter une exécution avec les cardinalités locales, puisque DBpedia possède un schéma RDFS. La figure 4.8 donne le



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21
var	52	65	35	50	114	5.4E4	412	2	5	9	19	55	188	322	16	24	32	253	1352	1288	4793
global	27	42	27	30	28	2823	65	4	7	11	14	43	109	103	13	9	16	62	861	180	395
cs	463	1202	2300	4711	1.9E4	7.7E4	9.5E4	2	6	453	2101	3120	9484	2.3E4	1244	1.1E4	3591	2.6E4	2.5E4	1.8E5	4.6E5

FIGURE 4.7 – Temps d'exécution WatDiv 11M triplets



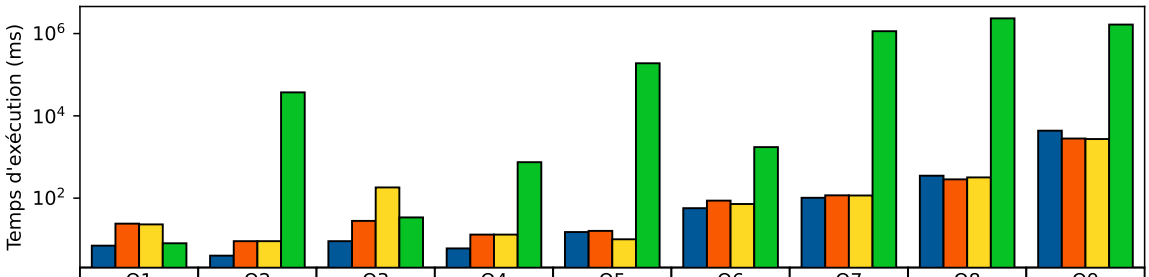
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
var	8	5	16	8	9	49	118	243	130
global	8	3	16	8	5	49	118	243	130
class	8	3	16	8	3	49	118	243	130
cs	8	3	16	8	3	49	118	243	130
global_anyCard	4	3	6	4	4	17	59	115	55

FIGURE 4.8 – # Requêtes exécutées DBpedia

nombre de requêtes exécutées et le temps d'exécution est présenté en figure 4.9. Cette fois, nous remarquons que l'utilisation des cardinalités, qu'elles soient globales, de classe ou de CS, ne permet pas d'éviter beaucoup d'exécutions de requêtes. C'est uniquement pour les requêtes Q2 et Q5 que nous constatons une diminution du nombre de requêtes exécutées. Ainsi, les temps d'exécution des algorithmes FULL et VAR sont très proches. Cela s'explique par les cardinalités des prédicats dans DBpedia. En effet, très peu de prédicats ont une cardinalité maximale de 1 (environ 0.67%), alors que 66% des prédicats ont une cardinalité maximale de 2. Plus précisément, des prédicats qui auraient théoriquement une cardinalité maximale de 1, tels que *BirthDate* pour la date de naissance, ont en réalité une cardinalité maximale de 2. Cela peut être dû à des erreurs dans les données ou des informations incertaines (Giacometti et al., 2019 ; Muñoz et Nickles, 2017). En travaillant avec un jeu de données nettoyé, les performances des déductions basées sur les cardinalités seraient améliorées.

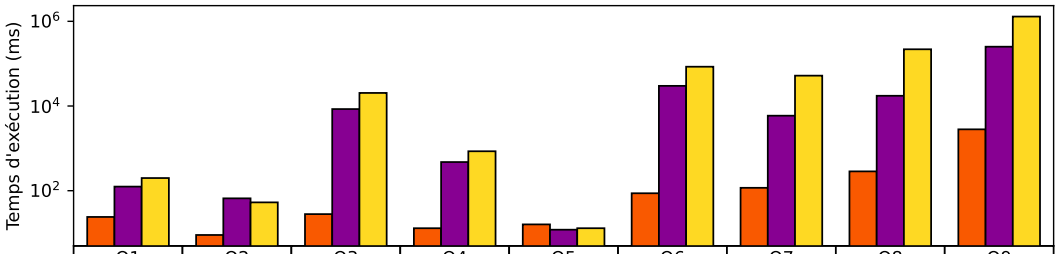
4.4.1.2 Utilisation des cardinalités supérieures à 1

Nous avons vu que DBpedia intègre très peu de prédicats avec une cardinalité maximale de 1. En conséquence, nous essayons d'appliquer plutôt le corollaire 4.2 qui s'applique pour n'importe quelle cardinalité maximale. Nous utilisons donc pour cette expérience la base de



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
var	7	4	9	6	15	57	102	351	4,360
global	24	9	28	13	16	87	117	286	2,813
class	23	9	182	13	10	72	116	319	2,732
cs	8	3.7E4	34	748	1.9E5	1,739	1.1E6	2.3E6	1.7E6

FIGURE 4.9 – Temps d'exécution DBpedia



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
card1-countlimit	24	9	28	13	16	87	117	286	2813
anyCard-count	125	66	8470	475	12	3.0E4	5908	1.7E4	2.5E5
card1-count	199	53	2.0E4	854	13	8.5E4	5.2E4	2.2E5	1.3E6

FIGURE 4.10 – Temps d'exécution DBpedia

connaissances DBpedia et les cardinalités globales. La figure 4.8 montre que l'utilisation de ce corollaire (étiqueté comme *global_anyCard*) permet effectivement de diminuer le nombre de requêtes exécutées de 45% en moyenne par rapport à l'utilisation du corollaire 4.1 qui s'applique uniquement pour des cardinalités maximales égales à 1 (étiqueté comme *global*).

Le problème de l'utilisation de ce corollaire est qu'il nécessite de connaître le nombre exact de réponses des requêtes. Il n'est plus suffisant de savoir si la requête réussit ou échoue, c'est-à-dire si son nombre de réponses dépasse le seuil fixé. Ainsi, il n'est plus possible d'appliquer l'optimisation présentée dans le chapitre 3 qui consiste à ajouter l'opérateur LIMIT à la requête pour diminuer le temps d'exécution. Pour pouvoir utiliser cette propriété, nous devons donc choisir la méthode d'exécution COUNT. La figure 4.10 donne le temps d'exécution associé (étiquette *anyCard-count*), qui est à comparer au temps d'exécution si le corollaire 4.1 est appliqué. Pour l'utilisation de cette dernière, nous donnons le temps d'exécution pour les deux méthodes d'exécution COUNT (étiquette *card1-count*) et COUNT+LIMIT (étiquette *card1-countlimit*). Cela nous permet de voir que l'utilisation du corollaire 4.2 représente une amélioration significative (50%) par rapport au corollaire 4.1 pour la même méthode d'exécution.

Cependant, cette amélioration n'est pas suffisante pour compenser la perte de performance associée au choix de la méthode d'exécution. L'utilisation du corollaire 4.1 avec la méthode d'exécution COUNT+LIMIT est en moyenne 100 fois plus rapide.

4.4.2 Intégration des opérateurs

Comme pour les expériences précédentes, les algorithmes sont implémentés en Java 1.8 et sont exécutés sur un serveur Ubuntu 16.04 LTS avec un processeur Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz et 32 GB RAM. Les temps d'exécution présentés dans les figures sont une moyenne de cinq exécutions consécutives, après une première exécution non comptabilisée. Nous utilisons la méthode d'exécution COUNT+LIMIT.

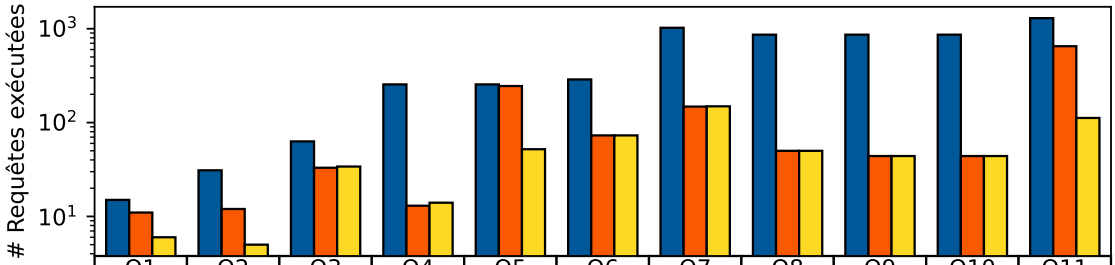
Pour les expériences concernant les opérateurs, nous avons uniquement utilisé la base de connaissances DBpedia, car les requêtes associées à la base de connaissances WatDiv sont uniquement sous forme conjonctive. Les journaux de requêtes du projet LSQ contiennent des requêtes avec les opérateurs FILTER, UNION et OPTIONAL qui nous intéressent. Les onze requêtes utilisées dans cette expérience contiennent entre quatre et douze patrons de triplets, et chacune contient au moins un de ces opérateurs. Nous avons reformulé les requêtes qui ne se trouvaient pas déjà sous forme factorisée afin de respecter la contrainte spécifiée dans la section 4.3. Nous comparons trois méthodes pour la recherche des MFIS et XSS.

— La méthode naïve où toutes les requêtes du treillis sont exécutées : Base.
— L'algorithme Var appliqué à la requête avec opérateurs, en intégrant la définition étendue des sous-requêtes, mais sans appliquer les propriétés identifiées dans la section 4.3.2.
— L'algorithme Operator, où l'on commence par rechercher les MFIS et XSS de la partie conjonctive de la requête.

Les figures 4.11 et 4.12 montrent respectivement le nombre de requêtes exécutées par chaque algorithme et le temps d'exécution. Nous pouvons nous attendre a avoir 2^n exécutions pour l'algorithme Base, où n est le nombre de patrons de triplet de la requête initiale. Cela n'est pas le cas lorsque FILTER est impliqué car certaines requêtes du treillis ne sont pas valides donc ne sont pas exécutées.

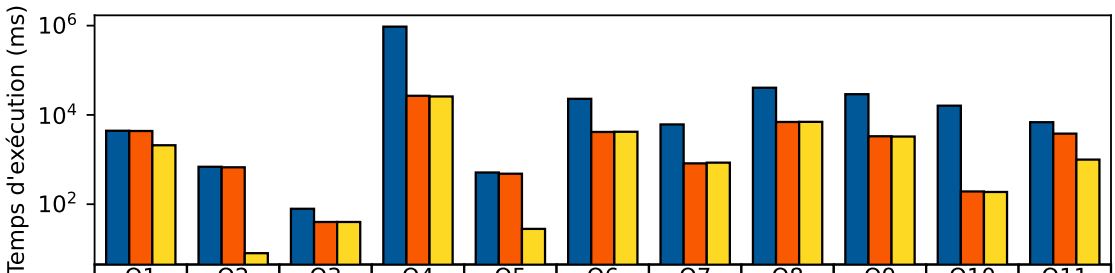
En utilisant l'algorithme Var, 66% de requêtes de moins sont exécutées et le temps d'exécution diminue de 58% par rapport à la méthode naïve. L'algorithme Operator exécute 82% de requêtes de moins que la méthode naïve, et calcule les MFIS et XSS 83% plus rapidement que la méthode naïve et 29% plus rapidement que l'algorithme Var.

L'algorithme Operator est particulièrement efficace lorsque la requête comporte un opérateur avec un grand nombre de patrons de triplet par rapport à la taille de la requête. Pour Q5, il y a 4 patrons de triplet dans l'opérateur OPTIONAL, pour 8 patrons au total et Operator est plus de dix fois plus rapide que Var, qui a des performances proches de Base. Les requêtes Q6, Q8, Q9 et Q10 utilisent uniquement l'opérateur FILTER, donc l'algorithme



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
base	15	31	63	255	255	288	1023	864	864	864	1296
var	11	12	33	13	245	73	148	50	44	44	651
operator	6	5	34	14	52	73	149	50	44	44	112

FIGURE 4.11 – # Requêtes exécutées DBpedia



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11
base	4420	689	79	9.5E5	513	2.3E4	6122	4.1E4	2.9E4	1.6E4	6869
var	4364	671	40	2.7E4	482	4152	822	6948	3325	193	3811
operator	2092	8	40	2.6E4	28	4192	853	6993	3279	188	998

FIGURE 4.12 – Temps d'exécution DBpedia

OPERATOR exécute les mêmes requêtes que VAR puisque la partie conjonctive de la requête est la requête elle-même. Nous obtenons le même nombre de requêtes exécutées, et des temps d'exécution proches.

4.5 Conclusion

Dans ce chapitre, nous avons considéré les spécificités des langages RDF et SPARQL. Nous avons commencé par utiliser les cardinalités des prédicats afin de diminuer le nombre de requêtes à exécuter dans notre recherche des MFIS et XSS. Nous avons vu que cette propriété supplémentaire peut permettre de diviser par deux le nombre de requêtes exécutées pour chaque patron de triplet auquel elle s'applique. Cependant, une étude expérimentale a montré que si l'utilisation des cardinalités se montre pertinente avec des données synthétiques (diminution de 39% du temps d'exécution), ce n'est pas le cas pour les données réelles de DBpedia. En effet, la présence de données erronées peut faire disparaître la majorité des prédicats de cardinalité 1, ce qui nous empêche d'appliquer la propriété. Ainsi, même si l'utilisation des cardinalités supérieures à 1 semble permettre une amélioration, celle-ci ne suffit pas pour

compenser le coût supplémentaire dû à l'utilisation d'une méthode d'exécution des requêtes moins efficace.

Les éléments du schéma que nous avons utilisés pour déduire l'échec de requêtes restent assez simples : calcul des cardinalités, utilisation des propriétés *rdf:type* et *rdfs:domain*. Nous pouvons envisager d'exploiter des schémas plus riches. Par exemple le langage OWL propose la propriété *owl:maxCardinality* dans la définition d'un prédicat. Une difficulté est que les données peuvent ne pas correspondre au schéma dans le cas où la BC contient des erreurs, ce qui implique que les déductions basées sur le schéma ne correspondent pas à la réalité des données.

Dans un second temps, nous avons étendu notre approche aux requêtes non conjonctives, c'est-à-dire aux requêtes contenant les opérateurs FILTER, OPTIONAL ou UNION de SPARQL. Cela a impliqué de redéfinir la notion de sous-requête. Pour une interprétation cohérente des MFIS et XSS, nous imposons également que les requêtes soient traitées sous forme factorisée. Une évaluation expérimentale a montré que la stratégie de chercher d'abord les MFIS de la partie conjonctive de la requête est plus performante. Cet algorithme calcule les MFIS et XSS 83% plus rapidement que la méthode naïve et 29% plus rapidement que l'algorithme Var.

Il reste à considérer l'effet des modificateurs de résultats sur les causes d'échec. Par exemple, la sélection d'un nombre restreint de variables, à la place d'un SELECT *, l'utilisation de DISTINCT ou encore de l'opérateur GROUP BY diminuent tous les trois le nombre de réponses.

Chapitre 5

Méthode unifiée pour les réponses insatisfaisantes

Sommaire

Résumé

Nous avons adapté les notions de causes d'échec introduites pour le problème des réponses vides au problème des réponses pléthoriques. Dans ce chapitre, nous élargissons

le champ de notre approche aux réponses insatisfaisantes de tous types. Nous détaillons les caractéristiques communes et les différences entre ces problèmes, puis nous montrons comment en calculer les MFIS et XSS. Dans un premier temps, nous nous intéressons aux problèmes élémentaires, puis nous étudions les cas où plusieurs types d'échec sont combinés. Dans ce second cas, nous proposons deux stratégies pour identifier les MFIS et XSS. Une comparaison expérimentale avec des méthodes existantes spécifiques à un problème montre que notre généralisation n'introduit pas de surcoût notable lors de l'exécution.

5.1 Introduction

Dans les deux chapitres précédents, nous avons vu comment les MFIS et XSS peuvent être calculées pour aider à expliquer les raisons des réponses pléthoriques obtenues par des utilisateurs. Nous étendons dans ce chapitre la méthode aux autres problèmes de réponses insatisfaisantes présentés dans le chapitre 2.

Pour ces problèmes, les approches existantes (Fokou et al., 2016 ; Wang et al., 2019) s'intéressent chacune à un problème particulier. Aucune approche ne permet de gérer des problèmes combinés, c'est-à-dire des requêtes pour lesquelles les critères d'insatisfaction sont multiples et de plusieurs types. Ainsi, nous voulons montrer que les MFIS et XSS peuvent être utilisées pour tous types de réponses non satisfaisantes comme causes d'échec et requêtes alternatives, et que ces notions sont également adaptées pour traiter des problèmes impliquant de multiples conditions d'échec.

Un utilisateur peut être insatisfait par plusieurs types de problèmes. Nous avons vu un cas dans le chapitre 2, où la résolution d'un problème de réponses vides fait apparaître un problème de réponses pléthoriques. Il s'agit ici d'une disjonction de conditions d'échec : $\text{FAIL}_{\emptyset}(Q) \vee \text{FAIL}_{>K}(Q)$. Il existe d'autres situations qui impliquent la combinaison de problèmes.

— Lorsqu'un utilisateur s'attend à trouver une réponse parmi plusieurs qui ne figurent pas dans les résultats. Il s'agit ici d'une conjonction de conditions d'échec : $\text{FAIL}_{\not\subseteq \mu_{w_1}}(Q) \wedge \text{FAIL}_{\not\subseteq \mu_{w_2}}(Q)$.

— Lorsqu'une application nécessite un nombre de réponses exact, la requête échoue si elle produit strictement plus ou strictement moins de réponses que le nombre attendu (Vartak et al., 2010). Il s'agit d'une disjonction de conditions d'échec : $\text{FAIL}_{<K}(Q) \vee \text{FAIL}_{>K}(Q)$.

Nous identifions donc un besoin d'approches pour traiter ces problèmes combinés. À notre connaissance, ce problème n'a jamais été abordé. C'est l'objet de ce chapitre.

Nous commençons dans la section 5.2 par identifier des propriétés liées aux conditions d'échec élémentaires et établissons les points communs et différences entre celles-ci. Dans la section 5.3, nous proposons ensuite deux approches pour traiter les problèmes combinés. Ces approches sont confrontées expérimentalement dans la section 5.4 où nous comparons

Problème	Cardinalité ou contenu	Condition d'échec
réponses vides	cardinalité	$\text{FAIL}_{\emptyset}(Q) : [[Q]]_D = \emptyset$
réponses pléthoriques	cardinalité	$\text{FAIL}_{>K}(Q) : \lvert[[Q]]_D\rvert > K$
réponses insuffisantes	cardinalité	$\text{FAIL}_{<K}(Q) : \lvert[[Q]]_D\rvert < K$
réponses manquantes	contenu	$\text{FAIL}_{\not\subseteq \mu_w}(Q) : \forall \mu \in [[Q]]_D \mu_w \not\subseteq \mu$
réponses présentes	contenu	$\text{FAIL}_{\subseteq \mu_w}(Q) : \exists \mu \in [[Q]]_D \mu_w \subseteq \mu$

FIGURE 5.1 – Les cinq conditions d'échec élémentaires

également notre solution générale à des solutions spécifiques pour les problèmes de réponses vides, de réponses manquantes et de réponses pléthoriques.

5.2 Liens entre les problèmes de réponses insatisfaisantes

Les cinq conditions d'échec élémentaires ont été définies dans le chapitre 2. Nous rappelons leurs caractéristiques dans la table 5.1. Ces conditions d'échec sont associées à des fonctions booléennes qui calculent la réussite ou l'échec d'une requête Q en fonction du résultat de son évaluation $[[Q]]$. Les notions des MFIS et XSS introduites dans le chapitre 3 ne sont pas spécifiques à une condition d'échec particulière. Elles correspondent aux causes d'échec et requêtes alternatives. Nous en donnons les définitions ci-dessous pour une condition d'échec quelconque.

Définition 5.1 (FIS, MFIS, XSS)**.** *Pour une requête Q et une condition d'échec FAIL :*

$$\mathit{fis}(Q, \mathit{FAIL}) = \{Q^* \mid Q^* \subseteq Q \wedge \mathit{FAIL}(Q^*) \wedge \forall Q' \subseteq Q, Q^* \subset Q' \Rightarrow \mathit{FAIL}(Q')\}$$

$$\mathit{mfis}(Q, \mathit{FAIL}) = \{Q^* \mid Q^* \in \mathit{fis}(Q, \mathit{FAIL}) \wedge \nexists Q' \subset Q^*,\ Q' \in \mathit{fis}(Q, \mathit{FAIL})\}$$

$$\mathit{xss}(Q, \mathit{FAIL}) = \{Q^* \mid Q^* \subseteq Q \wedge \neg \mathit{FAIL}(Q^*) \wedge \forall Q' \mid Q^* \subset Q',\ Q' \in \mathit{fis}(Q, \mathit{FAIL})\}$$

Dans cette section, nous détaillons les propriétés de ces conditions d'échec pour mettre en évidence leurs similarités et leurs différences, avant de proposer une méthode commune de traitement basée sur les MFIS et XSS.

5.2.1 Propriétés des conditions d'échec

Dans le chapitre 2, nous avons séparé les conditions d'échec en problèmes de contenu et de cardinalité. Nous montrons ici que les deux types de problèmes sont liés, et nous introduisons une autre distinction entre problèmes monotones et non monotones.

5.2.1.1 Reformulation des problèmes de contenu en problèmes de cardinalité

Le problème des réponses manquantes peut être transformé en problème de réponses vides. Cela permettra d'exploiter les propriétés déjà identifiées pour le problème des réponses vides

```
SELECT * WHERE {
  ?p age ?a .      #t1
  ?p pole ?s .     #t2
  ?n soigne ?p .   #t3
  ?n service ?s } #t4
```

(a) Requête Q

?p	?a	?s	?n
p_2	47	Urgences	n_2
p_3	33	soinsIntensifs	n_1

(b) Résultats de Q

```
SELECT * WHERE {
  p1 age ?a .      #t1
  p1 pole ?s .     #t2
  ?n soigne p1 .   #t3
  ?n service ?s }  #t4
```

(c) Requête Q'

FIGURE 5.2 – Transformation d'un problème de réponses manquantes en réponses vides

lors de la recherche des MFIS et XSS. Considérons une requête Q et une question de réponses manquante μ_w qui associe des variables v à $\mu_w(v)$. Nous construisons une nouvelle requête Q' à partir de la requête Q en remplaçant chaque variable v par leur image par μ_w. Cette technique a déjà été appliquée pour le problème des réponses manquantes dans les bases de connaissances (Wang et al., 2019). La propriété suivante transforme un échec de Q en un échec de Q'.

Propriété 5.1 (réponse manquante et réponses vides)**.** *Soit une requête Q, une question de réponse manquante μ_w, et une requête Q' construite en remplaçant chacune des variables $v \in dom(\mu_w) \cap var(Q)$ par $\mu_w(v)$. $FAIL_{\not\subseteq \mu_w}(Q) = FAIL_{\emptyset}(Q')$.*

Démonstration. Si $\text{FAIL}_{\not\subseteq \mu_w}(Q)$ est vrai, supposons $\exists \mu' \in [[Q']]$. D'après la définition de Q', $\text{dom}(\mu') \cap \text{dom}(\mu_w) = \emptyset$, donc μ' et μ_w sont compatibles. Donc $\mu = \mu' \cup \mu_w$ est un mapping, $\mu \in [[Q]]$ et μ et μ_w sont compatibles, ce qui contredit $\nexists \mu \in [[Q]], \mu$ et μ_w sont compatibles. Donc $[[Q']] = \emptyset$ et $\text{FAIL}_{\emptyset}(Q')$ est vrai.
Si $\text{FAIL}_{\emptyset}(Q')$ est vrai, supposons $\exists \mu \in [[Q]]$ tel que μ_w et μ sont compatibles. D'après la définition de Q', $\mu_{|\text{var}(Q')} \in [[Q']]$ ce qui contredit $[[Q']] = \emptyset$. Donc $\nexists \mu \in [[Q]], \mu$ et μ_w sont compatibles et $\text{FAIL}_{\not\subseteq \mu_w}(Q)$ est vrai. □

Exemple 5.1 (réponse manquante et réponses vides)**.** *Avec la base de connaissances de la figure 2.1, considérons la requête Q donnée en figure 5.2a avec ses résultats en figure 5.2b. Un utilisateur peut se demander pourquoi aucune réponse ne contient la valeur p_1 pour la variable p. Dans ce cas, $\mu_w : \{p \to p_1\}$. La requête modifiée implique de remplacer chaque occurrence de la variable p selon μ_w, c'est-à-dire par p_1. La requête correspondante Q' est donnée en figure 5.2c. Rechercher les causes d'échec pour Q et $FAIL_{\not\subseteq \mu_w}$ revient à rechercher les causes d'échec pour Q' et $FAIL_{\emptyset}$.*

Un problème de réponse présente peut également être transformé en problème de réponses pléthoriques avec un seuil K=0. Dans ce cas, nous cherchons à obtenir zéro réponse. Il faudra transformer la requête de la même façon.

Propriété 5.2 (réponse présente et réponses pléthoriques)**.** *Soit une requête Q, une question de réponse présente μ_w, et une requête Q' construite en remplaçant chaque $v \in dom(\mu_w) \cap var(Q)$ par $\mu_w(v)$. $FAIL_{\subseteq \mu_w}(Q) = FAIL_{>0}(Q')$.*

Démonstration. Avec les définitions d'échec, $\text{FAIL}_{\subseteq\mu_w}(Q) = \neg(\text{FAIL}_{\not\subseteq\mu_w}(Q))$. Puis, en appliquant la propriété 5.1 $\text{FAIL}_{\not\subseteq\mu_w}(Q) = \text{FAIL}_{\emptyset}(Q')$ donc $\text{FAIL}_{\subseteq\mu_w}(Q) = \neg(\text{FAIL}_{\emptyset}(Q')) = \neg(\text{FAIL}_{<1}(Q')) = \text{FAIL}_{>0}(Q')$. □

Ces deux propriétés témoignent de l'équivalence entre une requête avec une variable pour laquelle on recherche une valeur en particulier, et cette même requête où la variable est remplacée par sa valeur. L'utilisation de ces deux propriétés permet de traiter les problèmes de contenu en les transformant en problèmes de cardinalité. Dans la section 5.2.2, nous donnerons des propriétés de déduction d'échec pour les problèmes de cardinalité. En utilisant les transformations évoquées ici, ces propriétés s'appliqueront également aux problèmes de contenu.

5.2.1.2 Monotonie des conditions d'échec

Nos conditions d'échec sont des fonctions booléennes. Un cas particulier des fonctions booléennes, les fonctions booléennes monotones (MBF), ont été largement étudiées (Crama et Hammer, 2011). Une fonction booléenne $f(x_1, x_2, \ldots, x_n)$ est dite *positive* si pour chacune de ses variables x, $f_{x=0} \leq f_{x=1}$. Une fonction booléenne f est dite *négative* si pour chacune de ses variables x, $f_{x=0} \geq f_{x=1}$. Les fonctions booléennes positives ou négatives sont dites *monotones*.

Nous étendons ces termes aux conditions d'échec pour les requêtes. Nous appelons donc *positive* une condition d'échec dont la fonction d'échec est positive. Cela implique que si une requête échoue, toutes ses super-requêtes échouent, et par contraposée, si une requête réussit, toutes ses sous-requêtes réussissent. Nous appelons *négative* une condition d'échec dont la fonction d'échec est négative. Cela implique que si une requête réussit, toutes ses super-requêtes réussissent et si une requête échoue, toutes ses super-requêtes échouent. Nous appelons *monotones* les conditions d'échec qui sont positives ou négatives. La monotonie de trois conditions d'échec peut être établie grâce à la propriété suivante :

Propriété 5.3 (lien entre ensembles de réponses)**.** *Pour deux requêtes $Q' \subseteq Q$, si $\mu \in [[Q]]_D$ alors $\mu_{|var(Q')} \in [[Q']]_D$.*

Démonstration. Considérons deux requêtes Q et $Q' \subseteq Q$ où $[[Q]]_D \neq \emptyset$ et $\mu \in [[Q]]_D$. Comme $Q' \subseteq Q$, nous pouvons écrire $Q = Q' \wedge Q''$. Ainsi $[[Q]]_D = [[Q']]_D \bowtie [[Q'']]_D$ et nous pouvons écrire $\mu = \mu_1 \cup \mu_2$ où $\mu_1 \in [[Q']]_D$ et $\mu_2 \in [[Q'']]_D$. Nous avons $\text{dom}(\mu_1) = \text{var}(Q')$ et $\forall v \in \text{var}(Q')\ \mu(v) = \mu_1(v)$. Finalement, $\mu_{|\text{var}(Q')} \in [[Q']]_D$. □

Nous pouvons en déduire la monotonie des problèmes des réponses vides et des réponses manquantes. La monotonie du problème des réponses présentes s'ensuit comme négation du problème des réponses manquantes.

Propriété 5.4 (conditions positives)**.** *$FAIL_{\emptyset}$ et $FAIL_{\not\subseteq\mu_w}$ sont des conditions positives.*

Démonstration. Considérons d'abord $\text{FAIL}_{\emptyset}(Q) = ([[Q]] = \emptyset)$. Soit Q' une requête qui échoue et Q une requête telle que $Q' \subseteq Q$. Supposons que Q réussit, c'est-à-dire $\exists \mu \in [[Q]]$. D'après la propriété 5.3, $\mu_{|\text{var}(Q')} \in [[Q']]$ qui contredit l'hypothèse que Q' échoue. Donc Q échoue, et $\text{FAIL}_{\emptyset}$ est positive.

Considérons maintenant $\text{FAIL}_{\not\subseteq \mu_w}(Q) = \text{dom}(\mu_w) \subseteq \text{var}(Q) \wedge \forall \mu \in [[Q]] : \mu_w$ *et* μ ne sont pas compatibles. Soit Q' une requête qui échoue et Q une super-requête de Q'. Comme $\text{var}(Q') \subseteq \text{var}(Q)$ et $\text{dom}(\mu_w) \subseteq \text{var}(Q')$, $\text{dom}(\mu_w) \subseteq \text{var}(Q)$. Supposons par l'absurde que Q réussisse, c'est-à-dire $\exists \mu \in [[Q]], \mu_w$ et μ sont compatibles. Nous allons montrer que sous cette hypothèse, Q' réussit.
Considérons $\mu' = \mu_{|\text{var}(Q')}$. D'après la propriété 5.3, $\mu' \in [[Q']]$. Nous montrons maintenant que μ' et μ_w sont compatibles. Nous avons $\text{dom}(\mu') = \text{var}(Q')$ et $\text{dom}(\mu_w) \subseteq \text{var}(Q') \subseteq \text{dom}(\mu)$, donc $\text{dom}(\mu') \cap \text{dom}(\mu_w) = \text{dom}(\mu_w)$ et $\text{dom}(\mu) \cap \text{dom}(\mu_w) = \text{dom}(\mu_w)$. Considérons $x \in \text{dom}(\mu_w)$. Comme μ et μ_w sont compatibles, $\mu(x) = \mu_w(x)$. D'après la définition de μ', $\mu'(x) = \mu(x)$. Donc $\mu'(x) = \mu_w(x)$. Ainsi μ' et μ_w sont compatibles, donc Q' réussit, ce qui contredit notre hypothèse. Ainsi Q échoue, et $\text{FAIL}_{\not\subseteq \mu_w}$ est positive. □

Comme $\neg(\text{FAIL}_{\subseteq \mu_w}(Q)) = \text{FAIL}_{\not\subseteq \mu_w}(Q)$, nous en déduisons que le problème des réponses présentes a une condition d'échec négative. Nous montrons que le problème des réponses insuffisantes et le problème des réponses pléthoriques n'ont pas une condition d'échec monotone avec un contre-exemple.

Exemple 5.2 (conditions non monotones)**.** *Nous utilisons la base de connaissances D, la requête Q de la figure 2.1, les conditions d'échec $FAIL_1(Q) = FAIL_{>2}(Q)$, $FAIL_2(Q) = FAIL_{<2}(Q)$. Nous donnons les nombres de réponses de chaque requête dans le treillis de la figure 5.3. Considérons les sous-requêtes $Q_1 = t_5$, $Q_2 = t_2t_5$, $Q_3 = t_2t_4t_5$. Donc $Q_1 \subseteq Q_2 \subseteq Q_3$. Comme $FAIL_1(Q_1)$ est vrai, $FAIL_1(Q_2)$ est faux, et $FAIL_1(Q_3)$ est vrai, $FAIL_1$ n'est pas monotone. Considérons les sous-requêtes $Q_1 = t_2$, $Q_2 = t_2t_4$, $Q_3 = t_2t_3t_4$. Donc $Q_1 \subseteq Q_2 \subseteq Q_3$. Comme $FAIL_2(Q_1)$ est vrai, $FAIL_2(Q_2)$ est faux, et $FAIL_2(Q_3)$ est vrai, $FAIL_2$ n'est pas monotone.*

5.2.2 Propriétés de déduction

Nous avons identifié deux types de propriétés de déduction. Le premier type peut permettre de déterminer l'échec d'une requête basé sur la cardinalité. Ce type de propriété a déjà été utilisé pour le problème des réponses pléthoriques dans le chapitre précédent. Le deuxième type de propriétés de déduction utilise la monotonie du problème. Ce type de propriété a été utilisé pour traiter le problème des réponses vides.

Les premières propriétés sont une généralisation des propriétés 3.4 et 4.1. Elles peuvent être utilisées pour obtenir une borne supérieure et inférieure du nombre de réponses d'une

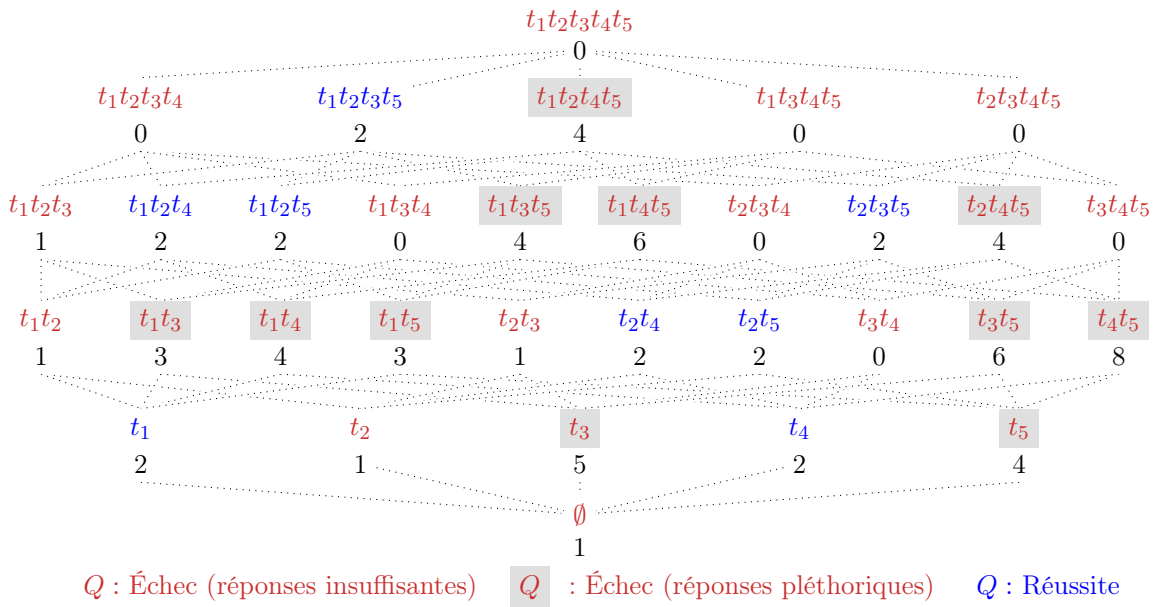


FIGURE 5.3 – Treillis des sous-requêtes de Q

requête en fonction du nombre de réponses de l'une de ses super-requêtes. Cela permettra de déduire l'échec d'une sous-requête à partir de l'échec d'une super-requête ou inversement.

Propriété 5.5 (variables)**.** *Pour deux requêtes Q et Q', si $var(Q') = var(Q)$ et $Q \subseteq Q'$ alors* $|[[Q']]| \leq |[[Q]]|$.

Démonstration. Nous allons montrer que $\forall \mu \in [[Q']]$, $\mu \in [[Q]]$. Soit $\mu' = \mu_{|\text{var}(Q)}$. Montrons que $\mu' \in [[Q]]$. Supposons par l'absurde que $\mu' \notin [[Q]]$. Soit $\text{dom}(\mu') \neq \text{var}(Q)$ où $\exists v \in \text{var}(Q), \mu'(v) \notin D$. Par définition de la restriction, $\text{dom}(\mu') = \text{var}(Q)$. Soit $v \in \text{var}(Q), \mu'(v) \notin D$. Par définition, $\mu'(v) = \mu(v)$. Donc $\mu(v) \notin D$, ce qui contredit $\mu \in [[Q \wedge t]]$. Donc $\mu' \in [[Q]]$, et comme $\text{var}(Q') = \text{var}(Q)$, alors $\mu' = \mu$. Donc $[[Q']] \subseteq [[Q]]$ et $|[[Q']]| \leq |[[Q]]|$. □

Propriété 5.6 (cardinalités)**.** *Soit une requête Q et un patron de triplet t avec un prédicat constant $p(t)$, un objet variable $o(t) \notin var(Q)$ et un sujet variable $s(t) \in var(Q)$. Nous avons :*

$$\text{cardinalite}_{\min}(p(t), s(t), Q \wedge t) \cdot |[[Q]]| \leq |[[Q \wedge t]]| \leq \text{cardinalite}_{\max}(p(t), s(t), Q \wedge t) \cdot |[[Q]]|$$

Démonstration. La partie droite de l'inégalité a déjà été démontrée dans le chapitre 4 (preuve de la propriété 4.1). Pour montrer $\text{cardinalite}_{\min}(p(t), s(t), Q \wedge t).|[[Q]]| \leq |[[Q \wedge t]]|$, nous réutilisons les fonctions $f_{Q,t}$ et $a_{Q,t}$ définies pour la preuve de la propriété 4.1.

$$f_{Q,t} : \begin{cases} [[Q \wedge t]] \rightarrow [[Q]] \\ \mu \mapsto \mu_{|\text{var}(Q)} \end{cases}$$

```
SELECT * WHERE {
  ?p age ?a .   #t1
  ?p pole ?s } #t2
```

(a) Requête Q

?p	?a	?s
p_1	25	soinsIntensifs
p_2	47	Urgences
p_3	33	soinsIntensifs

(b) Résultats de Q

?p	?a	?s	?m
p_1	25	soinsIntensifs	brasCassé
p_1	25	soinsIntensifs	grippe
p_2	47	Urgences	arrêtCardiaque
p_3	33	soinsIntensifs	fièvre

(c) Résultats de $Q \wedge t$

FIGURE 5.4 – Évolution du nombre de résultats d'une requête en ajoutant un patron de triplet

Nous avons $a_{Q,t}(\mu) = |\{\mu' \mid f_{Q,t}(\mu') = \mu\}|$ et d'après la preuve de la propriété 4.1, $a_{Q,t}(\mu) = \text{count}(\mu(s(t)), p(t))$. Pour chaque définition de la cardinalité (globale, classe, Characteristic Set) nous montrons que $\text{cardinalite}_{\min}(p(t), s(t), Q \wedge t) \leq \min_{s \in \{\mu(s(t)), \mu \in [[Q]]\}} \text{count}(s, p(t))$.

Notons $S = \{\mu(s(t)), \mu \in [[Q]]\}$.

Pour les cardinalités globales, $\text{cardinalite\_globale}_{\min}(p(t)) = \min_{s \in \text{subject}(D)} \text{count}(s, p)$ et $S \subseteq \text{subject}(D)$. Ainsi $\text{cardinalite\_globale}_{\min}(p(t)) \leq \min_{s \in S} \text{count}(s, p(t))$.

Pour les cardinalités de classe, considérons $s \in S$ et une classe C telle que $\exists t_1 \in Q, s(t_1) = s(t) \wedge C \subseteq \text{domain}(p(t_1))$. D'après la définition de s, $\exists o \mid (s, p(t_1), o) \in D$, donc s appartient à toutes les classes qui sont des domaines de $p(t_1)$, en particulier, $s \in \text{instances}(C)$. Donc $\text{count}(s, p) \geq \min_{s \in \text{instances}(C)} \text{count}(s, p) = \text{cardinalite\_classe}_{\min}(p(t), C)$. Comme cette propriété est vérifiée pour toutes les telles classes C, $\text{count}(s, p) \geq \text{cardinalite\_classe}_{\min}(p(t), s(t), Q \wedge t)$, pour tout $s \in S$. Donc $\text{cardinalite\_classe}_{\min}(p(t), s(t), Q \wedge t) \leq \min_{s \in S} \text{count}(s, p(t))$.

Pour les cardinalités CS, considérons $s \in S$. Notons P le CS de $s : S_C(s) = P$. Nous avons $\text{count}(s, p(t)) \geq \min_{s', S_C(s') = P} \text{count}(s', p(t)) = \text{cardinalite\_CS}_{\min}(p(t), P)$.
Le CS P contient les prédicats de chaque triplet $t' \in Q \wedge t$ tel que $s(t) = s(t')$, donc $\forall t' \in Q \wedge t, s(t') = s(t) \Rightarrow p(t') \in P$. Ainsi, $P \in S_C(p(t), s(t), Q \wedge t)$ (définition 4.4). Comme nous avons $\text{cardinalite\_CS}_{\min}(p(t), s(t), Q \wedge t) = \min_{P \in S_C(p(t), s(t), Q \wedge t)} \text{cardinalite\_CS}_{\min}(p, P)$, nous obtenons $\text{cardinalite\_CS}_{\min}(p(t), s(t), Q \wedge t) \leq \text{cardinalite\_CS}_{\min}(p(t), P)$.
Avec l'inégalité $\text{count}(s, p(t)) \geq \text{cardinalite\_CS}_{\min}(p(t), P)$, nous obtenons $\text{count}(s, p(t)) \geq \text{cardinalite\_CS}_{\min}(p(t), s(t), Q \wedge t)$. Comme cette affirmation est vérifiée pour tout $s \in S$, $\text{cardinalite\_CS}_{\min}(p(t), s(t), Q \wedge t) \leq \min_{s \in S} \text{count}(s, p(t))$.

Comme nous avons $\forall \mu \in [[Q]]$, $\text{count}(\mu(s(t)), p(t)) \geq \min_{s \in S} \text{count}(s, p(t))$, alors $\forall \mu \in [[Q]], a_{Q,t}(\mu) \geq \text{cardinalite}_{\min}(p(t), s(t), Q \wedge t)$.

Enfin, par définition de $f_{Q,t}$ et $a_{Q,t}$, $|[[Q \wedge t]]| = \sum_{\mu \in [[Q]]} a_{Q,t}(\mu)$, puis $\sum_{\mu \in [[Q]]} a_{Q,t}(\mu) \geq \sum_{\mu \in [[Q]]} \text{cardinalite}_{\min}(p(t), s(t), Q \wedge t) = |[[Q]]| \cdot \text{cardinalite}_{\min}(p(t), s(t), Q \wedge t)$.

□

Exemple 5.3 (utilisation des cardinalités)**.** *Considérons la requête Q donnée dans la figure 5.4 avec ses résultats quand elle est exécutée sur la BC du chapitre 2. Notons t le patron de triplet (?p maladie ?m). Avec les cardinalités de CS, nous avons* $\text{cardinalite}_{\min}(\textit{maladie}, ?p, Q \wedge$

$t) = 1$ *et* $\text{cardinalite}_{\max}(maladie, ?p, Q \wedge t) = 2$. *D'après les définitions de cardinalité minimale et maximale, chacun des sujets* p_1, p_2 *et* p_3 *est associé à au moins un objet pour le prédicat maladie. Si chaque sujet est associé à un seul objet, on obtient une valeur possible pour la variable* m *pour chacun des mappings existants. On obtient la borne minimale* $\text{cardinalite}_{\min}(maladie, ?p, Q \wedge t).|[[Q]]| = |[[Q \wedge t]]|$. *Si chaque sujet est associé à deux objets, on obtient deux valeurs possibles pour la variable* m *pour chacun des mappings existants. Chaque résultat de* $[[Q]]$ *apparaît donc deux fois dans le résultat de* $[[Q \wedge t]]$ *avec les deux valeurs pour* m. *On obtient alors la borne maximale* $\text{cardinalite}_{\max}(maladie, ?p, Q \wedge t) \cdot |[[Q]]| = |[[Q \wedge t]]|$. *Dans notre exemple, les résultats de* $Q \wedge t$ *sont donnés dans la figure 5.4c. On a* $|[[Q \wedge t]]| = 4$, *donc l'inégalité est bien vérifiée avec la borne inférieure* $\text{cardinalite}_{\min}(maladie, ?p, Q \wedge t) \cdot |[[Q]]| = 1 * 3 = 3$ *et la borne supérieure* $\text{cardinalite}_{\max}(maladie, ?p, Q \wedge t) \cdot |[[Q]]| = 2 * 3 = 6$.

Si ces propriétés s'appliquent directement aux problèmes de cardinalités, en utilisant les transformations de problème de contenu en problème de cardinalité, elles peuvent être appliquées à tous les problèmes. Pour les conditions d'échec monotones, il existe des propriétés de déduction supplémentaires. Les propriétés suivantes sont une traduction directe de la définition de la monotonie.

Propriété 5.7 (échec pour les conditions positives)**.** *Si FAIL est une condition positive* $FAIL(Q') \wedge Q' \subseteq Q \implies FAIL(Q)$.

Propriété 5.8 (échec pour les conditions négatives)**.** *Si FAIL est une condition négative* $FAIL(Q') \wedge Q \subseteq Q' \implies FAIL(Q)$.

Nous avons évoqué dans le chapitre 3 que le problème des réponses vides, qui est un exemple de condition d'échec monotone positive, a été traité en utilisant les MFS comme causes d'échec. Montrons que pour les problèmes avec une condition d'échec positive, les notions de MFS et de MFIS sont équivalentes.

Définition 5.2 (MFS ; Godfrey 1997)**.**

$$mfs(Q, FAIL) = \{Q^* \mid FAIL(Q^*) \wedge \nexists Q' \subset Q^*, FAIL(Q')\}$$

Propriété 5.9 (équivalence entre MFS et MFIS)**.** *Pour une condition d'échec FAIL positive,* $mfs(Q, FAIL) = mfis(Q, FAIL)$.

Démonstration. Nous montrons l'égalité des ensembles en montrant la double inclusion.

Soit $Q^* \in \text{mfis}(Q, \text{FAIL})$, montrons que $Q^* \in \text{mfs}(Q, \text{FAIL})$. Comme $Q^* \in \text{mfis}(Q, \text{FAIL})$, alors $Q^* \in \text{fis}(Q, \text{FAIL})$, donc $\text{FAIL}(Q^*)$. Supposons qu'il existe $Q' \subset Q^*$ tel que $\text{FAIL}(Q')$. D'après la propriété 5.7, $\forall Q'' \subseteq Q$, $Q' \subset Q'' \implies \text{FAIL}(Q'')$. Ainsi $Q' \in \text{fis}(Q, \text{FAIL})$, ce qui contredit $Q^* \in \text{mfis}(Q, \text{FAIL})$, car Q^* n'est pas minimale. Donc $\nexists Q' \subset Q^*, \text{FAIL}(Q')$, et $Q^* \in \text{mfs}(Q, \text{FAIL})$.

Soit $Q^* \in \text{mfs}(Q, \text{FAIL})$, et montrons que $Q^* \in \text{mfis}(Q, \text{FAIL})$. Comme $Q^* \in \text{mfs}(Q, \text{FAIL})$, $\text{FAIL}(Q^*)$. D'après la propriété 5.7, $\forall Q'' \subseteq Q,\ Q' \subset Q'' \implies \text{FAIL}(Q'')$. Ainsi $Q' \in \text{fis}(Q, \text{FAIL})$. Supposons qu'il existe $Q' \subset Q^*$ tel que $Q' \in \text{fis}(Q, \text{FAIL})$. Nous avons donc $Q' \subset Q^*, \text{FAIL}(Q')$, ce qui contredit $Q^* \in \text{mfs}(Q, \text{FAIL})$. Donc $\nexists Q' \subset Q^*,\ Q' \in \text{fis}(Q, \text{FAIL})$, et $Q^* \in \text{mfis}(Q, \text{FAIL})$. □

Pour identifier une MFS dans le problème des réponses vides en exécutant au plus n requêtes (où n est le nombre de patrons de triplet de la requête initiale), l'algorithme *a_mel_fast* utilise une propriété basée sur la propriété 5.7. Nous avons repris cette propriété en l'adaptant pour une condition d'échec positive quelconque.

Propriété 5.10 (identification des MFS ; Godfrey 1997)**.** *Pour une condition d'échec positive FAIL, deux requêtes Q et Q_i et un patron de triplet t_i tels que $Q = Q_i \wedge t_i$. Si $FAIL(Q)$ et $\neg FAIL(Q_i)$, pour tout $Q' \in mfis(Q, FAIL)$, $t_i \in Q'$.*

Démonstration. Soit $Q' \in \text{mfis}(Q, \text{FAIL})$, et supposons $t_i \notin Q'$. Cela implique $Q' \subseteq Q_i$. Comme Q' est une MFIS, nous avons $\text{FAIL}(Q_i)$ ce qui contredit notre hypothèse. Ainsi, il n'existe pas $Q' \in \text{mfis}(Q, \text{FAIL})$ telle que $t_i \not\subseteq Q'$. □

Pour les conditions d'échec négatives, la propriété 5.8 nous permet de déterminer les MFIS et XSS à partir de l'échec de la requête initiale. En effet, comme l'échec d'une requête implique l'échec de toutes ses sous-requêtes, si la requête initiale de l'utilisateur échoue, la totalité du treillis échoue.

Propriété 5.11 (MFIS et XSS pour les conditions négatives)**.** *Considérons FAIL, une condition d'échec négative. Si $FAIL(Q)$, $xss(Q, FAIL) = \emptyset$, et les MFIS sont toutes les requêtes minimales.*

Comme les MFIS et XSS d'un problème avec une condition d'échec négative peuvent toujours être déterminées sans exécuter de requête supplémentaire, elles ne fournissent pas d'informations utiles pour la résolution des problèmes. Cette propriété nous permet simplement d'affirmer qu'un problème de réponse présente, c'est-à-dire le seul problème élémentaire négatif, ne peut pas être résolu par suppression de patrons de triplet de la requête initiale. Dans ce cas, il faudra corriger la requête en considérant l'ajout ou la modification de patrons de triplets.

5.2.3 Recherche des MFIS et XSS

Nous utilisons deux algorithmes pour identifier les MFIS et XSS d'une requête. Le premier est issu des travaux sur le problème des réponses vides et pourra être utilisé pour les problèmes avec une condition d'échec monotone, le second est une adaptation de l'algorithme Full présenté dans le chapitre 4.

Algorithme 5.1: Algorithme pour l'identification d'une MFS (Fokou, 2016)

```
1  FindAnMFS(FAIL, Q)
       entrées: Une condition d'échec positive FAIL
                Une requête Q = t_1 ∧ ... ∧ t_n telle que FAIL(Q) ;
       sorties : Une MFS de Q
2      mfs ← ∅, Q' ← Q;
3      foreach t_i ∈ Q do
4          Q' ← Q − t_i;
5          if FAIL(Q' ∧ t_i) then
6              mfs ← mfs ∧ t_i ;
7      return mfs ;
```

Algorithme 5.2: Algorithme de recherche des MFS et XSS (Fokou, 2016)

```
1  LBA(FAIL, Q)
       entrées: Une condition d'échec positive FAIL
                Une requête Q = t_1 ∧ ... ∧ t_n telle que FAIL(Q) ;
       sorties : Les MFS et XSS de Q
2      Q* ← FindAnMFS (FAIL, Q);
3      pxss ← pxss(Q, Q*);
4      mfs ← {Q*};
5      xss ← ∅;
6      while pxss ≠ ∅ do
7          Q' ← pxss.element() ; // élément de pxss
8          if isValid (Q')∧¬FAIL(Q') then
9              xss ← xss ∪ {Q'} ;
10             pxss ← pxss − {Q'} ;
11         else
12             if isValid (Q') then
13                 Q** ← FindAnMFS (FAIL,Q');
14                 mfs ← mfs ∪ {Q**};
15                 foreach Q'' ∈ pxss telle que Q** ⊆ Q'' do
16                     pxss ← pxss − {Q''};
17                     pxss ← pxss ∪ {Q_j ∈ pxss(Q'', Q**) | ∄Q_k ∈ pxss ∪ xss : Q_j ⊆ Q_k};
18             else
19                 pxss ← pxss − {Q'};
20     return mfs, xss ;
```

Considérons d'abord la recherche des MFIS et XSS pour les problèmes avec une condition d'échec positive. Il existe un algorithme nommé LBA (algorithme 5.2), proposé pour le problème des réponses vides (Fokou, 2016) qui calcule les MFS et les XSS. Il repose sur la

propriété 5.7. Cet algorithme identifie une première MFIS, avec la fonction FindAnMFS (algorithme 5.1), en retirant les patrons de triplets de la requête initiale un par un. Dans LBA, les XSS possibles sont identifiées avec la fonction pxss(Q, Q^*), où Q est la requête initiale et Q^* est une MFIS, définie par :

$$\text{pxss}(Q, Q^*) : \begin{cases} \emptyset & \text{si } Q \text{ contient un unique patron de triplet} \\ \{Q - t_i | t_i \in Q^*\} & \text{sinon} \end{cases}$$

Le processus est répété à partir des XSS potentielles, jusqu'à ce que la liste des XSS possibles soit vide.

Nous avons montré que dans le cas d'une condition d'échec positive, les MFIS et MFS sont équivalentes, donc cet algorithme peut être utilisé pour identifier les MFIS et XSS pour une condition d'échec positive. Nous avons également montré que les problèmes de réponses vides et de réponses manquantes ont des conditions d'échec positives. Pour adapter l'algorithme au problème des réponses manquantes, il est nécessaire de vérifier la validité des requêtes, c'est-à-dire que les variables indiquées dans le mapping de la réponse manquante doivent faire partie de la requête. Ces vérifications sont donc ajoutées à l'algorithme LBA (lignes 8 et 12).

Pour l'algorithme Shiny (algorithme 5.3) basé sur Full (algorithme 4.1), les changements concernent : la vérification de la validité de la requête (ligne 11), qui permet de vérifier que la requête contient toutes les variables demandées en cas de problème de contenu ; l'exécution de la requête où la condition d'échec utilisée est paramétrable (ligne 13) et l'utilisation des propriétés de déduction (ligne 19). Nous enregistrons dans les conditions d'échec les propriétés de déduction applicables. Elles sont listées dans la table 5.1. Pour les utiliser, il faut commencer par vérifier que la condition de la propriété est respectée. Puis, si l'état de la super-requête Q' correspond à l'hypothèse, l'état de la sous-requête $Q' - t$ est déduit. Celui-ci est renseigné dans le dictionnaire (queryStatus) qui enregistre les échecs et réussites des requêtes étudiées. Dans les chapitres précédents, nous avons uniquement appliqué des propriétés déduisant un échec à partir d'un échec, car si une requête réussit, ses sous-requêtes n'étaient pas étudiées. Nous ajoutons ici des applications des propriétés permettant de déduire la réussite d'une sous-requête à partir de la réussite d'une super-requête, qui seront utiles pour les conditions d'échec combinées étudiées dans la section suivante.

L'algorithme peut être étendu si d'autres propriétés de déduction sont identifiées. Il suffira alors d'ajouter des propriétés aux conditions d'échec concernées.

5.3 Combinaison de conditions d'échec

La problématique de conditions d'échec combinées apparaît lorsque l'utilisateur a plusieurs attentes pour les résultats d'une requête. Cela peut correspondre à un besoin de cardinalité

Algorithme 5.3: Énumération des MFIS et XSS d'une requête Q

```
1  SHINY(FAIL, Q)
       entrées: Une condition d'échec FAIL
                Une requête Q = t_1 ∧ ... ∧ t_n
       sorties : MFIS et XSS de Q
2      mfis ← ∅, xss ← ∅, fis ← ∅, queryStatus ← ∅;
3      list ← {lattice(Q)};
4      while list ≠ ∅ do
5          Q' ← première requête de list en parcours en largeur;
6          list ← list − {Q'};
7          parents_fis ← vrai;
8          superqueries ← superQueries(Q');
9          foreach Q'' ∈ superqueries do
10             parents_fis ← parents_fis ∧ ((Q'') ∈ fis);
11         if isValid (Q') ∧ parents_fis then
12             if Q' ∉ queryStatus then
13                 queryStatus [Q'] ← FAIL(Q');
14             if queryStatus [Q'] then // si Q' échoue
15                 fis ← fis ∪ {Q'};
16                 mfis ← mfis − superqueries;
17                 mfis ← mfis ∪ {Q'};
18                 foreach t ∈ triplePatterns(Q') do
19                     status ← infer_status(FAIL, Q', Q' − t) ; // application des
                         propriétés de déduction
20                     if status is not UNKNOWN then
21                         queryStatus [Q'] ← status;
22             else // Q' réussit et est donc une XSS
23                 xss ← xss ∪ {Q'};
24     return mfis, xss;
```

Échec	Condition de la propriété	Q'	$Q' - t$
$\text{FAIL}_{>K}$	$\text{var}(Q') = \text{var}(Q' - t)$	Échec	Échec
$\text{FAIL}_{<K}$	$\text{var}(Q') = \text{var}(Q' - t)$	Réussite	Réussite
$\text{FAIL}_{>K}$	$\text{cardinalite}_{\max}(p(t), s(t), Q) = 1,\ s(t) \in \text{var}(Q - t)$	Échec	Échec
$\text{FAIL}_{<K}$	$\text{cardinalite}_{\max}(p(t), s(t), Q) = 1,\ s(t) \in \text{var}(Q - t)$	Réussite	Réussite
$\text{FAIL}_{<K}$	$\text{cardinalite}_{\min}(p(t), s(t), Q) = 1,\ s(t) \in \text{var}(Q - t)$	Échec	Échec
$\text{FAIL}_{>K}$	$\text{cardinalite}_{\min}(p(t), s(t), Q) = 1,\ s(t) \in \text{var}(Q - t)$	Réussite	Réussite

TABLE 5.1 – Propriétés de déduction d'échec ou de réussite

spécifique, qui combine alors une condition d'échec pléthorique et une condition d'échec insuffisante, ou un ensemble de réponses souhaitées et non souhaitées, qui combine alors des

conditions d'échec de réponses manquantes et présentes.

Nous rappelons les définitions données au chapitre 2 :

— $(\text{FAIL}_1 \wedge \text{FAIL}_2)(Q) = \text{FAIL}_1(Q) \wedge \text{FAIL}_2(Q)$

— $(\text{FAIL}_1 \vee \text{FAIL}_2)(Q) = \text{FAIL}_1(Q) \vee \text{FAIL}_2(Q)$

Avec ces définitions pour la combinaison des conditions d'échec, nous pouvons appliquer les MFIS et XSS pour la recherche de causes d'échec et d'alternatives lorsque ces problèmes interviennent. Dans la partie suivante, nous proposons deux techniques pour adapter les algorithmes précédents à la gestion de problèmes avec des conditions d'échec combinées. Soit chaque condition d'échec est traitée individuellement, puis les résultats sont fusionnés ; soit les conditions d'échec sont évaluées simultanément. L'objectif est de pouvoir comparer ces deux méthodes, ce qui sera fait dans la section 5.4.

5.3.1 Exécution successive de plusieurs algorithmes

Disposant d'une méthode pour identifier les causes d'échec pour un problème élémentaire, une première méthode pour traiter un problème combiné consiste à exécuter séparément un algorithme pour chaque partie du problème, puis à combiner les MFIS et XSS. Nous commençons donc par considérer individuellement chacune des conditions d'échec impliquées. Cela implique de déterminer les échecs et réussites de toutes les sous-requêtes pour chaque condition d'échec, puis de les combiner, et enfin d'identifier les MFIS et XSS. Pour les conjonctions, nous pouvons cependant limiter le nombre de requêtes à exécuter.

5.3.1.1 Combinaisons de MFIS et XSS

Les MFIS et XSS d'une requête pour une conjonction de conditions d'échec peuvent être calculées à partir des MFIS et XSS des requêtes pour chaque condition d'échec.

Propriété 5.12 (MFIS pour les conditions d'échec combinées)**.** *Soit une requête Q, une condition d'échec $FAIL = FAIL_1 \wedge FAIL_2$, et Q' une MFIS de Q pour $FAIL$. Q' est une super-requête d'une MFIS de Q pour $FAIL_1$ et une super-requête d'une MFIS de Q pour $FAIL_2$.*

Démonstration. Considérons Q' une MFIS de Q pour FAIL. Puisque $Q' \in \text{mfis}(Q, \text{FAIL})$, alors $Q' \in \text{fis}(Q, \text{FAIL})$. Donc $Q' \in \text{fis}(Q, \text{FAIL}_1)$ et $Q' \in \text{fis}(Q, \text{FAIL}_2)$. D'après la définition de MFIS, $\exists Q_1 \in \text{mfis}(Q, \text{FAIL}_1)$ et $\exists Q_2 \in \text{mfis}(Q, \text{FAIL}_2)$ tel que $Q_1 \subseteq Q'$ et $Q_2 \subseteq Q'$. □

Propriété 5.13 (XSS pour les conditions d'échec combinées)**.** *Soit une requête Q, une condition d'échec $FAIL = FAIL_1 \wedge FAIL_2$, et Q' une XSS de Q pour $FAIL$. Q' est une XSS de Q pour $FAIL_1$ ou pour $FAIL_2$.*

Démonstration. Considérons Q' une XSS de Q pour FAIL. Supposons que Q' n'est pas une XSS de Q pour FAIL_1. Soit Q' réussit pour FAIL_1 mais n'est pas maximale, soit Q' échoue

pour FAIL_1. Si $\exists Q'', Q' \subseteq Q'' \wedge \neg\text{FAIL}_1(Q'')$, alors $\neg\text{FAIL}(Q'')$ ce qui contredit l'assertion que Q' est une XSS de Q pour FAIL.

Si $\text{FAIL}_1(Q')$, nous allons montrer que Q' est nécessairement une XSS pour FAIL_2. Comme $\neg\text{FAIL}(Q')$, nous devons donc avoir $\neg\text{FAIL}_2(Q')$. Si $\exists Q''', Q' \subseteq Q''' \wedge \neg\text{FAIL}_2(Q''')$, cela implique $\neg\text{FAIL}(Q''')$ et contredit l'assertion que Q' est une XSS de Q pour FAIL. Donc Q' est une XSS de Q pour FAIL_2. □

Montrons qu'il n'existe pas de propriété similaire pour une disjonction de conditions d'échec avec un contre-exemple issu de l'exemple 2.3. La condition d'échec est $\text{FAIL} = \text{FAIL}_\emptyset \vee \text{FAIL}_{>2}$. Pour la requête initiale Q, nous avons $\neg\text{FAIL}_{>2}(Q)$. Donc Q est la XSS et il n'y a pas de MFIS pour $\text{FAIL}_{>2}$. Cependant, la condition $\text{FAIL}_{>2}$ ne peut pas être ignorée pour l'identification des MFIS et XSS pour FAIL. Donc pour une disjonction de conditions d'échec, il est nécessaire de connaître l'état du treillis pour chacune des conditions d'échec. Pour appliquer les algorithmes précédents, il est donc nécessaire de les modifier afin qu'ils retournent le treillis complet, plutôt que seulement les MFIS et XSS.

Pour une disjonction de conditions d'échec, une fois l'état du treillis connu pour l'une des conditions d'échec, il est uniquement nécessaire de connaître l'état pour la deuxième condition des requêtes qui réussissent pour la première condition. En effet, les requêtes qui échouent pour FAIL_1 échouent pour $\text{FAIL}_1 \vee \text{FAIL}_2$, quel que soit leur état pour FAIL_2. Ainsi, dans notre algorithme, nous pouvons exécuter uniquement les requêtes qui sont des sous-requêtes des XSS pour la deuxième condition d'échec d'une disjonction.

5.3.1.2 Algorithme Shiny-Split

Nous avons vu que le fonctionnement d'une conjonction de conditions d'échec est différent de celui d'une disjonction. Pour une conjonction, nous pouvons appliquer l'algorithme LBA si la condition d'échec est monotone, ou Shiny dans le cas contraire pour chaque condition d'échec, puis combiner les MFIS et XSS. Les MFIS correspondent aux requêtes minimales parmi les conjonctions deux à deux des MFIS de chaque condition de la conjonction. Les XSS sont les requêtes maximales parmi l'union des ensembles des XSS de chaque condition de la conjonction.

Pour la disjonction de conditions d'échec, il est nécessaire de connaître l'état complet du treillis pour l'une des conditions d'échec, puis, pour les requêtes qui réussissent avec cette première condition, leur état selon la seconde condition. L'algorithme 5.4 montre l'adaptation de l'algorithme Shiny afin de retourner le treillis des sous-requêtes de Q. La modification principale est la suppression de la vérification que les parents de la requête étudiée sont bien des FIS (ligne 7). Les autres changements concernent la gestion des listes de MFIS et XSS qui ne sont plus nécessaires. L'optimisation consistant à exécuter uniquement les sous-requêtes des XSS pour la première condition d'échec lors de l'exécution de l'algorithme pour la seconde

Algorithme 5.4: Identification de l'état des sous-requêtes de Q avec SHINY

```
1  SHINY-SPLIT(FAIL, Q)
       entrées: Une condition d'échec FAIL
                Une requête Q = t_1 ∧ ... ∧ t_n
       sorties : Treillis de l'état des sous-requêtes de Q
2      queryStatus ← ∅;
3      list ← {lattice(Q)};
4      while list ≠ ∅ do
5          Q' ← première requête de list en parcours en largeur;
6          list ← list − {Q'};
7          if isValid (Q') then
8              if Q' ∉ queryStatus then
9                  queryStatus [Q'] ← FAIL(Q');
10             foreach t ∈ triplePatterns(Q') do
11                 useProperties (FAIL, Q', Q' − t) ; // application des propriétés de
                       déduction
12     return queryStatus;
```

s'intègre facilement dans l'algorithme. Pour cela, il suffit de remplacer l'initialisation de list du treillis à partir des XSS identifiées (ligne 3). En effet, l'ensemble des requêtes qui réussissent est inclus dans l'ensemble des sous-requêtes des XSS. Pour LBA, comme la condition d'échec est nécessairement monotone, la connaissance des MFIS et XSS suffit pour déduire l'intégralité du treillis.

Exemple 5.4 (exécutions séparées)**.** *Nous présentons dans les figures 5.5 et 5.6 un exemple de problème combiné du chapitre 2. La condition d'échec associée est $FAIL_{\emptyset} \vee FAIL_{>2}$. Nous commençons par considérer la condition d'échec $FAIL_{\emptyset}$. La figure 5.5 donne l'état du treillis pour cette condition d'échec, et les requêtes exécutées par l'algorithme LBA. Cette étape implique l'exécution de 6 requêtes. Les XSS obtenues sont $t_1t_2t_3t_5$ et $t_1t_2t_4t_5$. La deuxième étape exécute l'algorithme* SHINY *à partir de ces deux requêtes pour la condition d'échec $FAIL_{>2}$. La figure 5.6 donne l'état du treillis pour cette condition d'échec. L'utilisation de la propriété 5.5 a permis d'éviter 6 exécutions de requêtes. Cette étape implique l'exécution de 17 requêtes. La figure 5.7 présente les résultats pour la disjonction. Nous obtenons finalement deux MFIS : t_3t_4 et t_4t_5 et deux XSS : $t_1t_2t_3t_5$ et $t_1t_2t_4$. En tout, il y a eu 23 exécutions de requêtes avec cet algorithme.*

Notons que dans cet exemple, les sous-requêtes des XSS pour $FAIL_{\emptyset}$ réussissent pour cette condition d'échec à cause de la monotonie, mais ce n'est pas le cas de façon générale. Par ailleurs, dans cet exemple, les conditions d'échec sont disjointes, le nombre de réponses ne pouvant pas être simultanément strictement supérieur à deux et nul. Pour des problèmes

impliquant un problème de contenu et un problème de cardinalité, ce n'est pas non plus le cas.

5.3.2 Exécution d'un algorithme avec une condition d'échec combinée

La deuxième méthode que nous proposons pour traiter les conditions d'échec combinées consiste à exécuter une fois l'algorithme BASE en utilisant la condition combinée pour déterminer l'échec. Ainsi, chaque requête n'est exécutée qu'une fois, en vérifiant simultanément chaque partie de la condition d'échec.

5.3.2.1 Utilisation des propriétés de déduction

Dans cette méthode, l'optimisation consistant à éviter l'exécution des sous-requêtes des requêtes qui réussissent est possible. Nous pouvons appliquer la propriété 3.3, car nous utilisons une seule condition d'échec.

La monotonie des conditions d'échec est conservée pour des conditions d'échec combinées. Pour ces cas, l'algorithme LBA pourra être utilisé.

Propriété 5.14 (combinaison et monotonie)**.** *Une conjonction ou disjonction de conditions positives (resp. négatives) est positive (resp. négative).*

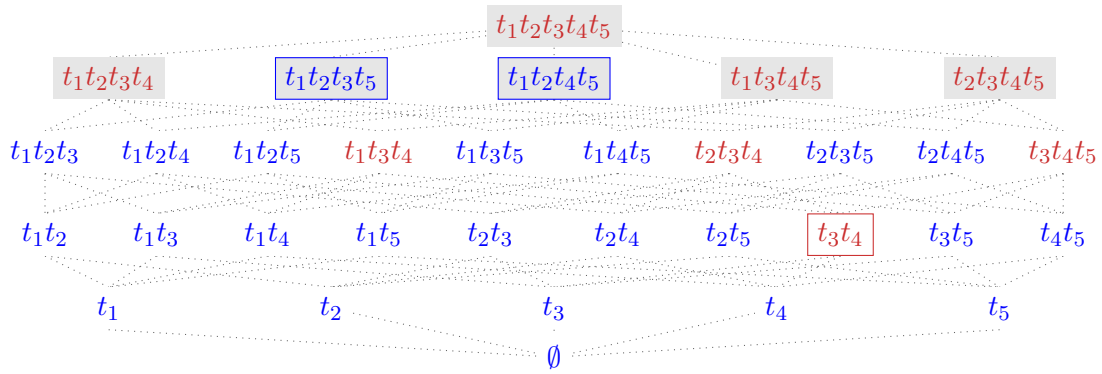
Démonstration. Considérons FAIL_1 et FAIL_2 deux conditions d'échec monotones.

Notons $\text{FAIL} = \text{FAIL}_1 \wedge \text{FAIL}_2$, et considérons Q et Q' telles que $\text{FAIL}(Q)$, et $Q \subseteq Q'$. D'après la définition de FAIL, $\text{FAIL}_1(Q)$ est vrai. Donc d'après la monotonie de FAIL_1, $\text{FAIL}_1(Q')$ est vrai. De même, $\text{FAIL}_2(Q')$ est vrai. Donc $\text{FAIL}(Q')$ est vrai, et FAIL est monotone.

Notons $\text{FAIL} = \text{FAIL}_1 \vee \text{FAIL}_2$, et considérons Q et Q' telles que $\text{FAIL}(Q)$, et $Q \subseteq Q'$. Soit $\text{FAIL}_1(Q)$ est vrai ou $\text{FAIL}_2(Q)$ est vrai. Sans perte de généralité, supposons que $\text{FAIL}_1(Q)$ soit vrai. D'après la monotonie de FAIL_1, $\text{FAIL}_1(Q')$ est vrai. Donc $\text{FAIL}(Q')$ est vrai et FAIL est monotone. □

Les propriétés de déduction liées aux cardinalités bornent le nombre de réponses d'une requête en fonction du nombre de réponses d'une autre requête. Ces propriétés peuvent être utilisées pour une condition d'échec combinée, mais ne permettront pas toujours d'éviter l'exécution d'une requête. Nous donnons un exemple où l'application de la propriété 5.5 ne permet pas d'éviter l'exécution d'une requête.

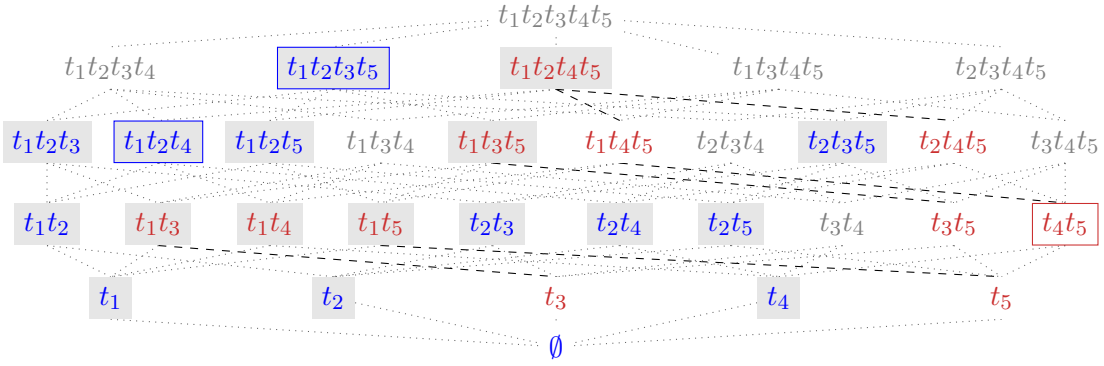
Exemple 5.5 (utilisation des propriétés)**.** *Soit une condition d'échec disjonctive* $FAIL = FAIL_{<10} \vee FAIL_{>100}$*. Considérons* Q' *une requête qui échoue, et* $Q \subset Q'$ *telle que* $var(Q') = var(Q)$*. Nous avons donc* $|[[Q']]| \leq |[[Q]]|$ *d'après la propriété 5.5. Cependant, si* $|[[Q']]| = 5$*, nous pouvons par exemple avoir* $|[[Q]]| = 50$*, donc* Q *réussit, ou* $|[[Q]]| = 150$*, donc* Q *échoue.*



Q : Échec (réponses vides) Q : Réussite $\boxed{Q}$: MFIS $\boxed{Q}$: XSS

Q : Requêtes exécutées

FIGURE 5.5 – Treillis des sous-requêtes pour $\text{FAIL}_{\emptyset}$

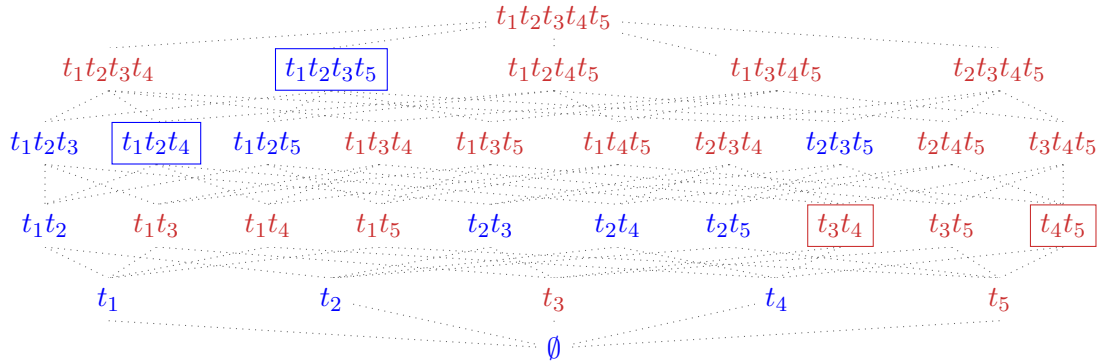


Q : Échec (réponses pléthoriques) Q : Réussite $\boxed{Q}$: MFIS $\boxed{Q}$: XSS

Q : Requêtes exécutées Q : Requêtes non exécutées car échouant pour $\text{FAIL}_{\emptyset}$

- - - - - - utilisation de la propriété 5.5

FIGURE 5.6 – Treillis des sous-requêtes pour $\text{FAIL}_{>2}$



Q : Échec Q : Réussite $\boxed{Q}$: MFIS $\boxed{Q}$: XSS

FIGURE 5.7 – Treillis combiné $\text{FAIL}_{\emptyset} \vee \text{FAIL}_{>2}$

Pour les disjonctions comme pour les conjonctions, l'utilisation de cette propriété permet de déterminer une borne du nombre de résultats, mais cela n'est pas toujours suffisant pour connaître sa réussite ou son échec. Le fonctionnement est le même pour l'utilisation de la propriété 5.6.

5.3.2.2 Algorithme Shiny-Joint

Pour appliquer la méthode d'exécution avec une condition d'échec combinée, nous pouvons appliquer l'algorithme Shiny (algorithme 5.3). Il suffira d'utiliser la condition d'échec combinée pour déterminer l'état de la requête à la ligne 13. Pour cela, nous commençons par exécuter la requête, puis ses résultats sont étudiés en fonction de chacune des conditions d'échec impliquées. Cela signifie que dans cette méthode, la résolution d'un problème de contenu en passant par un problème de cardinalité n'est pas possible, car cela nécessiterait d'exécuter plusieurs requêtes. De la même façon, l'optimisation introduite au chapitre 3 consistant à effectuer le décompte des réponses dans la requête (méthodes d'exécution COUNT et COUNT-LIMIT) n'est pas utilisée dans Shiny-Joint à moins de considérer l'exécution de plusieurs requêtes. L'utilisation des propriétés (ligne 19) applique les propriétés énoncées dans la section précédente. Le dictionnaire utilisé dans l'algorithme pour stocker l'échec ou la réussite de chaque requête (queryStatus) est modifié afin d'enregistrer pour chaque partie élémentaire de la condition d'échec les requêtes qui réussissent et échouent. Il peut être nécessaire de savoir non seulement qu'une requête réussit ou échoue, mais également pour quelle(s) partie(s) de la cause d'échec elle échoue afin d'appliquer l'une des propriétés de déduction.

Exemple 5.6 (exécution combinée)**.** *La figure 5.8 présente l'exécution de l'algorithme* Shiny-Joint *pour l'exemple des problèmes combinés du chapitre 2. La condition d'échec associée est* $FAIL_{\emptyset} \vee FAIL_{>2}$*. Ici, la non-exécution des requêtes avec une super-requête qui réussit permet d'éviter 18 requêtes. La propriété 5.5 permet d'éviter les exécutions des requêtes* $t_1t_4t_5$*,* $t_2t_4t_5$ *et* t_4t_5*. Les MFIS et XSS obtenues sont bien les mêmes que celles obtenues avec l'algorithme* Shiny-Split*. Au total, 11 requêtes sont exécutées par l'algorithme* Shiny-Joint*, tandis que l'algorithme* Shiny-Split *en exécute 23.*

5.4 Implémentation et évaluation expérimentale

Nous proposons deux expériences. Dans un premier temps, nous allons comparer les variations sur nos algorithmes dans l'objectif de quantifier les améliorations de performance liées à chacune des propriétés présentées dans ce chapitre. Par la suite, nous comparerons notre algorithme générique avec des solutions spécifiques pour les problèmes des réponses vides (Fokou et al., 2016), des réponses pléthoriques (Parkin et al., 2021) et des réponses manquantes (Wang

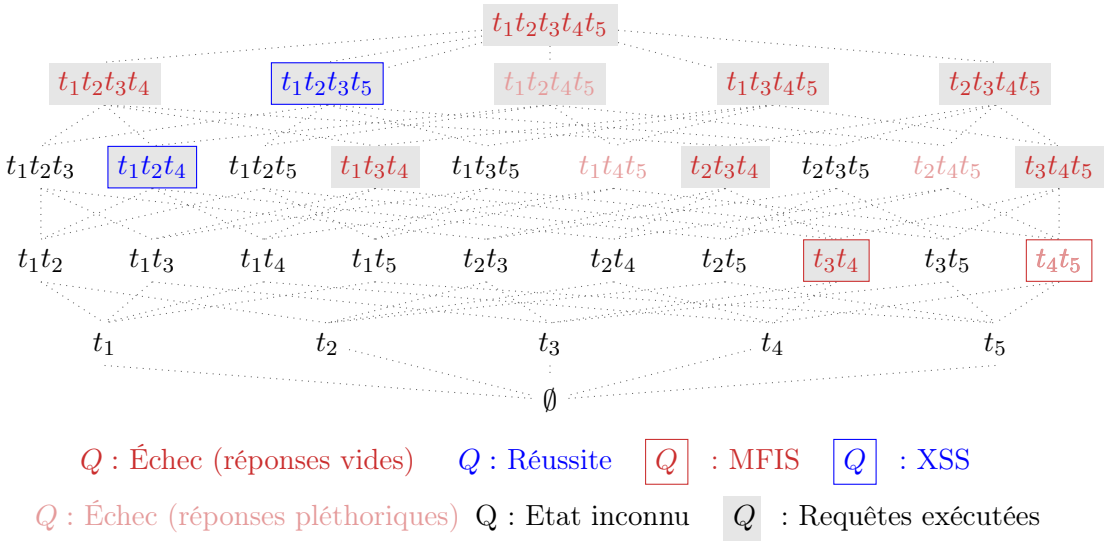


FIGURE 5.8 – Treillis de l'exécution de l'algorithme SHINY-JOINT

et al., 2019). Nous comparerons à la fois les performances des algorithmes et les informations retournées à l'utilisateur.

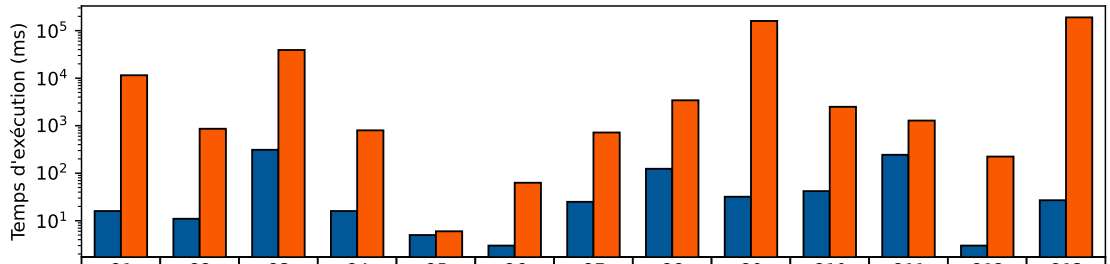
Les algorithmes sont implémentés en Java 1.8 et exécutés sur un serveur Ubuntu 16.04 LTS avec un processeur Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40 GHz et 32 GB RAM. Les temps d'exécution présentés dans les figures sont une moyenne de cinq exécutions consécutives, après une première exécution non comptabilisée. Pour les deux premières expériences, nous avons utilisé la base de connaissances DBpedia, et des requêtes du projet LSQ (Saleem et al., 2015). Les requêtes de chacune des expériences sont fournies dans l'annexe A.

5.4.1 Traitement des problèmes de contenu via la cardinalité

Dans cette expérience, nous étudions les problèmes de contenu, plus particulièrement le problème des réponses manquantes, et nous évaluons l'impact de l'utilisation de la propriété 5.1 pour traiter ce problème en le transformant en problème de cardinalité. La méthode *cardinalité* crée une nouvelle requête en remplaçant les variables de la requête par leur résultat attendu et recherche les MFIS et XSS de cette nouvelle requête pour le problème des réponses vides. La méthode *contenu* recherche le mapping attendu parmi les réponses de la requête.

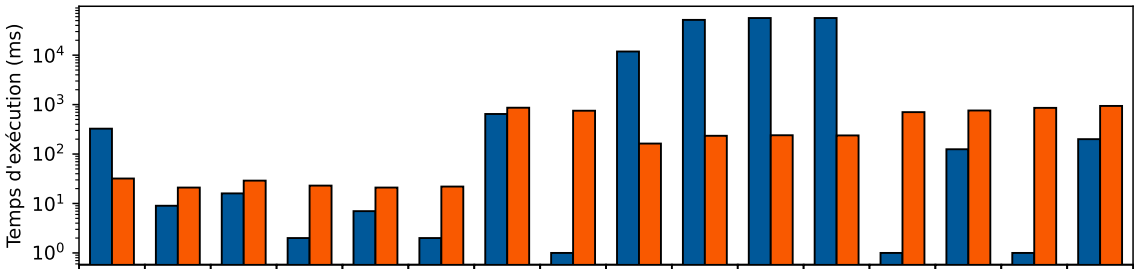
Les réponses manquantes utilisées pour cette expérience proviennent des ensembles de réponses d'une des sous-requêtes, ce qui a permis d'obtenir des réponses manquantes plausibles. La figure 5.9 donne les temps d'exécution pour l'identification des MFIS et XSS de treize problèmes de réponses manquantes.

Le traitement du problème des réponses manquantes via la transformation en problème de cardinalité représente une amélioration significative. Les temps d'exécution sont réduits jusqu'à cinq ordres de grandeur. En moyenne, la méthode *cardinalité* est exécutée en 9% du



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13
cardinalité	16	11	311	16	5	3	25	124	32	42	244	3	27
contenu	11512	862	39148	798	6	63	719	3424	159847	2496	1282	224	190074

FIGURE 5.9 – Temps d'exécution pour des problèmes de réponses manquantes par la méthode de contenu ou de cardinalité



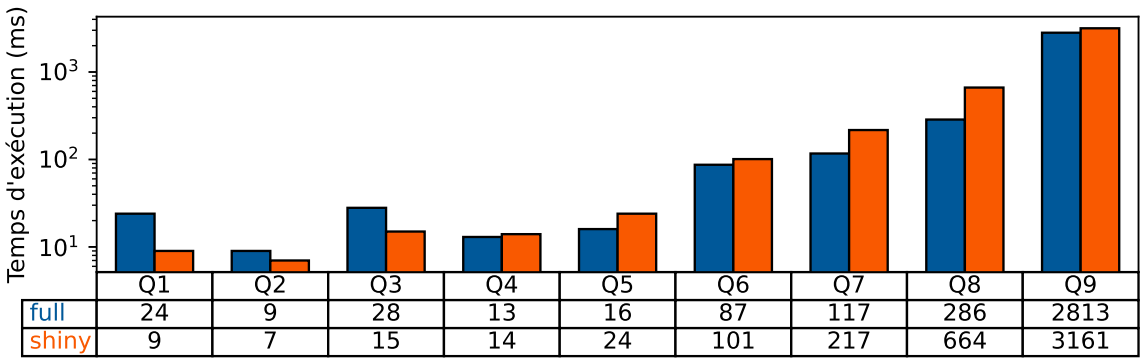
	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16
joint	326	9	16	2	7	2	646	1	11857	51688	56345	56396	1	125	1	200
split	32	21	29	23	21	22	865	752	163	234	240	238	704	760	859	940

FIGURE 5.10 – Temps d'exécution pour des problèmes combinés avec un algorithme combiné ou séparé

temps d'exécution de la méthode *contenu*. Pour la méthode *contenu*, le temps d'exécution est lié au nombre de réponses, car pour déterminer qu'un résultat est manquant, il est nécessaire de vérifier chacune des réponses. Dans la méthode *cardinalité*, il suffit de savoir si une requête produit au moins une réponse, sans les examiner une à une. En transformant le problème de contenu en problème de cardinalité, les résultats qui ne sont pas liés au mapping recherché ne sont pas générés et la majorité du traitement est réalisé par le triplestore.

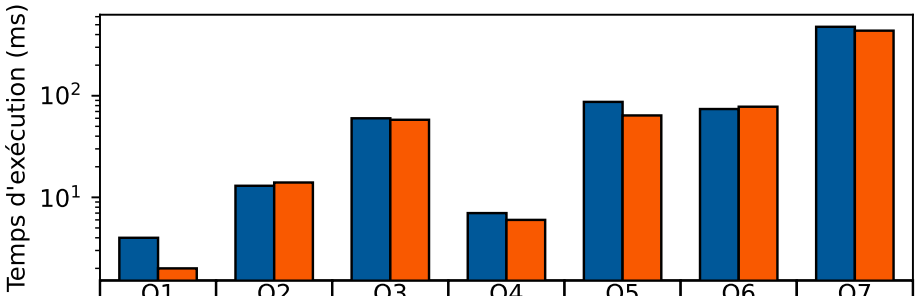
5.4.2 Comparaison des méthodes pour les problèmes combinés

Dans cette deuxième expérience, nous comparons nos deux stratégies d'exécution pour un problème avec une condition d'échec combinée. La méthode SHINY-JOINT utilise une unique exécution d'algorithme avec l'algorithme SHINY en appliquant les propriétés d'élagage. Elle détermine l'échec en utilisant une combinaison des conditions d'échec. La méthode SHINY-SPLIT exécute un algorithme par condition d'échec puis combine les résultats. La figure 5.10 donne le temps d'exécution pour identifier les MFIS et XSS pour seize combinaisons de deux causes d'échec. Dans la plupart des cas, l'exécution *joint* a de meilleures performances, en particulier pour les conjonctions. Avec la méthode SHINY-JOINT, une réussite est détectée



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
full	24	9	28	13	16	87	117	286	2813
shiny	9	7	15	14	24	101	217	664	3161

FIGURE 5.11 – Temps d'exécution pour le problème des réponses pléthoriques avec les algorithmes Full et Shiny



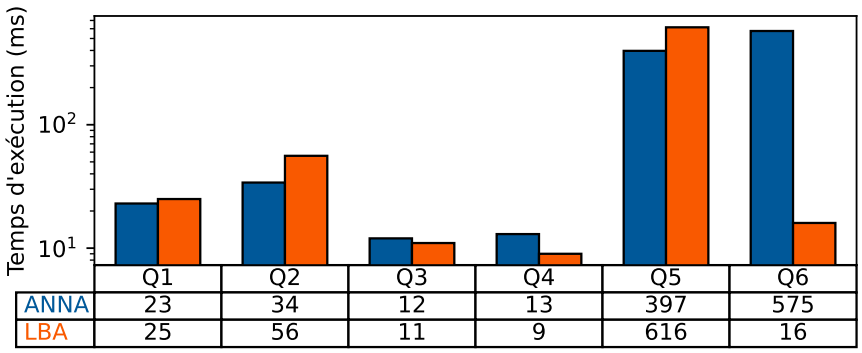
	Q1	Q2	Q3	Q4	Q5	Q6	Q7
QARS	4	13	60	7	87	74	477
LBA	2	14	58	6	64	78	437

FIGURE 5.12 – Temps d'exécution pour le problème des réponses vides avec les algorithmes QARS et LBA

plus rapidement dans le cadre d'une conjonction, puisqu'il suffit qu'une condition d'échec ne soit pas vérifiée pour qu'une requête réussisse. Pour les requêtes 9 à 12, la condition d'échec implique un problème de réponse manquante. Dans l'algorithme SHINY-SPLIT, ce problème est traité en le transformant en problème de cardinalité ce qui entraîne une amélioration de performances significative comme illustré dans l'expérience de la section 5.4.1. Dans ce cas, la méthode d'exécution SHINY-SPLIT a des performances jusqu'à trois ordres de grandeur plus rapides.

5.4.3 Comparaison avec les méthodes existantes

Nous comparons également l'algorithme général avec des approches spécifiques. Nous avons utilisé le système QARS (Fokou et al., 2016) qui traite le problème des réponses vides, notre méthode pour les réponses pléthoriques, présentée dans les chapitres 3 et 4, et le système ANNA (Wang et al., 2019), qui traite le problème des réponses manquantes. L'implémentation de QARS est disponible en ligne, ainsi que les indications de base de connaissances et de



	Q1	Q2	Q3	Q4	Q5	Q6
ANNA	23	34	12	13	397	575
LBA	25	56	11	9	616	16

FIGURE 5.13 – Temps d'exécution pour le problème des réponses manquantes avec les algorithmes ANNA et LBA

Requête initiale	`SELECT * WHERE {?f dbo:director ?dir . ?f dbo:starring ?dir . ?dir dbp:name ?name . ?f dbp:name ?fname . ?dir dbp:gender Female . ?dir dbp:awards dbr:Academy_Award_for_Best_Picture}`
Mapping	`?fname` $\rightarrow$ `Argo`
XSS	`SELECT * WHERE {?f dbo:director ?dir . ?f dbo:starring ?dir . ?dir dbp:name ?name . ?f dbp:name ?fname}`
ANNA modification	`SELECT * WHERE {?f dbo:director ?dir . ?f dbo:starring ?dir . ?dir dbp:name ?name . ?f dbp:name ?fname . ?dir ?v0 dbr:Academy_Award_for_Best_Picture}`

FIGURE 5.14 – Résultats pour Q6 dans l'expérience sur les réponses manquantes

requêtes utilisées pour son évaluation. Nous avons donc repris ces mêmes requêtes, sur la BC LUBM pour notre comparaison. Pour le système ANNA, en l'absence d'une implémentation de référence, nous avons ré-implémenté la méthode en utilisant les indications fournies par les auteurs. Les requêtes utilisées sont issues de cette publication et s'appliquent à la BC DBpedia.

La figure 5.11 donne les temps d'exécution pour identifier les MFIS et XSS pour le problème des réponses pléthoriques, en utilisant l'algorithme spécifique à ce problème défini au chapitre 4 (FULL), et l'algorithme générique (version SHINY). La figure 5.12 donne le temps d'exécution des algorithmes QARS (Fokou et al., 2016) pour identifier les MFS et XSS, et de notre algorithme pour identifier les MFIS et XSS. Pour le problème des réponses vides, nous avons montré que les MFIS et MFS sont équivalentes. Dans ces deux cas, les algorithmes exécutent les mêmes requêtes, et ont donc des temps de traitement très proches, aux variations de mesure près. L'ajout de généricité a peu d'impact sur le temps d'exécution total.

Pour le problème des réponses manquantes, la figure 5.13 donne le temps de modification des requêtes en utilisant l'algorithme ANNA (Wang et al., 2019), et le temps d'identification des MFIS et XSS avec la version LBA de l'algorithme générique. Un exemple des résultats est donné dans la table 5.14. La réponse modifiée retournée par ANNA est proche d'une XSS, mais nous observons des légères différences si l'un des patrons de triplets a été relaxé. Dans

l'exemple de la table 5.14, le dernier patron de triplet de la requête initiale a été supprimé dans la XSS. Dans la modification de ANNA, ce patron de triplet a été relâché en remplaçant son prédicat constant par une variable. Il s'agit ici d'une relaxation par modification, plus proche de la requête initiale que la relaxation par suppression réalisée pour former la XSS. Nous observons de faibles différences de temps de traitement selon les requêtes, mais nous pouvons conclure que les deux approches fournissent des résultats dans un temps raisonnable, inférieur à la seconde.

5.5 Conclusion

Dans ce chapitre, nous avons mis en évidence des propriétés des différentes conditions d'échec, afin d'identifier des méthodes communes pour identifier efficacement les MFIS et XSS. Le formalisme des fonctions booléennes nous a permis d'étudier la monotonie de chacun des problèmes élémentaires. Nous avons montré que les causes d'échec appliquées au problème des réponses vides sont équivalentes aux MFIS dans le cadre d'une condition d'échec positive, et que les MFIS et XSS ne sont pas suffisantes pour expliquer l'échec dans le cas d'une condition d'échec négative. Nous avons montré comment un problème de contenu peut être traité en le transformant en problème de cardinalité. Une comparaison expérimentale a montré que cette méthode permet de diminuer de 93% le temps nécessaire pour en identifier les causes d'échec. Par ailleurs, nous avons proposé deux méthodes pour déterminer les MFIS et XSS avec des conditions d'échec combinées. Nous identifions que dans les cas où un problème de contenu est impliqué, il est préférable de traiter séparément chacune des conditions d'échec, tandis que pour des problèmes de cardinalité, une exécution combinée est plus rapide. Le résultat de nos propositions est l'algorithme Shiny, une approche d'identification des causes d'échec (MFIS), et requêtes alternatives (XSS) applicable pour toutes les conditions d'échec, avec une variation (LBA) pour les problèmes positifs. Une comparaison avec des approches existantes spécifiques à un problème a montré que cette approche générale conserve des temps d'exécution raisonnables et induit un surcoût de traitement négligeable.

Les propriétés identifiées permettent de rechercher les MFIS et XSS en exécutant les requêtes dans un ordre autre que le parcours en largeur descendant. Nous considérons comme perspective le développement d'algorithme de recherche avec différents ordres d'exécution des requêtes. Nous avons vu ici une variante du parcours en profondeur. Une proposition d'un nouvel algorithme consiste à déterminer pour chaque requête du treillis non encore exécutée le nombre d'échecs et de réussites qu'elle nous permet de déduire dans le cas où elle réussit et dans le cas où elle échoue. Le choix de la prochaine requête à exécuter permet ainsi de choisir la requête fournissant le plus d'informations quel que soit son état. Nous pouvons également intégrer dans cette approche l'utilisation d'un optimiseur de requêtes afin d'estimer le temps d'exécution des requêtes et ainsi pondérer le choix des requêtes à exécuter.

Pour le problème des conditions d'échec combinées, nous envisageons également un algorithme intermédiaire permettant de bénéficier des avantages des deux stratégies. Ainsi, le treillis serait parcouru une fois, mais pour chaque sous-requête du treillis, nous acceptons d'exécuter plusieurs requêtes afin de déterminer l'échec ou la réussite. Cela permettrait d'éviter l'exécution des sous-requêtes qui réussissent, comme avec l'algorithme Shiny-Joint, tout en intégrant la transformation des problèmes de contenu en problèmes de cardinalité et l'exécution des requêtes avec COUNT permises par l'algorithme Shiny-Split.

Conclusion

Le développement du Web sémantique a permis des avancées en termes de partage et de diffusion de l'information, avec de plus en plus de domaines qui adoptent les bases de connaissances comme support de leurs travaux. Par conséquent, un enjeu majeur est l'utilisabilité des bases de connaissance. En accompagnant les utilisateurs lorsque leurs requêtes ne produisent pas les résultats souhaités, nos travaux s'intègrent dans ce contexte. Plus particulièrement, nous nous sommes intéressés au problème des réponses pléthoriques, en proposant plusieurs algorithmes pour déterminer les causes d'échec et des requêtes alternatives. Nous avons ensuite élargi notre approche afin de traiter tout problème de réponses non satisfaisantes, y compris si ce problème combine plusieurs types d'insatisfaction.

Nous avons identifié que les approches de la littérature ne permettent pas d'identifier de causes d'échec pour le problème des réponses pléthoriques, ni pour des problèmes combinés. Cependant, l'étude d'autres problèmes de réponses non satisfaisantes, en particulier les réponses vides, a montré que l'identification des causes d'échec est une première étape pertinente afin de reformuler la requête pour produire les résultats attendus. La notion de cause d'échec pour le problème des réponses vides, MFS, nécessitait une adaptation pour ces nouveaux problèmes.

Causes d'échec pour le problème des réponses pléthoriques

Nos premières contributions concernent le problème des réponses pléthoriques. Nous avons proposé une nouvelle définition de causes d'échec, appelée MFIS, et présenté une première méthode naïve pour l'énumération des MFIS et XSS (requêtes alternatives). L'utilisation de propriétés liées à la structure de la requête ou aux cardinalités des prédicats a permis d'optimiser cette méthode en réduisant le nombre de requêtes à exécuter. Notre algorithme Bfs permet de faire diminuer le temps d'exécution de 7% en moyenne et l'algorithme Var de 35%.

En parallèle, nous avons proposé des solutions d'optimisation lors de l'implémentation permettant de réduire le temps d'exécution de chaque requête. En ajoutant les opérateurs COUNT et LIMIT, le temps d'exécution diminue de 82%. Nous avons mené des expériences en utilisant des données synthétiques et réelles avec des requêtes conjonctives de formes diverses comportant jusqu'à douze patrons de triplets. Nous avons ainsi montré que les MFIS et XSS peuvent être calculées dans une durée de l'ordre de la seconde, et ce, pour de bases de

connaissances de taille pouvant dépasser les 800 millions de triplets.

Nos algorithmes sont implémentés dans l'outil SHINY. Celui-ci enrichit un point d'accès SPARQL afin de permettre aux utilisateurs confrontés à des réponses pléthoriques d'afficher le résultat d'une requête alternative ou de corriger leur requête à l'aide des causes d'échec.

Prise en compte des spécificités de RDF et SPARQL

Par défaut, les propriétés de RDF sont multi-valuées. Cependant, via les instances, il est possible d'identifier des cardinalités précises sur une BC. Ce calcul dépend du fait que la BC soit saturée ou non et la nature du schéma utilisé permet de calculer des cardinalités plus ou moins précises. Avec les cardinalités, nous avons montré qu'il est possible de déduire l'échec de requêtes sans les exécuter.

Cependant, l'étude expérimentale de notre algorithme qui exploite les cardinalités a montré que sur les données réelles de DBpedia cet ajout ne permet pas d'améliorer les performances. Cela s'explique par les cardinalités imprécises qui apparaissent en présence de données erronées. Ainsi, la qualité des bases de connaissances est importante pour que la méthode proposée soit efficace.

Les approches précédentes se sont focalisées sur des requêtes conjonctives. Cependant, le langage SPARQL propose des opérateurs spécifiques au langage RDF, par exemple OPTIONAL qui répond à l'incomplétude des BC ou UNION qui permet de combiner des propriétés qui font référence au même concept, mais portent différents noms car elles sont issues de sources différentes. Ces opérateurs sont utilisés en pratique, donc l'extension des approches était importante.

Pour supporter les requêtes contenant les opérateurs FILTER, OPTIONAL ou UNION, nous avons redéfini la notion de sous-requête. Pour une interprétation cohérente des MFIS et XSS, nous imposons également que les requêtes soient traitées sous forme factorisée. Cette contrainte ne limite pas les requêtes que nous pouvons traiter, puisque les règles de distributivité des requêtes SPARQL permettent d'écrire n'importe quelle requête sous cette forme.

Généralisation aux réponses insatisfaisantes

Habituellement, les problèmes de réponses insatisfaisantes sont traités indépendamment les uns des autres. Pourtant, en pratique, plusieurs types de réponses peuvent être considérées non satisfaisantes par un utilisateur. Par exemple, en tentant de corriger une requête produisant des réponses pléthoriques, un utilisateur peut obtenir des réponses insuffisantes, voir vides, qu'il faudra de nouveau réparer. Nous avons donc cherché à identifier les liens qui existaient entre les différents problèmes et qui permettraient de les traiter conjointement.

Ainsi, nous avons proposé deux méthodes pour identifier les MFIS et XSS pour des conditions d'échec combinées. Dans les cas où un problème de contenu est impliqué, il est préférable

de traiter séparément chacune des conditions d'échec, car cela permet de bénéficier d'une optimisation consistant à transformer les problèmes de contenu en problèmes de cardinalité. En revanche, pour des problèmes de cardinalité, une exécution unique d'algorithme s'avère plus efficace.

Nous avons terminé par une comparaison expérimentale de notre algorithme général, qui permet de traiter toute condition d'échec élémentaire, avec des algorithmes existants traitant un problème unique. Les performances proches nous permettent de conclure que la généricité supplémentaire n'impacte pas significativement la performance.

Perspectives

Au cours de la réalisation des travaux présentés dans ce document, plusieurs perspectives ont été identifiées. Les perspectives à court terme concernent des améliorations directes des approches proposées. Celles-ci ont déjà été évoquées dans les chapitres correspondants.

— Utiliser un schéma plus riche pour améliorer l'apport des cardinalités. Jusqu'ici, nous avons calculé les cardinalités en utilisant le schéma RDFS. Le schéma OWL est plus expressif et contient deux propriétés liées aux cardinalités : *owl:minCardinality* et *owl:maxCardinality*.

— Changer l'ordre d'exécution des requêtes pour la recherche des causes d'échec. Il s'agit ici de sortir du parcours en largeur descendant utilisé pour nos algorithmes, pour choisir les requêtes à exécuter au fur et à mesure afin de déduire le maximum de réussites ou d'échec selon l'état de la requête exécutée. Nous proposons d'utiliser des techniques d'estimation du nombre de réponses d'une requête pour choisir judicieusement l'ordre d'exécution.

— Intégrer des modificateurs de requête pour le problème des réponses pléthoriques. Il s'agit des opérateurs intervenant en dehors des patrons de graphe. Certains peuvent modifier le nombre de réponses, comme DISTINCT, LIMIT ou GROUP BY. Nous proposons également d'intégrer les opérateurs dans le traitement du problème général.

Au-delà de ces perspectives spécifiques à court terme, nous envisageons plusieurs axes de recherche de façon plus large à la suite de nos travaux. Ces nouveaux axes portent sur l'utilisation des MFIS et XSS pour résoudre les réponses non satisfaisantes, sur la manière de les présenter aux utilisateurs ou pour la reformulation automatiquement de leurs requêtes, et sur l'adaptation des algorithmes dans de nouveaux contextes de BC distribuées, d'exécution de requêtes multiples ou de recherche par mots clés.

Explication des MFIS et XSS en langue naturelle

Les MFIS et XSS produites par nos algorithmes permettent d'identifier les causes d'échec dans une requête. Cependant, cela implique un certain niveau de compréhension du langage SPARQL de la part de l'utilisateur afin de déterminer les modifications à effectuer au sein de la requête. Or, nos travaux visent à faciliter l'interrogation des bases de connaissances par des utilisateurs novices. Dans ce cadre, il est nécessaire de guider les utilisateurs davantage que simplement fournir les causes d'échec sous forme de patrons de triplets.

Dans l'outil SHINY qui enrichit le point d'accès SPARQL d'un triplestore, nous souhaitons développer le module d'interprétation des MFIS afin de fournir aux utilisateurs des explications en langue naturelle pour décrire précisément les changements à opérer sur leurs requêtes. Il existe des travaux concernant l'interprétation des requêtes SPARQL en langage naturel (Ngonga Ngomo et al., 2013). Ces techniques peuvent être appliquées pour l'explication des XSS, afin de présenter à l'utilisateur ce que calculeront les requêtes alternatives proposées. En revanche, elles ne sont pas suffisantes pour expliquer les causes d'échec. Par exemple, si un échec est dû à la présence d'un produit cartésien, nous voulons que cette information liée à la forme de la requête soit fournie à l'utilisateur, plutôt que lui donner uniquement la traduction de ce que calcule la requête.

Utilisation des MFIS et XSS pour guider la reformulation des requêtes

Dans les contributions que nous avons présentées, les requêtes alternatives proposées aux utilisateurs proviennent exclusivement de la suppression de patrons de triplet de la requête initiale. Nous avons présenté dans le deuxième chapitre des techniques de reformulation de requête qui impliquent l'ajout de nouveau patrons de triplet ou la modification des triplets existants. Ces techniques sont prometteuses, d'autant plus que dans le cadre des réponses vides, l'identification des MFS permet de guider le processus de relaxation (Fokou et al., 2016).

L'obstacle principal pour le problème des réponses pléthoriques est que, contrairement aux MFS, une super-requête d'une MFIS n'échoue pas nécessairement si ce n'est pas également une sous-requête de la requête initiale. Cette implication n'étant vraie que dans le treillis des sous-requêtes de la requête proposée par l'utilisateur, la prise en compte de requêtes extérieures à ce treillis nécessite d'adapter les propriétés de déduction, et potentiellement d'exécuter ces nouvelles requêtes sans pouvoir garantir que leurs résultats seront satisfaisants.

Exécution en lot des requêtes

Comme nos approches déterminent la réussite ou l'échec d'un treillis de sous-requêtes, elles exécutent souvent de nombreuses requêtes similaires. Dans ce contexte, des techniques d'optimisation multi-requêtes peuvent être appliquées pour optimiser nos algorithmes. Ces techniques, introduites pour les bases de données relationnelles, s'adaptent difficilement au

contexte RDF et SPARQL. Cependant, il existe des travaux proposant des heuristiques de regroupement des requêtes afin d'optimiser l'exécution de chaque groupe de requêtes (Le et al., 2012).

Nous avons montré que dans notre approche toutes les requêtes du treillis ne sont pas nécessairement exécutées, car nous pouvons exploiter des propriétés de déduction. Ainsi, pour appliquer l'exécution multiple de requêtes, il s'agira de déterminer à l'avance quelles requêtes doivent être exécutées ensemble. Pour cela, il faudra trouver un équilibre entre exécuter d'emblée trop de requêtes, dont les résultats auraient pu être déduits, et en exécuter trop peu et se retrouver avec des requêtes à l'état inconnu qu'il faudra exécuter individuellement ou dans un nouveau lot.

Passage à l'échelle

Les expérimentations menées sur nos algorithmes ont montré que leurs temps d'exécution sont raisonnables pour des requêtes jusqu'à douze patrons de triplets avec des bases de connaissances centralisées de plusieurs millions de triplets (812 millions de triplets pour DBpedia). Dans le contexte des données massives, des tailles de données plus importantes doivent être considérées. Pour répondre à ce besoin de passage à l'échelle, des solutions de stockage distribuées impliquant plusieurs nœuds ont été proposées. Cela implique de paralléliser nos approches afin de déterminer l'échec ou la réussite de requêtes en minimisant la communication entre les nœuds, qui est coûteuse.

Application aux recherches par mots clés

Nous avons suggéré d'utiliser les transformations entre langue naturelle et requêtes SPARQL pour aider à interpréter les MFIS et XSS. Il est également possible d'utiliser les MFIS et XSS pour améliorer l'interrogation des bases de connaissances en langue naturelle qui utilise la recherche par mots clés. Si l'utilisateur détecte des réponses insatisfaisantes, cela peut être dû à une mauvaise traduction de la recherche par mots clés. L'identification de MFIS et XSS pourra alors servir pour compléter le modèle de traduction des requêtes. Il s'agira de considérer si les notions de MFIS et XSS peuvent s'appliquer à la recherche par mots clés directement. Il peut être préférable de traduire l'ensemble des mots en une requête SPARQL, puis d'y appliquer les méthodes proposées.

Bibliographie

Agrawal, Sanjay, Surajit Chaudhuri, Gautam Das, and Gionis Aristides (2003). "Automated ranking of database query results". In: *Conference on Innovative Data Systems Research (CIDR2003)*. Citeseer.

Ali, Waqas, Muhammad Saleem, Bin Yao, Aidan Hogan, and Axel-Cyrille Ngonga Ngomo (2021). "A survey of RDF stores & SPARQL engines for querying knowledge graphs". In: *The VLDB Journal*, pp. 1–26. DOI: 10.1007/s00778-021-00711-3.

Aluç, Güneş, Olaf Hartig, M Tamer Özsu, and Khuzaima Daudjee (2014). "Diversified stress testing of RDF data management systems". In: *International Semantic Web Conference (ISWC2014)*. Springer, pp. 197–212. DOI: 10.1007/978-3-319-11964-9_13.

Arnaout, Hiba and Shady Elbassuoni (2018). "Effective searching of RDF knowledge graphs". In: *Journal of Web Semantics* 48, pp. 66–84. DOI: 10.1016/j.websem.2017.12.001.

Auer, Sören, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives (2007). "Dbpedia: A nucleus for a web of open data". In: *International Semantic Web Conference (ISWC2007)*. Springer, pp. 722–735. DOI: 10.1007/978-3-540-76298-0_52.

Auer, Sören, Jens Lehmann, and Sebastian Hellmann (2009). "LinkedGeoData: Adding a Spatial Dimension to the Web of Data". In: *International Semantic Web Conference (ISWC2009)*. Springer, pp. 731–746. DOI: 10.1007/978-3-642-04930-9_46.

Bebee, Bradley R, Daniel Choi, Ankit Gupta, Andi Gutmans, Ankesh Khandelwal, Yigit Kiran, Sainath Mallidi, Bruce McGaughy, Mike Personick, Karthik Rajan, et al. (2018). "Amazon Neptune: Graph Data Management in the Cloud". In: *Internation Semantic Web Conference (ISWC2018) P&D/Industry/BlueSky*.

Bechhofer, Sean, Frank van Harmelen, Jim Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein (2004). "OWL Web Ontology Language Reference". In: *W3C Recommendation*. URL: https://www.w3.org/TR/owl-ref/..

Berners-Lee, Tim and Mark Fischetti (1999). *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. Harper San Francisco.

Bidoit, Nicole, Melanie Herschel, and Katerina Tzompanaki (2014). "Query-based why-not provenance with nedexplain". In: *Extending database technology (EDBT)*.

Bigerl, Alexander, Felix Conrads, Charlotte Behning, Mohamed Ahmed Sherif, Muhammad Saleem, and Axel-Cyrille Ngonga Ngomo (2020). "Tentris–A Tensor-Based Triple Store". In: *International Semantic Web Conference (ISWC2020)*. Springer, pp. 56–73. DOI: 10.1007/978-3-030-62419-4_4.

Bonifati, Angela, Wim Martens, and Thomas Timm (2020). "An analytical study of large SPARQL query logs". In: *The International Journal on Very Large Data Bases* 29.2, pp. 655–679. DOI: 10.1007/s00778-019-00558-9.

Borsje, Jethro and Hanno Embregts (2006). "Graphical query composition and natural language processing in an RDF visualization interface". In: *Erasmus School of Economics and Business Economics, Vol. Bachelor. Erasmus University, Rotterdam*, p. 76.

Bosc, Patrick, Allel Hadjali, and Olivier Pivert (2009). "Incremental controlled relaxation of failing flexible queries". In: *Journal of Intelligent Information Systems* 33.3, pp. 261–283. DOI: 10.1007/s10844-008-0071-6.

Bosc, Patrick, Allel Hadjali, Olivier Pivert, and Grégory Smits (2010). "Une approche fondée sur la corrélation entre prédicats pour le traitement des réponses pléthoriques." In: *Conférence Francophane sur l'Extraction et la Gestion des Connaissances (EGC2010)*, pp. 273–284.

Braga, Daniele, Alessandro Campi, and Stefano Ceri (2005). "XQBE (XQuery By Example) A visual interface to the standard XML query language". In: *ACM Transactions on Database Systems (TODS2005)* 30.2, pp. 398–443. DOI: 10.1145/1071610.1071613.

Brickley, Dan and R.V. Guha (2014). "RDF Schema 1.1". In: *W3C Recommendation*. URL: https://www.w3.org/TR/rdf-schema/..

Brin, Sergey and Lawrence Page (1998). "The anatomy of a large-scale hypertextual web search engine". In: *Computer networks and ISDN systems* 30.1-7, pp. 107–117. DOI: 10.1016/S0169-7552(98)00110-X.

Brisaboa, Nieves R, Ana Cerdeira-Pena, Guillermo De Bernardo, and Antonio Fariña (2020). "Revisiting compact RDF stores based on k2-trees". In: *Data Compression Conference (DCC2020)*. IEEE, pp. 123–132. DOI: 10.1109/DCC47342.2020.00020.

Broekstra, Jeen, Arjohn Kampman, and Frank van Harmelen (2002). "Sesame: A generic architecture for storing and querying rdf and rdf schema". In: *International semantic web conference*. Springer, pp. 54–68. DOI: 10.1007/3-540-48005-6_7.

Buneman, Peter, Sanjeev Khanna, and Tan Wang-Chiew (2001). "Why and where: A characterization of data provenance". In: *International conference on database theory (ICDT2001)*. Springer, pp. 316–330. DOI: 10.1007/3-540-44503-X_20.

Chakrabarti, Kaushik, Surajit Chaudhuri, and Seung-won Hwang (2004). "Automatic Categorization of Query Results". In: *SIGMOD'04*, pp. 755–766. DOI: 10.1145/1007568.1007653.

Chapman, Adriane and HV Jagadish (2009). "Why not?" In: *Proceedings of the ACM SIGMOD International Conference on Management of data*, pp. 523–534. DOI: 10.1145/1559845.1559901.

Chaudhuri, Surajit, Gautam Das, Vagelis Hristidis, and Gerhard Weikum (2004). "Probabilistic ranking of database query results". In: *Proceedings of the international conference on Very large data bases (VLDB2004)*. Vol. 30, pp. 888–899. DOI: 10.5555/1316689.1316766.

Chen, Zhiyuan and Tao Li (2009). "Addressing Diverse User Preferences: A Framework for Query Results Navigation". In: *IEEE Data Engineering Bulletin* 32.4, pp. 41–48.

Chen, Zhiyuan, Tao Li, and Yanan Sun (2012). "A Learning Approach to SQL Query Results Ranking Using Skyline and Users' Current Navigational Behavior". In: *IEEE Transactions on Knowledge and Data Engineering (TKDE2012)* 25.12, pp. 2683–2693. DOI: 10.1109/TKDE.2012.128.

Complexible Inc. (2010). https://docs.stardog.com/.

Crama, Yves and Peter L Hammer (2011). *Boolean functions: Theory, algorithms, and applications*. Cambridge University Press.

Cui, Yingwei, Jennifer Widom, and Janet L Wiener (2000). "Tracing the lineage of view data in a warehousing environment". In: *ACM Transactions on Database Systems (TODS2000)* 25.2, pp. 179–227. DOI: 10.1145/357775.357777.

Dellal, Ibrahim (2019). "Management and Exploitation of Large and Uncertain Knowledge Bases". PhD thesis. ISAE-ENSMA - Poitiers.

Dellal, Ibrahim, Stéphane Jean, Allel Hadjali, Brice Chardin, and Mickaël Baron (2017). "On Addressing the Empty Answer Problem in Uncertain Knowledge Bases". In: *Database and Expert Systems Applications (DEXA2017)*, pp. 120–129. DOI: 10.1007/978-3-319-64468-4_9.

Dong, Xin, Evgeniy Gabrilovich, Geremy Heitz, Wilko Horn, Ni Lao, Kevin Murphy, Thomas Strohmann, Shaohua Sun, and Wei Zhang (2014). "Knowledge Vault: A Web-scale Approach to Probabilistic Knowledge Fusion". In: *SIGKDD international conference on Knowledge discovery and data mining (KDD2014)*, pp. 601–610. DOI: 10.1145/2623330.2623623.

Dong, Xin Luna, Xiang He, Andrey Kan, Xian Li, Yan Liang, Jun Ma, Yifan Ethan Xu, Chenwei Zhang, Tong Zhao, Gabriel Blanco Saldana, et al. (2020). "Autoknow: Self-driving knowledge collection for products of thousands of types". In: *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD2020)*, pp. 2724–2734. DOI: 10.1145/3394486.3403323.

Elbassuoni, Shady, Maya Ramanath, Ralf Schenkel, Marcin Sydow, and Gerhard Weikum (2009). "Language-model-based ranking for queries on RDF-graphs". In: *Proceedings*

of the 18th ACM conference on Information and knowledge management (CIKM2009), pp. 977–986. DOI: 10.1145/1645953.1646078.

Erling, Orri and Ivan Mikhailov (2010). "Virtuoso: RDF support in a native RDBMS". In: *Semantic web information management*. Springer, pp. 501–519. DOI: 10.1007/978-3-642-04329-1_21.

Faye, David C, Olivier Cure, and Guillaume Blin (2012). "A survey of RDF storage approaches". In: *Revue Africaine de la Recherche en Informatique et Mathématiques Appliquées* 15, pp. 11–35. DOI: 10.46298/arima.1956.

Fokou, Géraud (2016). "Conception d'un famework pour la relaxation des requêtes SPARQL". PhD thesis. ISAE-ENSMA Ecole Nationale Supérieure de Mécanique et d'Aérotechique-Poitiers.

Fokou, Géraud, Stéphane Jean, Allel Hadjali, and Mickaël Baron (2015). "Cooperative Techniques for SPARQL Query Relaxation in RDF Databases". In: *European Semantic Web Conference (ESWC2015)*. Springer, pp. 237–252. DOI: 10.1007/978-3-319-18818-8_15.

Fokou, Géraud, Stéphane Jean, Allel Hadjali, and Mickaël Baron (2016). "RDF query relaxation strategies based on failure causes". In: *European Semantic Web Conference (ESWC2016)*. Springer, pp. 439–454. DOI: 10.1007/978-3-319-34129-3_27.

Franz Inc. (2004). https://allegrograph.com/products/allegrograph/.

Gallego, Mario Arias, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente (2011). "An Empirical Study of Real-World SPARQL Queries". In: *International Workshop on Usage Analysis and the Web of Data (USEWOD2011) in the International World Wide Web Conference (WWW2011)*. DOI: 10.48550/arXiv.1103.5043.

Gawich, M, A Badr, A Hegazy, and H Ismail (2012). "A methodology for ontology building". In: *International Journal of Computer Applications* 56.2.

Giacometti, Arnaud, Béatrice Markhoff, and Arnaud Soulet (2019). "Mining Significant Maximum Cardinalities in Knowledge Bases". In: *International Semantic Web Conference (ISWC19)*, pp. 182–199. DOI: 10.1007/978-3-030-30793-6_11.

Goasdoué, François, Ioana Manolescu, and Alexandra Roatis (2012a). "Répondre aux requêtes par reformulation dans les bases de données RDF". In: *Reconnaissance des Formes et Intelligence Artificielle (RFIA2012)*, pp. 978–2.

Goasdoué, François, Ioana Manolescu, and Alexandra Roatiş (2012b). "Getting more RDF support from relational databases". In: *Proceedings of the International Conference on World Wide Web (WWW2012)*, pp. 515–516. DOI: 10.1145/2187980.2188104.

Godfrey, Parke (1997). "Minimization in Cooperative Response to Failing Database Queries". In: *International Journal of Cooperative Information Systems* 6.2, pp. 95–149. DOI: 10.1142/S0218843097000070.

Green, Todd J, Grigoris Karvounarakis, and Val Tannen (2007). "Provenance semirings". In: *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 31–40. DOI: 10.1145/1265530.1265535.

Groppe, Jinghua, Sven Groppe, and Andreas Schleifer (2011). "Visual query system for analyzing social semantic web". In: *Proceedings of the international conference companion on World Wide Web (WWW2011)*, pp. 217–220. DOI: 10.1145/1963192.1963293.

Gruber, Thomas R (1995). "Toward principles for the design of ontologies used for knowledge sharing?" In: *International journal of human-computer studies* 43.5-6, pp. 907–928. DOI: 10.1006/ijhc.1995.1081.

Guha, Ramanathan V, Dan Brickley, and Steve Macbeth (2016). "Schema. org: evolution of structured data on the web". In: *Communications of the ACM* 59.2, pp. 44–51. DOI: 10.1145/2844544.

Guo, Qingyu, Fuzhen Zhuang, Chuan Qin, Hengshu Zhu, Xing Xie, Hui Xiong, and Qing He (2020). "A survey on knowledge graph-based recommender systems". In: *IEEE Transactions on Knowledge and Data Engineering (TKDE2020)*. DOI: 10.1109/TKDE.2020.3028705.

Guo, Yuanbo, Zhengxiang Pan, and Jeff Heflin (2005). "LUBM: A benchmark for OWL knowledge base systems". In: *Journal of Web Semantics* 3.2-3, pp. 158–182. DOI: 10.1016/j.websem.2005.06.005.

Haag, Florian, Steffen Lohmann, Steffen Bold, and Thomas Ertl (2014). "Visual SPARQL querying based on extended filter/flow graphs". In: *Proceedings of the 2014 International Working Conference on Advanced Visual Interfaces (AVI2014)*, pp. 305–312. DOI: 10.1145/2598153.2598185.

Harris, Stephen and Nicholas Gibbins (2003). "3store: Efficient bulk RDF storage". In: *1st International Workshop on Practical and Scalable Semantic Systems*, pp. 1–15.

Harris, Steve, Nick Lamb, and Nigel Shadbolt (2009). "4store: The design and implementation of a clustered RDF store". In: *5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2009)*. Vol. 94, pp. 81–96.

Hayes, Patrick J and Peter F Pater-Schneider (2014). "RDFS entailment". In: *W3C Recommendation*. URL: https://www.w3.org/TR/rdf11-mt/#rdfs-entailment.

Herschel, Melanie (2015). "A hybrid approach to answering why-not questions on relational query results". In: *Journal of Data and Information Quality (JDIQ2015)* 5.3, pp. 1–29. DOI: 10.1145/2665070.

Herschel, Melanie and Mauricio A Hernández (2010). "Explaining missing answers to SPJUA queries". In: *Proceedings of the VLDB Endowment* 3.1-2, pp. 185–196. DOI: 10.14778/1920841.1920869.

Hogan, Aidan (2020). "Web of data". In: *The Web of Data*. Springer, pp. 15–57. DOI: 10.1007/978-3-030-51580-5_2.

Hogenboom, Frederik, Viorel Milea, Flavius Frasincar, and Uzay Kaymak (2010). "RDF-GL: a SPARQL-based graphical query language for RDF". In: *Emergent Web Intelligence: Advanced Information Retrieval*. Springer, pp. 87–116. DOI: 10.1007/978-1-84996-074-8_4.

Horrocks, Ian and Peter F Patel-Schneider (2003). "Reducing OWL entailment to description logic satisfiability". In: *International semantic web conference (ISWC2003)*. Springer, pp. 17–29. DOI: 10.1007/978-3-540-39718-2_2.

Huang, Hai, Chengfei Liu, and Xiaofang Zhou (2012). "Approximating query answering on RDF databases". In: *World Wide Web* 15.1, pp. 89–114. DOI: 10.1007/s11280-011-0131-7.

Huang, Jiansheng, Ting Chen, AnHai Doan, and Jeffrey F Naughton (2008). "On the provenance of non-answers to queries over extracted data". In: *Proceedings of the VLDB Endowment* 1.1, pp. 736–747. DOI: 10.14778/1453856.1453936.

Hurtado, Carlos A, Alexandra Poulovassilis, and Peter T Wood (2006). "A relaxed approach to RDF querying". In: *International Semantic Web Conference (ISWC2006)*. Springer, pp. 314–328. DOI: 10.1007/11926078_23.

Hurtado, Carlos A, Alexandra Poulovassilis, and Peter T Wood (2008). "Query relaxation in RDF". In: *Journal on data semantics X (JODS)*, pp. 31–61. DOI: 10.1007/978-3-540-77688-8_2.

Hyvönen, Eero, Tuukka Ruotsalo, Thomas Häggström, Mirva Salminen, Miikka Junnila, Mikko Virkkilä, Mikko Haaramo, Eetu Mäkelä, Tomi Kauppinen, and Kim Viljanen (2006). "Culturesampo–finnish culture on the semantic web: The vision and first results". In: *Developments in Artificial Intelligence and the Semantic Web-Proceedings of the 12th Finnish AI Conference STeP*, pp. 63–72.

Ilyas, Ihab F., George Beskales, and Mohamed A. Soliman (2008). "A Survey of Top-k Query Processing Techniques in Relational Database Systems". In: *ACM Computing Surveys* 40.4, pp. 1–58. DOI: 10.1145/1391729.1391730.

Islam, Md Saiful, Chengfei Liu, and Rui Zhou (2012). "On modeling query refinement by capturing user intent through feedback". In: *Proceedings of the Twenty-Third Australasian Database Conference-Volume 124*, pp. 11–20.

Jagadish, H. V., Adriane Chapman, Aaron Elkiss, Magesh Jayapandian, Yunyao Li, Arnab Nandi, and Cong Yu (2007). "Making Database Systems Usable". In: *SIGMOD internatioanl conference on Management of data (SIGMOD2007)*, pp. 13–24. DOI: 10.1145/1247480.1247483.

Janke, Daniel and Steffen Staab (2018). "Storing and querying semantic data in the cloud". In: *Reasoning Web International Summer School*. Springer, pp. 173–222. DOI: 10.1007/978-3-030-00338-8_7.

Jannach, Dietmar (2006). "Techniques for fast query relaxation in content-based recommender systems". In: *Annual Conference on Artificial Intelligence*. Springer, pp. 49–63. DOI: 10.1007/978-3-540-69912-5_5.

Jayaram, Nandish, Arijit Khan, Chengkai Li, Xifeng Yan, and Ramez Elmasri (2014). "Towards a query-by-example system for knowledge graphs". In: *Proceedings of workshop on graph data management experiences and systems*, pp. 1–6. DOI: 10.1145/2621934.2621937.

Jean, Stéphane (2007). "OntoQL, un langage d'exploitation des bases de données à base ontologique". PhD thesis. Université de Poitiers.

Kaplan, Samuel Jerrold (1979). "Cooperative responses from a portable natural language data base query system." PhD thesis. University of Pennsylvania.

Kießling, Werner (2002). "Foundations of preferences in database systems". In: *Proceedings of the International Conference on Very Large Databases (VLDB2002)*. Elsevier, pp. 311–322. DOI: 10.1016/B978-155860869-6/50035-4.

Kiryakov, Atanas, Damyan Ognyanov, and Dimitar Manov (2005). "OWLIM–a pragmatic semantic repository for OWL". In: *International Conference on Web Information Systems Engineering (WISE2005)*. Springer, pp. 182–192. DOI: 10.1007/11581116_19.

Knorr, Matthias, Carlos Viegas Damásio, Ricardo Gonçalves, and João Leite (2022). "Towards Provenance in Heterogeneous Knowledge Bases". In: *International Conference on Logic Programming and Nonmonotonic Reasoning*. Springer, pp. 287–300. DOI: 10.1007/978-3-031-15707-3_22.

Krishna, Ranjay, Yuke Zhu, Oliver Groth, Justin Johnson, Kenji Hata, Joshua Kravitz, Stephanie Chen, Yannis Kalantidis, Li-Jia Li, David A Shamma, et al. (2017). "Visual genome: Connecting language and vision using crowdsourced dense image annotations". In: *International journal of computer vision* 123.1, pp. 32–73. DOI: 10.1007/S11263-016-0981-7.

Le, Wangchao, Anastasios Kementsietsidis, Songyun Duan, and Feifei Li (2012). "Scalable multi-query optimization for SPARQL". In: *International Conference on Data Engineering (ICDE2012)*. IEEE, pp. 666–677. DOI: 10.1109/ICDE.2012.37.

Lee, Jae-won and Jaehui Park (2019). "An approach to constructing a knowledge graph based on Korean open-government data". In: *Applied Sciences* 9.19. DOI: 10.3390/app9194095.

Lee, Joon Ho, Myoung Ho Kim, and Yoon Joon Lee (1993). "Information retrieval based on conceptual distance in IS-A hierarchies". In: *Journal of documentation* 49.2, pp. 188–207. DOI: 10.1108/eb026913.

Lehmann, Jens, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N. Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer (2015). "DBpedia - A Large-scale, Multilingual Knowledge Base Extracted from Wikipedia". In: *Semantic Web* 6.2, pp. 167–195. DOI: 10.3233/SW-140134.

Luo, Xusheng, Luxin Liu, Yonghua Yang, Le Bo, Yuanpeng Cao, Jinghang Wu, Qiang Li, Keping Yang, and Kenny Q Zhu (2020). "AliCoCo: Alibaba e-commerce cognitive concept net". In: *Proceedings of the 2020 ACM SIGMOD international conference on management of data*, pp. 313–327. DOI: 10.1145/3318464.3386132.

Ma, Li, Chen Wang, Jing Lu, Feng Cao, Yue Pan, and Yong Yu (2008). "Effective and efficient semantic web data management over DB2". In: *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 1183–1194. DOI: 10.1145/1376616.1376735.

Manola, Frank, Eric Miller, Brian McBride, et al. (2004). "RDF primer". In: *W3C recommendation* 10, p. 6.

MarkLogic (2017). https://docs.marklogic.com/.

Maron, Melvin Earl and John Larry Kuhns (1960). "On relevance, probabilistic indexing and information retrieval". In: *Journal of the ACM (JACM)* 7.3, pp. 216–244. DOI: 10.1145/321033.321035.

McSherry, David (2004). "Incremental relaxation of unsuccessful queries". In: *European Conference on Case-Based Reasoning*. Springer, pp. 331–345. DOI: 10.1007/978-3-540-28631-8_25.

Meliou, Alexandra, Wolfgang Gatterbauer, Katherine F Moore, and Dan Suciu (2010). "The complexity of causality and responsibility for query answers and non-answers". In: *Proceedings of the VLDB Endowment* 4.1, pp. 34–45. DOI: 10.14778/1880172.1880176.

Moises, Samyr Abrahão and S do L Pereira (2014). "Dealing with empty and overabundant answers to flexible queries". In: *Journal of Data Analysis and Information Processing*, pp. 12–18. DOI: 10.4236/jdaip.2014.21003.

Möller, Knud, Tom Heath, Siegfried Handschuh, and John Domingue (2007). "Recipes for semantic web dog food–the ESWC and ISWC metadata projects". In: *The Semantic Web*. Springer, pp. 802–815. DOI: 10.1007/978-3-540-76298-0_58.

Motro, Amihai (1984). "Query Generalization: A Method for Interpreting Null Answers". In: *Expert Database Workshop*, pp. 597–616.

Motro, Amihai (1986). "SEAVE: A mechanism for verifying user presuppositions in query systems". In: *ACM Transactions on Information Systems (TOIS)* 4.4, pp. 312–330. DOI: 10.1145/9760.9762.

Motro, Amihai (1996). "Cooperative database systems". In: *International Journal of Intelligent Systems* 11.10, pp. 717–731. DOI: 10.1002/(SICI)1098-111X(199610)11:10<717::AID-INT1>3.0.CO;2-1.

Mottin, Davide, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas (2014). "Exemplar queries: Give me an example of what you need". In: *Proceedings of the VLDB Endowment* 7.5, pp. 365–376. DOI: 10.14778/2732269.2732273.

Muñoz, Emir and Matthias Nickles (2017). "Mining Cardinalities from Knowledge Bases". In: *International Conference on Database and Expert Systems Applications (DEXA2017)*, pp. 447–462. DOI: 10.1007/978-3-319-64468-4_34.

Neumann, T. and G. Moerkotte (2011). "Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins". In: *International Conference on Data Engineering (ICDE11)*, pp. 984–994. DOI: 10.1109/ICDE.2011.5767868.

Neumann, Thomas and Gerhard Weikum (2008). "RDF-3X: a RISC-style engine for RDF". In: *Proceedings of the VLDB Endowment* 1.1, pp. 647–659. DOI: 10.14778/1453856.1453927.

Ngonga Ngomo, Axel-Cyrille, Lorenz Bühmann, Christina Unger, Jens Lehmann, and Daniel Gerber (2013). "Sorry, i don't speak SPARQL: translating SPARQL queries into natural language". In: *Proceedings of the 22nd international conference on World Wide Web*, pp. 977–988. DOI: 10.1145/2488388.2488473.

Ozawa, J. and K. Yamada (1995). "Discovery of global knowledge in a database for cooperative answering". In: *International Conference on Fuzzy Systems*. Vol. 2, pp. 849–854. DOI: 10.1109/FUZZY.1995.409782.

Özsu, M Tamer (2016). "A survey of RDF data management systems". In: *Frontiers of Computer Science* 10.3, pp. 418–432. DOI: 10.1007/s11704-016-5554-y.

Parkin, Louise, Brice Chardin, Stéphane Jean, Allel Hadjali, and Mickaël Baron (2021). "Dealing with Plethoric Answers of SPARQL Queries". In: *International Conference on Database and Expert Systems Applications (DEXA2021)*. Springer, pp. 292–304. DOI: 10.1007/978-3-030-86472-9_27.

Parkin, Louise, Brice Chardin, Stéphane Jean, Allel Hadjali, and Mickael Baron (2022). "A cooperative treatment of the plethoric answers problem in RDF". In: *Knowledge and Information Systems* 64.9, pp. 2481–2514. DOI: 10.1007/s10115-022-01710-8.

Pellissier Tanon, Thomas, Gerhard Weikum, and Fabian Suchanek (2020). "Yago 4: A reasonable knowledge base". In: *European Semantic Web Conference*. Springer, pp. 583–596. DOI: 10.1007/978-3-030-49461-2_34.

Pérez, Jorge, Marcelo Arenas, and Claudio Gutierrez (2009). "Semantics and Complexity of SPARQL". In: *ACM Trans. Database Syst.* 34.3, pp. 1–45. DOI: 10.1145/1567274.1567278.

Pham, Minh-Duc, Linnea Passing, Orri Erling, and Peter A. Boncz (2015). "Deriving an Emergent Relational Schema from RDF Data". In: *International Conference on World Wide Web (WWW2015)*, pp. 864–874. DOI: 10.1145/2736277.2741121.

Ramakrishnan, Cartic, William H Milnor, Matthew Perry, and Amit P Sheth (2005). "Discovering informative connection subgraphs in multi-relational graphs". In: *ACM SIGKDD Explorations Newsletter* 7.2, pp. 56–63. DOI: 10.1145/1117454.1117462.

Resnik, Philip (1995). "Using information content to evaluate semantic similarity in a taxonomy". In: *Proceedings of the 14th international joint conference on Artificial intelligence (IJCAI1995)* 1, pp. 448–453. DOI: 10.48550/arXiv.cmp-lg/9511007.

Ronzhin, Stanislav, Erwin Folmer, Pano Maria, Marco Brattinga, Wouter Beek, Rob Lemmens, and Rein van't Veer (2019). "Kadaster knowledge graph: Beyond the fifth star of open data". In: *Information*, p. 310. DOI: 10.3390/info10100310.

Rubin, Daniel L, Dilvan A Moreira, Pradip Kanjamala, and Mark A Musen (2008). "BioPortal: A Web Portal to Biomedical Ontologies". In: *Association for the Advancement of Artificial Intelligence Spring Symposium: Symbiotic Relationships between Semantic Web and Knowledge Engineering*. Vol. 4, pp. 74–77.

Russell, Alistair, Paul R Smart, Dave Braines, and Nigel R Shadbolt (2008). "Nitelight: A graphical tool for semantic query construction". In: *Semantic Web User Interaction Workshop (SWUI2008)*.

Saïs, Fatiha (2007). "Intégration Sémantique de Données guidée par une Ontologie". PhD thesis. Paris 11.

Saleem, Muhammad, Muhammad Intizar Ali, Aidan Hogan, Qaiser Mehmood, and Axel-Cyrille Ngonga Ngomo (2015). "LSQ: The Linked SPARQL Queries Dataset". In: *International Semantic Web Conference (ISWC2015)*, pp. 261–269. DOI: 10.1007/978-3-319-25010-6_15.

Salton, Gerard (1975). "A vector space model for information retrieval". In: *Journal of the ASIS*, pp. 613–620.

Smits, Grégory (2018). "Personnalisation et enrichissement des méthodes d'accès aux données". Habilitation à diriger des recherches en informatique. Université Rennes 1.

Song, Q., M. H. Namaki, and Y. Wu (2019). "Answering Why-Questions for Subgraph Queries in Multi-attributed Graphs". In: *International Conference on Data Engineering (ICDE19)*, pp. 40–51. DOI: 10.1109/ICDE.2019.00013.

Sperner, Emanuel (1928). "Ein satz über untermengen einer endlichen menge". In: *Mathematische Zeitschrift* 27.1, pp. 544–548.

Stadlera, Claus, Muhammad Saleema, Qaiser Mehmoodb, Carlos Buil-Arandac, Michel Dumontierd, Aidan Hogane, and Axel-Cyrille Ngonga Ngomoa (2022). "LSQ 2.0: A Linked Dataset of SPARQL Query Logs". In: *Semantic Web Journal*.

Suchanek, Fabian M, Gjergji Kasneci, and Gerhard Weikum (2007). "Yago: a core of semantic knowledge". In: *Proceedings of the 16th international conference on World Wide Web*, pp. 697–706. DOI: 10.1145/1242572.1242667.

Suchanek, Fabian M and Nicoleta Preda (2014). "Semantic culturomics". In: *Proceedings of the VLDB Endowment* 7.12, pp. 1215–1218. DOI: 10.14778/2732977.2732994.

Systap (2008). https://blazegraph.com/.

Tao, Tao and ChengXiang Zhai (2006). "Best-k queries on database systems". In: *Proceedings of the ACM international conference on Information and knowledge management*, pp. 790–791.

The Apache Software Foundation (2004). https://jena.apache.org/documentation/tdb/.

Tran, Quoc Trung and Chee-Yong Chan (2010). "How to conquer why-not questions". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 15–26. DOI: 10.1145/1807167.1807172.

UniProt Consortium (2015). "UniProt: a hub for protein information". In: *Nucleic acids research* 43.D1, pp. D204–D212. DOI: 10.1093/nar/gku989.

Vartak, Manasi, Venkatesh Raghavan, and Elke A Rundensteiner (2010). "Qrelx: generating meaningful queries that provide cardinality assurance". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pp. 1215–1218. DOI: 10.1145/1807167.1807323.

Vasilyeva, Elena, Maik Thiele, Christof Bornhövd, and Wolfgang Lehner (2016). "Answering "why empty?" and "why so many?" queries in graph databases". In: *Journal of Computer and System Sciences* 82.1, pp. 3–22. DOI: 10.1016/j.jcss.2015.06.007.

Vicknair, Chad (2010). "Research issues in data provenance". In: *Proceedings of the Annual Southeast Regional Conference*, pp. 1–4. DOI: 10.1145/1900008.1900037.

Vieira, Marcos R, Humberto L Razente, Maria CN Barioni, Marios Hadjieleftheriou, Divesh Srivastava, Caetano Traina, and Vassilis J Tsotras (2011). "On query result diversification". In: *International Conference on Data Engineering (ICDE2011)*. IEEE, pp. 1163–1174. DOI: 10.1109/ICDE.2011.5767846.

Vrandečić, Denny and Markus Krötzsch (2014). "Wikidata: a free collaborative knowledgebase". In: *Communications of the ACM* 57.10, pp. 78–85. DOI: 10.1145/2629489.

Wang, Meng, Jun Liu, Bifan Wei, Siyu Yao, Hongwei Zeng, and Lei Shi (2019). "Answering why-not questions on SPARQL queries". In: *Knowledge and Information Systems* 58, pp. 169–208. DOI: 10.1007/s10115-018-1155-4.

Weiss, Cathrin, Panagiotis Karras, and Abraham Bernstein (2008). "Hexastore: sextuple indexing for semantic web data management". In: *Proceedings of the VLDB Endowment* 1.1, pp. 1008–1019. DOI: 10.14778/1453856.1453965.

Widom, Jennifer (2004). *Trio: A system for integrated management of data, accuracy, and lineage*. Tech. rep. Stanford InfoLab.

Woodruff, Allison and Michael Stonebraker (1997). "Supporting fine-grained data lineage in a database visualization environment". In: *International Conference on Data Engineering (ICDE1997)*. IEEE, pp. 91–102. DOI: 10.1109/ICDE.1997.581742.

Wylot, Marcin, Philippe Cudre-Mauroux, and Paul Groth (2014). "Tripleprov: Efficient processing of lineage queries in a native rdf store". In: *Proceedings of the international conference on World wide web (WWW2014)*, pp. 455–466. DOI: 10.1145/2566486.2568014.

Xie, Min, Raymond Chi-Wing Wong, Peng Peng, and Vassilis J Tsotras (2020). "Being Happy with the Least: Achieving α-happiness with Minimum Number of Tuples". In: *International Conference on Data Engineering (ICDE2020)*. IEEE, pp. 1009–1020. DOI: 10.1109/ICDE48307.2020.00092.

Yang, Shengqi, Yinghui Wu, Huan Sun, and Xifeng Yan (2014). "Schemaless and structureless graph querying". In: *Proceedings of the VLDB Endowment* 7.7, pp. 565–576. DOI: 10.14778/2732286.2732293.

Zeng, Kai, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang (2013). "A distributed graph engine for web scale RDF data". In: *Proceedings of the VLDB Endowment* 6.4, pp. 265–276. DOI: 10.14778/2535570.2488333.

Zhang, Xiaofei, Lei Chen, Yongxin Tong, and Min Wang (2013). "EAGRE: Towards scalable I/O efficient SPARQL query evaluation on the cloud". In: *International Conference on Data Engineering (ICDE2013)*. IEEE, pp. 565–576. DOI: 10.1109/ICDE.2013.6544856.

Zloof, Moshé M (1975). "Query by example". In: *National computer conference and exposition*, pp. 431–438.

Zou, Lei, Jinghui Mo, Lei Chen, M Tamer Özsu, and Dongyan Zhao (2011). "gStore: answering SPARQL queries via subgraph matching". In: *Proceedings of the VLDB Endowment* 4.8, pp. 482–493. DOI: 10.14778/2002974.2002976.

Annexe A

Requêtes des expériences

Nous fournissons les requêtes utilisées pour les évaluations expérimentales. Pour la lisibilité, les préfixes des IRI ont été retirés.

— Les figures A.1 et A.2 contiennent les requêtes exécutées sur la BC WatDiv pour le problème des réponses pléthoriques. Elles sont utilisées dans les expériences des sections 3.4.4, 3.5 et 4.4.1.1 (figures 3.7, 3.11 à 3.16, 3.21, 4.6 et 4.7).

— La figure A.3 fournit les requêtes conjonctives exécutées sur la BC DBpedia pour le problème des réponses pléthoriques. Ces requêtes sont utilisées dans les expériences des sections 3.4.3, 3.4.4, 3.5, 4.4.1 et 5.4.3 (figures 3.6, 3.7, 3.17 à 3.20, 3.22, 3.23, 4.8, 4.9, 4.10 et 5.11).

— Les figures A.4 et A.5 donnent les requêtes contenant des opérateurs FILTER, UNION ou OPTIONAL exécutées sur la BC DBpedia pour le problème des réponses pléthoriques. Elles sont utilisées dans l'expérience de la section 4.4.2 (figures 4.11 et 4.12).

— La figure A.6 contient les requêtes exécutées sur la BC DBpedia pour le problème des réponses manquantes et les réponses manquantes associées. Elles sont utilisées dans l'expérience de la section 5.4.1 (figure 5.9).

— La figure A.7 donne les requêtes, exécutées sur la BC DBpedia, et conditions d'échec utilisées dans l'expérience de la section 5.4.2 (figure 5.10).

— La figure A.8 fournit les requêtes, issues de l'article sur QARS (Fokou et al., 2016), exécutées sur la BC LUBM pour le problème des réponses vides. Ces requêtes sont utilisées dans l'expérience de la section 5.4.3(figure 5.12).

— La figure A.9 fournit les requêtes, issues de l'article sur ANNA (Wang et al., 2019), exécutées sur la BC DBpedia pour le problème des réponses vides. Ces requêtes sont utilisées dans l'expérience de la section 5.4.3 (figure 5.13).

Q1 (4 TP)
```
SELECT * WHERE {
  ?v0 telephone ?v1 .
  ?v0 email ?v2 .
  ?v0 makesPurchase ?v3 .
  ?v0 likes Product0
}
```

Q2 (5 TP)
```
SELECT * WHERE {
  ?v0 friendOf ?v1 .
  ?v0 subscribes ?v2 .
  ?v0 email ?v3 .
  ?v0 likes Product0 .
  ?v0 telephone ?v4
}
```

Q3 (6 TP)
```
SELECT * WHERE {
  ?v0 likes Product0 .
  ?v0 friendOf ?v2 .
  ?v0 Location ?v3 .
  ?v0 age AgeGroup5 .
  ?v0 gender ?v5 .
  ?v0 givenName ?v6
}
```

Q4 (7 TP)
```
SELECT * WHERE {
  ?v0 userId "6347025" .
  ?v0 friendOf ?v1 .
  ?v0 likes ?v2 .
  ?v0 makesPurchase ?v3 .
  ?v0 gender ?v4 .
  ?v0 email ?v5 .
  ?v0 likes ?v6
}
```

Q5 (8 TP)
```
SELECT * WHERE {
  ?v0 follows ?v1 .
  ?v0 friendOf User585 .
  ?v0 nationality ?v3 .
  ?v0 gender ?v4 .
  ?v0 email ?v5 .
  ?v0 familyName ?v6 .
  ?v0 subscribes ?v7 .
  ?v0 givenName ?v8
}
```

Q6 (9 TP)
```
SELECT * WHERE {
  ?v0 follows ?v1 .
  ?v0 friendOf ?v1 .
  ?v0 nationality ?v3 .
  ?v0 gender ?v4 .
  ?v0 email ?v5 .
  ?v0 familyName ?v6 .
  ?v0 subscribes ?v7 .
  ?v0 givenName ?v8 .
  ?v0 likes ?v9
}
```

Q7 (10 TP)
```
SELECT * WHERE {
  ?v0 follows ?v1 .
  ?v0 friendOf User1035 .
  ?v0 nationality ?v3 .
  ?v0 gender ?v4 .
  ?v0 email ?v5 .
  ?v0 familyName ?v6 .
  ?v0 subscribes ?v7 .
  ?v0 givenName ?v8 .
  ?v0 likes ?v9 .
  ?v0 type ?v10
}
```

Q8 (4 TP)
```
SELECT * WHERE {
  User758 follows ?v1 .
  ?v1 likes ?v2 .
  ?v2 actor ?v4 .
  ?v4 friendOf ?v5
}
```

Q9 (5 TP)
```
SELECT * WHERE {
  User758 follows ?v1 .
  ?v1 likes ?v2 .
  ?v2 actor ?v4 .
  ?v4 friendOf ?v5 .
  ?v5 makesPurchase ?v6
}
```

Q10 (6 TP)
```
SELECT * WHERE {
  User758 follows ?v1 .
  ?v1 likes ?v2 .
  ?v2 actor ?v4 .
  ?v4 friendOf ?v5 .
  ?v5 makesPurchase ?v6 .
  ?v6 purchaseFor ?v7
}
```

Q11 (7 TP)
```
SELECT * WHERE {
  User758 follows ?v1 .
  ?v1 likes ?v2 .
  ?v2 hasReview ?v3 .
  ?v3 reviewer ?v4 .
  ?v4 friendOf ?v8 .
  ?v8 Location ?v9 .
  ?v9 parentCountry ?v10
}
```

Q12 (8 TP)
```
SELECT * WHERE {
  User758 follows ?v1 .
  ?v1 likes ?v2 .
  ?v2 actor ?v4 .
  ?v4 friendOf ?v5 .
  ?v5 makesPurchase ?v6 .
  ?v6 purchaseFor ?v7 .
  ?v7 author ?v8 .
  ?v8 Location ?v9
}
```

Q13 (9 TP)
```
SELECT * WHERE {
  ?v0 follows ?v1 .
  ?v1 likes ?v2 .
  ?v2 hasReview ?v3 .
  ?v3 reviewer ?v4 .
  ?v4 makesPurchase ?v6 .
  ?v6 purchaseFor ?v7 .
  ?v7 author ?v8 .
  ?v8 Location ?v9 .
  ?v9 parentCountry
          Country0
}
```

Q14 (10 TP)
```
SELECT * WHERE {
  User758 follows ?v1 .
  ?v1 likes ?v2 .
  ?v2 hasReview ?v3 .
  ?v3 reviewer ?v4 .
  ?v4 friendOf ?v5 .
  ?v5 makesPurchase ?v6 .
  ?v6 purchaseFor ?v7 .
  ?v7 author ?v8 .
  ?v8 Location ?v9 .
  ?v9 parentCountry ?v10
}
```

FIGURE A.1 – Requêtes en étoiles (Q1–Q7) et en chaîne (Q8–Q14) pour les expériences WatDiv

Q15 (6 TP)
```
SELECT * WHERE {
  ?v0 tag ?v1 .
  ?v0 type ?v2 .
  ?v3 trailer ?v4 .
  ?v3 keywords ?v5 .
  ?v3 hasGenre ?v0 .
  ?v3 type ProductCategory2
}
```

Q16 (7 TP)
```
SELECT * WHERE {
  ?v0 title ?v2 .
  ?v0 type ?v3 .
  ?v0 caption ?v4 .
  ?v0 description ?v5 .
  ?v1 url ?v6 .
  ?v1 hits ?v7 .
  ?v0 hasGenre ?v8
}
```

Q17 (8 TP)
```
SELECT * WHERE {
  User758 follows ?v1 .
  ?v1 likes ?v2 .
  ?v2 hasReview ?v3 .
  ?v3 reviewer ?v4 .
  ?v5 makesPurchase ?v6 .
  ?v6 purchaseFor ?v7 .
  ?v7 author ?v8 .
  ?v8 Location City207
}
```

Q18 (9 TP)
```
SELECT * WHERE {
  ?v0 homepage ?v1 .
  ?v2 includes ?v0 .
  ?v0 tag ?v3 .
  ?v0 description ?v4 .
  ?v0 contentSize ?v8 .
  ?v1 url ?v5 .
  ?v1 hits ?v6 .
  ?v1 language Language0 .
  ?v7 likes ?v0
}
```

Q19 (10 TP)
```
SELECT * WHERE {
  ?v0 legalName ?v1 .
  ?v0 offers ?v2 .
  ?v2 eligibleRegion ?v10 .
  ?v2 includes ?v3 .
  ?v4 jobTitle ?v5 .
  ?v4 homepage ?v6 .
  ?v4 makesPurchase ?v7 .
  ?v7 purchaseFor ?v3 .
  ?v3 hasReview ?v8 .
  ?v8 totalVotes ?v9
}
```

Q20 (11 TP)
```
SELECT * WHERE {
    ?v0 subscribes ?v2 .
    ?v0 email ?v3 .
    ?v0 likes Product0 .
    ?v0 age AgeGroup2 .
    ?v0 givenName ?v5 .
    ?v0 familyName ?v1 .
    ?v7 director ?v6 .
    ?v7 actor ?v8 .
    ?v7 hasReview ?v9 .
    ?v7 tag Topic144 .
    ?v7 title ?v10
}
```

Q21 (12 TP)
```
SELECT * WHERE {
    ?v0 subscribes ?v2 .
    ?v0 email ?v3 .
    ?v0 likes Product0 .
    ?v0 telephone ?v4 .
    ?v0 age AgeGroup2 .
    ?v0 givenName ?v5 .
    ?v0 familyName ?v1 .
    ?v7 director ?v6 .
    ?v7 actor ?v8 .
    ?v7 hasReview ?v9 .
    ?v7 tag Topic144 .
    ?v7 title ?v10
}}
```

FIGURE A.2 – Requêtes composites pour les expériences WatDiv (Q15-Q21)

Q1 (4 TP)
```
SELECT * WHERE {
  ?person birthPlace London .
  ?person birthDate ?birth .
  ?person name ?name .
  ?person deathDate ?death
}
```

Q2 (4 TP)
```
SELECT * WHERE {
  ?lang type Language .
  ?nation language ?lang .
  ?nation type Country .
  ?nation comment ?com
}
```

Q3 (5 TP)
```
SELECT * WHERE {
  ?s name ?player .
  ?s type SoccerPlayer .
  ?s position ?position .
  ?s clubs ?club .
  ?s birthPlace ?place
}
```

Q4 (5 TP)
```
SELECT * WHERE {
  ?person type Person .
  ?person type Writer .
  ?person name ?name .
  ?person birthPlace ?birthPlace .
  ?person influenced ?person2
}
```

Q5 (6 TP)
```
SELECT * WHERE {
  ?subject type Film .
  ?subject starring Paul\_Newman .
  ?subject starring ?actors .
  ?subject comment ?abstract .
  ?subject label ?label .
  ?subject releaseDate ?released
}
```

Q6 (7 TP)
```
SELECT * WHERE {
  ?s name ?name .
  ?s type SoccerPlayer .
  ?s position ?position .
  ?s clubs ?club .
  ?club capacity ?cap .
  ?s birthPlace ?place .
  ?s number ?tricot
}
```

Q7 (8 TP)
```
SELECT * WHERE {
  ?m type MusicalArtist .
  ?m activeYearsStartYear
          ?activeyearsstartyear .
  ?m associatedBand ?associatedband .
  ?m birthPlace ?birthplace .
  ?m genre ?genre .
  ?m recordLabel ?recordlable .
  ?m voiceType ?voicetype .
  ?artist artist ?m
}
```

Q8 (9 TP)
```
SELECT * WHERE {
  ?m type MusicalArtist .
  ?m activeYearsStartYear
          ?activeyearsstartyear .
  ?m associatedBand ?associatedband .
  ?m birthPlace ?birthplace .
  ?m genre ?genre .
  ?m recordLabel ?recordlable .
  ?m voiceType ?voicetype .
  ?artist artist ?m .
  ?starring starring ?m
}
```

Q9 (10 TP)
```
SELECT * WHERE {
  ?m type MusicalArtist .
  ?m activeYearsStartYear
          ?activeyearsstartyear .
  ?m associatedBand ?associatedband .
  ?m birthPlace ?birthplace .
  ?m genre ?genre .
  ?m recordLabel ?recordlable .
  ?m voiceType ?voicetype .
  ?artist artist ?m .
  ?starring starring ?m .
  ?writer writer ?m
}
```

FIGURE A.3 – Requêtes conjonctives pour les expériences DBpedia (Q1-Q9)

Q1 (4 TP - Union)
```
SELECT * WHERE {
?s homepage colo
{ ?s ?p ?o .
?p label ?label }
UNION { ?s sameAs ?same } }
```

Q2 (5 TP - Union)
```
SELECT * WHERE {
?Food type Food .
?Food ingredient ?Ingredient1 .
?Ingredient1 ingredient ?Ingredient2 .
{ ?Food origin United_States }
UNION { ?Food origin Canada } }
```

Q3 (6 TP - Optional)
```
SELECT * WHERE {
?person award Nobel_in_Economics .
?person name ?sname .
?person birthDate ?dob
OPTIONAL { ?person thumbnail ?img }
OPTIONAL { ?person nationality ?nat }
OPTIONAL { ?person almaMater ?am } }
```

Q4 (8 TP - Union)
```
SELECT * WHERE {
?subject name ?name .
?subject label ?label .
?subject type Album .
?subject cover ?coverFilename
{ ?subject prop-artist ?artist }
UNION { ?subject artist ?artist }
{ ?artist label "Prince"@en }
UNION { ?artist name "Prince
"@en } }
```

Q5 (8 TP - Optional)
```
SELECT * WHERE {
?person1 type Person .
?person1 birthPlace ?bplace1 .
?person influenced Friedrich_Nietzsche .
Friedrich_Nietzsche birthPlace ?bplace2
OPTIONAL {
?bplace1 lat ?lat1 .
?bplace1 long ?long1 .
?bplace2 lat ?lat2 .
?bplace2 long ?long2 } }
```

Q6 (9 TP - Filter)
```
SELECT * WHERE {
?movie type Film .
?movie name ?movieTitle .
?movie director ?directorName .
?movie distributor ?DistributorName .
?movie country ?Country .
?movie releaseDate ?ReleaseDate .
?movie genre ?Genre
FILTER ( ( ! isLiteral(?movieTitle) )
|| langMatches(lang(?movieTitle), "EN") )
FILTER ( ( ?ReleaseDate >= "2005-04-01" )
&& ( ?ReleaseDate < "2006-01-01" ) ) }
```

Q7 (10 TP - Optional)
```
SELECT * WHERE {
?pers occupation Actor .
?pers birthPlace Belgium .
?pers name ?name .
?pers comment ?description .
?pers birthPlace ?bPlace .
?pers birthDate ?birthDate .
?pers occupation ?occ .
?occ label ?occupation .
?bPlace label ?birthPlace
OPTIONAL {
?pers placeOfBirth ?placeOfBirth } }
```

Q8 (11 TP - Filter)
```
SELECT * WHERE {
?person type Politician .
?person name ?surname .
?person birthPlace ?c .
?c country ?co .
?person birthDate ?by .
?person almaMater ?mater .
?mater label ?almaMater .
?co label ?country
FILTER langMatches(lang(?country), "en")
FILTER langMatches(lang(?almaMater),"en")
FILTER ( ?by > "1980-06-01" ) }
```

Q9 (11 TP - Filter)
```
SELECT * WHERE {
?person type Scientist .
?person name ?surname .
?person birthPlace ?c .
?c country ?co .
?person birthDate ?by .
?person almaMater ?mater .
?mater label ?almaMater .
?co label ?country
FILTER langMatches(lang(?country), "en")
FILTER langMatches(lang(?almaMater),"en")
FILTER ( ?by > "1977-01-01" ) }
```

FIGURE A.4 – Requêtes DBpedia avec opérateurs (Q1-Q9)

Q10 (11 TP - Filter)

```
SELECT * WHERE {
?person type Economist .
?person name ?surname .
?person birthPlace ?c .
?c country ?co .
?person birthDate ?by .
?person almaMater ?mater .
?mater label ?almaMater .
?co label ?country
FILTER langMatches(lang(?country), "en")
FILTER langMatches(lang(?almaMater),"en")
FILTER ( ?by > "1958-01-01" ) }
```

Q11 (12 TP - Optional & Filter)

```
SELECT * WHERE {
?a type Book .
?a label ?title .
?a numberOfPages ?numri .
?a publicationDate ?pyear .
?a abstract ?abstract .
?a author ?name .
?name name ?author
FILTER ( lang(?title) = "en" )
FILTER ( lang(?author) = "en" )
FILTER ( lang(?abstract) = "en" )
FILTER ( ?numri > 650 )
OPTIONAL { ?a label ?Title } }
```

FIGURE A.5 – Requêtes DBpedia avec opérateurs (Q10-Q11)

Q1 (?person : Curtis_Ackie)
```
SELECT * WHERE {
?person birthPlace London .
?person birthDate ?birth .
?person name ?name .
?person deathDate ?death }
```

Q2 (?lang : Aivilingmiutut)
```
SELECT * WHERE {
?lang type Language .
?nation language ?lang .
?nation type Country .
?nation comment ?com }
```

Q3 (?position : Pitcher)
```
SELECT * WHERE {
?s name ?player .
?s type SoccerPlayer .
?s position ?position .
?s clubs ?club .
?s birthPlace ?place }
```

Q4 (?person : A._A._Milne)
```
SELECT * WHERE{
?person type Person .
?person type Writer .
?person name ?name .
?person birthPlace ?birthPlace .
?person influenced ?person2 }
```

Q5 (?book : Mansfield_Park)
```
SELECT * WHERE {
Pride_and_Prejudice author ?author .
?book author ?author .
?book releaseDate ?date .
?book literaryGenre Romance_novel }
```

Q6 (?influence : Baruch_Spinoza)
```
SELECT * WHERE {
?N type Philosopher .
?N era ?epoque .
?N influenced ?influence .
?N influencedBy ?influence }
```

Q7 (?a : Woody_Allen)
```
SELECT * WHERE {
?movie director ?a .
?movie executiveProducer ?a .
?movie author ?a .
?movie creator ?a .
?movie starring ?a }
```

Q8 (?l : Russian_language)
```
SELECT * WHERE {
?movie type Film .
?movie country ?mcountry .
?mcountry language ?l .
?p birthPlace ?pcountry .
?pcountry language Spanish_language .
?movie starring ?p }
```

Q9 (type : WikicatCountriesInEurope)
```
SELECT * WHERE {
?x type Person .
?x birthPlace ?city .
?x award 2010_Nobel_Peace_Prize .
?city type Place .
?city country ?country .
?country type ?type }
```

Q10 (?book : Night_Flight(novel))
```
SELECT * WHERE {
The_Little_Prince author ?author .
?book author ?author .
?author birthDate ?birthDate .
?author birthPlace ?birthPlace .
?author deathPlace ?deathPlace .
?deathPlace label ?deathPlaceName .
?author deathDate ?deathDate }
```

Q11 (?u : Berlin_Mathematical_School)
```
SELECT * WHERE {
?c demonym ?Country .
?c type WikicatCountriesInEurope .
?c currency Euro .
?u type University .
?u country> ?c .
?u name ?University .
?u students ?Students .
?u staff ?Staff }
```

Q12 (?book : Mansfield_Park ;
?b : Pride_and_Prejudice)
```
SELECT * WHERE {
?b author ?author .
?book author ?author .
?book releaseDate ?date .
?book literaryGenre Romance_novel }
```

Q13 (?type : WikicatCountriesInEurope ;
?award : 2010_Nobel_Peace_Prize)
```
SELECT * WHERE {
?x type Person .
?x birthPlace ?city .
?x award ?award .
?city type Place .
?city country ?country .
?country type ?type }
```

FIGURE A.6 – Requêtes DBpedia et mappings pour le problème des réponses manquantes (Q1-Q13)

Query 1
```
SELECT * WHERE {
?b author ?author .
?book author ?author .
?book releaseDate ?date .
?book literaryGenre Romance_novel }
```

Query 2
```
SELECT * WHERE {
?author type Writer .
?author label ?author_name .
?author notableWork ?work .
?work label ?title }
```

Query 3
```
SELECT * WHERE {
?c demonym ?Country .
?c type WikicatCountriesInEurope .
?c currency Euro .
?u type University .
?u country ?c .
?u name ?University .
?u students ?Students .
?u staff ?Staff }
```

Q1 Query 1
(∉(?book : Mansfield_Park ;
?b : Pride_and_Prejudice)
AND (>100))

Q2 Query 1
(∉(?book : Mansfield_Park ;
?b : Pride_and_Prejudice)
OR (>100))

Q3 Query 2
(>100 **OR** <10)

Q4 Query 2
(>100 **AND** <10)

Q5 Query 2
(<100 **OR** >10)

Q6 Query 2
(<100 **AND** >10)

Q7 Query 3
(∉(?u : Berlin_Mathematical_School ;
?c : Germany)
OR (>100))

Q8 Query 3
(∉(?u : Berlin_Mathematical_School ;
?c : Germany)
AND (>100))

Q9 Query 3
(∉(?u : Berlin_Mathematical_School ;
?c : Germany)
AND (<1))

Q10 Query 3
(∉(?u : Berlin_Mathematical_School ;
?c : Germany)
OR (<1))

Q11 Query 3
(∉(?u : Berlin_Mathematical_School ;
?c : Germany)
OR (∉(?c : France ;
?u : Paris_Sciences_et_Lettres_University))

Q12 Query 3
(∉(?u : Berlin_Mathematical_School ;
?c : Germany)
AND (∉(?c : France ;
?u : Paris_Sciences_et_Lettres_University))

Q13 Query 3
(<1 **AND** >100)

Q14 Query 3
(<1 **OR** >100)

Q15 Query 3
(<100 **AND** >1)

Q16 Query 3
(<100 **OR** >1)

FIGURE A.7 – Requêtes DBpedia et conditions d'échec pour les problèmes combinés (Q1-Q16)

Q1 (2 TP)
```
SELECT * WHERE {
?X type FullProfessor .
?X title 'DR' }
```

Q2 (4 TP)
```
SELECT * WHERE {
UndergraduateStudent33 advisor ?Y13 .
?Y13 doctoralDegreeFrom ?Y14 .
?Y14 hasAlumnus ?Y15 .
?Y15 title ?Y16 }
```

Q3 (5 TP)
```
SELECT * WHERE {
?X type FullProfessor .
?X publicationAuthor ?Y1 .
?X worksFor ?Y2 .
?S advisor ?X .
?X title ?Y4 }
```

Q4 (7 TP)
```
SELECT * WHERE {
?X type UndergraduateStudent .
?X memberOf ?Y1 .
?X mastersDegreeFrom University822 .
?X emailAddress ?Y2 .
?X advisor FullProfessor0 .
?X takesCourse ?Y4 .
?X name ?Y5 }
```

Q5 (9 TP)
```
SELECT * WHERE {
?X type FullProfessor .
?X doctoralDegreeFrom ?Y3 .
?X memberOf ?Y2 .
?X headOf ?Y3 .
?X title ?Y4 .
?X officeNumber ?Y5 .
?X researchInterest ?Y6 .
?S advisor ?X .
?S name ?Y7 }
```

Q6 (12 TP)
```
SELECT * WHERE {
?X type Faculty .
?X doctoralDegreeFrom ?Y1 .
?X memberOf ?Y2 .
?X headOf ?Y3 .
?X title ?Y4 .
?X officeNumber ?Y5 .
?X researchInterest ?Y6 .
?X name 'FullProfessor3' .
?X emailAddress ?Y7 .
?X age ?Y8 .
?X mastersDegreeFrom University1 .
?X undergraduateDegreeFrom ?Y9 }
```

Q7 (15 TP)
```
SELECT * WHERE {
?X type Professor .
?X teacherOf Course2 .
?X name ?Y1 .
?X age ?Y2 .
?X emailAddress ?Y3 .
?X mastersDegreeFrom ?Y4 .
?X worksFor ?Y5 .
?Y5 subOrganizationOf ?Y6 .
?Y6 name ?Y7 .
?Y8 advisor ?X .
?Y8 mastersDegreeFrom ?Y4 .
?Y8 memberOf ?Y9 .
?Y8 emailAdress ?Y10 .
?Y8 takesCourse ?Y11 .
?Y8 name ?Y12 }
```

FIGURE A.8 – Requêtes LUBM pour la comparaison avec QARS (Q1-Q7)

Q1 (?name : "Batman"@en)

```
SELECT * WHERE {
?film releaseDate ?date .
?film director ?person .
?film name ?name .
?person name "Tim Burton"@en }
```

Q2 (?actorname : "Spike Lee"@en)

```
SELECT * WHERE {
?actor name ?actorname .
?actor type WikicatActors .
?actor country ?c .
?actor gender "male"@en .
?actor awards Academy_Award_Best_Actor }
```

Q3 (?filmname : "Titanic"@en)

```
SELECT * WHERE {
?film name ?filmname .
?film type WikicatAmericanRomanceFilms .
?film type WikicatAmericanDramaFilms .
?film releaseDate ?date }
```

Q4 (?fname : "The English Patient"@en)

```
SELECT * WHERE {
?film name ?fname .
?film type WikicatAmericanRomanceFilms .
?film type WikicatAmericanDramaFilms .
?film releaseDate ?date }
```

Q5 (?filmname : "Atonement"@en)

```
SELECT * WHERE {
?f name ?filmname .
?f type WikicatAmericanWarFilms .
?f type WikicatAmericanAdventureFilms .
?f country "United States"@en }
```

Q6 (?filname : "Argo"@en)

```
SELECT * WHERE {
?film director ?director .
?film starring ?director .
?director name ?directorname .
?film name ?filmname .
?director awards ?award .
?director gender "Female"@en}
```

FIGURE A.9 – Requêtes DBpedia et mappings pour la comparaison avec ANNA (Q1-Q6)

Table des figures

Liste des tableaux

Résumé

Avec le développement des bases de connaissances dans de nombreux domaines industriels comme académiques, des utilisateurs novices sont confrontés à la nécessité de formuler des requêtes, sans forcément maîtriser le langage de requête, SPARQL, ou la structure de données sous-jacente, généralement décrite avec le langage RDF. Ces utilisateurs peuvent ainsi commettre des erreurs lors de l'écriture de leurs requêtes et obtenir des résultats inattendus ou difficiles à traiter. Parmi les situations d'insatisfaction des utilisateurs, le problème des réponses vides a été largement étudié. L'explication des raisons de l'absence de réponse peut permettre soit à l'utilisateur de progresser dans l'écriture de ses requêtes, soit à les corriger automatiquement. Mais l'absence de réponse n'est pas la seule source possible d'insatisfaction et peu de travaux existants se sont intéressés à l'identification des causes d'échec pour des problèmes différents. Dans un premier temps nous nous intéressons au problème des réponses pléthoriques, c'est-à-dire lorsqu'une requête produit un très grand nombre de réponses alors que l'utilisateur ne s'y attendait pas, et qu'il ne peut alors pas en extraire l'information pertinente. Nous montrons que des notions de cause d'échec et de requête alternatives introduites pour le problème des réponses vides peuvent être étendues au problème des réponses pléthoriques, et nous introduisons des algorithmes de calcul adaptés. Nous avons ensuite considéré les apports spécifiques de SPARQL en utilisant les cardinalités de prédicats pour améliorer les algorithmes de recherche et en adaptant notre formalisme pour accepter les requêtes contenant plusieurs opérateurs spécifiques à ce langage. Enfin, la méthode est généralisée pour un problème quelconque d'insatisfaction de l'utilisateur avec les résultats obtenus. Nous montrons comment traiter cinq problèmes élémentaires de réponses insatisfaisantes et comment les combiner pour décrire des problèmes plus complexes. Nos contributions ont été validées expérimentalement en utilisant des données et requêtes synthétiques de WatDiv et des données et requêtes réelles de DBpedia.

Mots-clés : Bases de données–Interrogation / Optimisation mathématique / Questions et réponses / Resource Description Framework (informatique) / SPARQL (langage de programmation) / Systèmes, Analyse de / Systèmes experts (informatique) / Réponses pléthoriques / Réponses insatisfaisantes / Causes d'échec

Abstract

The development of knowledge bases across multiple industrial and academic contexts has led to novice users being faced with writing queries without always fully understanding the query language, SPARQL, or the underlying data structure, often defined with the RDF language. These users can therefore make mistakes when writing a query and consequently receive unexpected or unwanted answers. A possible source of user dissatisfaction is the absence of results. This problem has been widely studied and the explanation of empty answers can be used either to help the user improve their query formulation, or to automatically correct them. But the absence of results is not the only possible reason for user dissatisfaction and few existing works focus on identifying failure causes for other problems. We first consider the overabundant answers problem, where a query produces a very large number of answers which the user was not expecting, meaning that they cannot extract useful information. We show that failure causes and alternative queries introduced to deal with empty answers can be extended to the overabundant answers problem and introduce suitable algorithms. We then consider the specificities of SPARQL with improved versions of our algorithms based on predicate cardinality as well as additional support for operators. Finally, the method is generalized to any unsatisfactory answer problem. We show how five elementary problems can be dealt with and how they can be combined to describe intricate problems. Our contributions are experimentally validated using synthetic data and queries from the WatDiv benchmark, and using real data and queries from DBpedia.

Keywords: Querying (Computer science) / Mathematical optimization / Question-answering systems / RDF (Document markup language) / SPARQL (Computer program language) / System analysis / Expert systems (Computer science) / Overabundant answers / Unsatisfactory answers / Failure causes