Conception d'un famework pour la relaxation des requêtes SPARQL

Géraud Fokou Pelap

THESE

Pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

(Diplôme National — Arrêté du 25 mai 2016)

Ecole Doctorale : Science et Ingénierie pour l'Information, Mathématiques (S2IM)
Secteur de Recherche : INFORMATIQUE ET APPLICATIONS

Présentée par :

Géraud FOKOU PELAP

****

CONCEPTION D'UN FRAMEWORK POUR LA RELAXATION DES REQUETES SPARQL

****

Directeurs de Thèse : **Allel HADJALI** et **Stéphane JEAN**

Soutenue le 21 novembre 2016
devant la Commission d'Examen

JURY

Présidente :	**Marie-Christine ROUSSET**	Professeur, Université de Grenoble, Grenoble
Rapporteurs :	**Nadine CULLOT**	Professeur, Université de Bourgogne, Dijon
	Farouk TOUMANI	Professeur, Université Blaise Pascal, Clermont-Ferrand
Examinateurs :	**Olivier CORBY**	Chargé de Recherche, INRIA, Sophia Antipolis
	François GOASDOUE	Professeur, Université de Rennes, Rennes
	Allel HADJALI	Professeur, ISAE-ENSMA, Poitiers
	Stéphane JEAN	Maître de Conférences, Université de Poitiers, Poitiers

THESE

pour l'obtention du Grade de

DOCTEUR DE L'ÉCOLE NATIONALE SUPÉRIEURE DE MÉCANIQUE ET D'AÉROTECHNIQUE

(Diplôme National — Arrêté du 25 mai 2016)

Ecole Doctorale : Science et Ingénierie pour l'Information, Mathématiques (S2IM)
Secteur de Recherche : INFORMATIQUE ET APPLICATIONS

Présentée par :

Géraud FOKOU PELAP

****

CONCEPTION D'UN FRAMEWORK POUR LA RELAXATION DES REQUETES SPARQL

****

Directeurs de Thèse : **Allel HADJALI** et **Stéphane JEAN**

Soutenue le 21 novembre 2016

devant la Commission d'Examen

JURY

Présidente :	**Marie-Christine ROUSSET**	Professeur, Université de Grenoble, Grenoble
Rapporteurs :	**Nadine CULLOT**	Professeur, Université de Bourgogne, Dijon
	Farouk TOUMANI	Professeur, Université Blaise Pascal, Clermont-Ferrand
Examinateurs :	**Olivier CORBY**	Chargé de Recherche, INRIA, Sophia Antipolis
	François GOASDOUE	Professeur, Université de Rennes, Rennes
	Allel HADJALI	Professeur, ISAE-ENSMA, Poitiers
	Stéphane JEAN	Maître de Conférences, Université de Poitiers, Poitiers

Remerciements

Au terme de ma thèse, je tiens à remercier :

— Mes directeurs de thèse **Allel Hadjali** et **Stéphane Jean**, pour m'avoir suivi jour après jour, guidé et soutenu durant ces trois années. Je les remercie pour le savoir et l'expertise dans la recherche et l'enseignement qu'ils m'ont transmis.

— Mes rapporteurs **Nadine Cullot** et **Farouk Toumani** pour avoir accepté cette lourde tâche, avoir pris le temps de comprendre ma thèse et avoir apporté des corrections pour l'améliorer.

— Mes examinateurs **Olivier Corby** et **François Goasdoué** d'avoir accepté d'être dans mon jury et de m'avoir enrichi d'autres points de vues sur mon travail à travers leurs questions.

— La président de Jury **Marie Christine Rousset** de m'avoir fait l'honneur d'accepter cette mission.

— **Mickael Baron**, pour l'encadrement, le suivi et les conseils dont j'ai bénéficié de sa part depuis mon arrivée au laboratoire en 2011. Également **Ladjel Bellatreche**, responsable de l'équipe IDD, pour ses conseils, son enthousiasme et sa passion de la recherche qu'il m'a transmis.

— **Claudine Rault** et **Frédéric Carreau** pour leur convivialité, leur générosité et toutes les tâches ingrates qu'ils ont réalisées pour moi. Leur aide a facilité mon travail pendant ces trois années.

— Tous les permanents du LIAS, **Brice Chardin**, **Emmanuel Grolleau**, **Laurent Guittet**, **Yassine Ouhamou**, **Michael Richard** et **Pascal Richard** pour leur convivialité et leurs précieux conseils.

— Tous mes amis doctorants qui m'ont accompagné, soutenu et grâce à qui ces trois années m'ont paru courtes : **Abdallah**, **Abdelkrim**, **Anh Toan**, **Aymen**, **Cyrille**, **Ibrahim**, **Lahcene**, **Nadir**, **Okba**, **Olga**, **Saida**, **Selma**, **Thomas**, **Yves**, **Zahira**, **Zouhir** et un merci spécial à **Guillaume**.

— **Le personnel administratif de L'ENSMA** pour l'assistance et les services dont j'ai bénéficié dans la convivialité et la bonne humeur.

— Mes amis qui m'ont aidé dès mon arrivé à Poitiers et durant tout mon séjour **Valery Teguiak**, **Georges Kemayo**, **Raoul Taffo**, **Gabriel Ngadou**, **Gabriel et Carole Kemje**.

— Mes grandes sœurs **Mawé Pelap**, **Meko Pelap**, **Komtchueng Pelap**. Les familles **KAMGA**, **FOTSI**, **MOMO**, **MBUDA**, **DJOKO**, **NGAPA** et ma famille, **la famille PELAP**.

— **L'Association des Étudiants et Stagiaires Camerounais de Poitiers (ASSECAM)** qui a facilité mon intégration et mon épanouissement dans le Grand Poitiers. Je remercie également **Eugénie Gaborieau**, **Steve Joumessi**, **Chancelline Kengho**, **Hortense keubou**, **Laure Mawoue**, **Linda Ugille**, **Hermann Sohtsinda** pour le soutien et les encouragements que vous m'avez apporté.

— Ceux qui sont partis sans voir l'aboutissement de ce travail pour lequel ils ont ardemment contribué, **Maman Pelap Clémentine**, **Maman Fotsi Jacqueline**, **Tonton Brian** et ma grand-mère **Maman Mengue** et ceux qui de près ou de loin ont contribué à la réalisation de ce travail.

— Mon fils **Louange Matys Leumeleu Fokou**, ma source inépuisable de motivation et de courage et à sa mère, ma future femme, **Danielle Lako** pour son soutien inestimable durant les moments difficiles de cette thèse.

— Mes parents, **Bernadette et Édouard Pelap** qui ont fait d'énormes sacrifices pour mon éducation, qui m'ont toujours montré la valeur du travail et appris le goût de l'effort. Ils m'ont constamment encouragé et donné l'amour et la confiance dont j'avais besoin à chaque fois pour aller de l'avant. À vous mes parents, je vous dis mille fois merci. Je remercie le seigneur de vous avoir gardé en vie et de m'avoir donné la force et l'opportunité de vous offrir cette fierté.

Merci à tout-e-s !

À mon père qui m'a appris la valeur du travail, le goût de l'effort et qui a toujours cru en moi.
À ma mère qui m'a montré la persévérance et la patience dans les épreuves de la vie et m'a toujours encouragé à aller de l'avant.

Pour qu'un enfant grandisse il faut tout un village.

Proverbe Africain

La connaissance parle, mais la sagesse écoute.

Proverbe Africain

Je ne perds jamais, soit je gagne, soit j'apprends.

Nelson Mandela

Table des Matières

Introduction

Contexte

Le Web sémantique a été créé, dans les années 90, afin de permettre aux utilisateurs de rechercher, partager et combiner les informations plus aisément et sans aucun intermédiaire, directement à travers le Web. Son créateur Tim Berners-Lee dans son livre Weaving the Web [1] disait à ce sujet : "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". Au cœur du Web sémantique se situe la notion d'*ontologie (ou base de connaissance (BC))* [2]. Une ontologie est une représentation formelle des connaissances sous la forme d'entités et de faits sur ces entités. Dans ces dernières années, de nombreux projets issus à la fois de l'industrie et de la recherche ont été menés pour concevoir des ontologies de très grandes tailles. Comme exemples bien connus d'ontologies, nous pouvons citer YAGO [3], DBpedia [4] et DBLP [5] qui résultent de projets académiques et les ontologies développées par Google [6] et Walmart [7] dans le contexte industriel. Proposant un ensemble d'informations sur un domaine d'étude, qui peuvent être traitées par machine, les ontologies sont utilisées dans de nombreux domaines de recherche comme l'intégration de données, l'expansion de requêtes, la traduction automatique de textes ou la conception de bases de données.

Les ontologies sont généralement représentées avec le langage RDF sous la forme de triplet *(sujet, predicat, objet)*. L'objet d'un triplet pouvant être le sujet d'un autre triplet, ces données peuvent être vues sous la forme d'un graphe. Un schéma peut être associé à ces données. Ce schéma est défini dans des langages plus ou moins expressifs tels que RDF Schema (RDFS) [8], Ontogy Web Language (OWL) : OWL Lite, OWL DL ou OWL Full [9]. Ces langages permettent de définir des classes et propriétés ainsi que des hiérarchies associées. Ils sont équipés de *règles de raisonnement* qui permettent de déduire de nouveaux triplets RDF à partir de ceux définis dans l'ontologie. Le langage standard d'interrogation des ontologies est SPARQL [10]. De manière simplifiée, SPARQL permet d'exprimer des requêtes sous la forme de conjonction de patron de triplets. Un patron de triplet est un triplet RDF pouvant contenir une ou plusieurs variables. Les réponses à un patron de triplet sont l'ensemble des triplets RDF présents dans l'ontologie qui sont conformes aux patrons de triplet. Les occurrences de la même variable dans plusieurs patrons de triplet indiquent une jointure. SPARQL fournit d'autres possibilités pour définir de requêtes telles que rendre optionnels certains patrons de triplet ou ajouter des filtres.

Problématique

Pour la plupart des utilisateurs, rechercher des informations dans une ontologie n'est pas une tâche facile. Cette difficulté est due aux trois principales raisons suivantes.

— Une ontologie est souvent construite automatiquement à partir d'un ensemble de sources de données hétérogènes. En conséquence, les utilisateurs ont souvent une connaissance partielle du contenu et de la structure d'une ontologie.
— Le schéma d'une ontologie est défini avec des langages basés sur les logiques de description dont la sémantique et les hypothèses sous-jacentes ne sont pas forcément bien comprises par les utilisateurs.
— Malgré leur taille très importante (par exemple, Knowledge Vault [6] de Google contient 1.6 milliard de triplets RDF, YAGO [3] décrit plus de 10 millions d'entités et 120 millions de faits sur ces entités et Freebase [1] comportait 1.9 milliard de triplets RDF), les ontologies sont incomplètes. Par exemple, nous avons observé que 60% des institutions éducatives dans l'ontologie Freebase n'ont pas de numéro de téléphone connu et le nombre de personnel enseignant est inconnu pour 95% d'entres elles [2].

Ces difficultés d'interrogation des ontologies peuvent conduire au problème suivant : alors que les utilisateurs attendent des résultats de qualité en réponse à leurs requêtes, ils n'en obtiennent aucun. Ce problème, appelée *problème des réponses vides* (*empty answer problem*), n'est pas anodin. Par exemple, une étude récente de plusieurs points d'accès SPARQL (des interfaces vers des ontologies) montrent que 10% des requêtes soumises entre mai et juin 2010 sur DBpedia ont retourné un résultat vide [11]. Confronté à ce problème, l'utilisateur ne peut pas savoir si sa requête est trop sélective, mal formulée ou si le résultat attendu n'est tout simplement pas présent dans l'ontologie. Il peut alors se lancer dans un processus d'essai et d'erreur durant lequel il essaie de modifier ou rendre *optionnelles* les conditions de la requête et étudie les résultats éventuellement obtenus, afin de trouver une reformulation lui permettant d'obtenir des résultats satisfaisants. Même dans le meilleur des cas, où l'utilisateur finit par trouver un résultat adéquat, ce processus peut être vu comme long et fastidieux et les résultats retournés peuvent ne pas être les plus proches de la requête initialement formulée. Nous notons aussi qu'il engendre souvent une frustration des utilisateurs puisque ces derniers ne comprennent pas pourquoi, avec un aussi grand volume de données, leurs requêtes ne retournent aucune réponse. Face à ce constat, nous pensons que les utilisateurs ont besoin d'assistance afin de retrouver des réponses répondants à leurs besoins même si, pour diverses raisons, la requête initialement formulée ne retourne pas de réponse. Pour assister les utilisateurs dans leur recherche, des approches d'interrogation dites *coopératives* [12] ont été développées, parmi lesquelles *la relaxation de requêtes* que nous abordons dans nos travaux. La relaxation de requêtes consiste à affaiblir les conditions exprimées dans les requêtes pour retourner des résultats à l'utilisateur. Ces techniques sont généralement associées à des mesures de similarité afin de préserver autant que possible l'intention de l'utilisateur et de retourner des résultats avec un degré de similarité par rapport à la requête utilisateur. La relaxation de requêtes s'appuie ainsi sur des requêtes dites *relaxées* qui sont les plus proches possibles de la requête utilisateur.

1. Fermée en 2015. Son contenu a été transféré à Wikidata
2. Valeur en février 2015

Motivation et contributions

Si la relaxation de requêtes a été largement étudiée dans le contexte des bases de données relationnelles, ce n'est pas le cas pour les données RDF. Pourtant, ce contexte apporte de nouvelles dimensions à prendre en compte.

- Le schéma d'une ontologie offre une sémantique plus riche qu'un schéma relationnel. Cette sémantique avec notamment les règles de raisonnement associées peuvent être utilisées pour relaxer une requête.
- La sémantique de SPARQL et de SQL déjà de part la différence des représentations, RDF et relationnelle, des données interrogées. Ces deux langages possèdent également des différences plus subtiles dans le traitement des requêtes. Par exemple, le traitement lors des jointures des résultats avec des attributs non renseignées c'est-à-dire possédant une valeur NULL [13].
- Les requêtes SPARQL sont décomposées en patron de triplets, ce qui fait que la taille d'une requête SPARQL peut être importante. Par exemple, des requêtes ayant jusqu'à 15 patrons de triplets ont été trouvées dans les journaux de requêtes de DBpedia [14].
- De même, la décomposition des données en triplets fait que la taille d'une ontologie peut être de millions voir de milliards de triplets. Ainsi, les approches conçues dans le contexte relationnel, pour des tables ayant des milliers de lignes et des requêtes ayant peu de conditions dans la clause WHERE, sont difficilement applicables en pratique dans le contexte RDF.

Les travaux proposant des approches de relaxation pour des données RDF, qui prennent en compte ces nouvelles dimensions, sont peu nombreux [15]. Ils ont introduit des opérateurs de relaxation basés sur la sémantique de RDFS (par exemple, la généralisation de patrons de triplet par l'utilisation des hiérarchies de classes et de propriétés) [16], sur des mesures de similarités [17] ou sur les préférences utilisateurs [18]. Ces opérateurs sont principalement utilisés pour obtenir les *TOP-K* réponses (*K* meilleures réponses, où *K* est un paramètre défini par l'utilisateur) [17]. En effet, ils sont appliqués à la requête initiale de l'utilisateur pour générer un ensemble de requêtes relaxées. Celles-ci sont ensuite triées, avec une mesure de similarité, puis exécutées par similarité décroissante (de la plus à la moins similaire). D'autres approches ont proposé une extension de SPARQL pour inclure des opérateurs de relaxation directement dans les requêtes [19, 20] ou des règles de réécriture de requêtes pour effectuer la relaxation [16].

Nous avons observé plusieurs limitations à ces travaux. Ces approches ne permettent généralement pas de définir précisément la relaxation à effectuer dans une requête SPARQL tout en permettant de contrôler le processus de relaxation qui est, par nature, itératif. Une autre limitation de ces approches est que les *causes d'échec* de la requête exprimée par l'utilisateur ne sont pas identifiées. En conséquence, le processus de relaxation se fait *à l'aveugle*, c'est-à-dire qu'il relaxe la requête sans savoir pourquoi la requête a échoué. L'intérêt d'identifier ces causes d'échec est double. D'une part, il permet d'éviter d'exécuter des requêtes relaxées qui contiennent encore une cause d'échec de la requête initiale puisqu'elles échoueront forcément. Ceci permet ainsi d'accélérer le processus de relaxation. D'autre part, l'utilisateur peut, à partir des causes d'échec, mieux comprendre la distribution des données dans l'ontologie et ainsi formuler des requêtes plus pertinentes. Enfin, nous notons que peu d'approches ont proposé des outils permettant aux utilisateurs de construire et éventuellement de relaxer leurs requêtes.

Pour répondre aux limitations précédentes, nos principales contributions, décrites dans cette thèse, sont les suivantes.

- La définition d'un ensemble d'opérateurs de relaxation des requêtes SPARQL, afin de relaxer des parties précises des requêtes, de façon incrémentale et en contrôlant la pertinence des réponses alternatives par rapport aux besoins exprimés par les utilisateurs dans leurs requêtes initiales.

— La conception d'algorithmes de recherche des causes d'échec d'une requête SPARQL qui échoue, afin de fournir aux utilisateurs des explications sur cet échec. Ces algorithmes retournent aussi les requêtes qui réussissent (c'est-à-dire, qui ont des résultats) et ont un nombre maximal de patron de triplet de la requête utilisateur. Ces requêtes peuvent être directement exécutées par l'utilisateur pour trouver des réponses alternatives à sa requête.
— La proposition de stratégies de relaxation de requêtes SPARQL qui se basent sur les causes d'échec de ces requêtes pour trouver des réponses alternatives. Ces stratégies réduisent le nombre de requêtes relaxées exécutées pendant le processus de relaxation par rapport à l'approche habituellement utilisée. En conséquence, elles réduisent le temps du processus de relaxation.
— Le développement d'un outil utilisant des éléments graphiques pour aider les utilisateurs dans la formulation des requêtes SPARQL et également pour les relaxer lorsqu'elles échouent.

Ces contributions ont été validées par des scénarios et expérimentations réalisées sur le banc d'essai LUBM [21], couramment utilisé dans les travaux sur RDF. Ils montrent l'impact de nos contributions par rapport à l'état de l'art.

Organisation du mémoire

Ce mémoire est organisé en six chapitres.

Le premier chapitre présente les technologies du Web sémantique utilisées dans ce mémoire. Nous y décrivons les ontologies, leurs rôles, leurs formalisations et présentons des exemples d'ontologies existantes et leur processus de construction. Nous décrivons également les langages RDF et RDFS de formalisation des ontologies, les règles et techniques de raisonnement. Une description des représentations des données RDF dans des systèmes de stockage est aussi réalisée. Ces systèmes sont divisés en deux grandes familles : la famille des systèmes *non natifs*, développés au-dessus des systèmes de gestion des bases de données relationnelles et la famille des systèmes *natifs* qui sont développés sans s'appuyer sur des systèmes de gestion de bases de données existants. Ce chapitre se termine par la description du langage d'interrogation de données RDF : le langage SPARQL.

Pour positionner la relaxation de requêtes parmi les approches d'interrogation coopératives, nous réalisons, dans le second chapitre, une synthèse des approches d'interrogation coopératives existantes. Nous présentons les différentes approches d'interrogation coopératives à savoir la suggestion des requêtes, l'interrogation par l'exemple, l'explication des réponses d'une requête, le raffinement des requêtes par les utilisateurs et la redéfinition des requêtes. Ce chapitre décrit les différentes techniques développées dans chaque approche d'interrogation coopérative et dans plusieurs types de bases de données : des bases de données relationnelles aux bases de données RDF. Ce chapitre décrit également les systèmes de recommandation et montre comment ils combinent plusieurs approches d'interrogation coopératives.

Afin d'analyser en détails les différentes contributions faites dans la redéfinition des requêtes et plus précisément dans la relaxation des requêtes, nous réalisons, dans le troisième chapitre, un état de l'art sur les techniques de relaxation développées dans différents contextes . Dans cet état de l'art, nous présentons tour à tour les approches de relaxation développées dans les bases de données relationnelles, les systèmes de recommandation et les bases de données RDF. Nous analysons les techniques de transformation de requêtes, les processus de relaxation implémentés et l'intégration de la relaxation dans des langages d'interrogation de données afin d'être accessible aux utilisateurs. Ce chapitre nous permet de mettre en évidence des limites des approches de relaxation des requêtes sur des données RDF, parmi lesquelles :

— l'absence d'opérateurs de relaxation fins et précis pour les requêtes SPARQL ;
— l'absence d'outils intuitifs et graphiques d'interrogation et de relaxation ;
— l'absence de processus de recherche des causes d'échec des requêtes SPARQL ;
— l'absence de processus de relaxation guidé par les causes d'échecs des requêtes SPARQL.

Dans le chapitre 4, nous développons notre contribution sur les opérateurs de relaxation et l'implémentation de l'outil graphique d'interrogation et de relaxation *QaRS*. Nous proposons dans ce chapitre trois opérateurs de relaxation : *PRED*, pour la relaxation de littéraux numériques ; *GEN*, pour la généralisation de concepts en se basant sur le schéma des données RDF et *SIB*, l'opérateur de substitution de concepts. Ces opérateurs se basent sur des mesures de similarité que nous définissons. Ils sont intégrés dans l'outil d'interrogation et de relaxation QaRS. L'outil QaRS offre également un processus automatique de relaxation basé sur ces opérateurs et fournit des explications aux utilisateurs sur les causes d'échec d'une requête à réponse vide.

Le chapitre 5 présente les algorithmes de recherche des causes d'échec d'une requête SPARQL qui ne retourne aucune réponse. Dans ce chapitre, nous décrivons deux algorithmes pour la recherche des causes d'échec des requêtes encore appelées *Minimal Failing Subquery (MFS)*. Le premier algorithme, textitLattice-Based Approach (LBA), est basé sur le treillis des sous-requêtes de la requête qui échoue et le second, *Matrice-Based Approach (MBA)*, sur la matrice des réponses de chaque patron de triplet de la requête SPARQL . Nous avons également implémenté des heuristiques pour optimiser ces algorithmes et avons mené une évaluation des performances de ces algorithmes sur le banc d'essai LUBM.

Le chapitre 6 développe des stratégies de relaxation de requêtes SPARQL en s'appuyant sur les causes d'échec de ces requêtes. Dans ce chapitre, nous proposons trois stratégies qui exploitent, à différents niveaux, les causes d'échec des requêtes à réponse vide. La première stratégie, *MFS-Based Search (MBS)*, est basée uniquement sur les MFS de la requête initiale qui échoue. La seconde, *Optimized-MBS (O-MBS)*, vérifie si les MFS relaxées retournent des réponses ou échouent également, auquel cas elles sont aussi des MFS. La dernière stratégie, *FULL MBS (F-MBS)*, recherche toutes les MFS dans les requêtes qui, malgré la relaxation, continuent à échouer. Nous avons également mené une évaluation de ces algorithmes et une comparaison de ces algorithmes avec ceux existants.

Ce mémoire comprend également trois annexes. La première liste les requêtes utilisées pour l'évaluation des opérateurs de relaxation étudiés dans le chapitre 3 durant nos expérimentations. La seconde annexe donne les algorithmes complets des stratégies de relaxation O-MBS et F-MBS. La dernière annexe liste les publications relatives aux contributions présentées dans ce mémoire.

Chapitre 1

Ontologies : représentation, stockage et interrogation

Sommaire

Résumé : Dans cette thèse, nous abordons le problème des requêtes portant sur des ontologies (ou bases de connaissances) qui retournent une réponse vide considérée comme insatisfaisante par l'utilisateur. Il est donc nécessaire de présenter les notions sous-jacentes à cette problématique c'est-à-dire les ontologies, les langages de représentation associés, le stockage d'ontologies dans des bases de données et l'interrogation de ces dernières via des requêtes dans des langages dédiés. Cette présentation n'est pas exhaustive mais ciblée sur des notions qui interviennent dans la suite de notre thèse et dont la connaissance est une condition préalable pour la compréhension de notre travail.

Introduction

Nos travaux visent à proposer des solutions pour éviter la frustration de l'utilisateur lorsque ses requêtes, portant sur des ontologies (aussi appelées bases de connaissances) retournent des réponses vides. Plusieurs notions importantes sont sous-jacentes à cette problématique. L'objet de ce chapitre est la présentation de ces notions dont la compréhension est impérative pour la suite. Nous commençons donc par présenter la notion d'ontologie. En effet, cette notion ayant été abordée par plusieurs communautés de recherche, plusieurs définitions sont apparues. Néanmoins, pour toutes ces communautés les ontologies permettent de modéliser explicitement les aspects structurels et descriptifs des concepts d'un domaine à travers des modèles consensuels. Nous précisons ainsi dans la section 1.2 la définition que nous retenons dans nos travaux et donnons quelques exemples d'ontologies qui ont été développées dans différents domaines.

Afin de pouvoir exploiter entièrement les capacités des ontologies, tant les capacités descriptives que structurelles, les ontologies sont représentées à l'aide de langages dédiés dits formels. Ces dernières années, plusieurs langages ont été développés pour la représentation formelle des ontologies. Par exemple, dans le contexte du Web sémantique, les langages Resource Description Framework (RDF) [22], RDF Schema (RDFS) [23] et OWL [24] ont été définis et standardisés par le World Wide Web Consortium (W3C). Dans le contexte de l'ingénierie, le modèle PLIB [25] fait figure de référence et permet la standardisation du domaine via la norme ISO 13584 [26]. Ces langages permettent une représentation consensuelle et explicite des ontologies dans ces différents contextes d'application. Nos contributions traitent des ontologies définies dans les langages RDF et RDFS car ce sont des langages simples et fortement utilisés pour la définition d'ontologies sur le Web [27]. Ces deux langages sont donc présentés en détails dans la section 1.3.

Certaines ontologies disponibles sur le Web ont été construites automatiquement à partir de données structurées ou non, publiées sur Internet. Ces ontologies constituent un volume de données important. Par exemple, l'ontologie Knowledge Vault [6] construite par Google extrait, structure et stocke 1.6 milliards de faits provenant de sources de données plus ou moins structurées (pages web, bases de données, etc.). La gestion en mémoire centrale d'un tel volume de données posant des problèmes de performance, des bases de données spécifiques, appelées *bases de données RDF (ou triple stores)*, ont donc été conçues pour remédier à ce problème. Nous décrivons dans la section 1.4 ces bases de données RDF et présentons deux exemples que nous avons utilisés pour nos expérimentations. Ces dernières ont été réalisées sur des volumes importants de données afin d'être le plus proche possible des cas réels d'exploitation de nos contributions.

Enfin, pour exploiter les ontologies stockées dans les bases de données RDF, plusieurs langages d'interrogation d'ontologies ont été proposés. Parmi ces langages nous avons le langage SPARQL [10] qui est un standard du W3C et le langage OntoQL [28] développé dans notre laboratoire. Les bases de données RDF sont généralement interrogées avec le langage SPARQL. Dans nos travaux, nous considérons des requêtes écrites avec un sous-ensemble d'opérateurs de ce langage présenté dans la section 1.5.

1.1 Notion d'ontologie

La notion d'ontologie trouve son origine dans une branche de la philosophie traitant des sciences de l'être. Cette discipline philosophique, initiée par Aristote, essaye de définir l'être à travers ce qui le caractérise de façon essentielle. Le terme *ontologie* n'apparaît qu'en 1962, emprunté au latin scientifique

ontologia. Ce terme a été introduit en informatique dans les années 70 par McCarthy dans le domaine de l'intelligence artificielle [29]. Il a ensuite été repris dans le domaine de la représentation des connaissances dans les années 90. De nos jours, le nombre de domaines dans lesquels le concept d'ontologie est utilisé a encore augmenté, incluant la fouille des données, le Web sémantique, l'intégration des données, le traitement des langages naturels etc. Chacun de ces domaines d'utilisation des ontologies a sa propre interprétation du terme *ontologie*. Dans cette section nous présentons la notion d'ontologie telle que nous la définissons dans nos travaux.

1.1.1 Définition et caractéristiques d'une ontologie

Les ontologies ont été conçues pour formaliser les connaissances d'un domaine donné. Une *ontologie* est définie par Gruber comme une spécification explicite d'une conceptualisation [30]. Cette définition a été précisée par Jean et al. [28], en caractérisant une ontologie comme étant une représentation *formelle*, *consensuelle* et *référençable* de l'ensemble des concepts partagés d'un domaine. Ces caractéristiques sont définies comme suit [28] :

— *formelle* : une ontologie est une conceptualisation basée sur une sémantique formelle qui permet d'en vérifier la consistance et/ou de réaliser des raisonnements et déductions à partir de ses concepts et de ses instances ;
— *consensuelle* : une ontologie est une conceptualisation acceptée par une communauté qui peut être plus ou moins large. Elle n'est ainsi pas conçue spécifiquement pour un système particulier. Au contraire, deux systèmes développés au sein de la même communauté et portant sur le même domaine d'application peuvent être basés sur la même ontologie ;
— *référençable* : une ontologie associe à chacun de ses concepts un identifiant unique permettant de le référencer à partir de n'importe quel environnement, indépendamment de l'ontologie dans laquelle il a été défini.

1.1.2 Exemples d'ontologies

Les ontologies sont utilisées pour de nombreux problèmes informatiques tels que l'intégration de base de données [31], l'échange de données [32], le traitement de la langue naturelle [33] ou la recherche d'information [34, 35]. De nombreuses ontologies ont ainsi été développées ces dernières années dans des domaines variés.

Dans le domaine du Web sémantique, nous trouvons par exemple l'ontologie Yago [3]. Cette ontologie a été créée automatiquement par extraction des données dans Wikipédia et Wordnet. Yago est décrite formellement dans le langage OWL [36]. Bien que construite automatiquement, l'ontologie Yago reste consensuelle du fait du caractère consensuel de ses sources Wikipedia et Wordnet. Il en est de même pour l'ontologie DBpedia [4], créée automatiquement à partir de données extraites de Wikipédia et formalisée dans le langage OWL. Dans le domaine du Web sémantique, nous pouvons également citer Knowledge vault [6] qui est une ontologie créée de façon automatique en se basant sur les ontologies déjà existantes et d'autres sources de données consensuelles, comme Wikipédia, qui sont fusionnées à l'aide de distribution de probabilité.

Dans le domaine de l'ingénierie, l'ontologie IEC61360-4 [26] a été définie pour représenter des composants électriques et électroniques. Elle a été créée manuellement par une commission d'experts et approuvée par l'organisation internationale de la normalisation (ISO) d'où son caractère consensuel. Elle a été formalisée dans langage le PLIB [28].

Dans le domaine de la biologie, nous trouvons l'ontologie universelle des protéines [3] (UniProt) [37] créée manuellement et de façon consensuelle par des experts du domaine des protéines, à savoir European Bioinformatics Institute (EMBL-EBI), Swiss Institute of Bioinformatics (SIB) et Protein Information Resource (PIR). Elle est formalisée dans le langage OWL et maintenue par environ 100 experts faisant partie du groupe de travail nommé Uniprot Consortium. Dans le même domaine, nous retrouvons également l'ontologie d'annotation des gènes (GOA) [38].

Pour des besoins de tests et d'expérimentation, des ontologies ont également été conçues dans le cadre de la définition de *bancs d'essais* (ou *benchmark*). Par exemple, le banc d'essai LUBM [21] définit une ontologie OWL sur le domaine universitaire. Ce banc d'essai étant utilisé pour évaluer nos propositions, nous utilisons par la suite un exemple d'ontologie portant sur ce domaine et dans ce document nous utiliserons des illustrations basées sur ce banc d'essai.

Comme nous le voyons dans ces exemples, ils existent plusieurs langages de définitions des ontologies. Nous présentons ceux utilisés dans nos travaux dans la section suivante.

1.2 Langages de définition des ontologies

Pour définir une ontologie, plusieurs langages sont disponibles tels que RDF [22], RDFS [23], OWL [24] ou PLIB [39]. Dans cette section, comme nos travaux portent sur des ontologies définies en RDF et RDFS, nous présentons ces deux langages.

1.2.1 Langage RDF

Le langage RDF est le premier langage standard de représentation de connaissances dont le premier draft a été publié en 1997 par le W3C [40]. En 1999, la même organisation publie une nouvelle version qui sera adoptée comme standard et recommandation du W3C. Il est maintenu et mis à jour par le W3C. La dernière version connue est RDF 1.1 [41]. Ce langage a été créé pour représenter les données, les échanger sur le web et faciliter l'intégration des données entre différentes sources, même si elles sont hétérogènes (de schémas différents) [42]. Sa structure est issue des réseaux sémantiques introduits dans le domaine de la représentation des connaissances [43]. RDF représente la connaissance sous forme de *statements* (ou *assertions* en français) dont la structure est un *triplet* défini dans la section suivante.

1.2.1.1 Triplet RDF

Une *assertion* RDF est un triplet ayant la forme *(sujet, prédicat, objet)* où :

— le *sujet* est une ressource identifiée de façon unique à l'aide d'un URI (*Uniform Resource Identifier*). Une ressource peut être un objet, une personne, une page web ou même un lieu, tous identifiés avec un URI. Par exemple dans l'ontologie LUBM, la ressource *Professor* a pour URI *http://swat.cse.lehigh.edu/onto/univ-bench.owl#Professor*.
Pour simplifier l'écriture des URI, le mécanisme *des espaces de noms* (ou *namespaces*) est utilisé. Ainsi l'URI de la ressource précédente peut s'écrire *uba:Professor* où *uba* est l'espace de nom *http://swat.cse.lehigh.edu/onto/univ-bench.owl#*. Par soucis de lisibilité, nous omettons l'espace de noms *uba* dans la suite de ce mémoire. Ainsi, *uba:Professor* sera désigné par *Professor* ;

3. `http://www.uniprot.org/`

— le *prédicat* est une propriété permettant de caractériser une ressource. Dans l'ontologie LUBM nous trouvons par exemple le prédicat *worksFor* qui décrit la ressource *Professor*. Un prédicat est également identifié par une URI, *http://swat.cse.lehigh.edu/onto/univ-bench.owl#worksFor* ;

— l'*objet* est une valeur prise par une propriété pour une ressource donnée. Cette valeur peut être une valeur primitive (entier, réel, chaine de caractères, ...) appelée *littéral*, éventuellement typée par les types primitifs définis par XML schéma tels que le type entier défini par *xsd:integer*[4] ou le type date défini par *xsd:date*. La valeur de l'objet peut aussi être une autre ressource identifiée par son URI. Par exemple, la valeur du prédicat *worksFor* peut être la ressource *university*$_1$ ayant comme URI, *http://www.University1.edu*.

Notons également la possibilité d'utiliser des *nœuds blancs* pour représenter des ressources non spécifiés. Par exemple, on peut utiliser un nœud blanc pour modéliser qu'un professeur travaille pour une université donnée sans avoir à préciser l'URI de cette personne.

1.2.1.2 Graphe RDF

L'objet d'un triplet RDF pouvant être le sujet d'un autre triplet, les données RDF peuvent se représenter sous la forme d'un graphe. La figure 1.1 illustre un graphe RDF portant sur le domaine universitaire. Dans ce graphe, les sujets et objets des triplets RDF sont des nœuds et les prédicats sont des arcs. Cette figure est ainsi composée des trois triplets RDF suivants : *(fullProfessor*$_1$*, work_for, university*$_1$*)*, *(undergraduateStudent*$_1$*, member_of, university*$_1$*)* et *(graduateStudent*$_1$*, member_of, university*$_1$*)*.

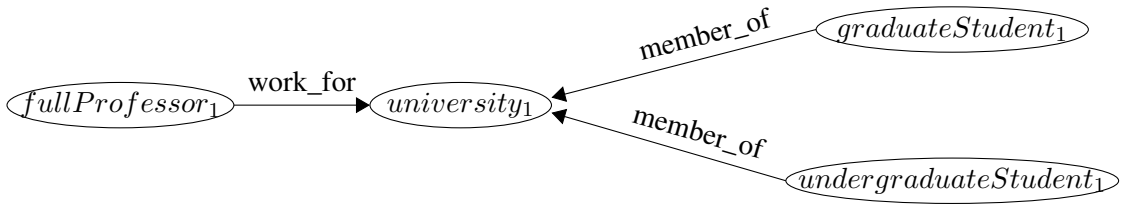


FIGURE 1.1 – Graphe de données RDF

RDF n'introduit que très peu de ressources prédéfinies pour définir le schéma d'une ontologie. Il permet principalement de créer des propriétés à l'aide de la ressource *rdf:Property*[5] et de typer les ressources avec *rdf:type*. Par exemple, le triplet *(work_for, rdf:type, rdf:Property)* permet de spécifier que *work_for* est une propriété de l'ontologie. RDF ne permet ainsi pas de créer la hiérarchie de classes d'une ontologie. Il a donc été étendu par le langage *RDF Schema (RDFS)*.

1.2.2 Langage RDFS

Le langage RDFS [8] étend RDF pour permettre de définir plus précisément le schéma d'une ontologie. Il introduit également des règles d'inférences permettant de déduire de nouveaux triplets RDF à partir de ceux définis dans l'ontologie. Le langage RDFS est une technologie issue du Web sémantique offrant des constructeurs de modélisation orientés objets pour la définition des concepts. Il permet de bâtir des concepts qui seront éventuellement définis à partir d'autres et pourront être partagés via leurs identifiants.

4. xsd est l'espace de noms pour XML Schema. xsd=http://www.w3.org/2001/XMLSchema#
5. *rdf* est l'espace de noms contenant les ressources prédéfinies de RDF.

1.2.2.1 Définition du schéma d'une ontologie en RDFS

Les principales ressources prédéfinies de RDFS permettant de définir le schéma d'une ontologie sont :

— *rdfs:Class* [6] et *rdfs:subClassOf* permettent respectivement de définir des classes et des relations de subsomption entre ces classes. La subsomption d'une classe par plusieurs (héritage multiple) est permis par *rdfs:subClassOf*. Par exemple, le triplet *(Professor, rdf:type, rdfs:Class)* indique que *Professor* est une classe. La création du triplet *(Professor, rdfs:subClassOf, Employee)*, signifie qu'entre la classe *Professor* et *Employee*, il existe une relation de subsomption dans laquelle la classe *Employee* subsume la classe *Professor*, *Professor* est une sous-classe de *Employee*.

— *rdfs:domain* et *rdfs:range* permettent respectivement de définir le *domaine* et le *codomaine* d'une propriété RDF (*rdf:Property*). Ces deux propriétés permettent de spécifier le type, d'une part, de l'ensemble des ressources qui peuvent être décrites par une propriété (*domaine*) et, d'autre part, de l'ensemble des valeurs que peut prendre une propriété donnée (*codomaine*). Par exemple, les triplets *(work_for, rdfs:domain, Employee)* et *(work_for, rdfs:range, University)* définissent le domaine et le codomaine de *work_for* comme étant respectivement *Employee* et *University*.

— *rdfs:subPropertyOf*, de manière similaire à *rdfs:subClassOf*, permet de définir une relation de spécialisation entre les propriétés. Il autorise également l'héritage multiple. Par exemple le triplet, *(work_for, rdfs:subPropertyOf, member_of)* permet d'indiquer que la propriété *work_for* est une sous-propriété de *member_of*.

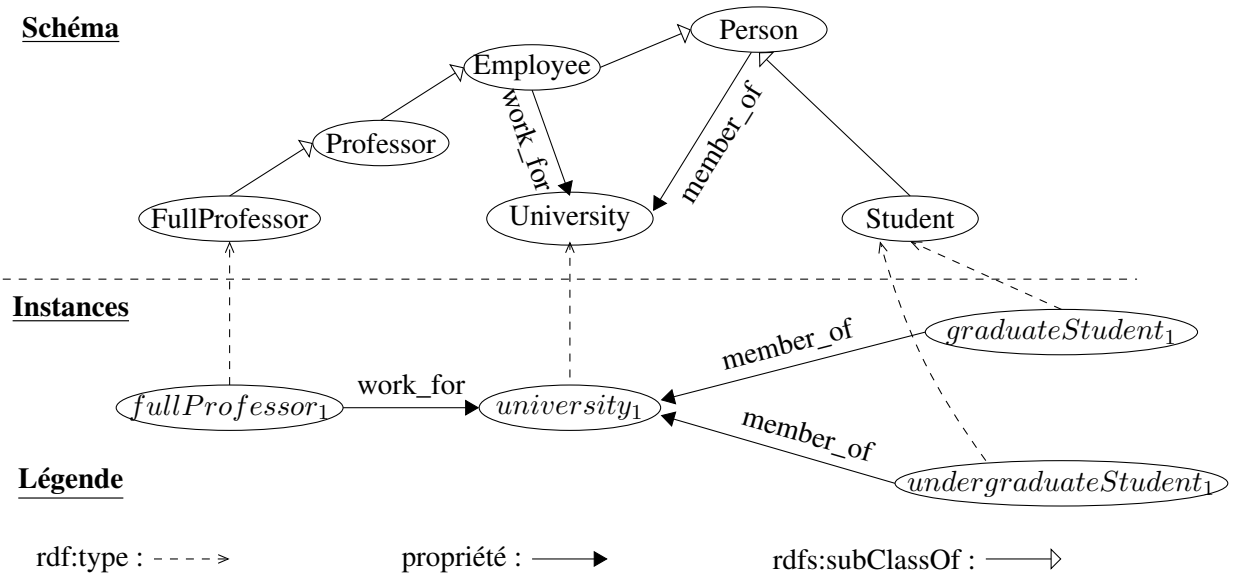


FIGURE 1.2 – Graphe RDF des données et du schéma RDFS associé

Grâce aux possibilités offertes par RDFS, l'exemple précédemment donné en figure 1.1 peut être complété en définissant le schéma de l'ontologie. La figure 1.2 présente ainsi des instances et le schéma d'une ontologie sur le domaine universitaire. Le schéma décrit une hiérarchie de classes ($FullProfessor \rightarrow Professor \rightarrow Employee \rightarrow Person$) et permet de typer les instances de l'ontologie (par exemple, $university_1$ est une instance de la classe *University*).

6. Comme pour l'espace de nom *rdf*, *rdfs* est l'espace de noms contenant les ressources prédéfinies de RDFS.

1.2.2.2 Raisonnement sur une ontologie

Une des caractéristiques du langage RDFS est sa capacité d'inférer de nouveaux triplets RDF à partir de ceux définis dans une ontologie. Ces capacités de raisonnement sont définies par des formules logiques appelées *règles d'inférence*. Ces règles sont des implications logiques du type : *"si hypothèses alors conclusion"* [44]. Lorsque les hypothèses de ces règles sont vérifiées, on peut déduire des conclusions valides qui sont de nouveaux triplets RDF. le tableau 1.1 donne quelques règles d'inférences prédéfinies pour le langage RDFS. La liste complète de ces règles a été définie par Hayes et al. [45].

	$Hypothèses \Longrightarrow Conclusion$
rdf_2	$(s,\ p,\ o), (p,\ rdfs:domain,\ u) \Longrightarrow (s,\ rdf:type,\ u)$
rdf_3	$(s,\ p,\ o), (p,\ rdfs:range,\ u) \Longrightarrow (o,\ rdf:type,\ u)$
rdf_5	$(p_1,\ rdfs:subPropertyOf,\ p_2), (p_2,\ rdfs:subPropertyOf,\ p_3)$ $\Longrightarrow (p_1,\ rdfs:subPropertyOf,\ p_3)$
rdf_7	$(s,\ p_1,\ o), (p_1,\ rdfs:subPropertyOf,\ p_2) \Longrightarrow (s,\ p_2,\ o)$
rdf_9	$(s,\ rdf:type,\ c_1), (c_1,\ rdfs:subClassOf,\ c_2) \Longrightarrow (s,\ rdf:type,\ c_2)$
rdf_11	$(c_1,\ rdfs:subClassOf,\ c_2), (c_2,\ rdfs:subClassOf,\ c_3)$ $\Longrightarrow (c_1,\ rdfs:subClassOf,\ c_3)$

TABLE 1.1 – Exemples de règles d'inférences RDFS [45]

Considérons l'ontologie et les données représentées en RDFS à la figure 1.2. Prenons les triplets *(Professor, rdfs:subClassOf, Employee)* et *(fullProfessor$_1$, rdf:type, Professor)*, par application de la règle d'inférence rdf_9 le triplet *(fullProfessor$_1$, rdf:type, Employee)* est déduit.

Des langages plus expressifs que RDFS existent comme OWL. Le langage OWL (Ontology Web Language) [9, 46] permet par exemple de définir une classe comme étant l'union ou l'intersection de plusieurs autres classes. Cette expressivité s'accompagne de nouvelles règles d'inférence. Pour simplifier le problème abordé dans cette thèse, c'est-à-dire la relaxation de requêtes exprimées sur des ontologies, nous avons fait le choix de considérer des ontologies définies dans le langage RDFS.

1.3 Base de données RDF (BD-RDF)

Les langages de définition des ontologies permettent une description textuelle des données dans une syntaxe comme XML, proche du langage humain. Cette description textuelle est ensuite sauvegardée dans des fichiers. Mais, avec le volume important des données à sauvegarder, ces fichiers deviennent rapidement illisibles pour les utilisateurs et difficilement exploitables. C'est dans ce contexte que les bases de données RDF (BD-RDF), ou *triplestores*, sont apparues, inspirées par l'exemple des bases de données relationnelles. Par abus de langage, on utilise le terme bases de données RDF pour désigner le système qui gère effectivement les données RDF à travers des bases de données. Parmi les objectifs des BD-RDF, nous avons le stockage des ontologies et l'exploitation des données RDF via des langages de requêtes. On distingue deux grands groupes de BD-RDF, les BD-RDF natives et les BD-RDF non natives. Nous présentons ces deux types de BD-RDF en montrant leurs différences.

1.3.1 Bases de données RDF non natives

La principale caractéristique des bases de données RDF non natives est leur architecture. Ces bases de données RDF sont implémentées au-dessus d'un système de gestion de bases de données relationnelles

(SGBDR), c'est pour cette raison qu'elles sont qualifiées de non natives. Le but de cette architecture est de capitaliser les travaux et techniques existants sur les SGBDR. Ainsi, les BD-RDF non natives permettent de profiter de l'efficacité des SGBDR pour exploiter les données RDF [47]. Il existe trois principaux types de stockage pour les BD-RDF : *la représentation verticale*, *la représentation binaire* et *la représentation horizontale*. Ces représentations possèdent des propriétés spécifiques et implémentent des techniques de stockage différentes. Les performances de ces représentations peuvent varier en fonction des données et des requêtes utilisées sur ces données [48].

1.3.1.1 Représentation verticale

Dans la représentation verticale, les données sont stockées dans une unique *table de trois colonnes* correspondantes chacune au *sujet, prédicat et objet* dans cet ordre. Cette table permet ainsi de stocker des triplets RDF. Pour interroger ces données, les requêtes sont traduites en SQL, le langage standard d'interrogation des bases de données relationnelles. Dans l'optique d'optimiser la traduction d'un langage de requête RDF tel que SPARQL vers le langage SQL [49, 50], d'autres tables auxiliaires peuvent être créées suivant les BD-RDF non natives [51, 52]. Les BD-RDF non natives utilisent aussi des outils d'optimisation de requêtes, tels que les index et les vues matérialisées, pour accélérer l'exécution des requêtes SQL [53]. L'usage de ces outils d'optimisation est un critère important de choix lorsque l'on doit sélectionner une BD-RDF particulière pour un problème donné [53].

Triplets		
Sujet	**Prédicat**	**Objet**
Person	*rdf:type*	*rdfs:Class*
Employee	*rdf:type*	*rdfs:Class*
Professor	*rdf:type*	*rdfs:Class*
Employee	*rdfs:subClassOf*	Person
Professor	*rdfs:subClassOf*	Employee
University	*rdf:type*	*rdfs:Class*
work_for	*rdf:type*	*rdf:Property*
work_for	*rdfs:domain*	Employee
work_for	*rdfs:range*	University
university_1	*rdf:type*	University
fullProf_1	*rdf:type*	Professor
fullProf_1	work_for	university_1

FIGURE 1.3 – Stockage des données RDF dans une table de triplets

Par exemple, la figure 1.3 donne un aperçu du stockage de l'ontologie de la figure 1.2 dans une table de triplets. Des exemples de BD-RDF utilisant cette représentation sont 3store [54] et RDFStore [55].

1.3.1.2 Représentation binaire.

La représentation binaire fait correspondre chaque prédicat à une table de deux colonnes, *(sujet, objet)*, qui contient les sujets ayant au moins une valeur pour ce prédicat et pour chaque sujet, le ou les objets correspondants pour ce prédicat. La figure 1.4 présente la représentation binaire des données RDF de la figure 1.2. Dans cette représentation une table est créée pour chaque propriété. Dans ces tables sont

stockées les valeurs des propriétés (*l'objet*) pour chaque ressource décrite (*sujet*). Des exemples de BD-RDF utilisant cette représentation sont *Jena SDB* [55], qui peut être couplé avec les SGBDR PostgreSQL ou MySQL, et 4Store [56].

rdf:type	
Sujet	**Objet**
Person	*rdfs:Class*
Employee	*rdfs:Class*
Professor	*rdfs:Class*
FullProfessor	*rdfs:Class*
Student	*rdfs:Class*
University	*rdfs:Class*
work_for	*rdf:Property*
university_1	University
fullProf_1	FullProfessor

rdfs:subClassOf	
Sujet	**Objet**
Employee	Person
Student	Person
Professor	Employee
FullProfessor	Professor

rdfs:Domain	
Sujet	**Objet**
work_for	Employee
member_of	Person

rdf:work_for	
Sujet	**Objet**
fullProf_1	university_1

rdfs:Range	
Sujet	**Objet**
work_for	University
member_of	University

FIGURE 1.4 – Représentation binaire des BD-RDF

1.3.1.3 Représentation horizontale

La représentation horizontale associe à chaque classe une table ayant pour colonnes les propriétés de la table. Cette représentation utilise le constructeur *rdfs:Class* pour créer les tables associées aux différentes classes. Le constructeur *rdfs:Domain* est utilisé pour identifier les propriétés des classes qui serviront à créer les colonnes des tables correspondantes à chaque classe. Et enfin, le constructeur *rdf:type* permet d'identifier les instances de chaque classe et de les insérer dans la table correspondante.

University
Sujet
university_1

FullProfessor	
Sujet	**work_for**
fullProf_1	university_1

Student	
Sujet	**member_of**
graduate_1	university_1
underGraduate_1	null

FIGURE 1.5 – Représentation horizontale des BD-RDF

Dans l'exemple présenté à la figure 1.5, nous montrons la représentation horizontale des données RDF du graphe de la figure 1.2. Pour chaque classe, une table est créée avec les propriétés de cette classe comme colonnes ainsi que la colonne *sujet* contenant l'URI de l'instance décrite. Des colonnes sont aussi créées pour les propriétés héritées. Lorsqu'une instance d'une classe n'affecte pas de valeur à une propriété de la classe, la valeur prédéfinie *Null* ou *UNBOUND* est affectée. C'est le cas de l'instance $graduate_1$ qui ne donne pas de valeur à la propriété *member_of*. Lorsqu'une instance de classe peut affecter plusieurs valeurs à une même propriété, on parle de *propriétés multivaluées*, les valeurs de cette propriété sont stockées dans une autre table. Cette table possède la structure des tables de la représentation binaire c'est-à-dire deux colonnes : *sujet* (URI de l'instance) et *objet* (valeur de la propriété). OntoDB [57, 58] est un exemple de BD-RDF utilisant la représentation horizontale.

Exemple de BD-RDF non native utilisée dans nos travaux : Virtuoso. Virtuoso est un framework multi-protocole, qui permet des accès ODBC, JDBC et HTTP, programmé dans le langage C. Ce fra-

mework a la capacité de stocker des données relationnelles et des données RDF. Pour les données RDF, Virtuoso peut les stocker en non natif à l'aide de tables relationnelles [59] ou en natif avec plusieurs index [55, 60, 47]. Le langage SPARQL a été implémenté au dessus du langage SQL. Virtuoso dispose d'un service HTTP d'interrogation des données RDF par des requêtes SPARQL.

1.3.2 Bases de données RDF natives

Les BD-RDF natives proposent un stockage spécifique qui ne repose pas sur un SGBDR. Elles sont regroupées en deux catégories :

1. la catégorie *in-memory* dans laquelle l'ontologie est stockée en mémoire centrale dès le lancement de l'application ;
2. la catégorie *non-memory* dans laquelle l'ontologie et les données sont stockées sur disque.

La représentation des données dans les BD-RDF natives diffère d'une BD-RDF à une autre. Cette représentation est, en effet, interne et propre à chaque BD-RDF native. Nous pouvons néanmoins les caractériser à partir des structures de données utilisées. Nous avons ainsi une famille de BD-RDF natives dont la représentation des données utilise des index et des structures de données de type arbre pour représenter les données. La BD-RDF RDF-3X [61] utilise cette représentation pour le stockage des données RDF. Nous pouvons aussi noter d'autres exemples comme Hexastore [55], istore [55] ou IBM DB2RDF [47]. Une autre famille de BD-RDF native utilise des structures de données basées sur les graphes afin de représenter les données définies en RDF [47]. C'est le cas de la BD-RDF gStore [47].

Exemple de BD-RDF native utilisée dans nos travaux : Jena TDB. Jena TDB est une BD-RDF native qui stocke les données sémantiques à l'aide d'index et de structures de données de type arbre [60]. Elle est implémentée en Java. Elle charge le schéma des données en mémoire dans un objet de type *Model* et intègre un moteur de raisonnement avec des règles d'inférences prédéfinies ou personnalisées par l'utilisateur. Jena TDB est disponible sous plusieurs formes : comme application standalone utilisée à l'aide de commande DOS , comme une API Java et également sous forme de serveur HTTP appelé Fuseki pour les requêtes SPARQL. Les requêtes sous Jena TDB se font en SPARQL ou en ARQ, le langage de requête interne propre à Jena.

Comme nous l'avons vu précédemment (section 1.2.2.2), RDFS permet de raisonner sur les données RDF. Ainsi, certaines BD-RDF embarquent un *moteur de raisonnement* pour réaliser ce raisonnement. La section suivante présente les techniques utilisées pour faire le raisonnement sur les BD-RDF.

1.3.3 Moteurs de raisonnement et saturation des données

Les moteurs de raisonnement ou *raisonneurs* ou *encore moteurs d'inférence* sont des applications qui permettent de réaliser des inférences sur un ensemble de données RDF. Pour un ensemble de données RDF, ces moteurs appliquent les règles d'inférences données afin de générer d'autres données sous forme de triplets ou de vérifier si l'ensemble de données est valide pour ces règles. La quasi-totalité des BD-RDF natives possèdent un *moteur de raisonnement*. Les moteurs de raisonnement utilisent deux principales techniques : *le chainage avant* et le *chainage arrière*. Ces processus de raisonnement sont très coûteux en temps suivant la taille des données à inférer et les règles d'inférence utilisées. À titre d'exemple, le problème de raisonnement utilisant les règles d'inférences prédéfinies pour le modèle RDFS est décidable [7] mais est de complexité NP-Hard [44, 9]. Ce raisonnement peut être fait avant l'exécution des requêtes,

7. Problèmes dont il existe un algorithme qui trouve une solution en un temps fini.

c'est *la saturation des données* ou au moment de l'interrogation des données, c'est *la reformulation des requêtes*.

1.3.3.1 Saturation des données.

La saturation d'un ensemble de données RDF $\mathbb{D}$ par un ensemble de règles d'inférence $\mathbb{R}$ est la production de tous les triplets possibles par application de toutes les règles d'inférence incluses dans $\mathbb{R}$. Elle se définit aussi comme la fermeture transitive de $\mathbb{R}$ sur $\mathbb{D}$. La production de ces triplets, appelés *triplets inférés*, se fait par exécution récursive des règles d'inférences sur les données RDF par un moteur d'inférence. Ces triplets inférés seront rajoutés dans l'ensemble des données $\mathbb{D}$, nous dirons alors que l'ensemble des données RDF $\mathbb{D}$ est saturé.

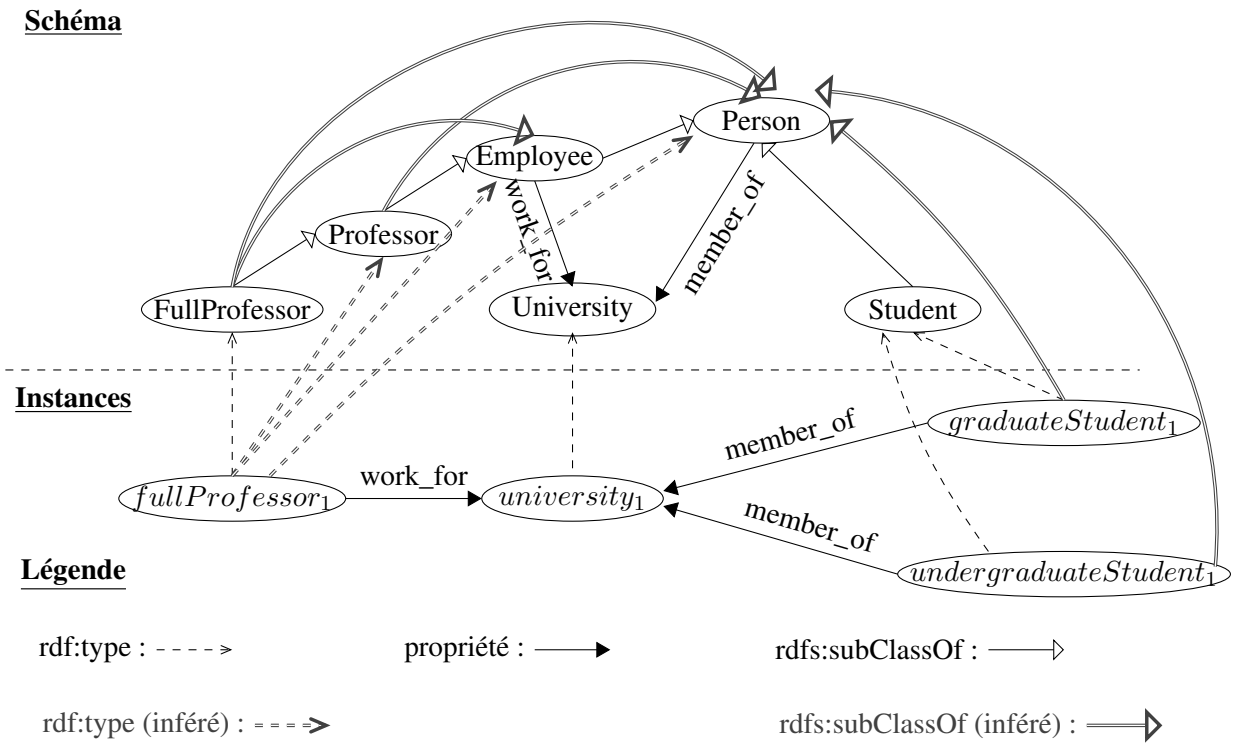


FIGURE 1.6 – Graphe RDF des données RDF saturées

Considérons l'ensemble des données RDF $\mathbb{D}$ présentées sur la figure 1.2 et $\mathbb{R}$ l'ensemble des règles du tableau 1.1. Avec la règle rdf_9 et les triplets *(Professor, rdfs:subClassOf, Employee)* et *(fullProfessor$_1$, rdf:type, Professor)*, nous obtenons le triplet *(fullProfessor$_1$, rdf:type, Employee)*. En appliquant de nouveau la règle rdf_9 sur *(Employee, rdfs:subClassOf, Person)* et le triplet inféré *(fullProfessor$_1$, rdf:type, Employee)* nous obtenons le triplet inféré *(fullProfessor$_1$, rdf:type, Person)*. Le processus est répété jusqu'à ce qu'on obtienne l'ensemble des données saturé décrit par la figure 1.6 où les arcs en rouge sont les arcs inférés.

Pour un ensemble de données $\mathbb{D}$ saturées avec un ensemble de règles d'inférence $\mathbb{R}$, aucun nouveau triplet ne peut être inféré à partir de $\mathbb{D}$ avec une ou plusieurs règles d'inférence de $\mathbb{R}$. Une fois les données

saturées, l'évaluation des requêtes ne nécessite aucun raisonnement. Par contre, toute modification des données RDF implique de refaire la saturation. Pour résoudre ce problème, surtout dans le cas des données RDF dynamiques [8], certaines BD-RDF proposent la technique dites de reformulation des requêtes.

1.3.3.2 Reformulation des requêtes

Une alternative à la saturation des BD-RDF est la reformulation des requêtes. La reformulation des requêtes est la réécriture de la requête en prenant en compte un ensemble de règle d'inférences. Le langage SPARQL définit plusieurs régimes d'inférence sous lesquels une requête peut être exécutée [62]. Le régime d'inférence de SPARQL est défini par un ensemble de règles d'inférences qui peuvent être appliquées à la requête durant la reformulation de cette dernière. Le régime d'inférence RDFS est l'un des régimes d'inférence défini par SPARQL. Dans ce régime, les règles d'inférences utilisées sont uniquement celles définies par RDFS. Pour un utilisateur ou un système souhaitant définir lui-même ses règles d'inférences, le langage SPARQL dispose du régime d'inférence RIF [62] (Rules Interchanged Format) qui permet de décrire et prendre en compte des règles d'inférences non prédéfinies dans le langage SPARQL. Quel que soit l'ensemble des règles d'inférence utilisées, le mécanisme de reformulation consiste à appliquer les règles d'inférences aux parties de la requête qui vérifient les hypothèses d'une ou plusieurs règles [63, 64].

Considérons le cas d'une requête qui recherche l'ensemble des employés de l'université $university_1$. Cette requête cherche des instances de la classe *Employee* dont la propriété *work_for* a pour valeur $university_1$. La règle rdf_9 dans le contexte de cette requête est traduite par la phrase suivante : *toutes les instances d'une sous-classe de la classe Employee sont aussi instances de Employee*. Nous allons donc rechercher aussi dans les sous-classes de la classe *Employee* les instances qui vérifient nos conditions. L'union de toutes les instances de toutes les sous-classes de *Employee* et de la classe *Employee* elle-même constituera le résultat de notre requête reformulée. D'après la figure 1.2, la classe *Employee* n'a qu'une seule sous-classe, la classe *Professor*. Donc, nous recherchons, parmi les instances des classes *Employee* et *Professor*, celles qui travaillent à $university_1$. Et, d'après la même figure 1.2, la requête ainsi formulée possède une seul réponse : l'instance $fullProfessor_1$.

Dans le but d'exploiter les données RDF, saturées ou non, stockées dans les BD-RDF des langages d'interrogation ont été proposés. Ces langages d'interrogation des ontologies sont regroupées en familles [65]. Nous avons, par exemple, la famille des langages qui traitent les données sous forme de triplets sans tenir compte de la sémantique associée aux éléments des triplets. Nous trouvons dans cette catégorie le langage SPARQL et aussi les langages RDQL [66] et SquishQL [67]. Dans une autre famille de langage, la séparation entre le schéma de l'ontologie et les données est bien marquée lors de l'interrogation. Nous pouvons citer trois exemples de langages de cette famille qui sont RQL [68], SeRQL [69] et OntoQL [70]. Il existe également une autre famille qui est celle des langages inspirés de XPath ou XQuery pour l'interrogation des ontologies et des données. Dans cette famille nous trouverons, par exemple, RDF-Path [71] et Versa [72]. Chaque langage possède des avantages et des inconvénients selon son contexte d'utilisation et les objectifs attendus [73]. Nos travaux portent sur un sous-ensemble de SPARQL, qui est présenté en détails dans la section suivante.

8. Les données dynamiques sont des données qui évoluent fréquemment dans le temps

1.4 Langage d'interrogation : SPARQL

Le langage SPARQL est une recommandation W3C largement utilisée pour l'interrogation des données RDF [10]. Il a été très vite considéré comme une technologie incontournable du Web sémantique. La conception du langage SPARQL a été fortement influencée par la structure du modèle RDF. C'est ainsi que les requêtes SPARQL, tout comme les données RDF, sont décrites par des triplets et elles peuvent aussi être représentées par des graphes. Le langage SPARQL définit quatre types de requêtes [10] :

— *ASK* : teste si, pour *un ensemble de contraintes*, il existe au moins une solution ou pas et renvoie un booléen, vrai ou faux, mais aucune information sur les résultats s'il y en a ;

— *DESCRIBE* : ce type de requête retourne des informations sur les résultats de la requête. Les informations décrivant les résultats, qui vérifient *les contrainte*s spécifiées dans la requête, sont retournées sous forme de graphe. Ce graphe est déterminé par le processus d'interrogation de SPARQL. L'utilisateur ne peut déterminer à l'avance la structure du graphe résultat ;

— *CONSTRUCT* : ce type de requête retourne les réponses, vérifiant *un ensemble de contraintes*, sous forme de graphe RDF. La structure du graphe résultat est décrit par un patron (ou template) dans la requête. Elle est comparable à une vue matérialisée dans les SGBDR ;

— *SELECT* : ce type de requête permet d'extraire des informations qui vérifient *les contraintes* spécifiées dans la requête.

Nos travaux se focalisent sur les requêtes de type SELECT. Notons que des études ont montré que les requêtes de type SELECT représentent 94% des requêtes utilisées pour interroger des ontologies usuelles et disponibles en ligne [74, 11] telles que DBPedia [4], Linked Geo Data [75] et Semantic Web Dog Food [76]. De plus, les requêtes de type SELECT partagent avec les trois autres types de requête la partie de la syntaxe qui permet de décrire les contraintes qui doivent être respectées par les réponses.

1.4.1 Requête SELECT : patron de triplet, patron de graphe et syntaxe

La partie principale d'une requête SELECT en SPARQL est spécifiée à l'aide d'un patron de graphe. Des études montrent que 45% des requêtes SELECT sur des bases de connaissances usuelles sont spécifiées uniquement par des patrons de graphe [74]. La définition d'un patron de graphe s'appuie sur celle d'un patron de triplet, qui est présentée dans la section suivante.

1.4.1.1 Patron de triplet

Le patron de triplet est l'élément atomique de description de contrainte en SPARQL. C'est un triplet RDF dans lequel un ou plusieurs des éléments *(sujet, prédicat ou objet)* peuvent être une *variable*. Une variable est un identifiant alphanumérique commençant par "?". Le patron de triplet sert à trouver les triplets RDF qui satisfont le modèle qu'il définit, processus appelé *matching*. Prenons par exemple le patron de triplet *(?pr, rdf:type, FullProfessor)*. Le matching de ce patron de triplet sur le graphe de la figure 1.2, consiste à trouver les valeurs de *?pr* pour lesquelles le triplet appartient au graphe RDF. Si, pour le graphe RDF de la figure 1.2, la variable *?pr* prend la valeur $fullProfessor_1$, alors le patron de triplet devient le triplet *(*$fullProfessor_1$*, rdf:type, FullProfessor)* qui appartient bien au graphe RDF de la figure 1.2 ; donc $pr = fullProfessor_1$ est bien une solution. La combinaison de plusieurs patrons de triplets donne un *patron de graphe*.

1.4.1.2 Patron de graphe

Un patron de graphe est formé par un ensemble de patrons de triplets. Les patrons de triplets sont connectés entre eux par les ressources (variables, littéraux, ...) qu'ils ont en commun pour former le patron de graphe. Dans un patron de graphe, les variables partagées par au moins deux patrons de triplets sont appelées *variables de jointures*. Interroger un graphe RDF avec un patron de graphe revient à chercher des valeurs pour chacune des variables du patron de graphe afin que le graphe RDF obtenu (en remplaçant les variables par ces valeurs) soit un sous-graphe du graphe RDF.

Exemple 1.
$?pr$ *rdf:type* $FullProfessor.\ (t_1)$
$?pr\ works\_for\ ?univ.\ (t_2)$

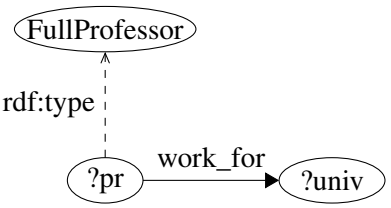


FIGURE 1.7 – Patron de graphe SPARQL

Dans l'exemple ci-dessus la variable *?pr* est une variable de jointure entre les deux patrons de triplets t_1 et t_2, tandis que *?univ* est juste une variable d'extraction de réponses. La figure 1.7 est une représentation du patron de graphe qui, au moment de l'exécution de la requête, doit correspondre à une partie du graphe de la figure 1.2. Pour les valeurs $\textit{fullProfessor}_1$ et $\textit{university}_1$ affectées respectivement aux variables *?pr* et *?univ*, nous obtenons un graphe qui correspond bien à une partie du graphe de la figure 1.2. Ce sont donc des réponses à la requête de l'exemple 1.

Il arrive que le patron de graphe soit composé d'au moins deux sous-graphes disjoints, c'est-à-dire sans variables communes, nous qualifions ce type de patron de graphe de *produit cartésien*. Le patron de graphe de l'exemple 2 est un exemple de produit cartésien constitué de deux sous-graphes disjoints, il s'agit des sous-graphes t_1 et t_2 d'une part et t_3 d'autre part. La figure 1.8 montre le graphe du produit cartésien. Nous remarquons bien qu'il s'agit de deux graphes disjoints : ils ne sont pas reliés par un arc et ne possèdent aucun sommet en commun. L'exécution de ce patron de graphe se fait en exécutant d'abord chaque sous-graphe disjoint et en réalisant un produit cartésien entre les réponses obtenues.

Exemple 2.
$?pr$ *rdf:type* $FullProfessor.\ (t_1)$
$?pr\ works\_for\ ?univ.\ (t_2)$
$?student$ *rdf:type* $Student.\ (t_3)$

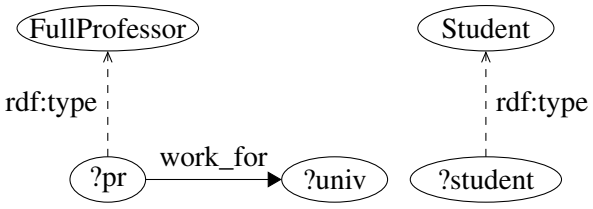


FIGURE 1.8 – Produit cartésien entre patrons de triplets

Comme la plupart des langages d'interrogation, le langage SPARQL possède une syntaxe textuelle. Dans la section suivante nous présentons la syntaxe de la requête SELECT du langage SPARQL.

1.4.1.3 Syntaxe et autres opérateurs de la requête SELECT

La spécification d'une requête SELECT en SPARQL utilise d'autres éléments que le patron de triplet et le patron de graphe, bien que ces deux éléments soient les principaux, comme nous l'avons dit précédemment. SPARQL offre ainsi d'autres opérateurs pour définir des requêtes SELECT. Nous pouvons citer les opérateurs FILTER, OPTIONAL, LIMIT, ORDER BY, DISTINCT et GROUP BY.

Précisons que cette présentation de SPARQL n'est pas exhaustive, nous présentons un sous ensemble de ce langage que nous considérons pertinent dans nos travaux et comme illustration.

— **FILTER.** Cet opérateur permet de spécifier des conditions sur la valeur des variables. Les variables sur lesquelles sont définies les conditions ou *variables filtrées* doivent intervenir dans au moins un patron de triplet (ou mapping ou bind) du patron de graphe. Cet opérateur permet de filtrer parmi les réponses qui correspondent au patron de graphe celles qui satisfont les conditions spécifiées. Les conditions sont décrites par des opérateurs de comparaison et des fonctions [10].

— **UNION.** Cet opérateur crée un multi-ensemble (multiset)de réponses à partir des ensembles de réponses de deux ou plusieurs patrons de graphe. Ces derniers peuvent partager des variables en commun ou pas.

— **OPTIONAL.** Il rend facultatif un ensemble de contraintes. Une solution, qui vérifie les contraintes obligatoires et optionnelles aura des valeurs pour toutes les variables correspondantes. Mais, si la solution ne vérifie que les contraintes obligatoires, alors seules les variables des contraintes obligatoires auront une valeur, les autres variables n'auront pas de valeur : *UNBOUND* (ou *Null*).

Exemple 3.
$?pr$ *rdf:type* $FullProfessor$. (t_1)
$?pr\ work\_for\ ?univ$. (t_2)
`OPTIONAL` $(?st\ member\_of\ ?univ)$. (t_3)

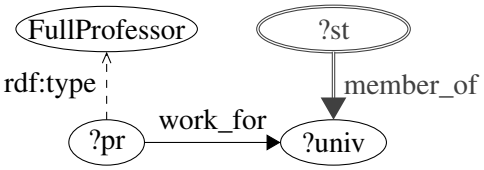


FIGURE 1.9 – Patron de graphe avec OPTIONAL

Considérons la requête de l'exemple 3 décrit par le graphe de la figure 1.9. Dans cet exemple, le patron de triplet t_3 est optionnel, ce qui veut dire que les contraintes obligatoires, qui doivent être vérifiées par toutes les réponses, sont t_1 et t_2. Si une réponse vérifie ces contraintes mais ne vérifie pas t_3, alors la variable $?st$ n'aura pas de valeur, c'est-à-dire *Null*. Si nous interrogeons les données RDF du tableau 1.3 avec cette requête nous trouverons ainsi trois réponses présentées dans le tableau 1.2. Nous utilisons le tableau 1.3 pour tous les exemples de cette section.

Numéro	?pr	?univ	?st
1	fullProfessor_1	univ_1	gStudent_1
2	fullProfessor_1	univ_1	uStudent_1
3	fullProfessor_2	univ_2	*Null*

TABLE 1.2 – Solutions d'une requête avec OPTIONAL

Tandis que les deux premières réponses vérifient tous les patrons de triplets, la troisième solution ne vérifie que les contraintes obligatoires d'où l'absence de valeur pour la variable ?st (= *Null*).

— **Les modificateurs.** Ce sont des opérateurs qui permettent de modifier la structure des réponses vérifiant les contraintes spécifiées. Il en existe plusieurs ; parmi les plus récurrents nous pouvons citer : LIMIT qui permet de limiter le nombre de réponses, ORDER BY permet de classer les réponses dans un ordre précis et paramétrable, DISTINCT permet d'éviter la redondance dans les réponses pour une ou plusieurs variables et enfin GROUP BY qui permet de regrouper les réponses par valeur d'une variable ou d'un groupe de variables.

Ces éléments se retrouvent dans la syntaxe de la requête SELECT dans le langage SPARQL. La syntaxe qui permet d'écrire textuellement une requête SELECT dans le langage SPARQL est la suivante [10] :

$$\langle \mathbf{SelectQuery} \rangle ::= \texttt{SELECT}\ \langle \texttt{DISTINCT}|\texttt{REDUCED} \rangle ?\ \langle \mathbf{Var}+ \rangle | *\\ \langle \mathbf{DatasetClause}* \rangle \langle \mathbf{WhereClause} \rangle \langle \mathbf{SolutionModifier} \rangle$$

Dans cette syntaxe, `SELECT` est le mot clé permettant d'identifier le type de la requête. Les mots clés `DISTINCT` et `REDUCED` permettent de modifier la structure des réponses. Le symbole $\mathbf{Var}+$ désigne la projection d'au moins une variable comme réponse recherchée par la requête et $*$ pour toutes les variables. Le symbole $\mathbf{DatasetClause}*$ permet de désigner l'ensemble des données interrogées, elle peut être omise et, dans ce cas, ce sera l'ensemble des données par défaut du système.
Le symbole $\mathbf{WhereClause}$ représente l'ensemble des contraintes de la requête. Ces dernières peuvent être des patrons de graphe avec ou sans filtre, optionnelles ou pas, des unions ou encore une combinaison de ces opérateurs dans le cas des requêtes complexes. La description de l'ensemble des contraintes est introduite par le mot clé `WHERE`. Enfin, le symbole $\mathbf{SolutionModifier}$ désigne un autre ensemble de modificateurs de réponse, précisément `ORDER BY` et/ou `GROUP BY`.

L'exemple 4 illustre une requête SELECT en SPARQL sous forme textuelle respectant la syntaxe de SPARQL hormis les préfixes qui ont été ignorés pour des raisons de lisibilité. Dans cet exemple, la fonction *BOUND* permet de vérifier si la variable $?univType$ correspond à une valeur dans le graphe RDF. Le mot clé `ASC` indique de faire un tri dans l'ordre croissant. La figure 1.10 représente une partie de la forme graphique de cette requête. Comme nous pouvons le voir, seul le patron de graphe est représenté graphiquement, les options telles que les filtres, les unions et les modificateurs ne sont pas illustrés.

Exemple 4.
`SELECT` $?pr\ ?univ\ ?univType\ ?st$
`WHERE{`
$?pr$ ***rdf:type*** $FullProfessor.\ (t_1)$
$?pr\ work\_for\ ?univ.\ (t_2)$
$?univ$ ***rdf:type*** $?univType.\ (t_3)$
`OPTIONAL`$(?st\ member\_of\ ?univ).\ (t_4)$
`FILTER (BOUND`$(?univType)$`)`
}
`GROUP BY` $(?univType)$
`ORDER BY ASC` $(?pr)$

Sujet	Prédicat	Objet
University	*rdf:type*	*rdfs:Class*
Professor	*rdf:type*	*rdfs:Class*
FullProfessor	*rdf:type*	*rdfs:Class*
Student	*rdf:type*	*rdfs:Class*
FullProfessor	*rdfs:subClassOf*	Professor
$univ_1$	*rdf:type*	University
$univ_2$	*rdf:type*	University
$fullProfessor_1$	*rdf:type*	FullProfessor
$fullProfessor_2$	*rdf:type*	FullProfessor
$fullProfessor_1$	work_for	$univ_1$
$fullProfessor_2$	work_for	$univ_2$
$gStudent_1$	*rdf:type*	Student
$uStudent_1$	*rdf:type*	Student
$gStudent_1$	member_of	univ_1
$uStudent_1$	member_of	univ_1

TABLE 1.3 – Données RDF : Table de triplets

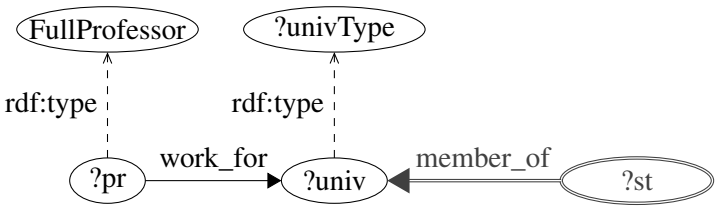


FIGURE 1.10 – Exemple de requête SELECT

L'exécution d'une requête SPARQL passe par l'interprétation de la sémantique du langage SPARQL, décrite dans la section suivante.

1.4.2 Requête SELECT : interprétation et évaluation

Dans cette section, nous présentons l'interprétation et l'évaluation des requêtes SELECT par les interprètes SPARQL. Ce processus d'interprétation et d'évaluation est basé sur une sémantique qui associe des actions à certains mots clés. Nous considérons dans cette partie que les patrons de graphe car ils sont au centre de nos travaux. Pour les requêtes contenant un ou plusieurs opérateurs tels que FILTER, UNION, OPTIONAL, Pérez et al. [77] présentent en détails la sémantique de ces opérateurs. Nous présentons successivement la définition formelle d'un patron de triplet et d'un patron de graphe.

1.4.2.1 Définition formelle d'un patron de triplet et d'un patron de graphe

Dans SPARQL, *un graphe RDF* est considéré comme un ensemble de triplets RDF. Pour un graphe RDF $\mathbb{D}$, nous considérons les ensembles suivants:

— U, l'ensemble des URI associées à $\mathbb{D}$.
— L, l'ensemble des litéraux définis dans $\mathbb{D}$.
— B, l'ensemble des nœud anonymes (Blank) de $\mathbb{D}$.

un triplet RDF est un élément de l'ensemble $(U \cup B) \times U \times (U \cup L \cup B)$. Comme nous le savons déjà, un patron de triplet est un triplet RDF contenant éventuellement des *variables*. Ainsi, *un patron de triplet* est un élément de l'ensemble $(U \cup V) \times (U \cup V) \times (U \cup L \cup V)$ où V est un ensemble de variables disjoint de U, B et L. On notera $var(t)$ l'ensemble des variables d'un patron de triplet t. Un patron de graphe est interprété comme une conjonction de patron de triplets, c'est-à-dire que si Q est un patron de graphe, alors $Q = t_1 \wedge t_2 \wedge \cdots \wedge t_n$ où chaque $t_i, 1 \leq i \leq n$ est un patron de triplet. On notera $|Q|$ le nombre de patrons de triplets d'un patron de graphe, dans notre cas $|Q| = n$. Et comme pour $var(t)$, nous notons $var(Q)$ l'ensemble des variables distinctes des patrons de triplet de Q. $var(Q)$ se calcule avec la formule suivante : $var(Q) = \bigcup_{i=1}^{n} var(t_i)$. L'évaluation des patrons de graphe retourne des *mappings* entre les variables et les valeurs.

1.4.2.2 Evaluation des patrons de triplet : mappings et propriétés

L'évaluation d'un patron de graphe est la recherche *des combinaisons des valeurs des variables* de ce patron de graphe telles qu'en remplaçant chaque variable par sa valeur dans le patron de graphe, on obtient un sous-graphe du graphe de données RDF interrogé. Cette combinaison de valeurs de variables qui permet d'obtenir une correspondance entre un patron de graphe et une partie du graphe de données RDF, est appelé *mapping* [77].

Mapping. Formellement un mapping μ se définit par une fonction partielle [9] de $V \longrightarrow U \cup L \cup B$ [77]. On dénote par $dom(\mu)$ le domaine du mapping μ, c'est-à-dire l'ensemble des variables pour lesquelles μ est défini. La propriété $dom(\mu) \subseteq V$ est toujours vérifiée. $\forall\ v \in dom(\mu)$, $\mu(v)$ retourne la valeur associée à la variable v. Par abus de langage, on considère que pour un patron de triplet t, la fonction $\mu(t)$ retourne le triplet obtenu en remplaçant, dans le patron de triplet t, toute les variables $v \in var(t)$ par le mapping $\mu(v)$. Ainsi, l'évaluation d'un patron de triplet t calcule l'ensemble des mappings Ω tel que $\Omega = \{\mu \mid dom(\mu) = var(t)\ and\ \mu(t) \in \mathbb{D}\}$.

Exemple 5.
SELECT $?pr\ ?univ\ ?univType\ ?st$
WHERE {
$?pr$ *rdf:type* $FullProfessor.\ (t_1)$
$?pr\ work\_for\ ?univ.\ (t_2)$
$?univ$ *rdf:type* $?univType.\ (t_3)$
$?st\ member\_of\ ?univ.\ (t_4)$}

Dans l'exemple 5, la requête contient quatre patrons de triplets t_1, t_2, t_3 et t_4. Chaque patron de triplet possède au moins une variable. $var(Q) = (?pr, ?univ, ?univType, ?st)$ est l'ensemble des variables de la requête Q ; cet ensemble correspond également au domaine de la fonction μ. Lorsque nous exécutons cette requête sur les données de la table 1.3, nous obtenons les deux mappings du taleau 1.4.

Mapping	?pr	?univ	?univType	?st
μ_1	fullProfessor_1	univ_1	University	gStudent_1
μ_2	fullProfessor_1	univ_1	University	uStudent_1

TABLE 1.4 – Exemple de mappings

Le tableau 1.4 contient les deux mappings vérifiant la requête de l'exemple 5. Nous voyons bien que ces deux mappings sont des fonctions qui associent à chaque variable de la requête une valeur de l'ensemble de données RDF.

Propriétés des mappings. Les mappings possèdent deux propriétés. Il s'agit de l'égalité et la compatibilité entre deux mappings.

Égalité entre deux mappings : deux mappings μ_1 et μ_2 sont dits égaux ssi $dom(\mu_1) = dom(\mu_2)$ et $\forall\ v \in dom(\mu_1); \mu_1(v) = \mu_2(v)$.
Dans le tableau de mappings 1.4, μ_1 et μ_2 ne sont pas égaux parce que $\mu_1(st) \neq \mu_2(st)$.

Compatibilité entre deux mappings : deux mappings μ_1 et μ_2 sont dits compatibles ssi pour toutes variables $v \in dom(\mu_1) \cap dom(\mu_2)$, $\mu_1(v) = \mu_2(v)$ c'est-à-dire $\mu_1 \cup \mu_2$ est aussi un mapping [77].

Conséquences : deux mappings égaux sont compatibles. Par ailleurs, deux mappings disjoints, $dom(\mu_1) \cap dom(\mu_2) = \emptyset$, sont aussi compatibles.

Pour illustrer la notion de compatibilité reprenons la requête de l'exemple 5. Nous divisons cette requête en deux, $Q_1 = t_1 \wedge t_2 \wedge t_3$ et $Q_2 = t_4$. Nous obtenons les requêtes ci-dessous.

9. La fonction μ est définie sur l'ensemble des variables du patron de graphe inclues dans V

SELECT $?pr$ $?univ$ $?univType$
WHERE {
$?pr$ *rdf:type* $uba : FullProfessor.$ (t_1)
$?pruba : work\_for$ $?univ.$ (t_2)
$?univ$ *rdf:type* $?univType.$ $(t_3)\}$

SELECT $?univ$ $?st$
WHERE {
$?st$ $member\_of$ $?univ.$ $(t_4)\}$

Les mappings des requêtes Q_1 et Q_2 sont données respectivement dans les tableaux 1.5 et 1.6.

Mapping	?pr	?univ	?univType
μ_1	fullProfessor_1	univ_1	University
μ_2	fullProfessor_2	univ_2	University

TABLE 1.5 – Mapping Q_1

Mapping	?univ	?st
μ_3	univ_1	gStudent_1
μ_4	univ_1	uStudent_1

TABLE 1.6 – Mapping Q_2

Nous remarquons que $dom(\mu_1) \cap dom(\mu_3) = ?univ$ et $\mu_1(?univ) = \mu_3(?univ) = univ_1$ donc μ_1 et μ_3 sont compatibles et $\mu_1 \cup \mu_3 = (?pr = fullProfessor_1,\ ?univ = univ_1,\ ?univType = University,\ ?st = gStudent_1)$. De même, nous avons μ_1 et μ_4 sont compatibles et $\mu_1 \cup \mu_4 = (?pr = fullProfessor_1,\ ?univ = univ_1,\ ?univType = University,\ ?st = uStudent_1)$. Par contre, $dom(\mu_2) \cap dom(\mu_3) = ?univ$ et $\mu_2(?univ) \neq \mu_3(?univ)$ donc μ_2 et μ_3 ne sont pas compatibles, de même que μ_2 et μ_4. Il existe aussi des opérations entre mappings. Ces opérations sont utilisées pour décrire la sémantique des patrons de graphe et des autres opérateurs de la requête SELECT dans SPARQL. Dans la section suivante, nous présentons la sémantique d'un patron de graphe et l'opérateur qui permet de la décrire.

1.4.2.3 Sémantique et évaluation d'un patron de graphe

Le langage SPARQL interprète un patron de graphe comme *une conjonction de patrons de triplets*. Ainsi, les mappings d'un patron de graphe sont obtenus à partir des mappings de chaque patron de triplet constituant le patron de graphe. Pour ce faire l'opérateur de jointure entre deux mappings a été crée. Cet opérateur se définit comme suit : soit Ω_1 et Ω_2 deux ensembles de mappings, la jointure de ces deux ensembles, notée $\Omega_1 \bowtie \Omega_2$, est $\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \wedge \mu_1, \mu_2\ sont\ compatibles\}$, c'est-à-dire, l'union des mappings compatibles.

Comme exemple de jointure, considérons celle entre les mappings des tables 1.5 et 1.6 des requêtes Q_1 et Q_2. Pour cette jointure, nous aurons $\mu_1 \cup \mu_3$ et $\mu_1 \cup \mu_4$ car seul μ_1 est compatible avec μ_3 et μ_4. Le résultat de cette jointure est donnée dans le tableau 1.7.

Mapping	?pr	?univ	?univType	?st
μ_1	fullProfessor_1	univ_1	University	gStudent_1
μ_2	fullProfessor_2	univ_1	University	uStudent_1

TABLE 1.7 – Mappings résultats de $Q_1 \bowtie Q_2$

Pour le cas particulier des patrons de graphe disjoints, c'est-à-dire lorsque $dom(\Omega_1) \cap dom(\Omega_2) = \emptyset$, et que Ω_1 et Ω_2 sont compatibles, la jointure devient un produit cartésien des mappings.

Plus généralement, pour un patron de triplet t nous noterons $[[t]]_{\mathbb{D}} = \{\mu \mid dom(\mu) = var(t)\ and\ \mu(t) \in \mathbb{D}\}$, l'ensemble des mappings de t. Pour un patron de graphe $Q = t_1 \wedge t_2 \wedge \cdots \wedge t_n$ où chaque patron de

triplet t_i a pour solution $[[t_i]]_{\mathbb{D}} = \Omega_i$, $1 \leq i \leq n$, l'ensemble des mappings du patron de graphe Q sur un ensemble de données $\mathbb{D}$, noté $[[Q]]_{\mathbb{D}}$, est $[[Q]]_{\mathbb{D}} = \Omega_1 \bowtie \Omega_2 \bowtie \ldots \bowtie \Omega_n$.

Pérez et al. [77] définit d'autres opérations sur les mappings, notamment la différence, l'union et également un autre type de jointures dites jointures externes. Ces opérations permettent de calculer les mappings pour des requêtes SELECT avec des opérateurs UNION, FILTER ou OPTIONAL par exemple.

Conclusion

Les bases de connaissances représentent de manière explicite la connaissance sur plusieurs domaines. Elles définissent des concepts ainsi que la structure de l'ensemble des concepts ainsi définis. Ces descriptions et structurations sont formalisées via l'utilisation des langages de formalisation. Nous avons notamment présenté comment la connaissance est formalisée avec les langages RDF et RDFS. Ces deux langages utilisent, pour représenter la connaissance, une structure de graphe dans laquelle les nœuds sont des concepts décris par des propriétés et les arcs sont des prédicats représentant les relations entre concepts. Ils permettent aussi la formalisation des règles de raisonnement qui peuvent être utilisées pour inférer de nouvelles connaissances. Dans ce chapitre nous avons aussi abordé le stockage de ces connaissances dans les BD-RDF. Certaines sont dites non-native s'appuyant sur les bases de données relationnelles et d'autres dites natives utilisent la structure en graphe des connaissances. Certaines BD-RDF embarquent dans leur système un moteur de raisonnement qui peut fonctionner sous deux modes distincts, la saturation des données et la reformulation des requêtes. Les requêtes exprimées sur ces connaissances sont décrites à l'aide des langages d'interrogation adaptés dont le plus utilisé est SPARQL.

Cependant, les ontologies et les technologies liées aux ontologies telles que le langage SPARQL sont encore inconnues ou peu maîtrisées par des utilisateurs non experts. Ces utilisateurs peuvent donc avoir besoin d'assistance dans l'exploitation de ces données. De plus les ontologies permettant de gérer des données hétérogènes et dynamiques, ces données peuvent rapidement évoluer avec le temps et rendre rapidement les connaissances des utilisateurs obsolètes pour l'exploitation de ces données RDF. Cette obsolescence de la connaissance des utilisateurs sur le contenu des BD-RDF, est l'une des causes des formulations incorrectes ou incomplètes des requêtes qui renverront des réponses insatisfaisantes aux utilisateurs. De cette insatisfaction peut naître une frustration chez les utilisateurs. Afin d'éviter cette frustration, les utilisateurs ont besoin d'être accompagnés dans la construction des requêtes et/ou dans le traitement de ces dernières. Pour palier ce problème, des recherches sur les systèmes coopératifs ont été menées. Plusieurs approches coopératives ont ainsi été développées afin de proposer des solutions alternatives aux utilisateurs même pour des requêtes incorrectes ou incomplètes. Le chapitre suivant présente les différentes approches coopératives existantes.

Chapitre 2

Synthèse des approches d'interrogation coopératives

Sommaire

Résumé : Nos travaux portent principalement sur la relaxation des requêtes, qui fait partie des approches d'interrogation coopératives. Nous réalisons dans ce chapitre un panorama des approches d'interrogation coopératives existantes pour positionner nos travaux. Cette synthèse présente les cinq principales approches d'interrogation coopératives, à savoir la suggestion de requêtes (Query Suggestion), l'interrogation par des exemples (Query By Example, QBE), l'explication des réponses d'une requête (Why-not Answers), le raffinement de la requête par les utilisateurs (Users Refinement) et la redéfinition des

requêtes dont l'une des variantes est la relaxation des requêtes. Si, toutes ces approches aident les utilisateurs dans la recherche des informations, elles ont chacune leurs avantages et leurs inconvénients que nous présentons dans ce chapitre. Ce chapitre expose les idées qui sous-tendent ces différentes approches sans toutefois entrer dans les détails. Il présente également les systèmes de recommandation qui sont des cas particuliers d'utilisation de certaines approches d'interrogation coopératives.

Introduction

Un problème important des utilisateurs des bases de données, en général, est la formulation et la soumission des requêtes aux systèmes de gestion des données pour rechercher des réponses à leurs besoins. Pour formuler leurs requêtes, les utilisateurs ne disposent que d'une connaissance limitée, très souvent, et même obsolète, des données à interroger. Ce problème devient encore plus complexe lorsque les utilisateurs se trouvent dans un environnement de bases de données distribuées avec des sources distantes et hétérogènes, dont les accès sont, très souvent, réglementés. De plus, pour faciliter la communication et l'intégration des données entre les sources hétérogènes, des ontologies ont vu le jour. L'intégration des données est réalisée à partir des schémas des données représentés dans des ontologies. Cette évolution accroit également la complexité de l'interrogation et de l'exploitation de ces bases de données. Dans ce contexte de données volumineuses, distribuées, hétérogènes, dynamiques et liées, l'ignorance des utilisateurs sur les données interrogées et les liens entres elles, rend plus complexe la tâche de formulation de la requête. Ce qui est particulièrement vrai pour le cas des ontologies. De plus, les systèmes d'interrogation existants n'offrent que très peu d'assistance aux utilisateurs, principalement les trois services suivants :

— l'interrogation avec des requêtes dans les langages tels que SQL [78, 79] ou SPARQL [10] ;
— l'interrogation des données par des applications dédiées ;
— l'interrogation des données via des interfaces graphiques telles que des formulaires ou des langages graphiques de composition de requêtes qui sont plus intuitifs [80, 81].

C'est dans ce contexte que les approches d'interrogation coopératives ont vu le jour. Leur objectif est d'assister les utilisateurs dans l'interrogation des données. Ces approches ont ensuite été intégrées dans des systèmes d'interrogation tels que les systèmes de recommandation.

Kaplan [12] est l'un des pionniers dans la recherche sur les approches d'interrogation coopératives. Dans le cadre de sa thèse, il a mené des travaux sur les réponses coopératives dans l'interrogation des données en langage naturel. Par la suite, d'autres chercheurs, parmi lesquels Gal [82], Janas [83] et Motro [84], ont développé des approches d'interrogation coopératives pour différents types de bases de données. Motro [85] catégorise ces approches en deux grands groupes qui sont :

1. les approches qui assistent les utilisateurs dans la formulation de leurs besoins en requêtes ;
2. les approches qui analysent la requête afin de détecter des anomalies dans les besoins exprimés par les utilisateurs ou dans leur formulation.

La première catégorie aide les utilisateurs qui ne savent pas vraiment ce qu'ils veulent ou ne savent pas comment formuler leurs besoins. Dans cette catégorie, nous retrouvons la suggestion de requêtes et les requêtes par l'exemple. La deuxième catégorie d'approches coopératives permet pour une requête bien formulée, d'alerter les utilisateurs en cas de problème pendant l'exécution. Ces problèmes peuvent être :

— aucune réponse n'est retournée, cas des *requêtes à réponse vide (empty answer query)* ;
— un utilisateur, familier de la base de données et aux données, n'est pas satisfait des réponses retournées par la requête. On parle alors de *réponses insatisfaisantes* ;
— un utilisateur qui ne comprend pas pourquoi il a obtenu des réponses mais pas celles qu'il attendait, c'est un autre aspect des *réponses insatisfaisantes* ;
— un utilisateur est partiellement satisfait par les réponses retournées : *réponses incomplètes*.

Nous trouvons, dans la deuxième catégorie, les approches d'interrogation coopératives suivantes : l'explication de requête, le raffinement par l'utilisateur et la redéfinition de requête.

Dans ce chapitre, nous présentons tour à tour les principales approches d'interrogation coopératives proposées dans la littérature. Il s'agit de la suggestion de requêtes (section 1), la construction de requêtes par

l'exemple (section 2), l'explication des réponses d'une requête (section 3), le raffinement de la requête par l'utilisateur (section 4) et la redéfinition d'une requête (section 5). Puis, nous présentons une famille de systèmes coopératifs intégrant ces approches : les systèmes de recommandation (section 6). Enfin, nous dressons, dans la conclusion, le bilan de ces approches suivants les comportements des utilisateurs, énumérés par Minker [86], que les systèmes coopératifs doivent considérer. Par la suite, nous utilisons l'acronyme AIC pour *Approches d'Interrogation Coopératives*.

2.1 Suggestion de requêtes

La suggestion de requêtes aide les utilisateurs à formuler leurs besoins sous forme de requêtes syntaxiquement et sémantiquement correctes. Elle a initialement été proposée pour éviter aux utilisateurs de formuler des requêtes qui ne pourront pas être satisfaisantes à cause des *fausses suppositions*. Dans son étude des systèmes de bases de données coopératifs, Wesley [87] définit deux types de fausses suppositions qui peuvent conduire à l'insatisfaction des utilisateurs. Le premier type est l'ensemble des suppositions faites par les utilisateurs sur le schéma des données, on parle de *présupposition*. Le deuxième type est l'ensemble des suppositions sur les données que les utilisateurs pensent exister ou pas dans la base de données, Wesley parle de *misconception*. Ainsi, pour aider les utilisateurs à formuler des requêtes ne contenant pas de fausses suppositions, des techniques d'édition de requêtes ont initialement été proposées dans ce qui deviendra l'AIC par suggestion de requêtes. Parmi ces techniques, nous trouvons des assistants d'édition de requête via des formulaires ou des diagrammes. Avec le temps, la suggestion de requêtes a évolué et s'est enrichie de plusieurs autres techniques pour assister les utilisateurs. Les outils de suggestion de requêtes ont également évolué pour suggérer aux utilisateurs plusieurs parties de la requête en cours de formulation. Enfin, une autre évolution permet de vérifier les réponses de la requête au fur et à mesure de sa construction.

Cette section présente les trois principales techniques de l'AIC par suggestion de requêtes. D'abord, nous présentons les interfaces graphiques de construction des requêtes ; ensuite, l'auto-complétion et enfin, nous verrons la technique d'évaluation et de vérification des réponses de la requête au fil de sa construction.

2.1.1 Interfaces de construction

La conception des interfaces utilisateurs pour la formulation des requêtes a été motivée par le besoin de simplifier la construction des requêtes. Motro [88] a proposé trois interfaces de construction de requêtes. Il s'agit des systèmes : BAROQUE [89], VAGUE [90] et FLEX [91]. Le système BAROQUE [89] explore les tables et les colonnes des bases de données relationnelles pour construire les requêtes. Cette exploration permet d'éviter des fausses présuppositions des utilisateurs et donc, de construire des requêtes qui pourraient renvoyer des réponses satisfaisantes pour les utilisateurs. Le système Flex [91] est globalement décrit comme l'union des systèmes BAROQUE et VAGUE. Cet outil expérimental propose des interfaces adaptées au niveau d'expertise des utilisateurs et intègre les techniques de réparation pour des requêtes avec des fausses suppositions ou misconceptions sur le contenu de la base de données. De nos jours, la plupart des SGBDR offrent des outils pour aider les utilisateurs dans la création des requêtes. C'est par exemple le cas de SQL Server qui offre dans sa boîte à outils (SQL Server Data Tools, SSDT [10]), une interface de construction de requête SQL. Il en est de même pour le SGBDR Oracle qui propose Oracle SQL Developer [92]. Toutes ces interfaces aident principalement dans la visualisation du schéma de la base de données et donnent la possibilité de sélectionner des concepts de ce schéma

10. https://msdn.microsoft.com/en-us/library/dd220607.aspx

Systèmes	Catégorie	Langage d'interrogation	Type de Données
BAROQUE [89]	Formulaire	SQL	relationnel
FLEX [91]	Formulaire	SQL	relationnel
Kaleidoquery [96]	Diagramme	OQL [11][97]	Objet
QGraph [98]	Diagramme	SQL	relationnel
visXcerpt [99]	Hybride	Xcerpt	XML
SPARQLViz [100]	Formulaire	SPARQL	RDF
NITELIGHT [95]	Diagramme	SPARQL	RDF
Xml in Graphics (Xing) [101]	Diagramme	Lorel & XML-QL	XML
FedViZ [102]	Hybride	SPARQL	Données RDF fédérées
QueryVOWL [103]	Diagramme	Logique de description	OWL

TABLE 2.1 – Systèmes graphiques d'interrogation des données

pour la requête en construction. Ils aident ainsi les utilisateurs à construire des requêtes syntaxiquement correctes. Néanmoins, ces outils possèdent un inconvénient. Ils sont, pour la plupart, adaptés pour un unique SGBDR même si les systèmes utilisent en arrière-plan le même langage de requête. Pour traiter cette limite, Manoj et al. [93] propose un outil capable d'être utilisé avec n'importe quel SGBDR utilisant le langage de requête SQL. De plus, des systèmes de construction graphique des requêtes ont été développés pour aider les utilisateurs inexpérimentés dans l'analyse, la compréhension et l'interprétation des schémas de données.

Les systèmes d'interrogation graphiques des bases de données (Visual Query Systems) sont des systèmes qui utilisent des objets graphiques pour formuler les requêtes [94]. Ces systèmes ont plusieurs avantages :

- ils traduisent le langage de requêtes textuel sous forme graphique. Ils offrent donc toutes les fonctionnalités du langage de requêtes textuel ;
- ils sont intuitifs et exploitent la mémoire visuelle des utilisateurs ;
- ils renforcent le dialogue entre les utilisateurs et le système qui devient ainsi plus coopératif.

Catarci et al. [94] présentent trois catégories de systèmes d'interrogation graphiques des bases de données : les formulaires ; les diagrammes et les systèmes hybrides qui combinent les deux précédents. Le tableau 2.1 montre que les systèmes d'interrogation graphiques permettent d'interroger tous les types de données qui possèdent préalablement un langage de requêtes. Les données ontologiques (RDF, OWL) n'échappent pas à cette règle. Smart et al. [95] ont proposé pour ces données, l'outil NITELIGHT d'édition graphique de requêtes SPARQL. NITELIGHT est un système de la catégorie des diagrammes dans lequel les éléments de la requête sont représentés à l'aide de diagrammes géométriques (carrés, rectangles, lignes etc.).

Néanmoins, la suggestion réalisée par ces outils reste très implicite. Elle est liée à la forme des diagrammes géométriques de l'outil et à la sémantique du langage. Par exemple, une ligne signifie presque toujours un lien entre deux éléments. Si, dans ces outils, la suggestion se fait par la présentation visuelle des composants de la requête, d'autres techniques plus pro-actives se proposent d'aider les utilisateurs à compléter la requête : ce sont les techniques *d'auto-complétion*.

11. Object Query Language

2.1.2 Auto-complétion

L'auto-complétion des requêtes n'aide pas seulement les utilisateurs à formuler leurs requêtes, pour éviter des fausses présuppositions, elle permet également de suggérer une ou plusieurs parties des requêtes aux utilisateurs. Ces derniers sélectionnent ensuite la suggestion qui correspond à leurs besoins. Les techniques d'auto-complétion utilisent un contexte qui est constitué des éléments suivants :

1. le schéma des données ;
2. l'historique des requêtes ;
3. les préférences des utilisateurs.

Le schéma de données : il est utilisé comme un dictionnaire de mots clés contenant l'ensemble des noms des concepts de la base de données. Ce dictionnaire permet ainsi de réaliser une auto-complétion lexicale des concepts saisis par les utilisateurs pendant la formulation de la requête. Cette technique de suggestion est la plus répandue et la quasi-totalité des systèmes d'interrogation l'utilisent comme le notent Bar-Yossef et Naama. [104]. Cette technique permet principalement d'éviter des erreurs syntaxiques liées à l'orthographe de certains concepts de la base de données. Par ailleurs, Campinas et al. [105] utilisent le schéma de données, défini en RDF, pour suggérer des concepts (sujet ou objet) ou des propriétés (prédicats) aux utilisateurs pour la construction des requêtes SPARQL. Ce principe est intégré dans l'outil de construction graphique de requêtes SPARQL, NITELIGHT de Russel et al. [106]. NITELIGHT exploite le schéma des données pour suggérer le prochain élément de la requête. Par exemple, pour une classe donnée, NITELIGHT proposera, comme prédicats liés à cette classe dans la requête, l'ensemble de ses propriétés définies dans le schéma ou une variable. Pour un prédicat donné, NITELIGHT utilise son codomaine pour suggérer toutes les classes valides ou une variable. De plus, en fonction des premiers caractères saisis par les utilisateurs, la liste de suggestion est réactualisée.

L'historique des requêtes : il est constitué de l'ensemble des requêtes récemment exécutées par un utilisateur ou l'ensemble des utilisateurs. L'une des techniques de suggestion utilisant l'historique des requêtes est celle de Bar-Yossef et al. [104]. Avec cette technique, à partir de l'état courant de la requête et en analysant l'historique des requêtes exécutées, le système suggère les suites les plus probables de la requête, en se basant sur des statistiques. Chuan Xiao et al. [107] présentent une autre technique qui cherche parmi les requêtes exécutées celles qui sont les plus proches de la requête en cours de construction. Cette technique, également présentée par Bar-Yossef et Naama [104] avec différentes mesures entre requêtes, est généralement utilisée pour la recherche par mots clés.

Les préférences de l'utilisateur : elles sont utilisées dans le cadre des techniques de suggestion similaires aux précédentes. Ces techniques sont centrées autour du modèle de préférence des utilisateurs. Jyun-Yu et al. [108] proposent une construction du modèle de préférence à partir des requêtes faites par les utilisateurs. D'autres, en plus de l'historique, considèrent également les séquences d'exécution des requêtes [109]. Ce modèle est ensuite exploité pour suggérer les prochains composants des requêtes.

L'auto-complétion aide les utilisateurs à formuler leurs requêtes par la suggestion d'une ou de plusieurs parties des requêtes en construction. Au terme de ce processus, les utilisateurs ont la garantie d'avoir des requêtes correctement formulées. Cependant, il est possible que cette requête retourne néanmoins un ensemble de réponses insatisfaisantes, ou même pas du tout de réponse. Ce résultat peut avoir plusieurs causes mais, dans tous les cas, l'auto-complétion ne permet pas aux utilisateurs de l'éviter. Pour y remédier, certains systèmes vérifient en continu les réponses des requêtes durant leurs constructions.

2.1.3 Vérification continue

Les techniques présentées jusqu'à présent se sont focalisées sur la formulation de requêtes. Mais, une requête peut être bien construite et retourner néanmoins un ensemble de réponses insatisfaisantes. Les techniques des réponses instantanées (Instant-Response) ont été développées pour cette raison. Ces techniques permettent d'avoir des informations sur les réponses, telles que le nombre de réponses et même un aperçu de quelques réponses. Arnab et Jagadish [110] proposent une solution qui utilise ce principe dans la recherche par mot clés. Le principe qui sous-tend les techniques des réponses instantanées est l'exécution fréquente de la requête pendant sa construction, pour informer les utilisateurs sur les réponses qu'ils recevront. Ainsi, si les réponses intermédiaires ne sont pas satisfaisantes, les utilisateurs peuvent immédiatement corriger la requête. Ce principe permet de construire la requête étape par étape : c'est une construction interactive et incrémentale de la requête. Le tableau 2.2 recense les travaux existants sur l'AIC par suggestion de requêtes (*Query Suggestion*). Ils sont décrits suivant quatre aspects :

— *le type de requête* : les techniques de suggestion de requêtes ont été proposées pour les requêtes par mots clés, des requêtes SQL, des requêtes SPARQL et bien d'autres ;
— *le mode d'interrogation* : l'expression des requêtes via des langages, formulaires dédiés, ou diagrammes ;
— *le mode d'auto-complétion* : l'auto-complétion peut exploiter le schéma ou contexte des données *(CD)*, les préférences ou contexte des utilisateurs *(CU)* ou encore l'historique des requêtes (HR) ;
— *la vérification des réponses* : la vérification en continu des réponses de la requête.

Approches	Type de requête	Mode d'interrogation	Mode d'auto-complétion	Vérification des réponses
Bar-Yossef et al. [104]	mots clés	textuelle	CD et HR	non
Xiao et al. [107]	mots clés	textuelle	CD	non
Khoussainova et al. [111]	SQL	textuelle	CD et HR	non
Whiting et al. [112]	mots clés	textuelle	HR	non
Jyun-Yu et al. [108]	mots clés	textuelle	CD et CU	non
Smart et al. [95]	SPARQL	visuelle	CD	non
Zhang et al. [113]	mots clés	textuelle	CD et CU	oui
Li et al. [114]	mots clés	textuelle	CD et CU	oui
Qumsiyeh et al. [115]	mots clés	textuelle	CD et HR	oui
Campinas et al. [116]	SPARQL	textuelle	CD	non

TABLE 2.2 – Solutions proposées pour la suggestion des requêtes

Nous observons que la suggestion des requêtes est plus traitée pour la recherche par mot clés. Elle est plus adaptée pour ce type de requête. Certains outils font une auto-complétion qui ne se limite pas à une partie de la requête mais qui concerne, au contraire, toute la requête : on parle alors de *recommandation de requêtes* [117].

2.2 Interrogation par l'exemple

Malgré l'aide apportée par les techniques de suggestion de requêtes dans la formulation des requêtes, Motro [85] montre que la conversion des besoins en requêtes est une tâche complexe. En effet, pour les utilisateurs n'ayant qu'une connaissance imprécise de leurs besoins, la formulation des requêtes correspondantes à ces besoins est très complexe. *L'interrogation par l'exemple (Query by Example, QBE)*

permet d'alléger cette difficulté. L'AIC d'interrogation par l'exemple a été initialement conçue par IBM comme un langage d'interrogation pour les données relationnelles. Zloof [118], le concepteur de cette approche, a eu l'idée d'un langage d'interrogation pour les non-programmeurs, afin de permettre aux utilisateurs de formuler leurs requêtes avec précision et d'éviter toutes fausses présuppositions. Dans cette section, nous présentons cette nouvelle approche d'interrogation sur les données relationnelles. Cette approche a ensuite été adaptée pour les autres types de données, notamment RDF et XML.

2.2.1 Interrogation par exemple des données relationnelles

Zloof [118] a proposé de construire une requête, non pas à partir des besoins mais, à partir d'exemples de réponses attendues par les utilisateurs. Cette approche demande aux utilisateurs de définir leurs besoins sous forme d'exemples de réponses attendues et de fournir ces réponses au système via une interface graphique dédiée. Cette interface présente tous les objets du SGBDR, c'est-à-dire les bases de données, les tables, les colonnes, etc. Dans cette approche, les utilisateurs peuvent spécifier plusieurs exemples de réponses et donc attendre des réponses semblables à au moins un des exemples. Ramakrisjnan et Gehrke [119] ont expliqué comment à partir de ces exemples et de la théorie du calcul relationnel, les systèmes d'interrogation retrouvent dans la base de données les réponses semblables aux exemples des utilisateurs. Illustrons le processus d'exécution d'une requête par l'exemple suivant, pris dans le contexte d'une base de données relationnelles.

Considérons la requête : *"Donnez la liste des cours suivis par l'étudiant nommé "Tim" et enseignés par un professeur de moins de "50 ans"* et les relations ci-dessous :

- *GraduateStudent (id:*INTEGER*, name:*STRING*, age:*REAL*)*;
- *Professor (id:*INTEGER*, name:*STRING*, age:*REAL*)*;
- *GraduateCourse (id:*INTEGER*, name:*STRING*, level:*STRING*)*;
- *TakeCourse (sid:*INTEGER*, cid:* INTEGER*)*.
- *TeacherOf (fid:*INTEGER*, cid:* INTEGER*)*.

Pour la construction de cette requête par des exemples, les exemples de solutions sont insérés dans les tables des relations. Pour cette opération, Zloof [118] adopte la convention suivante :

1. les variables des exemples de solutions doivent être soulignées ;
2. les constantes ou valeurs des exemples de solutions ne sont pas soulignées ;
3. les variables sélectionnées sont précédées de **P.** qui dénote la projection de la variable.

GraduateStudent		
id	**name**	**age**
ids	"Tim"	

TakeCourse	
sid	**cid**
ids	idc

GraduateCourse	
id	**name**
idc	P.NC

TeacherOf	
fid	**cid**
idf	idc

Professor		
id	**name**	**age**
idf	P.NF	<50

FIGURE 2.1 – Exemple de requête via le langage QBE

La figure 2.1 montre la traduction en QBE de notre requête. Pour notre requête nous avons un seul exemple de solution, elle correspond à la ligne des différentes tables. Pour la table **"GraduateStudent"**, la variable $\underline{ids}$ extrait le numéro unique de l'étudiant, la constante *"Tim"* dans la colonne **name** permet

de vérifier que le ou les étudiants retournés ont bien pour nom *"Tim"*. Enfin, aucune condition n'est spécifiée pour la colonne **age** de l'étudiant. Par contre, pour l'âge des professeurs nous avons spécifié une condition.

La description d'une requête en QBE se fait ainsi colonne par colonne dans les tables. Dans chaque colonne d'une table, les utilisateurs peuvent spécifier : (i) une condition que doivent remplir les valeurs de la colonne de cette table ; (ii) une variable pour récupérer la valeur de cette colonne ou (iii) rien du tout. La jointure est réalisée via des variables partagées entre plusieurs tables. Pour décrire un exemple sur plusieurs tables, il suffit de garder le même numéro de ligne entre toutes les tables. Autrement dit, chaque ligne i d'une table correspond à un exemple de solution des utilisateurs et toutes les lignes i de toutes les tables décrivent le même exemple sur chaque table. Lorsque les utilisateurs décrivent plusieurs exemples de solutions, le système cherche indépendamment les solutions correspondant à chaque exemple et fait l'union de toutes ces réponses pour les retourner aux utilisateurs.

Les travaux de Zloof [118] montrent également comment réaliser des opérations de mise à jour des bases de données en utilisant le langage QBE. Plusieurs systèmes ont intégré ce langage comme langage de définition et de manipulation des bases de données pour cette raison. Nous avons par exemple, le SGBD Access du pack office de Microsoft. Plus tard, Kacprzyk et al. [120] ont proposé une implémentation de l'interrogation avec imprécision, FQUERY, adaptée au QBE. Zadrozny et Kacprzyk [121] ont intégré cette extension de QBE qui inclut l'imprécision dans le SGBD Access. Par la suite, plusieurs travaux ont eu pour objectif d'adapter le QBE sur les données RDF et XML.

2.2.2 Adaptations du QBE : QFE, GQBE et XQBE

Nous avons identifié dans la littérature trois principales adaptations de QBE suivant l'environnement d'exploitation du langage :

- la requête à partir de l'exemple ou *Query From Example (QFE)*. L'objectif de cette adaptation est de retrouver la requête SQL qui correspond à cette requête par l'exemple ;
- l'interrogation des graphes RDF par l'exemple ou *Graph Query By Example (GQBE)*. C'est l'adaptation de QBE pour l'interrogation des données RDF ;
- l'interrogation des données XML par l'exemple : Un QBE pour interroger les données XML.

Requêtes à partir d'exemple (QFE) : Hao Li et al. [122] proposent cette déclinaison de QBE pour permettre aux utilisateurs de construire des requêtes SQL à partir des exemples. Dans cette adaptation, les exemples ne sont pas décrits comme dans le QBE original. Dans QFE, les utilisateurs donnent, pour des colonnes de la base de données, la valeur exemple. Ensuite, la base de données, notée $\mathbb{D}$, et les exemples de réponses, notés $\mathbb{R}$, sont exploités par QFE pour calculer l'ensemble des requêtes, $\mathbb{Q}$ sur $\mathbb{D}$ permettant aux utilisateurs d'obtenir $\mathbb{R}$. Enfin, pour raffiner l'ensemble des requêtes $\mathbb{Q}$ et ne garder que celles qui correspondent aux besoins des utilisateurs, QFE génère itérativement des ensembles de données $\mathbb{D}_i$ et calcule les réponses $\mathbb{R}_{ij}$ dans ces données pour chaque requête Q_j de $\mathbb{Q}$. Les réponses $\mathbb{R}_{ij}$ sont proposées aux utilisateurs afin qu'ils choisissent, parmi elles, celles qui sont satisfaisantes. QFE répète le processus en considérant ces nouvelles réponses satisfaisantes comme exemples de réponse. Le processus est répété jusqu'à ce que toutes les réponses $\mathbb{R}_{ij}$ soient satisfaisantes et donc que la requête Q_j décrive parfaitement les besoins de l'utilisateur.

Interrogation des graphes RDF par l'exemple (GQBE) : l'adaptation GQBE utilise un exemple de graphe RDF, considéré comme solution et fourni par les utilisateurs, pour rechercher des graphes *compatibles* avec cet exemple de solution. Nandish et al. [123, 124] ont proposé un système d'interrogation

de graphe par l'exemple dans lequel les utilisateurs donnent en entrée un graphe représentant le modèle de réponses attendues. Le système, dans une première étape, définit le graphe des voisins de la solution exemple dans le graphe de donnée. Pour réaliser cette tâche, le système parcourt des chemins d'une longueur $d = 2$ dans le graphe de données à partir des nœuds du graphe de réponse. Ensuite, le système recherche dans le graphe des données des sous- graphes ayant la même structure que le graphe exemple ou le graphe des voisins. Dans ces sous-graphes, les chemins peuvent être plus longs ou plus courts. Mais les nœuds sources et destinations de ces chemins doivent être sémantiquement proches de ceux décrits dans l'exemple de graphe solution. Cette proximité sémantique est calculée à l'aide de mesures de similarité entre entités du graphe. Enfin, les potentiels sous-graphes réponses sont classés en fonction d'un score qui est l'agrégation de la similarité entre entités et entre chemins du graphe exemple et du potentiel graphe solution. Par ailleurs, Mottin et al. [125, 126] ont proposé une déclinaison de GQBE, basée cette fois sur un exemple de requête (*examplar query*). En effet, les requêtes sur les données RDF sont des graphes comme le sont les exemples de graphes solutions de GQBE. Cette déclinaison utilise le graphe de requête comme un exemple de graphe solution, à la différence que, dans cette déclinaison, les variables du graphe de requête sont remplacées par des entités pertinentes afin d'obtenir des exemples de graphes solutions.

Interrogation des données XML par l'exemple (XQBE) : XQBE est un langage d'interrogation graphique des documents XML proposé et développé par Braga et al. [127]. Ce langage intègre l'édition graphique d'une requête en utilisant des diagrammes géométriques. Il aide les utilisateurs à formuler leurs requêtes en utilisant un exemple de réponse attendue. Une fois l'exemple décrit, XQBE le traite et génère une requête XQuery qui est exécutée. Néanmoins, Braga et al. [127] précisent que XQBE doit être utilisé, uniquement, avec des exemples de requêtes simples et découragent tout usage avec des requêtes complexes. Afin de pouvoir utiliser XQBE pour tous les types de requêtes XML, une évolution de cette adaptation a été proposée par Benzakin et al. [128].

Les AIC de suggestion de requête et d'interrogation par l'exemple aident les utilisateurs dans la construction de leurs requêtes afin que celles-ci décrivent le plus fidèlement possible leurs besoins. Ils permettent à l'utilisateur d'éviter de fausses présuppositions sur le schéma des données et souvent sur les données elles-mêmes dans le cas de l'auto-complétion avec vérification continue des réponses. Néanmoins, malgré cette assistance et ces précautions, l'exécution de la requête peut, dans certains cas, retourner des réponses non satisfaisantes. Les utilisateurs peuvent alors être amenés à ce poser les questions suivantes.

- Pourquoi n'ai-je pas obtenu l'ensemble de réponses attendues ?
- Pourquoi ai-je obtenu ces réponses ?
- Pourquoi ai-je obtenu une réponse vide ?
- Pourquoi ai-je obtenu énormément de réponses ?

Certaines approches cherchent à répondre à ces questions, afin de réduire la frustration des utilisateurs : ce sont les AIC par explications des réponses d'une requête.

2.3 Explications sur les réponses d'une requête

Les AIC présentées dans cette section ont pour but de fournir aux utilisateurs des explications aux questions qu'ils peuvent se poser devant les réponses retournées par leurs requêtes. Ces approches étaient, initialement, destinées à traiter de questions provenant de l'insatisfaction des utilisateurs. Cette insatisfaction se traduisait par la question : *pourquoi n'ai-je pas obtenu la réponse X que j'attendais ?* plus connue sous le terme *Why-Not Anwsers*. Les utilisateurs se posent cette question lorsque l'ensemble des

réponses retournées ne contient pas celles qu'ils attendaient. Ainsi, l'explication consiste à fournir aux utilisateurs les raisons pour lesquelles les réponses attendues ne se trouvent pas dans l'ensemble des réponses retournées. Par la suite, d'autre types d'explications de réponses de requête ont été développées. Il s'agit de l'explication de la présence de certaines réponses, inattendues, dans l'ensemble de réponses retournées *(Why Answers ?)*, de l'explication des réponses vides *(Why Empty Answers ?)* et de l'explication des réponses pléthoriques *(Why Too Many Answers ?)*. Cette section présente ces différentes approches d'explication des réponses d'une requête.

2.3.1 Explication de l'absence de réponses : Why-Not Answer

L'explication de l'absence des réponses d'une requête dépend du type de requête et donc du type de données sur lesquelles portent la requête. Nous présentons quelques travaux sur cette approche d'une part pour les requêtes SQL et, d'autre part, pour les autres types de requêtes.

2.3.1.1 Explication de l'absence de réponses pour les requêtes SQL

Pour expliquer l'absence de réponses attendues par les utilisateurs parmi les réponses retournées d'une requête SQL, trois principales familles de techniques ont été développées. Ces familles de techniques sont *les manipulations de la requête*, *les causes de l'absence des réponses attendues* dans la base de données et la troisième, dite *hybride*, est une combinaison des deux précédentes.

Famille de techniques basées sur les *manipulations* de la requête : *une manipulation* est une opération atomique du plan d'exécution de la requête. Cette famille d'explications considère les opérations du calcul relationnel comme atomiques. Les principales opérations atomiques sont la sélection (S), la projection (P), la jointure (J), l'union (U), l'agrégation (A) et la différence (D) : d'où l'acronyme (SPJUAD) pour les identifier. Le principe de ces techniques d'explications consiste à décomposer la requête sous forme de combinaison d'opérations atomiques et à trouver celles qui sont responsables de l'exclusion des réponses désirées par les utilisateurs pendant l'évaluation de la requête. Prendre en compte tous les opérateurs relationnels n'étant pas simple, Chapman et Jagadish [129] ont proposé une implémentation de l'explication des réponses d'une requête qui ne considère que les opérations de sélection, de projection et de jointure (SPJ). Herschel et Mauricio [130] ont proposé, plus tard, une extension de cette technique qui intègre les opérateurs Union et Agrégation (SPJUA). Ces techniques sont également nommées *explications basées sur la requête (Query based explanation)* car elles utilisent uniquement la requête pour expliquer l'absence des réponses attendues par les utilisateurs.

Les causes de l'absence des réponses attendues dans les données : cette famille d'explications considère que les réponses attendues par les utilisateurs sont absentes de l'ensemble des réponses retournées par la requête parce qu'elles ne sont pas dans la base de données. Ces techniques cherchent donc l'ensemble des modifications de la base de données qui ont causé l'absence des réponses attendues dans la base de données. Pour cela, ces techniques considèrent les opérations classiques de modifications d'une base de données : insertion, suppression et mise à jour. Huang et al. [131] nomment ce problème *Why-Not provenance* et ont développé un algorithme pour déterminer les modifications de la base de données qui ont provoqué l'absence des réponses attendues. Cette famille est également connue sous le nom *Instance-based* parce qu'elle est basée sur le contenu de la base de données.

Famille des techniques hybrides : cette troisième famille de techniques associe les explications basées sur la requête et celles basées sur les instances. Ces techniques ne sont pas très répandues. Parmi les

principales contributions proposées dans cette famille de techniques, nous trouvons celles de Herschel [132].

2.3.1.2 Explication de l'absence de réponses pour les autres types de requêtes

L'AIC *Why-Not Answers* est également utilisée pour d'autres langages de requête que SQL. Pour l'interrogation par mot-clés, Chen et al. [133] proposent de classer les réponses possibles à l'aide d'un algorithme dédié. Cet algorithme calcule la distance entre deux mots clés en prenant en compte la similarité entre les contextes sémantiques des deux mots clés (distance contextuelle) et la similarité lexicale entre ces deux mots clés. Cette distance est utilisée pour expliquer l'absence ou la présence d'une réponse parmi celles retournées. Dans les systèmes experts, les techniques développées dans le contexte du *Why-Not Answers* utilisent la connaissance modélisée dans les ontologies. Balder et al. [134] développent ainsi une approche exploitant les ontologies pour expliquer l'absence de réponses. Ces explications sont plus compréhensibles par les utilisateurs, car elles intègrent la sémantique explicitée dans les ontologies. Enfin, pour les graphes de données RDF, Islam et al. [135] et Siyu et al. [136] ont proposé des algorithmes pour l'explication des réponses. L'approche d'interrogation coopérative d'explication de l'absence de réponses attendues par les utilisateurs peut être exploitée pour éviter aux utilisateurs un processus fastidieux de modification/ré-exécution [129]. Ces explications peuvent donc servir pour aider les utilisateurs à retrouver les réponses qu'ils désirent.

2.3.2 Réparations basées sur les explications du Why Not Answers

La *réparation d'une requête* est le processus qui permet aux utilisateurs d'être satisfait des réponses retournées. Pour les requêtes SQL, trois techniques de réparation basées sur le *Why-Not Answers* ont été proposées. La première technique est *la redéfinition de la requête par les utilisateurs* en prenant en compte des explications fournies par le *Why-Not Answers*. Elle génère des requêtes redéfinies retournant les réponses désirées par les utilisateurs, ensuite les ordonne à l'aide d'un score avant de les proposer enfin aux utilisateurs [137, 138]. Dans leur implémentation de cette technique, Zhang et al. [139] calculent ce score en fonction des préférences des utilisateurs. Dans la seconde technique, les opérations atomiques ou manipulations qui sont en cause de l'exclusion des réponses attendues par les utilisateurs sont présentées à ces derniers, afin qu'ils opèrent manuellement une redéfinition [140]. Enfin, la troisième technique consiste à appliquer la plus petite transformation sur les sources de données, afin d'obtenir les réponses désirées parmi celles retournées [131]. Pour autant que nous le sachions, *la réparation des requêtes* basées sur les explications du *Why-Not Answers* n'a pas été proposée dans d'autres contextes que les bases de données relationnelles.

Le tableau 2.3 donne une synthèse des contributions sur l'explication du *Why-Not Answers* et sur la réparation des requêtes. Ces contributions portent sur différents types de requêtes tels que les requêtes SQL, les requêtes conjonctives (RC) avec des relations de comparaisons et les patrons de graphes basiques de SPARQL. Nous retrouvons également une variante de l'approche d'explication par manipulation de la requête : *l'approche polynomiale*. Cette approche est basée sur les contraintes de la requête sans considérer le plan d'exécution. Dans cette approche, une contrainte peut être impliquée dans plusieurs explications dont le nombre correspondra à son degré : d'où l'analogie avec les polynômes.

2.3.3 Autres explications des réponses d'une requête

Hormis la question *pourquoi n'ai-je pas obtenu les réponses que j'attendais parmi les réponses retournées par la requête*, les utilisateurs peuvent également se poser d'autres questions selon le contexte.

Algorithmes	Format des données	Type de requêtes	Approche d'explication	Approche de reparation	Expéri-mentation
Chapman et al. [129]	Relationnelles	SQL	Query based SPJU	-	oui
Bidoit et al. [140]	Relationnelles	RC avec inégalités	Query based polynomial	-	oui
Artemis [130]	Relationnelles	SQL	Instance-based	-	oui
NedExplain [141]	Relationnelles	SQL	Query based SPJUA	-	oui
EFQ [137]	Relationnelles	SQL	Query based polynomial	Ranked queries redefined	non
ConQueR [138]	Relationnelles	RC avec inégalités	Query based SPJA	Ranked queries redefined	oui
TALOS [142]	Relationnelles	SQL	Query based SPJ	Ranked queries redefined	oui
Jiansheng et al. [131]	Relationnelles	SQL	Instance-based SPJ	-	oui
Conseil [132]	Relationnelles	SQL	Hybrid SPJAD	User redifinition	oui
Chen et al. [133]	Web geo-tagged data	Mots clés	-	-	oui
Cate et al. [134]	Ontology + data	RC avec comparaison	Ontology based	-	non
ANNA [136]	RDF Data	Basic Graph pattern	Query based	-	non
Islam et al. [135]	Graph Matching	Basic Graph pattern	-	-	oui

TABLE 2.3 – Contributions sur les explications par le Why-Not Answers

L'explication de l'absence des réponses attendues d'une requête a donc été étendue, afin de proposer aux utilisateurs des explications pour d'autres questions. Nous identifions dans la littérature les trois autres questions suivantes que les utilisateurs peuvent se poser et pour lesquelles ils ont besoin d'explications.

Pourquoi ai-je obtenu ces réponses (Why Answers) ? Les explications données aux utilisateurs pour cette question sont les contraintes de sélection qui extraient ces réponses dans l'ensemble des données pendant l'exécution de la requête. Cette question se pose lorsque les requêtes retournent des réponses inattendues et parfois insatisfaisantes aux utilisateurs. Ainsi, le *Why Answers* permet d'expliquer aux utilisateurs pourquoi ils ont obtenu des réponses données contrairement au *Why-Not Answers*. Ces deux questions sont ainsi considérées comme duales. Plusieurs travaux ont proposé des techniques pour répondre à cette question et fournir des explications. Islam [143] propose de répondre à cette question avec les opérateurs atomiques (SPJ) de manipulation présentés précédemment. Pour les bases de données relationnelles associées à une ontologie, Glavic et al. [144] généralisent les contraintes de la requête afin de trouver plus efficacement les explications. Par exemple, si une contrainte généralisée ne permet pas la sélection d'une réponse, alors la contrainte initiale ne le permet pas non plus. Glavic et al. [144] répondent ainsi au *Why Answers* en testant les contraintes des plus générales au moins générales.

Pourquoi aucune réponse (Why Empty Answers) / Pourquoi autant de réponses (Why So Many Answers) ? Ces deux questions sont des cas particuliers des questions *Why-Not Answers* et *Why Answers*. Les utilisateurs se posent ces questions lorsque la requête ne retourne pas de réponse (*Empty Answer*) ou un nombre important de réponses (*Many Answers*). Bosc et al. [145] ont proposé des techniques de recherche d'explications pour ces questions dans les bases de données relationnelles. Ils ont également proposé des algorithmes pour réparer la requête en exploitant ces explications. La réparation consiste dans ce cas, à pouvoir retourner des réponses pour les requêtes qui retournent un ensemble vide de réponse ou à réduire les réponses en gardant les plus satisfaisantes dans le cas des réponses pléthoriques. Pour les données RDF, des travaux de Vasilyeva et al. [146] proposent des algorithmes pour répondre à ces questions. Ils calculent le sous-graphe du graphe de requête responsable de la réponse vide ou de la réponse pléthorique selon le cas.

En résumé, le *Why-Not Answers* permet de donner des explications aux utilisateurs sur le contenu des réponses renvoyées par leurs requêtes. Il en est de même pour *Why Answers*, *Why Empty Answers* et *Why So Many answers*. Dès lors que le système retourne ces explications, un processus de réparation de la requête peut être démarré afin de retourner des réponses satisfaisantes pour les utilisateurs. La réparation des requêtes peut se faire par une redéfinition de la requête par les utilisateurs : c'est une AIC plus connue sous le nom de *Users refinement*.

2.4 Raffinement de la requête par l'utilisateur (Users refinement)

L'AIC de raffinement de la requête par l'utilisateur (*Users refinement*) permet avant tout la réparation de la requête. Elle permet aux utilisateurs de redéfinir leurs requêtes de façon itérative jusqu'à être satisfait par les réponses retournées. Nous rappelons que l'ensemble des réponses peuvent ne pas satisfaire les utilisateurs parce qu'il ne contient pas les réponses attendues ou parce qu'il contient des réponses indésirables [143]. Les utilisateurs peuvent également être insatisfaits du nombre de réponses retournées [147]. Dans tous ces cas, la requête doit être réparée et le *Users refinement* peut le faire de deux façons :

- les techniques interactives font intervenir les utilisateurs et exploitent leurs retours ;
- les techniques automatiques sont basées sur les requêtes, les données et une modélisation de l'utilisateur.

2.4.1 Raffinement interactif

Islam et al. [148] proposent une solution interactive dans laquelle les utilisateurs sont sollicités pour annoter les réponses retournées. Ces annotations ou informations complémentaires sont appelées *User Feedbacks*. Elles sont utilisées pour raffiner la requête afin de retourner les réponses qui satisfont le mieux les utilisateurs. La figure 2.2 montre l'architecture générale de L'AIC *users refinement interactive*. Les techniques interactives possèdent deux modules importants qui servent à : (i) la modélisation des retours des utilisateurs (*Users FeedBacks*) sur les réponses et (ii) l'analyse et l'interprétation de ces retours (*Feedback analyzer*). Le premier module prépare les informations complémentaires, telles que les annotations, entrées par les utilisateurs. Le second module analyse les retours des utilisateurs et les réponses précédentes, les interprète et génère un ensemble d'instructions pour la définition de la requête. Ce module guide la redéfinition des requêtes. Il existe plusieurs *feedback utilisateurs*, à titre d'exemple, nous pouvons citer :

- les réponses attendues par les utilisateurs mais qui sont absentes des réponses de la requête [149] ;
- les réponses inattendues retournées par la requête mais non désirées par les utilisateurs [149] ;
- la hiérarchisation des contraintes pour définir implicitement un ordre de redéfinition ;

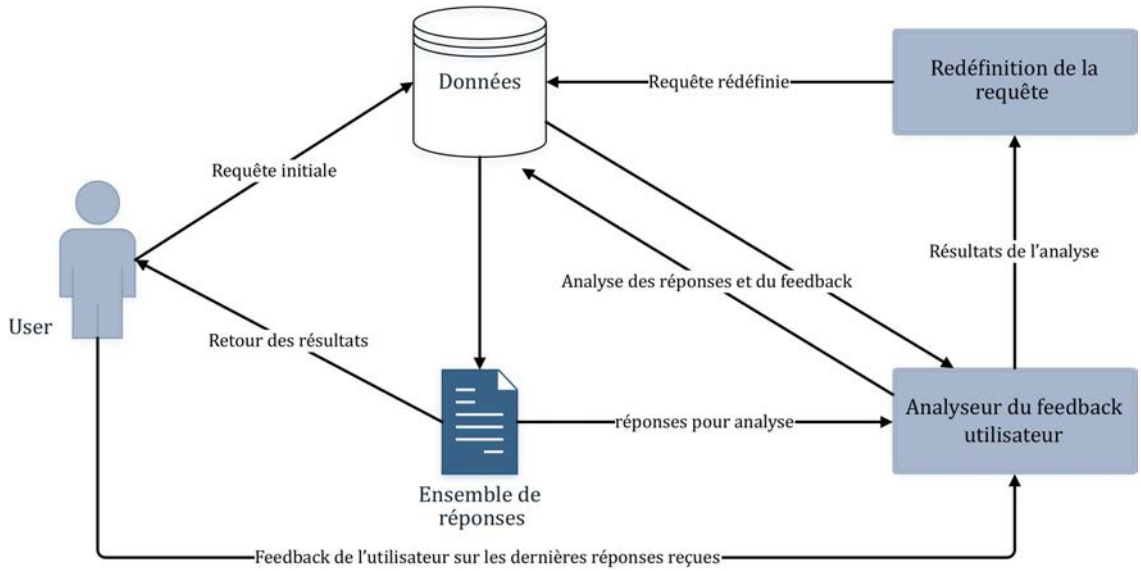


FIGURE 2.2 – Architecture de l'approche coopérative avec Feedbacks utilisateur (inspirée de [148])

— la catégorisation des contraintes en deux grands groupes, celles qui peuvent être modifiées (soft) et celles qui ne peuvent pas être modifiées (hard).

Le raffinement interactif de la requête par les utilisateurs pourrait néanmoins être limité par le nombre de feedbacks nécessaire pour obtenir des réponses satisfaisantes pour les utilisateurs. Si ce processus est très long, les utilisateurs perdront patience et seront frustrés. En effet, à force d'être trop sollicités les utilisateurs finiront par être agacés, ce qui peut les conduire à interrompre le processus. De plus, capturer ces sentiments afin d'implémenter un système de prévention et d'anticipation pour les utilisateurs est un problème très complexe. L'une des solutions qui a donc été proposée est l'automatisation du processus : le raffinement automatique de requête.

2.4.2 Raffinement automatique

Le principe fondamental du raffinement automatique de la requête est la substitution de l'utilisateur par son modèle. En effet, afin d'éviter de solliciter l'utilisateur, l'idée de modéliser le comportement de ce dernier a été proposée. Ainsi, les utilisateurs physiques sont remplacés par des modèles décrivant leurs préférences, leurs besoins, leurs contextes et bien d'autres éléments [150] qui dépendent de la modélisation. Koutrika et Ioannidis [151] exploitent le modèle de l'utilisateur afin de personnaliser la requête de ce dernier en retrouvant les *feedback* qu'il aurait renvoyés. La construction du modèle utilisateur se fait de plusieurs façons : à partir de ses préférences, de l'apprentissage automatique, des interactions de l'utilisateurs avec le système, etc. Par exemple, Lidong et al. [152] utilisent les fichiers logs des utilisateurs et Jiaul et al. [153] exploitent les annotations des utilisateurs sur les données. Plusieurs contributions ont proposé des algorithmes pour construire un modèle utilisateur [154, 155], parmi lesquelles les approches génériques [156] et les modèles pour le e-commerce [157]. De même, il existe plusieurs représentations des modèles utilisateurs comme par exemple celui proposé par Jelassi et al. [158] qui se base sur des folksonomies, qui sont des triplets de la forme $(user,\ tag,\ ressources)$, pour représenter le modèle utilisateur et réaliser des recommandations.

Le tableau 2.4 classifie les contributions sur l'AIC de raffinement par l'utilisateur *(user refinement)*. Cette

approche a également été développée pour d'autres types de données, et de requêtes (SQL sous forme normale conjonctive (FNC) ou forme normale disjonctive (FND)).

Approches	Format des données	Type de requête	Type d'approche	Type de feedback	Expérimentation
Ismael et al. [148]	Relationnelles	SQL (FNC/FND)	Interactive	Tuples	oui
FlexIQ [159]	Relationnelles	SQL (FNC/FND)	Interactive avec Skyline [12]	Tuples	oui
KISS[160]	Images	Mots clés sur metadata	Automatique	Préférences et Machine learning	oui
SFS[161]	Relationnelles	SQL	Automatique	Préférences utilisateurs	oui
DFA[162]	XML	XML Path	Interactive	Soft and Hard constraint	oui
Mishra et al. [147]	Relationnelles	SQL SPJ	Interactive	Hiérarchiser les contraintes	non

TABLE 2.4 – Algorithmes pour l'approche coopérative par raffinement de l'utilisateur

Les deux AIC précédentes se basent sur *la redéfinition de la requête*. Dans l'explication des réponses d'une requête, la redéfinition de la requête utilise les explications trouvées. Dans le raffinement par l'utilisateur, la redéfinition exploite les *feedbacks* de l'utilisateur. Mais que faire si nous ne disposons pas d'explications et si nous ne pouvons pas obtenir de retours des utilisateurs ? Cette question trouve sa réponse dans la dernière AIC que nous avons relevée dans la littérature : *la redéfinition des requêtes*. Cette AIC comprend notamment la relaxation de requêtes sur laquelle porte nos travaux.

2.5 Redéfinition des requêtes

La redéfinition d'une requête *reformule les contraintes* de cette requête afin de retourner des réponses satisfaisantes. Il existe trois types de reformulation de requête, ou *modification de requête* [163] :

- *la restriction de la requête* : la requête issue de cette modification est plus restrictive que la requête initiale. Toutes ses réponses sont contenues dans celles de la requête initiale ;
- *la relaxation de la requête* : la requête issue de cette modification est moins contraignante que la requête initiale. Ses réponses incluent toutes celles de la requête initiale et en rajoutent ;
- *l'approximation de la requête* : la requête issue de cette modification est une approximation de la requête initiale. Elle contient une partie des réponses de la requête initiale et rajoute d'autres réponses qui ne sont pas inclues dans celles de la requête initiale.

La figure 2.3 schématise les différents types de modification des requêtes. La requête Q est modifiée pour donner la requête Q' et chaque cercle correspond à l'ensemble des réponses de chaque requête.

Remarque. Il existe une relation entre l'approximation et la relaxation de requête. Certains travaux existants exploitent l'approximation d'une requête pour réaliser la relaxation de cette requête. En effet, si

12. le skyline de la requête retourne l'ensemble des tuples qui ne sont pas *dominés* (au sens de Pareto) par d'autres.

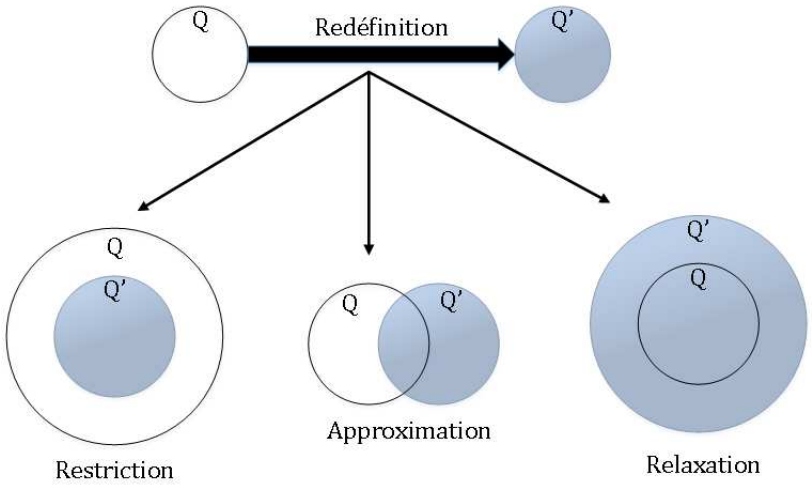


FIGURE 2.3 – Approche de modification de requête (source [163])

nous considérons une requête Q et Q' son approximation, alors toutes les réponses de Q ne sont pas incluses dans celles de Q' et vis-versa. Maintenant, pour Q" = Q ∪ Q' alors Q" est une relaxation de Q car elle contient toutes les réponses de Q et en ajoute d'autres. De même, Q" est une relaxation de Q'. Nous verrons dans nos contributions que nous avons proposé des opérateurs de relaxation et d'approximation.

La modification des requêtes se fait en général en deux grandes phases :

1. la phase de transformation. Elle utilise des techniques de transformation de contraintes pour modifier ces dernières selon *les objectifs du processus de modification* ;
2. la phase de classement. Elle utilise des mesures pour identifier, parmi les requêtes modifiées, celles qui remplissent le mieux *les objectifs du processus de modification*.

Remarque. Les objectifs d'un processus de modification peuvent être, par exemple, (i) la réduction du nombre de résultats, dans le cas des requêtes qui retournent des réponses pléthoriques, (ii) l'augmentation du nombre de réponses, tout en restant le plus proche possible des besoins des utilisateurs, dans le cas des réponses vides.

2.5.1 Techniques de transformation

Dans la littérature nous avons relevé deux principales techniques de transformation des contraintes.

La modification de contenu : Abdeslame [164] présente cette modification de requête comme une transformation progressive des constantes utilisées pour formuler les contraintes. Le processus de transformation est guidé par l'objectif de la redéfinition de la requête qui peut être soit la restriction, soit l'approximation ou encore la relaxation. Cette modification est plus intuitive pour les valeurs de type simple telles que les valeurs de type numérique, car elle sont munies d'une relation d'ordre.

La modification de structure : elle transforme la requête [164] en se basant sur la structure des données et de la requête initiale. Pour réaliser cette transformation, le système doit accéder au schéma des données ou au modèle conceptuel des données. Afin de rendre accessible le schéma de donnée pour des bases de données relationnelles, Wesley et al. [165] implémentent un *Type Abstrait Hiérarchique (TAH)* pour stocker et manipuler des concepts du modèle conceptuel des données. Le TAH est ensuite utilisé pour modifier la structure de la requête en opérant des substitutions entre des concepts. Par exemple, la table

FullPofessor dans une requête pourra être substituée par la table *Professor* si le concept *Professor* généralise le concept *FullPofessor* dans le TAH. Cette substitution est également valable pour les colonnes des tables contenues dans la requête. De leur côté, Fayech et Ounalli [166] ont exploité les ontologies comme modèles conceptuels des données pour réaliser la modification de structure des requêtes SQL. D'autres travaux proposent la suppression des jointures comme modification de la structure de la requête [167] pour retrouver des réponses pertinentes pour les utilisateurs.

Remarque. Avec la formalisation proposée par les ontologies basées sur des hiérarchies, la transformation basée sur la structure des données devient de plus en plus exploitée pour la redéfinition des requêtes. Par ailleurs, certains travaux réalisent une modification de requête en se basant sur des données des utilisateurs. Cela montre bien que la frontière entre la redéfinition des requêtes et le raffinement automatique des requêtes est très mince et souvent ces deux approches se confondent. Nous détaillons ces techniques dans le chapitre suivant.

2.5.2 Mesures de classement

La transformation des requêtes génère, très souvent, plusieurs requêtes modifiées. Afin de trier ces dernières et de ne considérer que celles qui sont *"les plus proches"* des besoins exprimés par les utilisateurs, les approches par redéfinition des requêtes utilisent des mesures dites de similarité. Ces mesures permettent de simuler l'intelligence humaine dans l'évaluation de la similarité entre deux éléments. Ces mesures évaluent la *proximité sémantique* entre la requête initiale et celle modifiée. La conception des mesures de similarité dépend de la technique de transformation et prend en considération plusieurs autres paramètres qui varient d'un environnement à un autre. Harispe [168] a réalisé une analyse approfondie et détaillée des mesures de similarité, de leur conception à leur implémentation. Il a également décrit leur contexte d'utilisation. Dans notre travail, nous présentons les mesures de similarités qui interviennent dans la redéfinition des requêtes. Elles évaluent la similarité entre requêtes et non entre données, même si la similarité entre les données peut être utilisée dans leurs calculs. Les mesures de similarité ont tous un point commun : ce sont tous des outils mathématiques utilisés pour estimer quantitativement et/ou qualitativement la proximité sémantique par comparaison des informations explicitement et/ou implicitement décrites par les éléments comparés [168].

Une mesure de similarité peut être définie comme une fonction de $E \times E \rightarrow \mathbb{R}$ qui prend deux éléments de l'ensemble E et renvoie un nombre réel correspondant à leur similarité. En tant qu'outil mathématique, les mesures de similarité doivent respecter certaines propriétés, dont une partie héritée des distances mathématiques. Pour les mesures de similarité, l'ensemble de leurs propriétés est présenté dans le tableau 2.5 où V est l'ensemble des valeurs possibles de la similarité, 0_V la valeur neutre de cet ensemble et max_V la valeur maximale.

Propriétés	Définitions
Non-négative	$sim(x, y) \geq 0_V$
Symétrie	$sim(x, y) = sim(y, x)$
Réflexive	$sim(x, x) = max_V$
Normalisée	$V = [0, 1]$
Éléments indiscernables	$sim(x, y) = max_V \; ssi \; x = y$
Intégrité	$sim(x, y) \leq sim(x, x)$

TABLE 2.5 – Propriétés caractéristiques des mesures de similarité (source [168])

Des bibliothèques de mesures de similarité entre valeurs numériques, chaînes de caractères, classes des

hiérarchies, nœuds de graphes, et bien d'autres types de données ont été développées. Citons comme exemples, la bibliothèque simPack [13] (Similarity Package) [169] et SML [14] (Semantic Measures Library & ToolKit) [170], une bibliothèque de mesures de similarité dans le domaine biomédical. Dans le chapitre suivant, nous présentons pour les techniques de transformation, les mesures de similarité associées.

Les AIC présentées dans ce chapitre, à savoir, la suggestion, l'interrogation par l'exemple, l'explication, le raffinement par l'utilisateur et la redéfinition des requêtes permettent, à différentes étapes du processus d'interrogation, d'assister d'une façon précise et particulière les utilisateurs. Certaines contributions sur ces AIC ont été intégrées dans des systèmes de recherche de données. Les systèmes de recommandation sont un exemple de cette intégration. Ces systèmes sont orientés utilisateurs et leur principal objectif est d'aider les utilisateurs à trouver des réponses à leurs besoins dans un ensemble de données connues ou inconnues. La section suivante présente les systèmes de recommandation et leurs principales approches de recommandation.

2.6 Systèmes de recommandation

Les systèmes de recommandation sont généralement définis comme des modules exploités par des applications, notamment ceux du E-commerce (Amazon, Netflix, etc.), pour suggérer des produits aux utilisateurs et/ou fournir des informations qui aideront les utilisateurs dans leur prise de décision. Les systèmes de recommandations sont utilisés dans divers domaines tels que les loisirs, le cinéma, les voyages [171] et le tourisme [172]. Ils utilisent une ou plusieurs sources de données pour effectuer la suggestion et/ou la recommandation [173]. Pour réaliser ces tâches, les systèmes de recommandation extraient et stockent toutes les données dans une seule table, dans laquelle chaque ligne correspond à un produit et chaque colonne ou attribut à une propriété descriptive du produit. Il existe trois principales approches de recommandation : l'approche collaborative (Collaborative filtering), l'approche basée sur les données (Content-based filtering) et l'approche basée sur la connaissance (Knowledge-based filtering) [173]. Ces approches appliquent et/ou adaptent des AIC pour recommander des produits aux utilisateurs.

2.6.1 Filtrage collaboratif

Cette approche se base sur les caractéristiques des utilisateurs pour qui la recommandation est effectuée et non sur les données. Elle exploite les actions des utilisateurs telles que l'historique des requêtes et l'évaluation sous forme d'annotations des produits par les utilisateurs [174]. Cette approche est aussi nommée *social filtering*. On distingue deux versions, l'une centrée sur l'utilisateur , *user-based*, et l'autre centrée sur les attributs du produit, *item-based*.

— Dans la version *user-based*, le système cherche les *k-meilleures* approximations de l'utilisateur parmi la population des utilisateurs existants et utilise les recommandations de ces utilisateurs pour suggérer des produits à l'utilisateur courant.
— Dans la version *item-based*, le système utilise les préférences de l'utilisateur pour réaliser une substitution entre les attributs similaires selon l'évaluation de l'utilisateur. Cette version peut également supprimer certains attributs dans la requête.

Comme nous pouvons l'observer, les approches collaboratives des systèmes de recommandation utilisent les principes des AIC suivants.

— Suggestion : ces approches suggèrent des réponses aux utilisateurs.

13. http://www.ifi.unizh.ch/ddis/simpack
14. http://www.semantic-measures-library.org/sml/

— Raffinement de requête par l'utilisateur : les versions *user-based* et *item-based* utilisent les préférences des utilisateurs pour modifier la requête avant de l'exécuter.
— Redéfinition des requêtes : elle est utilisée dans la substitution et/ou la suppression des attributs dans la requête.

La recommandation par filtrage collaboratif a néanmoins deux limites : d'une part les utilisateurs isolés et d'autre part le manque d'évaluation des produits. Un utilisateur est isolé s'il possède un profil qui n'est similaire à aucun autre et ne peut donc bénéficier des recommandations des autres utilisateurs. Le manque d'évaluation des produits survient lorsque les utilisateurs n'ont pas encore évalué les attributs contenus dans les contraintes de la requête. Le cas le plus fréquent d'apparition de cette limite est le démarrage à froid (cold-start) lorsque les utilisateurs utilisent le système pour la première fois.

2.6.2 Filtrage basé sur le contenu

Cette approche évalue la similarité entre le contenu des attributs des produits et les préférences des utilisateurs. Elle utilise le relâchement de valeur ou la substitution pour recommander des produits à l'utilisateur. Elle compare également, en fonction des utilisateurs, les attributs qui caractérisent différents produits afin de mesurer la similarité entre ces produits. La modification de contenu et la modification de structure sont donc exploitées dans cette approche pour suggérer des réponses pertinentes aux utilisateurs. La mesure de similarité utilisée est une agrégation des mesures de similarité associées aux transformations de contenu et de structure utilisées. Cette mesure permet d'ordonner les recommandations en fonction de l'utilisateur [172, 175].

2.6.3 Filtrage basé sur la connaissance

Dans cette approche, la sémantique des données sous forme d'ontologie est utilisée pour faire la recommandation. Cette approche utilise l'ontologie pour réaliser la redéfinition des requêtes. Elle utilise notamment les modifications de structure sur les attributs en s'appuyant sur la structure de l'ontologie [176]. Cette approche utilise également le raisonnement pour effectuer des substitutions entre des attributs et des données similaires.

Conclusion

Les AIC ont été développées pour aider les utilisateurs dans l'interrogation des données. Avec l'évolution des technologies, des volumes de données hétérogènes de plus en plus importants sont produits, ce qui rend la recherche des données plus complexe. Face à cette complexité dans l'interrogation des données, les utilisateurs ne savent très souvent pas les actions à mettre en œuvre pour obtenir les résultats qui satisferont leurs besoins. Au contraire, certains utilisateurs font des suppositions ou adoptent des comportements qui conduisent, inévitablement, à des requêtes insatisfaisantes. Jack Minker [86] a analysé et recensé certains comportements utilisateurs, qu'il serait important de prendre en considération afin d'apporter une assistance efficace aux utilisateurs pendant l'interrogation des données. Ces comportements sont : les suppositions des utilisateurs sur les schémas des données (*présupposition*), les suppositions des utilisateurs sur les données (*misconceptions*), la satisfaction des utilisateurs pour les réponses, la formulation des besoins par l'utilisateur et les réponses alternatives pour les utilisateurs. Ainsi, les AIC, qui ont pour objectif la simplification du processus d'interrogation des données via l'assistance des utilisateurs, doivent prendre en compte ces différents aspects du comportement des utilisateurs. Dans ce chapitre, nous avons présenté cinq AIC qui aident les utilisateurs soit dans la formulation de la requête, soit par l'analyse de cette dernière ou encore pendant l'exécution de la requête. Ces AIC sont : la suggestion de

requêtes (*Query Suggestion*), la construction des requêtes par l'exemple (*Query by Example*), l'explication des réponses des requêtes (*Query Explanation*), le raffinement des requêtes par les utilisateurs (*User refinement*) et la redéfinition des requêtes (*Query Relaxation*). Les deux premières interviennent dans la formulation de la requête tandis que les trois dernières interviennent dans le traitement de la requête. L'évaluation de ces approches suivant les comportements des utilisateurs relevés par Jack Minker [86] donne le tableau 2.6.

Approches coopératives	Présuppositions (Schéma)	Misconceptions (données)	Formulation des besoins	Satisfaction utilisateur	Réponses alternatives
Query Suggestion	✓	✓	✓	✗	✗
Query by Example	✓	✗	✓	✗	✗
Query Explanation	✓	✓	✗	✓	✗
User Refinement	✗	✗	✗	✓	✓
Query Relaxation	✓	✓	✗	✓	✓

TABLE 2.6 – Récapitulatifs des aspects considérés dans chaque approche coopérative

Ce tableau ne compare pas les différentes approches, car ces dernières ne sont pas toutes destinées à un même ensemble d'utilisateurs. Tandis que la suggestion de requête et la requête par l'exemple ciblent plus les utilisateurs sans aucune connaissance sur les données et même le langage de requête, les autres approches sont dédiées aux utilisateurs possédant cette connaissance et une expertise dans le langage d'interrogation. Ce tableau permet cependant de connaître les AIC qui considèrent un nombre important de comportements des utilisateurs qui peuvent influencer la qualité des réponses retournées. La redéfinition de requête est celle qui considère le maximum de comportements. C'est pour cette raison que nous avons choisie de travailler sur la redéfinition des requêtes et principalement sur la relaxation. Nous réalisons dans un premier temps un état de l'art de la relaxation de requêtes dans le chapitre suivant.

De plus, nous constatons dans ce chapitre que les contributions sur les AIC ne sont pas aussi riches et nombreuses pour les bases de données RDF qu'elles le sont pour les données relationnelles. Pourtant, le raisonnement qu'offrent les données RDF peut permettre une redéfinition des requêtes. Ainsi, nos travaux se focalisent sur la redéfinition des requêtes dans le contexte des données RDF. Nous développons également l'approche coopérative par suggestion de requête, à travers l'implémentation d'un outil d'interrogation graphique, intuitif, et qui intègre également le raffinement des requêtes par les utilisateurs via des opérateurs dits de relaxation. De plus, nous proposons des explications de type *why empty answer* pour des requêtes SPARQL qui ne retournent aucune réponse et proposons un processus de relaxation pour ces requêtes. Ces contributions sont regroupées dans un framework et détaillées dans les chapitres 4, 5 et 6. Le framework d'interrogation coopératif que nous proposons intègre ainsi plusieurs AIC tout comme les systèmes de recommandation présentés dans ce chapitre.

Chapitre 3

État de l'art sur la relaxation de requêtes

Sommaire

Résumé : Comme nous l'avons vu dans le chapitre précédent, la relaxation de requêtes est une approche coopérative qui consiste à transformer des requêtes, en situation d'échec ou d'insatisfaction, en de nouvelles requêtes moins contraignantes et ainsi plus susceptibles de retourner des réponses. Selon le type de données interrogées, différentes techniques de relaxation de requêtes ont été proposées. Dans ce chapitre, nous réalisons un état de l'art des différentes techniques de relaxation proposées dans les principaux types de bases de données. Il s'agit entre autres des bases de données relationnelles, des systèmes de recommandation et des bases de données RDF. Nous menons également une analyse de ces techniques de relaxation suivant différents axes qui sont : les techniques de transformation de la requête, les mesures de similarité, les processus de relaxation et l'intégration de la relaxation dans les langages de requêtes ou *query languages (QL)*. Les objectifs de ce chapitre sont la réalisation d'une synthèse des contributions sur la relaxation de requêtes et l'identification des insuffisances et des limites de ces contributions dans le contexte des bases de données RDF.

Introduction

Parmi les approches coopératives présentées dans le chapitre précédent, la redéfinition des requêtes est celle qui prend en considération le plus grand nombre de comportements utilisateurs nécessitant un traitement coopératif. Bien que la redéfinition des requêtes n'assiste pas les utilisateurs dans la formulation de leurs besoins, cette approche coopérative permet de réparer les fausses présuppositions des utilisateurs sur le schéma (*présuppositions*) et les données (*misconceptions*). La réparation est réalisée par une transformation de la requête basée sur les données et/ou le schéma des données. En plus de cette transformation, la redéfinition des requêtes utilise des mesures de similarité ou de satisfaction pour, d'une part, proposer des réponses alternatives proches des besoins des utilisateurs et, d'autre part, évaluer la satisfaction des utilisateurs par rapport aux réponses alternatives retournées. La redéfinition des requêtes est, de notre point de vue, l'approche coopérative, qui est la plus appropriée pour aider les utilisateurs dans l'interrogation des données RDF car l'utilisation des autres approches coopératives présente plusieurs difficultés et limites parmi lesquelles nous pouvons citer :

- *la complexité du schéma des données et le nombre important de relations entre les données* qui rendent complexe la visualisation des données RDF à des fins d'interrogation ;
- *le volume important des données RDF* qui rend quasi-impossible une connaissance complète et précise sur le schéma et les données pour un utilisateur usuel ;
- *la diversité des utilisateurs des données RDF*. Les utilisateurs des données RDF varient entre les experts, qui maîtrisent les données RDF à interroger, les schémas et, souvent, les technologies du Web sémantique et à l'inverse les novices, qui possèdent peu de connaissances sur ces aspects ;
- *la taille importante de l'historique des requêtes*. D'après les études menées par Saleem et al. [11] et Arias et al. [14], le fichier *Log* de DBpedia s'enrichirait de plus de cinq millions de requêtes en quelques mois. Cette taille rend difficile et très coûteux l'exploitation de ces fichiers *Log* dans l'approche de suggestion de requêtes telle que nous l'avons décrite dans le chapitre précédent.

L'approche coopérative de redéfinition des requêtes se décline en trois variantes qui sont :

- *la restriction des requêtes* qui réduit le nombre de réponses d'une requête pour ne garder que les plus satisfaisantes ou pertinentes pour les utilisateurs ;
- *l'approximation des requêtes* qui retourne de nouvelles réponses à partir de requêtes sémantiquement proches de la requête utilisateur ;
- *la relaxation des requêtes* qui accroit les réponses retournées aux utilisateurs (en ajoutant des réponses à celles de la requête utilisateur).

Comme nous l'avons indiqué précédemment, l'approximation des requêtes est, dans certains contextes, une autre forme de relaxation de requête. Ainsi, par la suite, nous utilisons le terme relaxation de requête pour désigner, abusivement et indifféremment, à la fois l'approximation et la relaxation proprement dite des requêtes. L'objectif principal de la relaxation de requête est de fournir des réponses alternatives aux utilisateurs lorsque leurs requêtes retournent, par exemple, un ensemble vide, insuffisant, incomplet ou insatisfaisant de réponses. Pour retrouver ces réponses alternatives, les approches de relaxation transforment la requête utilisateur en des requêtes dites *relaxées* et exécutent ces dernières pour avoir les réponses alternatives. Dans certains cas, la recherche des réponses alternatives nécessite l'exécution de plusieurs requêtes relaxées. Dans ces cas, les techniques de transformation de requêtes et les mesures de similarité sont utilisées dans un processus appelé *processus de relaxation*.

Les techniques de transformation des requêtes diffèrent suivant la requête qui elle-même dépend du type de données interrogées, de même que les mesures de similarités et les processus de relaxation. Nous présentons dans ce chapitre les méthodes de relaxation existantes pour différents types de données. Nous analysons les techniques de transformation des requêtes, les mesures de similarités associées et

les processus de relaxation associés. Nous commençons par les méthodes de relaxation dans les bases de données relationnelles (section 1). Ensuite, nous analysons la relaxation dans les systèmes de recommandation (section 2) et enfin dans les bases de données RDF (section 3). Dans la conclusion, nous réalisons une synthèse sur les similitudes et différences entre les méthodes de relaxation dans ces types de données. Nous discutons également des possibilités d'adaptation et d'exploitation des méthodes de relaxation des autres types de données dans le cadre des bases de données RDF. Les insuffisances et limites des méthodes de relaxation existantes pour les requêtes sur des données RDF sont ainsi identifiées et constituent les problématiques auxquelles nous proposons des solutions dans les prochains chapitres.

3.1 Relaxation des requêtes relationnelles

Le besoin d'interrogation coopérative est apparu avec les difficultés d'exploitation des données dans les bases de données relationnelles [12]. Rappelons que l'interrogation des données dans des bases de données relationnelles utilise des langages dédiés appelés *langage d'interrogation de bases de données* [15] ou *langage de requête (Query language, QL)*. Ces langages offrent un ensemble d'outils aux utilisateurs pour exprimer leurs besoins sous forme de requêtes. Ces dernières permettent aux utilisateurs de réaliser des recherches de données respectant les contraintes liées à leurs besoins. Motro [177, 88] montre que l'exploitation efficace d'une base de données relationnelle nécessite quelques pré-requis parmi lesquels :

— la connaissance préalable du modèle de données ;
— la connaissance préalable du langage d'interrogation ;
— la connaissance préalable du contenu de la base de données.

Mais, ces conditions ne sont pas toujours remplies ; d'où le besoin de systèmes d'interrogation flexibles, c'est-à-dire qui permettront aux utilisateurs de pouvoir interroger la base de données même s'ils ne remplissent pas toutes les conditions précédentes. C'est avec cet objectif que plusieurs travaux ont proposé des solutions et des extensions des langages de requêtes. Dans cette section nous présentons ces solutions.

3.1.1 Requêtes avec prédicats graduels

L'une des premières méthodes d'interrogation flexible proposée repose sur les *prédicats graduels*. Bosc et Pivert [178] proposent cette solution afin de permettre à l'utilisateur de rendre les contraintes plus flexibles, en les exprimant sous la forme de *prédicats flous ou graduels*. L'idée est d'exprimer les contraintes (ou prédicats) sous forme *graduelle* plutôt que booléenne. Les prédicats graduels permettent également aux utilisateurs d'exprimer leurs préférences dans la requête. Par exemple, un utilisateur pourra désormais rechercher *"tous les **jeunes** enseignants d'une université"*. Dans cet exemple, ***jeune*** est un prédicat graduel utilisé en lieu et place d'une inégalité de la forme *"age < X"* qui pourrait être source d'insatisfaction, suivant le contenu de la base de données et la valeur de *"X"*. Les prédicats graduels permettent, d'une part, d'éviter les réponses vides ou incomplètes dues aux *contraintes très sélectives* des utilisateurs et, d'autre part, de fournir un ensemble de résultats pertinents lorsque les utilisateurs spécifient des contraintes *assez générales*, c'est-à-dire qui correspondent à une grande partie des données présentes dans la base de données. Avec les prédicats graduels, l'interrogation devient plus flexible et proche de l'interrogation dans le langage naturel.

3.1.1.1 Principe de relaxation et transformation de la requête

Le principe fondamental de la méthode de relaxation avec les prédicats graduels est la transformation de contraintes booléennes en contraintes graduelles. Bosc et al. [179] proposent l'utilisation des ensembles

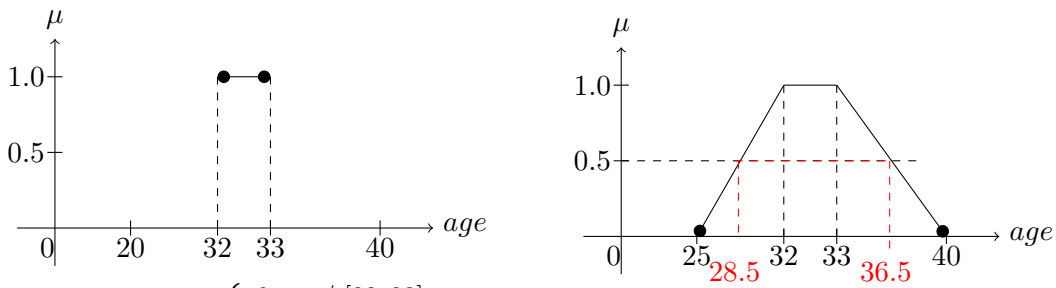
15. différent des langages de manipulation des bases de données

flous pour la définition et l'évaluation des prédicats graduels. Dans leur contribution, Bosc et al. [179] définissent la notion *d'ensemble flou* comme un ensemble, au sens de la théorie des ensembles, dont les frontières ne sont pas définies avec précision. Dans les ensembles flous, l'appartenance d'un élément n'est pas définie de manière booléenne (vrai ou faux), comme c'est la cas dans les ensembles classiques, mais par une fonction μ nommée *fonction d'appartenance*. La fonction d'appartenance μ_θ d'un ensemble flou θ associe à chaque valeur x de l'univers des valeurs possibles U, la valeur $\mu_\theta(x)$, telle que $\forall\ x \in U, 0 \leq \mu_\theta(x) \leq 1$, qui évalue l'appartenance de x à θ. Lorsque $\mu_\theta(x) = 1$ alors x appartient *pleinement* à θ et lorsque $\mu_\theta(x) = 0$, x n'appartient pas à θ. Enfin, $0 < \mu_\theta(x) < 1$ signifie une appartenance nuancée de x à θ. La fonction μ_θ peut-être discrète, monotone ou croissantes-décroissantes [179].

Ainsi, Bosc et al. [179] proposent d'exprimer les prédicats graduels tels que ***jeune*** ou ***pas cher*** en parlant d'un hôtel, avec des contraintes graduelles. Ainsi, l'ensemble des réponses d'une requête sera un ensemble flou, de données provenant de la base, dont les éléments x possèdent un degré de satisfaction $\mu(x)$ de la requête considérée. Soit le tableau 3.1 contenant les données sur des enseignants d'université.

ID	Université	Grade	Age	Ancienneté	Salaire (€)
associateProf_1	university_1	Associate Professor	31	6	3200
associateProf_2	university_1	Associate Professor	29	5	2600
fullProf_2	university_1	Full Professor	40	15	4500
fullProf_5	university_1	Full Professor	34	9	4000
graduateSt_1	university_1	Teacher Assistant	25	1	1200
assistantProf_1	university_1	Assistant Professor	28	3	1350

TABLE 3.1 – Exemple de données



(a) $\mu(x) = \begin{cases} 0 & x \notin [32, 33] \\ 1 & x \in [32, 33] \end{cases}$

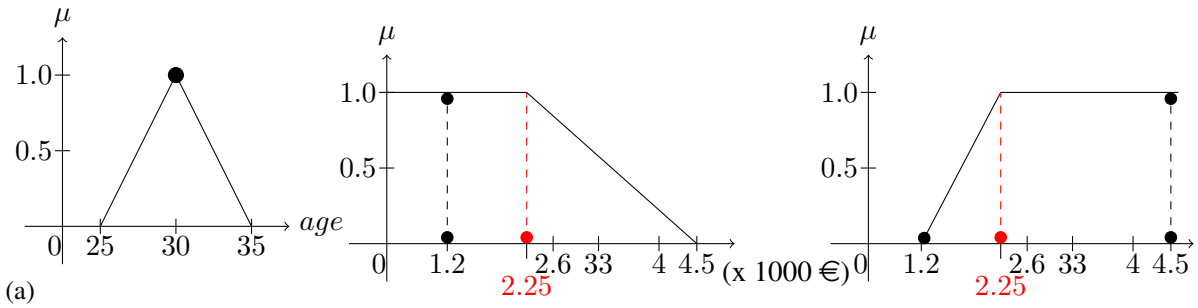
(b) Ensemble flou avec $\mu(x) \in [0, 1]$

FIGURE 3.1 – Fonction d'appartenance pour un ensemble classique (a) et flou (b)

Les prédicats graduels permettent, à défaut de chercher des enseignants ayant un âge compris entre 32 et 33 ans, de chercher des *jeunes* enseignants d'université. La fonction μ associée à ce prédicat, défini sur l'intervalle des âges de la base de données, est représentée graphiquement sur la figure 3.1b. Tandis que la figure 3.1a représente l'ensemble des réponses dans le cas de contrainte normale utilisant la théorie des ensembles classiques. Suivant la contrainte transformée en prédicat graduel, la représentation graphique de la fonction d'appartenance de ce dernier peut être un trapèze comme dans la figure 3.1b, un triangle (figure 3.2a) ou un demi-trapèze à gauche (figure 3.2b) ou à droite (figure 3.2c).

Lorsque plusieurs prédicats graduels sont utilisés dans une requête, les degrés de satisfaction doivent être agrégés afin d'avoir un ordre de satisfaction commun. Cette agrégation se fait via des opérateurs d'agré-

gation, des intégrales floues et des mesures floues [180] suivant le type de prédicats graduels utilisés. Abbaci [180] a donné une typologie de prédicats graduels : *les prédicats uni-polaires et bipolaires*.



(a)
$\mu(x) = 0 \quad x \notin [25, 35]$
$\mu(x) = 1 \quad x = 30$

(b) Trapèze ouvert à gauche

(c) Trapèze ouvert à droite

FIGURE 3.2 – Autres formes de fonctions d'appartenance

Prédicats uni-polaires : Abbaci [180] considère les prédicats uni-polaires comme des prédicats graduels comparables et basés sur une même échelle, on dit qu'ils sont *commensurables*. Par exemple, pour rechercher "*un **jeune** professeur ayant un **faible** salaire*", les prédicats ***jeune*** (figure 3.2a) et ***faible*** (figure 3.2b) sont uni-polaires. Pour ces prédicats, nous pouvons citer les techniques d'aggregation suivantes :

- la technique de Dubois et Prade [181] qui utilise les opérateurs d'agrégation de la théorie des ensembles flous tels que *min* ou le *produit* ;
- les intégrales floues telles que les intégrales de Sugeno [182] et Choquet [183] ;

Prédicats bipolaires : Tamani et al. [184] définissent un prédicat graduel bipolaire comme un prédicat contenant à la fois des conditions négatives, qui expriment des contraintes, et les conditions positives qui expriment des préférences. Prenons l'exemple de la requête qui cherche *"les professeurs **jeunes** et de préférence qui **n'ont pas un faible salaire**"*. Cette requête contient des prédicats bipolaires et elle est différente de rechercher *"les professeurs **jeunes** ayant un salaire **faible**"* (voir figures 3.2a et 3.2b) qui sont uni-polaires et surtout, il n'y a plus de distinction entre la contrainte et la préférence. Dubois et Prade [185] ont proposé des méthodes d'agrégation pour ces prédicats.

3.1.1.2 Intégration des prédicats graduels dans les langages de requêtes

Les principes sous-jacents aux prédicats graduels ayant été définis, ils doivent maintenant être intégrés dans un système ou langage d'interrogation de données afin d'être exploités par les utilisateurs. Bosc et al. [178, 179] ont proposé, dans cette optique, le langage SQLf *(SQL Fuzzy)* qui est une extention du langage d'interrogation des bases de données relationnelles SQL. SQLf permet d'évaluer les prédicats gaduels en utilisant les principes et les transformations présentés précédemment et retourne des réponses aux utilisateurs, chacune associée au degré de satisfaction correspondant. Dans SQLf, le bloc SELECT est le même que dans SQL, mais enrichi de deux éléments :

- **utilisation des prédicats graduels** : dans SQLf la clause `WHERE` reconnait et interprète des prédicats graduels ;
- **calibrage des réponses** : l'utilisateur a la possibilité de préciser dans la clause `ALPHACUT` un seuil $\alpha \in [0, 1]$ de satisfaction autorisé pour les réponses retournées [180].

Avec ces deux éléments, le bloc SELECT devient dans SQLf :

SELECT $< Attributs\ sélectionnés >$
FROM $< Relations\ contenant\ les\ données >$
WHERE $< Prédicats\ classiques\ ou\ graduels >$
ALPHACUT α.

SQLf garde ainsi toutes les fonctionnalités, les clauses et les opérateurs de SQL. En plus, il adapte les opérateurs, pour les prédicats graduels. Un exemple de requête SQLf avec prédicats graduels est :

SELECT $ID,\ Ancienneté$
FROM $Exemple\ de\ données$
WHERE $age = "jeune"\ AND\ salaire = "faible"\ AND\ Université = "university_1"$
ALPHACUT 0.6.

Smits et al. [186] ont intégré une implémentation de ce langage dans le SGBDR PostgreSQL afin de vérifier sa fiabilité et ses performances. Dans la littérature, il existe d'autres langages de requêtes qui intègrent des prédicats graduels. Nous avons par exemple le langage Fuzzy SQL [187].

3.1.1.3 Processus de relaxation

Dans le cas de la relaxation par les prédicats graduels, le processus de relaxation est intégré au processus d'évaluation de la requête. Ainsi, le processus de relaxation est lancé au moment même où un prédicat graduel est rencontré pendant l'évaluation de la requête, car le prédicat graduel est une condition à part entière de la requête. Ce processus de relaxation possède néanmoins plusieurs limites. La première est qu'il est invoqué par les utilisateurs, ce qui implique que ce dernier doit avoir une connaissance des attributs à relaxer et donc des données, ce qui n'est pas toujours le cas. De plus, en supposant qu'un utilisateur ait une connaissance correcte des données, il pourrait directement formuler sa requête SQL en tenant compte de ces données et, donc il n'aurait plus besoin de prédicats graduels. La deuxième limite est que cette relaxation ne prend pas en considération les potentielles préférences des utilisateurs sur les attributs pendant la relaxation. Dans ce processus, tous les attributs sont considérés comme étant de même importance et pertinence : on parle d'*attributs équipotents*. En plus de ces deux limites, nous constatons que la relaxation par les prédicats graduels reste uniquement applicable dans le cadre des attributs de type numérique. Par contre, pour les contraintes portant sur des attributs d'un type énuméré ou chaînes de caractères, les prédicats graduels sont difficilement exploitables.

3.1.2 Relaxation des requêtes SQL avec préférences

Afin de remédier à l'une des limites de la méthode de relaxation précédente, certains travaux proposent de prendre en compte les préférences des utilisateurs pendant la relaxation des requêtes SQL. Ces méthodes de relaxation diffèrent de l'approche coopérative de raffinement par l'utilisateur par le fait que dans celles-ci les préférences utilisateurs sont récupérées via des interactions avec l'utilisateur lui-même ou son modèle. Or, dans les techniques de relaxation avec préférences utilisateurs, ces dernières sont explicitement exprimées pendant la formulation de la requête.

3.1.2.1 Principe de relaxation et transformation de la requête

Les préférences utilisateurs sont exprimées sur les attributs et/ou les valeurs des attributs. Elles sont, par conséquent, non agrégatives. Ainsi, les travaux existants proposent deux familles de modèles de transformation de requêtes : *le modèle de Préférence* [151, 188, 189, 190] et *le modèle Skyline* [191, 192, 193].

Modèle de préférence : dans ce modèle de relaxation avec les préférences utilisateurs, Kießling [188] propose une catégorisation des contraintes d'une requête. Il propose ainsi deux catégories : les contraintes dures *(hard constraints)* et les contraintes molles *(soft constraints)*. Les contraintes dures décrivent des besoins qui, si les réponses venaient à être insatisfaisantes, ne doivent subir aucune forme de transformation . Tandis que les contraintes molles peuvent être relaxées. Kießling [188] propose pour cette relaxation, *des opérateurs de relaxation* qui relaxent les contraintes molles afin de retrouver des réponses proches de celles désirées par les utilisateurs. À la suite de ces travaux, Koutrika et Ioannidis [151] ont proposé un autre modèle de préférence avec deux principales propriétés :

— le degré de préférence utilisateur d'un attribut représenté par une valeur comprise entre -1 et 1 ;
— le degré d'intérêt porté par l'utilisateur pour la présence ou l'absence d'une valeur *v* d'un attribut donné parmi les solutions.

Koutrika et al. proposent à la fois des opérateurs pour combiner les préférences et/ou les ordonner. Ils intègrent également les préférences dans le processus d'évaluation de la requête [190].

Modèle Skyline : les requêtes Skylines sont utilisées afin de retourner les résultats les plus pertinents suivant un ensemble de critère dont les résultats sont comparables pour un même critère mais incomparable entre deux critères distincts donnés [194]. Dans la relaxation avec préférences, il s'agit de retourner les réponses préférentielles du point de vue utilisateur même si les préférences de ce dernier sont incomparables entre elles. Chomicki et al. [192] et Minolin et Chomicki [191] ont proposé de retourner les réponses préférentielles en utilisant le principe de la dominance au sens de Pareto. De leur côté, Hadjali et al. [195] proposent des requêtes skyline exploitant la théorie des ensembles flous pour relaxer les requêtes.

3.1.2.2 Intégration des préférences dans SQL et processus de relaxation

Les modèles de préférences ont été intégrés dans SQL ce qui conduit à la conception de PrefDB [190] et PreferenceSQL [189]. Pour autant que nous sachions, aucune extension du langage SQL n'a été proposée pour le modèle Skyline, même si des expérimentations sur ce modèle existent [191]. Seul l'opérateur Skyline a fait l'objet d'une intégration dans SQL par Börzsönyi [194]. Comme exemple, nous décrivons brièvement dans ce qui suit la requête `SELECT` dans PreferenceSQL.

PreferenceSQL, proposé par Kießling et al. [189], implémente des instructions pour la manipulation des préférences, principalement l'ordonnancement de ces préférences par priorité. La structure générale d'une requête dans PreferenceSQL est la suivante :

```
SELECT     < Attributs sélectionnés >
FROM       < Relations contenant les données >
WHERE      < Contraintes booléennes ou dures >
PREFERRING < Contraintes molles ou relaxées >
```

La clause **PREFERRING** contient la description de la relaxation à opérer suivant les préférences des utilisateurs. La description des préférences utilisateurs se fait avec des opérateurs selon la partie de la requête à relaxer. Nous avons des opérateurs pour la description des *préférences numériques* et *non-numériques*. Nous trouvons également des *agrégateurs de préférences* tels que la dominance de Pareto qui est utilisée par défaut pour des préférences de même importance ou la clause `CASCADE` qui permet de définir les niveaux d'importance entre les préférences. Dans la relaxation des requêtes SQL avec des préférences utilisateurs, le processus de relaxation est effectué au moment de l'exécution. Les réponses sont ensuite retournées en fonction de leur pertinence évaluée, premièrement, à partir des contraintes

obligatoires et, ensuite, à partir des préférences. Les opérateurs entre préférences sont utilisés pour déterminer l'ordre des réponses, même s'il peut arriver qu'il existe deux réponses incomparables, ce qui est le cas avec la dominance de Pareto par exemple. Dans ce cas, les deux réponses sont retournées à l'utilisateur. Cette approche présente des limites telles que la relaxation des jointures comme expliqué dans ce qui suit.

3.1.3 Relaxation des jointures

Les méthodes de relaxation précédentes se focalisent sur la relaxation des contraintes d'une requête relationnelle. Mais, dans certains cas, pour les requêtes à réponses vides principalement, il est utile d'analyser les jointures de la requête afin de voir si une possible relaxation d'une jointure ne serait pas plus pertinente que la relaxation de plusieurs autres contraintes. Nous illustrons à l'aide de l'exemple suivant ce problème et ses enjeux. Considérons les tables 3.2 et 3.3 d'une base de données contenant respectivement les *enseignements* et les *cours* dans des universités. Sur ces données, nous cherchons *les enseignants ayant un salaire compris entre 1500 € et 2500 € pour effectuer des cours d'un volume horaire compris entre 35H et 50H*, c'est-à-dire :

$$
\begin{array}{ll}
\texttt{SELECT} & P.ID,\ P.Salaire,\ C.Nom,\ C.Volume\ horaire \\
\texttt{FROM} & Professeurs\ P, Cours\ C \\
\texttt{WHERE} & 1500 < P.Salaire\ \texttt{AND}\ P.Salaire < 2500 \\
 & \quad \texttt{AND}\ P.ID = C.ID - Prof\ \texttt{AND} \\
 & \quad 35 < C.Volume\ horaire\ \texttt{AND}\ C.Volume\ horaire < 50
\end{array}
$$

ID	Université	Grade	Age	Ancienneté	Salaire(€)
associateProf$_1$	university$_1$	Associate Professor	31	6	3200
associateProf$_2$	university$_1$	Associate Professor	29	5	2600
fullProf$_2$	university$_1$	Full Professor	40	15	4500
fullProf$_5$	university$_1$	Full Professor	34	9	4000
graduateSt$_1$	university$_1$	Teacher Assistant	25	1	1200
assistantProf$_1$	university$_1$	Assistant Professor	28	3	1350

TABLE 3.2 – Professeurs

ID	Nom	Niveau	ID-Prof	Volume horaire	Type
cours$_1$	Java	M$_1$	graduateSt$_1$	24	TP
cours$_2$	Gestion Projet	L$_3$	associateProf$_1$	32	TD
cours$_3$	IDD	M$_2$	fullProf$_5$	54	CM
cours$_5$	Syst-emb	M$_2$	fullProf$_2$	54	CM
cours$_4$	OSEK/C++	M$_2$	assistantProf$_1$	26	TD

TABLE 3.3 – Cours

Cette requête exécutée sur les données des tables 3.2 et 3.3 ne retourne aucune réponse. L'usage de plusieurs tables dans une requête offre de nouvelles posibilités de relaxation basées sur les jointures. Des travaux se sont penchés sur ces possibilités, notamment ceux de Koudas et al. [167] que nous présentons dans ce qui suit.

P.ID	P.Salaire (€)	C.ID	C.Volume horaire
associateProf_1	3200	cours_2	32
fullProf_2	4500	cours_5	54
fullProf_5	4000	cours_3	54
graduateSt_1	1200	cours_1	24
assistantProf_1	1350	cours_4	26

TABLE 3.4 – Table de jointure

3.1.3.1 Principe de relaxation et transformation de la requête

Afin de prendre en compte les jointures pendant la relaxation d'une requête SQL, Koudas et al. [167] ont proposé deux algorithmes. Le principe du premier algorithme *JoinFirst* (jointure en premier) est de réaliser les jointures avant n'importe quelle autre opération sur la requête, ce qui permet de créer une table et donc de revenir au problème déjà connu de relaxer une requête sur une seule table. Une fois la table de jointure créée, une des méthodes de relaxation précédentes est appliquée sur la requête. Dans notre exemple, l'exécution de la jointure avant toutes autres opérations donne la table 3.4. Après la construction de la table de jointure, l'algorithme *JoinFirst* relaxe les contraintes de la requête en utilisant le modèle skyline vue précédemment et obtient ainsi des réponses alternatives pour les utilisateurs.

P.ID	P.Salaire
assistantProf_1	1350
associateProf_2	2600
graduateSt_1	1200

TABLE 3.5 – Tuples relaxés *Professeur*

C.ID	ID-Prof	C.Volume horaire
cours_2	associateProf_1	32
cours_3	fullProf_5	54
cours_5	fullProf_2	54
cours_4	assistantProf_1	26

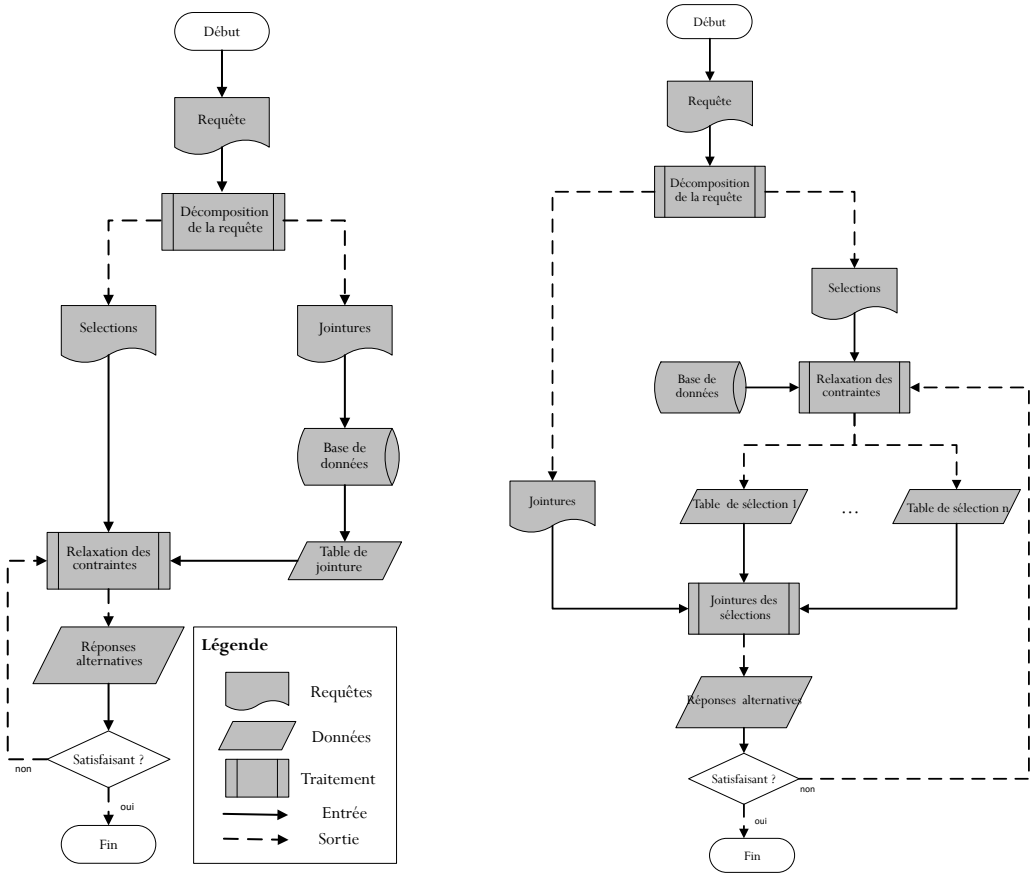
TABLE 3.6 – Tuples relaxés *Cours*

Le deuxième algorithme, *SortedAccessJoin*, proposé par Koudas et al. [167] exécute séparément les sélections sur chaque table. Ensuite, il opère des relaxations sur les tuples de chaque sélection. Enfin, il réalise la jointure entre les tuples relaxés. Dans notre exemple précédent, une relaxation sur les contraintes portant sur le salaire et le volume horaire est ainsi réalisée en premier. Ces contraintes pourraient devenir, par exemple, $1000 < P.Salaire$ `AND` $P.Salaire < 3000$ `AND` $25 < C.Volume\ horaire$ `AND` $C.Volume\ horaire < 60$. La sélection de ces contraintes relaxées sur chaque table retourne les tuples des tables 3.5 et 3.6. Une fois les deux tables obtenues, la jointure est réalisée entre les tuples relaxés. La jointure dans cet exemple se fait entre les tuples en gris. Une fois ces tuples trouvés, l'algorithme calcule le skyline sur ces tuples et attributs afin de les classer suivant un ordre tel que la dominance de Pareto. Cet algorithme permet donc de passer d'une requête à résultat vide à des requêtes relaxées retournant des réponses alternatives.

3.1.3.2 Intégration dans les langages de requête et processus de relaxation

Les deux algorithmes précédents n'ont été intégrés dans aucun langage de requête. Ces algorithmes ont en effet été implémentés sous forme de processus en amont de l'exécution d'une requête. Ce processus commence par la décomposition de la requête et la séparation des jointures et des sélections. Ensuite, le processus exécute soit la jointure soit les sélections et opère la relaxation sur les tuples obtenus. Enfin, il réalise la dernière opération qui peut être la jointure ou la sélection. Si cette opération retourne des résultats satisfaisants, alors le processus s'achève ; sinon, le processus retourne à l'étape de la relaxation des résultats intermédiaires. Les figures 3.3a et 3.3b schématisent les processus de relaxation de jointures

JoinFirst et *SortedAccessJoin* respectivement. Ces processus peuvent s'avérer très coûteux en terme de temps, à cause du processus itératif de relaxation. Ainsi, dans le but d'éviter d'entrer dans ce processus itératif, certains travaux proposent la suppression de contraintes comme méthode de relaxation.



(a) Processus *JoinFirst*

(b) Processus *SortedAccess Join*

FIGURE 3.3 – Processus de relaxation des jointures

3.1.4 Relaxation par suppression de contraintes

La suppression de contraintes est un moyen simple de relaxer une requête n'ayant pas de réponses. Cette méthode de relaxation doit cependant répondre à une question avant de pouvoir être implémentée : comment retrouver la ou les contrainte(s) à supprimer pour relaxer la requête ? Les algorithmes de relaxation par suppression des contraintes proposent plusieurs réponses à cette question. Dans ces contributions, la suppression est fortement dépendante de l'objectif de l'algorithme. Les algorithmes présentés, dans cette section, remplissent un des objectifs suivants : renseigner les utilisateurs sur le contenu de la base de données, retrouver des réponses désirées par les utilisateurs et enfin réparer au plus vite la requête.

3.1.4.1 Objectif : renseigner sur le contenu de la base de données

Inoue et Wiese. [196] ont proposé des opérateurs de généralisation de la requête dont l'opérateur *Dropping Condition (DC)* pour la suppression des contraintes. L'objectif de ces opérateurs est de rendre la requête plus générale et donc moins contraignante, ceci afin de pouvoir retrouver des réponses informatives, qui informerons les utilisateurs sur le contenu de la base de données. Dans cette contribution, Inoue

et al. se limitent aux requêtes conjonctives, c'est-à-dire à des conjonctions de contraintes. L'opérateur de relaxation *Dropping Condition (DC)* réalise alors une suppression aléatoire et itérative des contraintes et exécute les sous-requêtes obtenues. Cet opérateur n'insiste pas sur la similarité entre les requêtes car les réponses sont retournées pour informations et non comme réponses aux requêtes des utilisateurs. Bakhtyar et al. [197] ont ensuite développé une mesure de similarité pour les opérateurs de généralisation proposés par Inoue et Wiese [196]. Cette mesure de similarité évalue la pertinence d'une réponse r par rapport à la requête Q de l'utilisateur afin de pouvoir filtrer les réponses les moins pertinentes, ou trop générales pour les utilisateurs. Pour l'opérateur *Dropping Condition (DC)*, Bakhtyar et al. [197] ont développé la mesure de similarité, $SimQA_D(Q, r)$ suivante :

$$SimQA_D(Q, r) = \frac{m'}{m} \tag{3.1}$$

où Q est la requête initiale, r une réponse généralisée, m est la somme des arités [16] des prédicats de la requête Q et m' la somme des arités des prédicats de la requête relaxée ayant produit la réponse r. Considérons les données de l'exemple précédent (tableaux 3.2 et 3.3) et la requête :

SELECT $P.ID,\ P.Salaire,\ C.Cours,\ C.Volume\ horaire$
FROM $Professeurs\ P, Cours\ C$
WHERE $1500 < P.Salaire$ AND $P.Salaire < 2500$
AND $P.ID = C.ID - Prof$ AND
$35 < C.Volume\ horaire$ AND $C.Volume\ horaire < 50$

Cet exemple contient des opérateurs de comparaison binaires (arité 2) dans ses prédicats. Nous avons au total cinq prédicats binaires d'où $m = 10$. Une relaxation de cette requête est obtenue par la suppression de la jointure, $P.ID = C.ID - Prof$. Cependant, la requête relaxée ainsi obtenue retourne toujours un ensemble vide de réponse. Ce résultat persistera tant qu'une seule contrainte sera supprimée. Si, par contre, nous supprimons les contraintes $P.Salaire < 2500$ et $C.Volume\ horaire < 50$, la requête relaxée sera composée de trois prédicats binaires d'où $m' = 6$ et les réponses alternatives obtenues seront celles présentées dans le tableau 3.7. Ces deux réponses alternatives possèdent la même similarité, suivant la mesure de Bakhtyar et al. [197], et cette similarité est $SimQA_D(Q, r) = \frac{m'}{m} = \frac{6}{10} = 0,6$.

P.ID	P.Salaire (€)	C.ID	C.Volume horaire
fullProf_2	4500	cours_5	54
fullProf_5	4000	cours_3	54

TABLE 3.7 – Réponses alternatives obtenues par suppression de contraintes

3.1.4.2 Objectif : retrouver des réponses alternatives

Cette méthode de relaxation par suppression a pour objectif de supprimer les contraintes tout en respectant autant que possible les préférences de l'utilisateur. En effet, une requête relaxée sera évaluée par les utilisateurs sur la base des deux questions suivantes :

1. La requête relaxée me retournera-t-elle des réponses ou pas ?
2. Les réponses renvoyées sont-elles les plus désirées ?

Mottin et al. [198, 199] ont développé une approche basée sur les probabilités pour aider les utilisateurs à répondre à ces questions. Dans cette proposition chaque contrainte de la requête est dans l'un des quatre

16. Nombre de paramètres d'une fonction ou opération utilisée dans la requête.

états suivant : présente dans la requête notée 1, supprimée notée " − ", ayant été choisi par l'utilisateur pour ne pas être supprimé notée "#", et proposée à l'utilisateur pour suppression notée "?". La figure 3.4 présente les différentes relaxations par suppression possibles pour une requête possédant trois contraintes et pour chacune les réponses retournées.

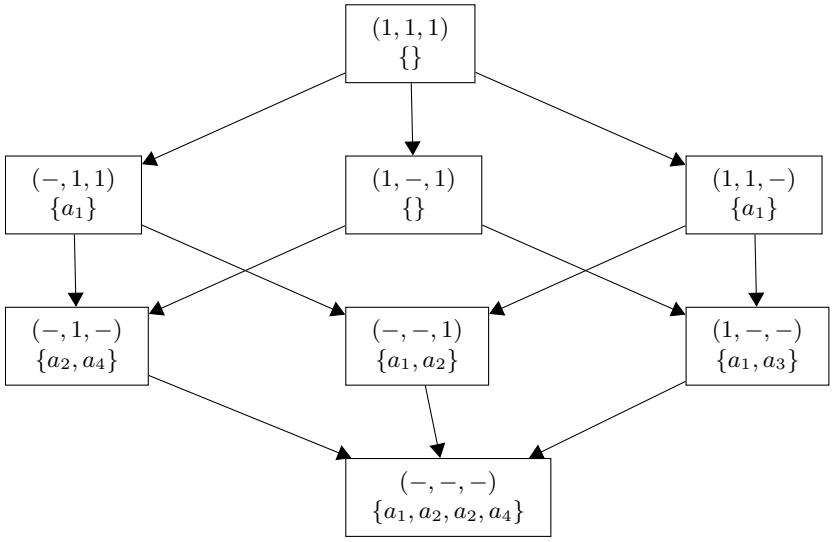


FIGURE 3.4 – Treillis de requêtes relaxées par suppression

Les travaux de Mottin et al. [199] permettent d'explorer ce treillis de requêtes relaxées et de trouver celles dont la probabilité de renvoyer des réponses désirées par les utilisateurs est la plus élevée. Pour cela, Mottin et al. [198, 199] définissent la probabilité notée $relPref_{no}(Q, Q')$ qu'une relaxation Q' d'une requête Q soit rejetée par l'utilisateur et en déduisent la probabilité $relPref_{yes}(Q, Q')$ qu'elle soit acceptée : $relPref_{yes}(Q, Q') = 1 - relPref_{no}(Q, Q')$. La probabilité $relPref_{no}(Q, Q')$ est calculée en fonction des préférences des utilisateurs et du contenu de la base de données.

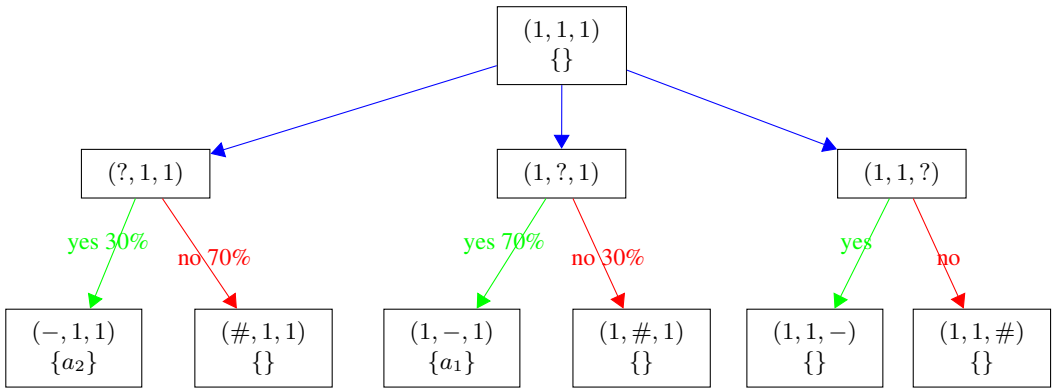


FIGURE 3.5 – Treillis de requêtes relaxées par suppression

Avec ces probabilités, le treillis de la figure 3.4 est transformé en celui de la figure 3.5 dans lequel une étape d'interrogation sur la suppression ou non d'une contrainte est ajoutée. En fonction des probabilités, l'algorithme peut déduire les arcs à explorer. Cette méthode possède néanmoins des limites. L'ensemble restreint des requêtes considérées, à savoir des requêtes conjonctives n'utilisant que des comparaisons d'égalité. Une autre limite est qu'elle n'évite pas systématiquement les requêtes relaxées qui retourneront

un ensemble vide de réponse. Par exemple, sur la figure 3.5 la proposition $(1, 1, ?)$ correspond à une requête relaxée qui retourne un ensemble vide de réponse.

3.1.4.3 Objectif : réparer la requête en se basant sur les causes d'échec

Pour réaliser la relaxation par suppression des contraintes pour des requêtes qui retournent des réponses vides, certains travaux proposent de chercher les contraintes responsables de *l'échec de ces requêtes* et de les utiliser dans la relaxation. Nous disons qu'une requête échoue (respectivement, réussie) si elle retourne un ensemble vide (respectivement, non vide) de réponse. Godfrey [200] est parmi les pionniers a avoir exploité cette approche. Godfrey considère des requêtes conjonctives $Q = c_1 \wedge c_2 \wedge ... \wedge c_n$ où les c_i sont les contraintes. Considérons la fonction $Ans(Q)$ qui renvoie l'ensemble des réponses de la requête Q. Pour toute requête Q qui échoue, c'est-à-dire $Ans(Q) = \emptyset$, Godfrey recherche les sous-requêtes minimales responsables de l'échec de Q encore appelées *Minimal Failing Subquery (MFS)*.

Définition 1

> *Une MFS d'une requête Q est une sous-requête Q^* de Q qui échoue et dont toutes les sous-requêtes réussissent.*

Les MFS respectent plusieurs propriétés dérivées de cette définition, principalement :

1. toute requête qui échoue contient au moins une MFS ;
2. si toutes les sous-requêtes de Q, qui échoue, réussissent alors la requête Q est sa propre MFS ;
3. la MFS d'une sous-requête Q' d'une requête Q est également une MFS de Q.

En s'appuyant sur ces propriétés, Godfrey [200] propose l'algorithme *a_mel_fast* de complexité $O(n)$ pour la recherche d'une MFS dans une requête qui échoue. Ensuite, pour énumérer toutes les MFS d'une requête, Godfrey propose l'algorithme *ISHAMEL* qui utilise *a_mel_fast*. Cet algorithme est de complexité exponentielle ce qui n'est pas surprenant vu que le problème de trouver toutes les MFS d'une requête qui échoue est NP-HARD [200]. Une fois toutes les MFS trouvées, la relaxation de requête par suppression doit relaxer toutes les MFS de la requête. Sachant qu'une sous-requête d'une MFS n'échoue pas, Godfrey propose pour réparer une requête Q qui échoue de supprimer au moins une contrainte à toute les MFS de Q. Afin que la requête relaxée reste proche de la requête initiale, Godfrey déduit, à partir des MFS de Q, les sous-requêtes avec un nombre de contraintes maximal et retournant des réponses non vides, encore appelées *maXimal Succeeding Sub-query (XSS)*.

Définition 2

> *Une XSS d'une requête Q est une sous-requête $Q^\#$ de Q de taille maximale qui réussit et dont toutes les sous-requêtes de Q, qui contiennent $Q^\#$, échouent.*

Les XSS relaxent donc les requêtes tout en conservant le plus de contraintes possibles de la requête utilisateur, restant ainsi le plus proche possible des besoins utilisateurs. Á noter que l'algorithme *ISHAMEL* calcule soit les MFS, soit les XSS, mais pas les deux en une seule exécution même si la déduction des XSS à partir des MFS soit discutée dans les travaux de Godfrey.

La suppression des contraintes est une forme de relaxation extrême qui pourrait retourner non seulement un ensemble de réponses pléthoriques pour les utilisateurs mais aussi insatisfaisantes pour ces derniers. Pour remédier à cela, Bosc et al. [201] et Pivert et Smits. [202] proposent d'utiliser la relaxation avec prédicats graduels sur les contraintes des MFS à la place de la suppression des contraintes. Ces méthodes de relaxation par suppression n'ont pas été intégrées dans un langage d'interrogation. Elles sont implémentées en amont des systèmes d'interrogation. Mais comme nous l'avons vu, les prédicats graduels ne

sont adaptés que pour les contraintes ayant des domaines numériques. Afin de pouvoir relaxer également les relations ou les attributs des relations, la relaxation par généralisation a été développée.

3.1.5 Généralisation de la requête

Dans les bases de données relationnelles, les liens (comme l'héritage) entres les relations exprimées dans le modèle conceptuel des données ne sont pas conservés dans la base de données. Ce contexte rend impossible toute exploitation de ces liens sémantiques entre relations dans une perspective de relaxation. Ainsi, afin d'avoir accès aux informations conceptuelles sur les relations dans les bases de données et même sur les données, plusieurs approches ont proposé de représenter les liens entre les relations sous forme d'une hiérarchie abstrait de types ou *Type Abstraction Hierarchy (TAH)*. Cette représentation permet d'interroger et d'extraire des informations qui serviront pour relaxer les requêtes par généralisation.

3.1.5.1 Principe de relaxation et transformation de la requête

Parmi les premières contributions sur la conception et l'utilisation des TAH pour relaxer les requêtes, figurent celles de Gaasterland et al. [203], Lee et al. [204] et Wesley et Qiming. [205]. Ces travaux considèrent, en effet, une structure de données hiérarchique qui décrit les liens entre les types de données. Ces liens sont des propriétés *IS-A* ou *sous-type*. Nous reviendrons dans la suite sur les techniques de génération des liens et de la structure de données. Un exemple de TAH sur les données universitaires est schématisé à la figure 3.6. Cette TAH peut être utilisée pour généraliser des requêtes portant sur les types *AssistantProfessor*, *AssociateProfessor* ou même *FullProfessor*.

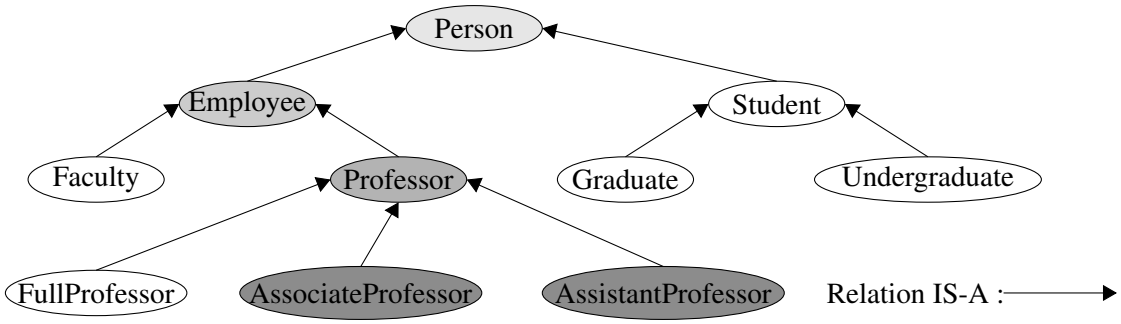


FIGURE 3.6 – Hiérarchie abstraite de types (TAH)

Dans la hiérarchie de la figure 3.6, les concepts *AssistantProfessor* et *AssociateProfessor* peuvent être substitués par les types en fond gris. Chaque nœud du TAH correspond à un type de données dans la base de données relationnelle et/ou à une relation. Ainsi, pour les requêtes sur *AssistantProfessor* et *AssociateProfessor*, le type *Professor* est utilisé pour les substituer dans une première étape. Si une généralisation est encore nécessaire, le type *Employee* est utilisé et ainsi de suite. Les types en fond gris sont ceux qui peuvent être utilisés dans cet exemple de généralisation. Cette relation sémantique entre les types permet de relaxer par substitution les relations dans une requête. La TAH permet ainsi de relaxer graduellement dans la requête les types par leurs super-types.

3.1.5.2 Intégration dans les langages de requête et processus de relaxation

Les bases de données relationnelles n'implémentant pas de façon native une TAH. Chu et al. [165] ont développé un système coopératif appelé *CoBase* qui utilise la généralisation pour relaxer les requêtes. Ce système s'appuie sur une TAH généré automatiquement à partir des données par le biais les techniques de

clustering et de partitionnement. De ce fait, la relaxation par généralisation n'a pas été intégrée dans un langage d'interrogation. La relaxation par généralisation est un processus itératif qui parcours un chemin de la TAH d'un nœud vers la racine et chaque nœud de la TAH correspond à une relation de la base de données. Ainsi, chaque fois qu'un nœud intermédiaire est rencontré pendant le parcours d'un chemin, une requête relaxée est obtenue par substitution de la relation correspondante au nœud initial par celle correspondant au nœud intermédiaire. Cette requête relaxée est exécutée et si elle retourne des réponses alternatives satisfaisantes, alors le processus s'arrête ; sinon, le parcours de la TAH continue et d'autres substitutions sont opérées. Ce processus ne s'arrête qu'à l'obtention des réponses alternatives ou à la substitution par la racine qui correspond à la suppression. Le processus de relaxation par généralisation est implémenté au-dessus d'un système d'interrogation afin de réaliser la relaxation si nécessaire.

3.2 Relaxation dans les systèmes de recommandation

Les systèmes de recommandation sont des systèmes dédiés pour aider les utilisateurs dans la recherche des informations. Les systèmes de recommandation ont généralement trois étapes de fonctionnement :

1. la collecte d'information sur les produits et les utilisateurs ;
2. la construction de la table des produits, si ce n'est déjà fait, et du modèle utilisateur ;
3. la réalisation de la recommandation des produits à un utilisateur en fonction des besoins qu'il décrit, de son modèle utilisateur et des produits existants dans la table.

La relaxation intervient dans la troisième et dernière étape du processus de recommandation. Les systèmes de recommandation utilisent principalement deux techniques de relaxation pour recommander des réponses. Il s'agit de la relaxation par suppression de contraintes et la relaxation par les prédicats flous.

3.2.1 Suppression des critères

Lorsque le système de recommandation ne trouve pas de produit correspondant à des critères formulés par des utilisateurs, le système peut supprimer certains critères pour trouver des réponses alternatives. Cette solution est proposée par Jannach [206, 207] et McSherry [208, 209]. Bien qu'ils utilisent tous les deux la suppression de critères, les méthodes proposées diffèrent.

3.2.1.1 Approche de Jannach

Jannach [206] recherche les relaxations minimales pour réparer la requête. Il montre que la réparation de la requête avec ces relaxations minimales donne les XSS définies par Godfrey [200]. Pour calculer les relaxations minimales, Jannach utilise une matrice présentée à la table 3.8 pour une requête $Q = c_1 \wedge c_2 \wedge c_3 \wedge c_4$. Cette matrice s'obtient en cherchant pour chaque contrainte c_i de la requête les produits la satisfaisant et pour chaque produit satisfaisant ans_j trouvée, l'affectation $M(c_i, ans_j) = 1$ est opérée, où M est la matrice.

	c_1	c_2	c_3	c_4
ans_1	0	0	0	1
ans_2	0	1	0	1
ans_3	0	0	1	1
ans_4	0	1	0	1

TABLE 3.8 – Évaluation individuelle des critères de recommandation

Grâce à cette matrice, nous pouvons déduire les produits qui satisfont tous les critères et donc la requête, ce sont les réponses telles que $\forall i, 1 < i < n; M(c_i, ans_j) = 1$, c'est-à-dire tous les éléments de la colonne ans_j valent 1. Par contre, si la requête ne retourne pas de réponse, alors aucun produit ne satisfait l'ensemble de ses critères, donc tous les produits possèdent au moins un critère évalué à faux, c'est-à-dire 0. Jannach [206] définit *la relaxation spécifique d'un produit* ans_j par rapport à la requête Q, $PSX(Q, ans_j)$, comme l'ensemble des critères que le produit ans_j ne satisfait pas c'est-à-dire $PSX(Q, ans_j) = \{c_i \mid M(ans_j, c_i) = 0\}$. Pour la matrice de la table 3.8 nous avons par exemple, $PSX(Q, ans_1) = c_1 \wedge c_2 \wedge c_3$, $PSX(Q, ans_2) = c_1 \wedge c_3$, $PSX(Q, ans_3) = c_1 \wedge c_2$ et $PSX(Q, ans_4) = c_1 \wedge c_3$. En s'appuyant sur *la relaxation spécifique d'un produit*, Jannach développe l'algorithme ***MinRelax*** qui permet de trouver les plus petites relaxations de la requête Q. Une relaxation est minimale si elle ne contient pas une autre relaxation.

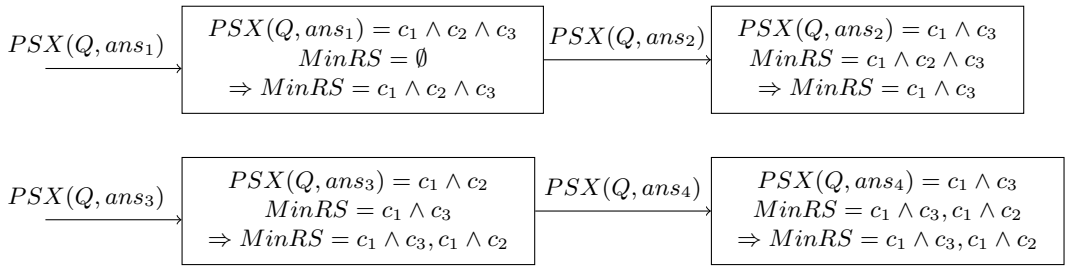


FIGURE 3.7 – Recherche des relaxations minimales

MinRelax initialise l'ensemble des relaxations minimales $MinRS$, à l'ensemble vide : $MinRS = \emptyset$. Pour chaque relaxation spécifique d'un produit, PSX, ***MinRelax*** vérifie si PSX n'est pas contenu dans une relaxation minimale incluse dans $MinRS$. Si c'est le cas ***MinRelax*** passe à la prochaine relaxation spécifique sinon PSX est inséré dans $MinRS$ après suppression dans $MinRS$ de toutes les relaxations qui contenaient PSX, car elles ne sont plus minimales. La figure 3.7 montre le déroulement du processus de recherche de relaxation minimale pour la requête Q. Au terme de ce processus, l'ensemble des relaxations minimales est $MinRS = c_1 \wedge c_3, c_1 \wedge c_2$. Il contient ainsi deux relaxations. Enfin, pour calculer les sous-requêtes maximales qui reussissent, Jannach [206] supprime les éléments de $MinRS$ de la requête utilisateur. Pour notre exemple, nous obtenons, $Q_1 = Q - (c_1 \wedge c_3) = c_2 \wedge c_4$ qui retourne ans_2, ans_4 et $Q_2 = Q - (c_1 \wedge c_2) = c_3 \wedge c_4$ qui retourne ans_3. Ces deux requêtes sont des XSS. Nous notons que cette approche ne permet de trouver que les XSS et non les MFS.

3.2.1.2 Approche de McSherry

McSherry [210, 209] comme dans l'approche de Godfrey cherche, dans un premier temps, les causes minimales d'échecs de la requête : les MFS. Ces MFS sont utilisées d'une part pour expliquer l'échec de la requête aux utilisateurs et d'autre part pour proposer des relaxations de la requête. Pour rechercher les MFS, McSherry explore le treillis des sous-requêtes de la requête utilisateur, en commençant par celles ayant les plus petits nombres de contraintes vers celles possédant les plus grandes. Pour chaque sous-requête, il vérifie si elle échoue donc si c'est une MFS car possédant, par construction, le plus petit nombre de contraintes. Lorsque une MFS est trouvée, toutes les sous-requêtes contenant cette MFS sont supprimées dans le treillis des sous-requêtes. Prenons l'exemple de la requête $Q = c_1 \wedge c_2 \wedge c_3 \wedge c_4$, le treillis des sous-requêtes de Q est donné par la figure 3.8. L'approche proposée par McSherry explore ce treillis de bas en haut. Dans cet exemple, appliqué sur la table 3.8, l'algorithme teste la sous-requête c_1 qui échoue donc $Q^* = c_1$ est une MFS, en rouge. Les sous-requêtes testées et qui réussissent sont en vert. Toutes les sous-requêtes contenant c_1 sont alors supprimées du treillis, grisées dans la figure 3.8, et

ne sont donc pas testées. Au final, on trouve deux MFS $Q^* = c_1$ et $Q^{**} = c_2 \wedge c_3$.

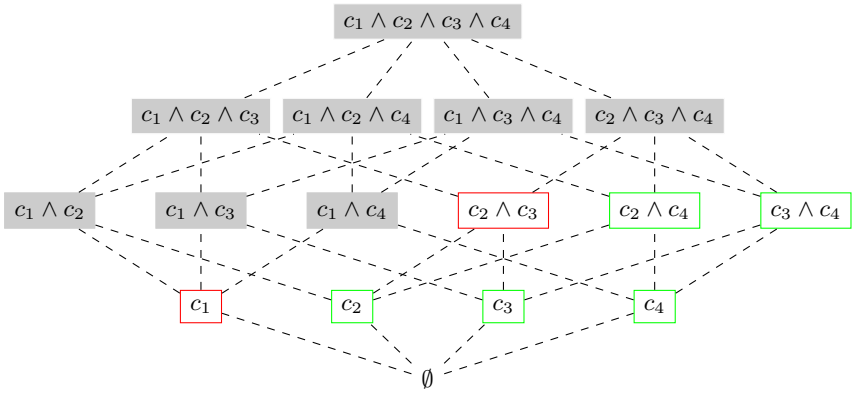


FIGURE 3.8 – Treillis des sous-requêtes de Q

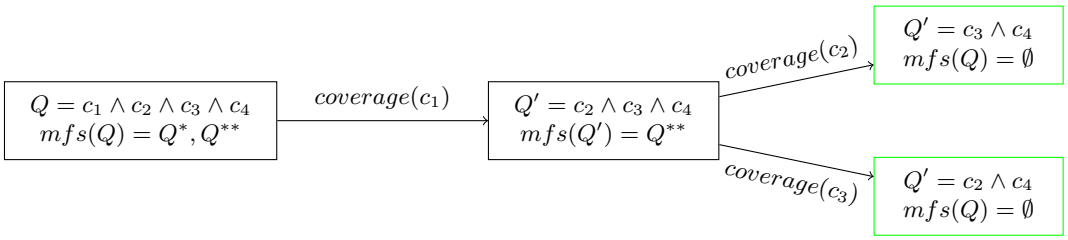


FIGURE 3.9 – Recherche de requêtes relaxées de Q avec Recover

Pour trouver ensuite les relaxations de Q, McSherry développe l'algorithme *Recover*. Il définit la notion de *couverture des critères (coverage)*. La couverture d'un critère est l'ensemble des MFS qui contiennent ce critère, $coverage(c) = \{Q' \in mfs(Q) \mid c \in Q'\}$ où $mfs(Q)$ est l'ensemble des MFS de la requête Q. Dans notre exemple, $coverage(c_1) = Q^*$, $coverage(c_2) = Q^{**}$, $coverage(c_3) = Q^{**}$ et $coverage(c_4) = \emptyset$. Ces ensembles sont ensuite utilisés pour supprimer toutes les MFS de Q. L'algorithme *Recover* commence par le critère ayant la plus grande couverture, propose de supprimer ce critère pour que Q ne contienne plus toutes les MFS de la couverture de ce critère. Si l'utilisateur accepte, ils sont donc soustrait de l'ensemble des MFS de Q. Le même processus est répété avec la requête relaxée et les MFS restant jusqu'à ce que l'ensemble des MFS de Q soit vide (ce qui voudrait dire que toutes les MFS ont été relaxées). La figure 3.9 montre un processus d'exécution de l'algorithme *Recover* sur notre exemple. Le processus commence par le critère c_1 pour lequel $|coverage(c_1) = 1|$ est la valeur maximale. La suppression de c_1 donne $Q' = c_2 \wedge c_3 \wedge c_4$ avec $mfs(Q') = mfs(Q) - coverage(c_1) = Q^{**}$. Ensuite, la suppression de c_2 ou c_3 est suffisant pour relaxer la dernière MFS de Q, nous obtenons donc les deux requêtes relaxées $c_2 \wedge c_4$ et $c_3 \wedge c_4$, en vert sur la figure 3.9.

Tandis que Jannach recherche directement les XSS, l'approche de McSherry se focalise donc sur la recherche des MFS pour proposer interactivement des requêtes relaxées qui ne sont pas forcément des XSS et qui peuvent échouer.

3.2.2 Relâchement des critères

La suppression des critères n'est pas la seule approche de relaxation des bases de données relationnelles adaptées dans les systèmes de recommandation, même si elle est la plus répandue. Nous notons

également l'utilisation des prédicats flous et la généralisation des requêtes. Pour les prédicats flous, Mirzadeh et al. [174] proposent une adaptation qui étend les contraintes numériques sur les critères avec des fonctions d'élargissement *(IncreaseRange Function)* et de restriction *(DecreaseRange Function)* des intervalles. Pour les critères non numériques, Mirzadeh et al. proposent de généraliser les attributs de ces critères en utilisant une hiérarchie abstraite des attributs nommée *Feature abstraction hierarchy (FAH)*. Dans le même ordre d'idée, Moreno et al. [176] proposent d'utiliser à la place de la *FAH* une ontologie. L'ontologie offre plusieurs avantages dont celui de permettre la généralisation des critères en s'appuyant sur leur capacité de raisonnement. Nous reverrons en détails cet aspect dans la section suivante.

3.3 Relaxation des requêtes sur des données RDF

Les données RDF contiennent généralement à la fois le schéma des données en s'appuyant sur un langage tel que RDFS, et les instances des concepts de ce schéma. La complexité de la structure représentée dans le schéma rend l'interrogation des données difficile, et plus ce schéma est grand plus l'interrogation est complexe. De plus, ce schéma peut contenir plusieurs concepts similaires mais pas identiques. Par conséquent, une confusion entre des concepts peut naître chez les utilisateurs et ainsi compromettre la requête et ses réponses. La relaxation permet d'identifier ces concepts similaires et de les modifier dans la requête afin de retourner à l'utilisateur des *réponses alternatives*. Plusieurs techniques de relaxation ont été conçues pour les requêtes SPARQL sur les données RDF. Pour aider à la compréhension des approches présentées dans cette section, nous avons représenté les données des tableaux 3.2 et 3.3 sous forme de graphe, à la figure 3.10. Pour des raisons de lisibilité, nous n'y avons pas intégré les données relatives aux cours et celles relatives aux âges.

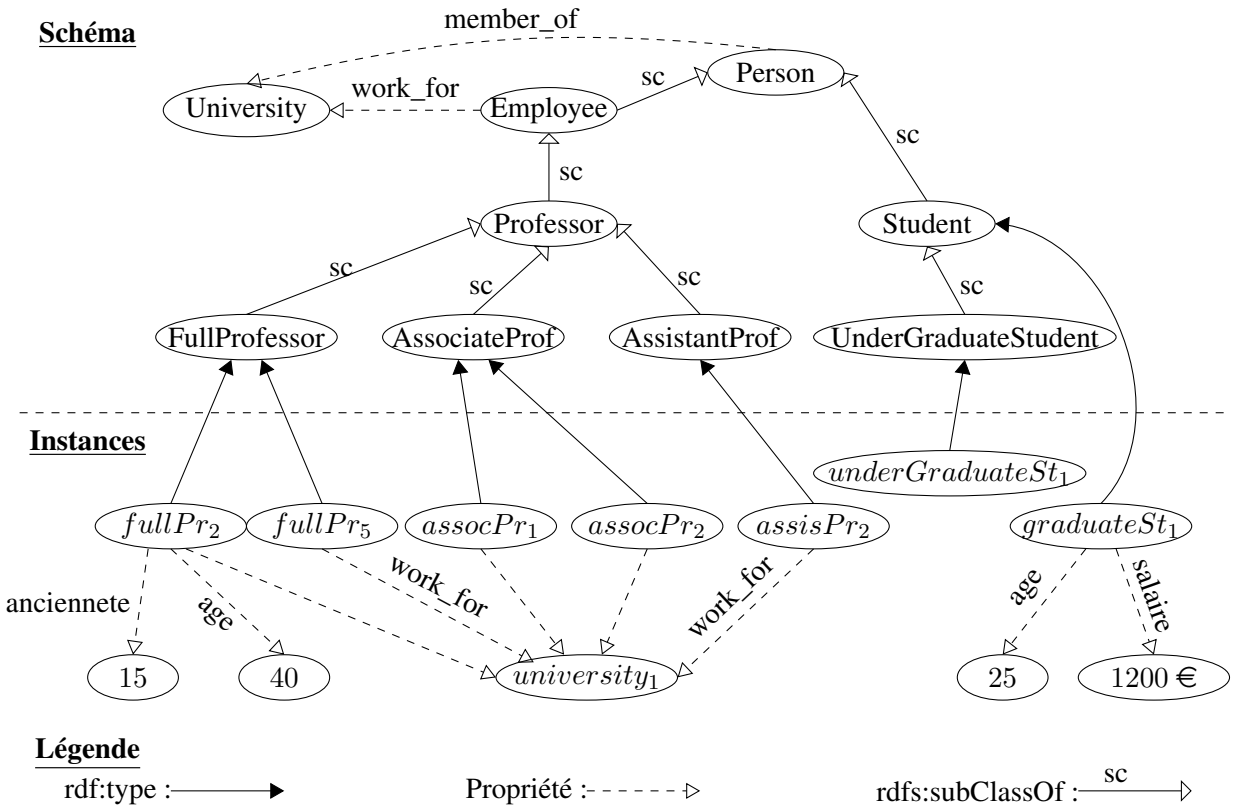


FIGURE 3.10 – Graphe de données

3.3.1 Relaxation par raisonnement

Le raisonnement permet de déduire des nouvelles connaissances et aussi de relaxer les requêtes SPARQL. Nous parlons de *relaxation par raisonnement* (appelée également *relaxation sémantique ou ontologique*). La relaxation par raisonnement est une méthode de relaxation basée sur le schéma des données et sur un ensemble de règles de raisonnement.

3.3.1.1 Règles de relaxation

Hurtado et al. [216, 217] se sont intéréssés à l'exploitation du raisonnement sur les ontologies pour relaxer les requêtes. Dans le contexte des ontologies définies en RDFS, ils ont identifié des règles de raisonnement qui peuvent servir pour la relaxation des requêtes. Ces règles sont celles de la table 3.9, dans laquelle les valeurs *sp* et *sc* représentent respectivement *rdfs:subPropertyOf* et *rdfs:subClassOf*. De plus, les valeurs *type*, *dom* et *range* représentent respectivement *rdf:type*, *rdfs:domain* et *rdfs:range*.

Groupe A (Sub-Property)	(1) $\frac{(a, \text{sp}, b)(b, \text{sp}, c)}{(a, \text{sp}, c)}$	(2) $\frac{(a, \text{sp}, b)(x, a, y)}{(x, b, y)}$
Groupe B (Sub-Class)	(3) $\frac{(a, \text{sc}, b)(b, \text{sc}, c)}{(a, \text{sc}, c)}$	(4) $\frac{(a, \text{sc}, b)(x, \text{type}, a)}{(x, \text{type}, b)}$
Groupe C (Typing)	(5) $\frac{(a, \text{dom}, c)(x, a, y)}{(x, \text{type}, c)}$	(6) $\frac{(a, \text{range}, d)(x, a, y)}{(y, \text{type}, d)}$

TABLE 3.9 – Règles RDFS de raisonnement pour relaxer

Hurtado et al. [216] utilisent par exemple les règles 2 et 4 pour substituer respectivement une propriété et une classe par une super-propriété et une super-classe respectivement. Cette substitution relâche la requête au niveau sémantique, car les instances des classes filles étant aussi instances de la classe mère, cette dernière possède donc, en plus de ses propres instances, celles de toutes ses classes filles. Toutes ces nouvelles instances pourront être des réponses alternatives de la requête. Hurtado et al. [217] ont proposées, à partir de ces règles, les relaxations suivantes, dites ontologiques, par remplacement d'un patron de triplet t_1 par un autre t_2, noté $t_1 \prec^*_{onto} t_2$.

- **Relaxation de type** : elle remplace les patrons de triplet de la forme $t_1 = (a, type, b)$ par $t_2 = (a, type, c)$ si le triplet (b, sc, c) existe dans l'ontologie *(règle 4)*.
- **Relaxation de prédicat** : elle remplace les patrons de triplet $t_1 = (a, p, b)$ par $t_2 = (a, q, b)$ si le triplet (p, sp, q) existe dans l'ontologie *(règle 2)*.
- **Relaxation de prédicat par son domaine** : elle remplace un prédicat par son domaine, c'est-à-dire les patrons de triplet de la forme $t_1 = (a, p, b)$ sont remplacés par $t_2 = (a, type, c)$ si le triplet (p, dom, c) existe dans l'ontologie *(règle 5)*.
- **Relaxation de prédicat par son codomaine** : elle remplace un prédicat par son codomaine, c'est-à-dire les patrons de triplet de la forme $t_1 = (a, p, b)$ sont remplacés par $t_2 = (b, type, c)$ si le triplet $(p, range, c)$ existe dans l'ontologie *(règle 6)*.

Pour illustrer la relaxation de type, prenons la requête SPARQL Q suivante.

Q :
SELECT $?pr$ $?a$
WHERE {
$?pr$ *rdf:type* $AssistantProfessor$. (t_1)
$?pr$ age $?a$. (t_2)
FILTER $(30 < ?a$ AND $?a < 35)\}$

Q' :
SELECT $?pr$ $?a$
WHERE {
$?pr$ *rdf:type* $Professor$. (t'_1)
$?pr$ age $?a$. (t_2)
FILTER $(30 < ?a$ AND $?a < 35)\}$

La relaxation de type sur t_1 permet de substituer *AssistantProfessor* par *Professor*, car *AssistantProfessor* est une sous-classe de *Professor*, nous obtenons la requête relaxée Q' ci-dessus. Cette règle permet également de substituer les classes *AssociateProfessor* et *FullProfessor* par la classe *Professor*. La requête relaxée Q' retourne ainsi les réponses alternatives suivantes : $\{(?pr = fullPr_5,\ ?a = 34)$ et $(?pr = associatePr_1,\ ?a = 31)\}$ alors que la requête utilisateur ne retourne aucune réponse. On notera que $t_1 \prec^*_{onto} t'_1$ et donc $Q \prec^*_{onto} Q'$.

Les règles de raisonnement du tableau 3.9 peuvent être combinées pour réaliser une relaxation. Par exemple, les règles 1 et 3, transitivité des propriétés *rdfs:subPropertyOf* et *rdfs:subClassOf*, permettent de retrouver les ancêtres d'un concept ou d'une propriété qui sont ensuite utilisés avec les règles 2 et 4 pour réaliser la relaxation de type. De même, les règles 5 et 6 permettent de déduire le type des concepts et ensuite les règles 2 et 4 pourront être utilisées pour relâcher la requête.

Hurtado et al. [216] ont également développé des relaxations dites simples, notée $\prec^*_{simple}$, il s'agit de :

- **la suppression de patron de triplet** : cette relaxation supprime un patron de triplet. Elle remplace toutes les constantes du patron de triplet par des nouvelles variables afin de conserver les jointures existantes avec les autres patrons de triplet. La clause *OPTIONAL* de SPARQL est une forme de suppression de patrons de triplet car elle considère la requête avec les patrons de triplet optionnels et également sans eux ;
- **la transformation des constantes en variables** : cette relaxation remplace une constante donnée en variable. C'est une forme de suppression de constante dans un patron de triplet. Cette relaxation est catégorisée en fonction de la constante supprimée, sujet, objet ou prédicat. Lorsque toutes les constantes d'un patron de triplet sont supprimées alors le patron de triplet peut lui même être considéré comme supprimé, c'est la suppression du patron de triplet ;
- **la suppression des jointures** : dans SPARQL les jointures peuvent se faire via des variables. Ainsi, pour relâcher la requête en supprimant les jointures, cette approche substitue toutes les occurrences de la variable de jointure par des variables deux à deux différentes.

Ces trois relaxations n'utilisent pas le raisonnement d'où leur nom de *relaxation simple*. Néanmoins, les contributions de cette approche présentent deux principales limites. La première est que ces techniques ne sont pas disponibles dans un langage d'interrogation. La deuxième est que l'exploitation de ces techniques nécessite un guide ou un contrôle, afin de savoir parmi plusieurs relaxations possibles laquelle est la plus pertinente, ce qui n'est pas proposé dans cette approche.

3.3.1.2 Relaxation des requêtes avec expressions régulières

La version 1.1 de SPARQL permet l'utilisation des expressions régulières à la place des prédicats lors de la formulation des requêtes. Les requêtes contenant des chemins sous forme d'expressions régulières sont appelées *conjunctives regular path queries*. Hurtado et al. [16] et Poulovassilis et al. [218] ont développé des techniques afin de relaxer les chemins, sous forme d'expressions régulières, dans les requêtes SPARQL. Si nous recherchons par exemple, *un enseignant, de grade* ***professeur*** *ou tout autre* ***grade semblable****, qui* ***enseigne*** *ou soit* ***assistant d'enseignement***, la requête suivante contenant plusieurs expressions régulières peut être utilisée :

SELECT $?teacher$ $?course$
WHERE {
$?class$ (*rdfs:subClassOf*$^+$) $Professor.$ (t_1)
$?teacher$ *rdf:type* $?class.$ (t_2)
$?teacher$ $(teacherOf \mid teacherAssistant)$ $?course.$ (t_3) }

Pour ces types de requête SPARQL, Hurtado et al. [16] et Poulovassilis et al. [218] ont développé des techniques d'approximation de chemins basées sur les automates non-déterministes. La figure 3.11 montre un exemple d'automate représentant la requête SPARQL précédente. Dans cet automate, nous observons que chaque arc est une partie d'un chemin de la requête et les s_i sont les états. Les deux états finaux de cet automate proviennent du patron de triplet t_3 qui contient une disjonction.

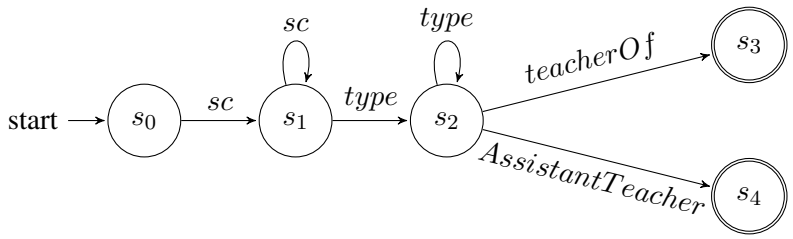


FIGURE 3.11 – Requête SPARQL avec chemin sous forme d'automate fini non-déterministe

À partir de l'automate correspondant à la requête (figure 3.11), Hurtado et al. construisent une approximation de cet automate en utilisant l'algorithme de construction d'automates non-déterministes de Thompson [219]. Cet algorithme conserve le même ensemble d'états et rajoute les transitions suivantes :

- pour chaque état s_i et chaque étiquette a, la transition (s_i, a, α, s_i) de s_i vers s_i est rajoutée avec α le coût d'insertion de cette transition ;
- pour chaque transition $(s, a, 0, t)$ de l'automate initial où a est une étiquette d'arc et t un état, on rajoute la transition (s, ϵ, α, t) où ϵ est la chaîne vide et α est le coût de suppression de a ;
- pour chaque transition $(s, a, 0, t)$ de l'automate initial où a est une étiquette d'arc et t un état, considérons une étiquette b de l'automate telle que $b \neq a$, la transition (s, b, α, t) est ajoutée avec un coût de substitution de a en b égal à α.

Les différents coûts sont calculés en utilisant la distance d'édition [164, 163], qui vaut zéro pour les arcs de l'automate initial. L'automate approximatif ainsi calculé est évalué sur le graphe de données. Cette évaluation est une forme d'alignement entre les deux automates. Cette évaluation se fait incrémentalement et en fonction du poids des réponses approximatives de la requête extraite après l'évaluation de la requête approximative.

L'alignement des graphes est également utilisé pour l'approximation des chemins dans les requêtes SPARQL. De Virgilio et al. [220] ont aussi développé une méthode de relaxation de chemins qui utilise l'alignement des graphes. Cette méthode adapte l'alignement ou appariement des graphes au contexte des requêtes SPARQL. De Virgilio et al. [221] ont également proposé une mesure de similarité pour l'alignement des graphes qui permet de classer les réponses alternatives provenant de l'approximation des chemins des requêtes SPARQL.

3.3.1.3 Opérateurs de relaxation

À la suite des travaux de Hurtado et al. [216, 217] et dans le but d'intégrer ces méthodes de relaxation dans SPARQL, Calì et al. [19] et Frosini et al. [15] ont proposé des opérateurs de relaxation pour le

langage SPARQL. Ces travaux développent une extension du langage SPARQL, appelée SPARQL^{AR}, qui inclut des clauses d'approximation et de relaxation. La clause de relaxation, ***RELAX***, implémente la relaxation de la requête via l'application de l'une des méthodes présentées dans la section 3.3.1.1. La clause d'approximation, ***APPROX***, implémente, quant à elle, des opérations d'approximation de chemins sous forme d'expressions régulières contenues dans des requêtes SPARQL. Ces deux clauses prennent comme paramètre un patron de triplet. Plus précisément, l'opérateur ***RELAX*** prend en paramètre un patron de triplet avec prédicat et, l'opérateur ***APPROX***, un patron de triplet avec chemin.

Par ailleurs, Cheng et al. [222] ont développé l'extension de SPARQL, nommée f-SPARQL (Fuzzy SPARQL), qui adapte les prédicats flous dans la relaxation des requêtes SPARQL. Cette extension concerne principalement la clause FILTER. Les prédicats flous permettent de relâcher les filtres sur des variables numériques, comme dans le cas des bases de données relationnelles. Cette extension utilise également la mesure de satisfaction associée aux prédicats flous pour trier les réponses alternatives. Dans le même ordre d'idée, Ruizhe et al. [223] ont proposé d'utiliser les préférences utilisateurs, en plus des prédicats flous. Comme dans le cas des bases de données relationnelles, les préférences utilisateurs sont intégrées dans la formulation de la requête. D'autres travaux, à défaut de proposer des extensions de SPARQL, ont développé des frameworks permettant aux utilisateurs de relaxer leurs requêtes.

3.3.2 Frameworks de relaxation des données RDF

Dans la littérature, plusieurs frameworks ont été proposés pour la gestion des données RDF et/ou la relaxation des requêtes SPARQL. Par exemple, nous pouvons citer le framework de relaxation développé par Hogan et al. [224] dans le cadre d'un projet de recherche de EADS (European Aeronautic Defence and Space Company), ou encore, les frameworks de relaxation proposés par Elbassuoni et al. [225] qui permettent de générer des requêtes relaxées à partir des sources de données RDF utilisateurs, des ontologies externes et d'autres sources de données textuelles. Dans cette section nous présentons deux frameworks : iSPARQL et Corese.

3.3.2.1 Imprecise SPARQL (iSPARQL)

iSPARQL (imprecise SPARQL) est une extension du langage SPARQL développée par Kiefer et al. [226] qui permet une évaluation flexible des requêtes. Ce framework est le successeur de iRDQL (imprecis RDQL) développé par Bernstein et Kiefer [227]. L'objectif est resté le même : interroger les données RDF en s'appuyant sur des mesures de similarité qui rendront l'opération flexible et permettront à l'utilisateur d'avoir des réponses approximatives. iSPARQL permet une évaluation flexible de la requête en intégrant dans SPARQL des mesures de similarité et des clauses de flexibilité qui permettent d'obtenir des réponses approximatives avec une similarité donnée. Par exemple, iSPARQL permet de formuler, sur le graphe de donnée 3.10 la requête Q de l'exemple 1 ci-dessous. Cette requête calcule la similarité entre $?pr\_1$ et $?pr\_2$ en utilisant la mesure de similarité entre chaines de caractères de Levenshtein *(isparql:lev)*. Dans la requête $Q_{newform}$, les réponses sont filtrées et uniquement celles ayant une similarité supérieure à 0.8 sont retenues.

Kiefer et al. [228] étendent le langage SPARQL afin de décrire de façon plus générique la relaxation. Ils intègrent, entre autre, les prédicats suivants dans SPARQL, dont certains sont utilisés dans $Q_{newform}$:

- *isparql:name* : pour nommer la mesure de similarité à utiliser ;
- *isparql:argument* : pour lister les deux arguments à évaluer ;
- *isparql:similarity* : pour retourner la valeur de la similarité ;
- *isparql:aggregator* : pour définir l'opérateur d'agrégation.

Exemple 1.

Q :
SELECT $?pr$ $?a$
WHERE {
$?pr\_1$ *rdf:type* $Professor.$ (t_1)
$?pr\_1$ age $?a.$ (t_2)
$?pr\_2$ *rdf:type* $Professor.$ (t_3)
$?pr\_2$ $salaire$ $?s.$ (t_4)
FILTER $(?a < 35$ AND $?s < 3500).$ }
IMPRECISE $(?pr\_1,\ ?pr\_2)$
SIMMEASURE (*isparql:lev*)

$Q_{newform}$:
SELECT $?pr$ $?a$
WHERE {
$?pr\_1$ *rdf:type* $Professor.$ (t_1)
$?pr\_1$ age $?a.$ (t_2)
$?pr\_2$ *rdf:type* $Professor.$ (t_3)
$?pr\_2$ $salaire$ $?s.$ (t_4)
FILTER $(?a < 35$ AND $?s < 3500).$ }
$\#Blocdepatrondetripletsimprcis$
$?relaxation$ *isparql:name* "lev".
$?relaxation$ *isparql:argument* $(?pr\_1, ?pr\_2)$
$?relaxation$ *isparql:similarity* $?sim$
FILTER $(?sim > 0.8)$ }

iSPARQL est lié à la bibliothèque simPack [17] [169] afin d'utiliser les mesures de similarité déjà implémentées dans cette bibliothèque, qui compte approximativement 40 mesures. En s'inspirant de simPack, Harpise et al. [170] ont développé une bibliothèque de mesure de similarité pour le domaine biomédicale : SML [18] (Semantic Measures Library & ToolKit).

iSPARQL et iRDQL possèdent néanmoins plusieurs limitations. Entre autres, elles forcent les utilisateurs à préciser les deux paramètres sur lesquels est évalué la similarité. Or, dans la relaxation, dans le meilleur des cas, un seul paramètre peut être disponible si la requête retourne au moins une réponse. De plus, ces extensions n'intègrent pas de techniques de relaxation, elles servent uniquement dans l'évaluation de la similarité des réponses imprécises.

3.3.2.2 Conceptual Resource Search Engine (Corese)

Corese [19] est un moteur de recherche sémantique pour le langage RDF développé au sein de l'équipe Wimmics de l'INRIA. Corese exploite les ontologies au format RDFS et OWL. Corby et al. [229] présentent Corese comme un moteur recherche d'information basé sur des modélisations de connaissances, telles que les ontologies ou des modèles conceptuels. Corese possède son propre langage d'interrogation construit sur le modèle RDF et proche syntaxiquement de SPARQL. Ce langage supporte le raisonnement sur l'ontologie par reformulation des requêtes. Corby et al. [20] ont également intégré des techniques d'approximation des requêtes dans Corese. Ce dernier propose deux types d'approximation : *l'approximation ontologique et structurelle*.

Approximation ontologique : il est basé sur la distance sémantique entre les concepts de l'ontologie afin d'identifier les concepts les plus similaires. Cette approximation ne considère pas uniquement les concepts ayant des relations de subsomption ou de spécialisation, elle considère tous les concepts sémantiquement *proches* suivant certaines mesures de similarité. Pour ce faire Corese définit une distance ontologique, basée sur l'observation suivante : dans une ontologie, les classes subsumées sont sémantiquement proches de celles subsumantes. Corese utilise la distance suivante :

$$\forall\ (c_1, c_2) \in H, D_H(c_1, c_2) = \min_{\{c \succeq c_1,\ c \succeq c_2\}} (l_H(< c_1, c >)) + (l_H(< c_2, c >)) \tag{3.2}$$

17. http://www.ifi.unizh.ch/ddis/simpack
18. http://www.semantic-measures-library.org/sml/
19. https://www-sop.inria.fr/acacia/soft/corese/manual/

où c est une super-classe commune à c_1 et c_2 dans la hiérarchie H et l_H est la longueur du chemin entre la classe c_1 et l'une de ses super-classes c dans la hiérarchie H [20].
Corese utilise également la similarité contextuelle pour rechercher des approximations de la requête. Pour cela, il exploite la sémantique du modèle RDFS. Il utilise les prédicats *rdfs:SubClassOf* et *rdfs:seeAlso* pour retrouver des concepts contextuellement *proches* d'un concept donné dans l'ontologie. Dans Corese, l'approximation ontologique est invoquée par le mot clé *MORE* dans la clause *SELECT*. La recherche des réponses approximatives dans Corese se fait par une approximation de tous les concepts de la requête. Afin de pouvoir les classer, Corese dispose d'une fonction pour convertir la similarité de contexte en distance ontologique.

Approximation structurelle : elle réalise une approximation de la structure de la requête. Pour cette approximation, les utilisateurs spécifient entre accolades devant le chemin à relaxer la longueur maximale autorisée pour l'approximation. Corese arrête les recherches après avoir trouvé un chemin valide avec la plus petite longueur.

3.3.3 Processus de relaxation

La relaxation ne peut être effective que si elle retourne des réponses alternatives les plus proches possibles des attentes des utilisateurs. Si, Corese implémente un processus qui permet d'atteindre ces deux objectifs, ce n'est pas le cas de toutes les méthodes de relaxation rencontrées jusqu'à présent. Les clauses *RELAX* et *APPROX*, [15] par exemple, n'offrent aucune garantie sur les réponses retournées par la requête relaxée en terme de quantité et de qualité. Afin de pouvoir garantir la quantité et la qualité des réponses alternatives, de façon automatique ou semi-automatique, des processus de relaxation ont été proposés. Ils peuvent être divisés en deux grandes familles, l'une centrée requête [17, 230, 225] et l'autre centrée utilisateur [18].

3.3.3.1 Processus basés sur les similarités

Huang et al. [17, 230] ont proposé un processus de relaxation afin de trouver des réponses alternatives pour des requêtes SPARQL qui ne retournent pas assez ou aucune réponse. Huang utilise les règles de raisonnement du Groupe A et B de la table 3.9 et la suppression des variables pour relaxer les requêtes SPARQL. Ils proposent l'algorithme *Best-First Search (BFS)* qui relaxe itérativement des requêtes conjonctives de la forme $Q = t_1 \wedge t_2 \wedge ... \wedge t_n$. Cet algorithme relaxe également chaque patron de triplet de la requête en appliquant des règles de relaxation. La figure 3.12 montre un exemple de relaxation sur le patron de triplet $t^{(0)} : (?X,\ type,\ FullProfessor)$.

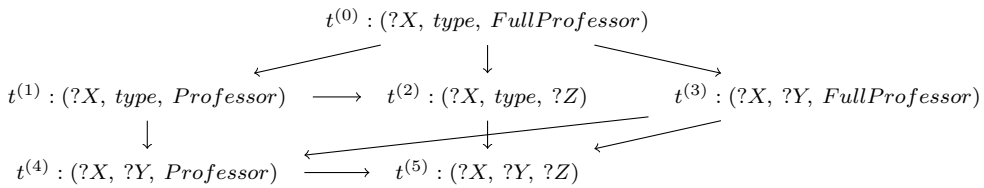


FIGURE 3.12 – Relaxation d'un patron de triplet

.

L'algorithme BFS relaxe chaque patron de triplet de la requête conjonctive Q comme dans l'exemple précédent. Toutes les requêtes relaxées forment le *graphe de relaxation* semblable à celui de la figure 3.12. Dans ce graphe, le nœud final d'un arc est une relaxation du nœud initial dans lequel un patron de

triplet est relaxé par une règle de relaxation. La racine du graphe est la requête utilisateur. Le graphe de relaxation est ensuite exploré et les requêtes relaxées sont exécutées des plus similaires au moins. Cette approche utilise pour cela une mesure de similarité que nous décrivons et utilisons au chapitre 6. Elle propose également des optimisations de BFS. Une optimisation est l'identification des inconsistances sémantiques dans les requêtes relaxées qui causent inévitablement l'échec de ces requêtes. Une autre est l'utilisation de la sélectivité et de l'exécution de batch de requêtes pour accélérer le processus.

3.3.3.2 Processus centrés utilisateurs

Dolog [18] propose un processus de relaxation qui exploite les préférences des utilisateurs. Dans cette approche, Dolog relaxe automatiquement les requêtes SPARQL en utilisant, en plus du modèle des données (schéma des données), un modèle de préférence utilisateur. Le relaxation se fait via des règles de réécriture définies dans [231]. Ces règles reformulent la requête utilisateur en requêtes relaxées en substituant des concepts de la requête par des variables ou par d'autres concepts tirés du modèle de préférence. Dolog utilise les triplets (X, Y, U) pour stocker les préférences d'un utilisateur U. Cette règle définit la préférence de X par rapport à Y pour l'utilisateur U. Ce modèle de préférence est utilisé pour guider la relaxation des concepts ou des valeurs les plus préférés par l'utilisateur vers les moins préférés. Ce niveau de préférence de l'utilisateur est exprimé sous forme de métrique. Comme pour le processus précédent, les requêtes relaxées forment un graphe qui est parcouru en largeur dans ce cas. Les requêtes relaxées sont exécutées jusqu'à l'obtention des réponses attendues par l'utilisateur. Dolog utilise l'opérateur skyline pour obtenir le niveau de préférence d'une requête relaxée issue de plusieurs réécritures de la requête utilisateur.

Conclusion

Dans ce chapitre, nous avons passé en revue les principales approches de relaxation des requêtes sur les données, en allant des données relationnelles aux données RDF en passant par les systèmes de recommandation. Nous pouvons constater que, dans le cadre des bases de données relationnelles, plusieurs méthodes de relaxation ont été développées. C'est ainsi le type de données disposant du plus grand nombre de méthodes de relaxation. Ces méthodes sont regroupées en cinq grandes familles selon l'action appliquée sur la requête et/ou les outils utilisés pour relaxer. Nous avons observé que certaines techniques ont été adaptées dans les autres types de bases de données. C'est le cas notamment de la relaxation avec des prédicats graduels et par suppression dans les systèmes de recommandation. Pour ces derniers, Jannach [207, 206] et McSherry [208, 209, 210] ont proposé des adaptations de la relaxation par suppression des contraintes et par les prédicats graduels. Pour les bases de données RDF, nous avons vu la relaxation par raisonnement [217, 216], qui est une extension de la généralisation dans les bases de données relationnelles. En plus de cette adaptation, certains travaux ont proposé des méthodes de relaxation propres aux requêtes SPARQL. C'est le cas de la relaxation des requêtes contenant des expressions régulières à partir des techniques proches de l'alignement des graphes [16, 220].
Le tableau 3.10 donne un résumé sur les méthodes de relaxation présentées dans ce chapitre. Ce tableau nous permet d'apprécier les contributions déjà réalisées dans la relaxation des requêtes et également ce qui reste à faire tel que l'adaptation de la relaxation avec prédicat graduel pour les données RDF. Ce tableau nous permet également de voir que les MFS et les XSS ont été utilisées dans les bases de données RDF et ensuite adaptées pour les systèmes de recommandation. Mais ces techniques n'ont, pour le moment, pas été encore exploitées dans les bases de données RDF.

La figure 3.13 catégorise les techniques de relaxation des requêtes. Elle permet de voir l'évolution chro-

	Relationnelles	Recommandation	RDF
Relaxation avec prédicats graduels	- Données numériques - Théorie des ensembles flous [178, 179]	- Adaptation pour la recommandation [174]	-
Relaxation des jointures	- *JoinFirst* et *SortedAccessJoin* [167]	-	- Suppression des jointures [216, 217]
Relaxation par suppression	- Mesure de similarité - Framework probabiliste - Causes d'échec (MFS) [196, 198, 200]	Adaptation de Godfrey : - Approche de Jannach et McSherry [206, 210]	- Supression des constantes et patrons de triplet [216, 217]
Relaxation par généralisation	- Utilisation des TAH [203, 204, 205]	Adaptation des TAH : - FAH et ontologies [174, 176]	- Extension avec le raisonnement [15]
Relaxation par raisonnement	-	-	- Règles de raisonnement et similarité [216, 217, 20]
Relaxation de chemin	-	-	- Structure des données et alignement des graphes [19, 229]
Processus de relaxation	- Interactive [198] - XSS [200]	-	- Basés sur les similarités et les préférences [17, 18]

TABLE 3.10 – Récapitulatifs des méthodes de relaxation dans les différents types de bases de données

nologique des techniques de relaxation, qui correspond également à l'évolution des types de stockage des données. Sur la figure 3.13, nous observons que moins la structure des données est figée plus grandes sont les techniques de relaxation exploitables. Les techniques de relaxation des requêtes des bases de données relationnelles appartiennent à la sous-catégorie des techniques de relaxation simples car elles relaxent les valeurs. La deuxième grande sous-catégorie de relaxation contient les techniques de relaxation qui exploitent la structure et la sémantique associées aux données. Dans cette sous-catégorie, nous retrouvons la généralisation ou l'alignement des graphes pour la relaxation de la structure de la requête. Ces techniques de relaxation sont implémentées naturellement pour les requêtes sur les données RDF. Même si la généralisation est présente dans les bases de données relationnelles, elle nécessite un type abstrait hiérarchique qui n'est pas stocké nativement dans la base de données. Certaines techniques de relaxation ont été proposées comme extension des langages d'interrogation des données. Nous avons vu le langage fuzzy SQL et fuzzy SPARQL pour ne citer que ceux-là. Néanmoins, l'extension des langages de requête par des techniques de relaxation posent des problèmes. D'une part, certaines extensions ne sont pas associées à des mesures de similarité et posent donc un problème de classement des réponses obtenues. De plus, certains opérateurs de relaxation ne permettent pas aux utilisateurs de faire des choix quand à la relaxation à utiliser. Par exemple, les opérateurs *RELAX* et *APPROX* proposés par Cali et al. [19] ne permettent pas de choisir entre les relaxations ontologiques ($\prec^*_{onto}$) et les relaxation simples ($\prec^*_{simple}$). D'autre part, la relaxation décrite par une instruction peut s'avérer insuffisante et donc nécessiter une relaxation plus poussée, d'où le besoin d'un processus de relaxation.

Les processus de relaxation guident la relaxation afin d'obtenir un ensemble de réponses non vides et satisfaisantes pour les utilisateurs en se basant sur des mesures de similarité. Nous avons présentés deux

processus de relaxation proposés par Huang et al. [17] et Dolog et al. [18] dans le contexte des données RDF. Ces processus ne sont néanmoins pas adaptés pour toutes les requêtes. Dans le cas des requêtes retournant un ensemble vide de réponses, ces processus peuvent s'avérer inefficaces parce qu'ils ne s'appuient pas sur les causes d'échecs des requêtes comme l'ont fait Godfrey [200] dans les bases de données relationnelles ou Jannach [206] et McSherry [209] dans les systèmes de recommandation. Si dans les deux cas précédents la relaxation se faisait par suppression ; dans les cas des données RDF, l'identification des causes d'échec permettrait d'orienter le processus de relaxation sur la réparation des causes d'échecs.

La suite de ce manuscrit présente nos contributions pour répondre à ces limitations notamment dans :

— la définition et l'intégration des opérateurs de relaxation précis et associés à des mesures de similarités dans le langage SPARQL et également l'intégration de la relaxation avec prédicat graduel (chapitre 4) ;
— la recherche simultanée des MFS et des XSS dans les requêtes sur des données RDF retournant un ensemble de réponse vide (chapitre 5) ;
— le développement d'un processus de relaxation des requêtes SPARQL retournant un ensemble vide de réponses en exploitant les causes d'échecs de ces requêtes (chapitre 6).

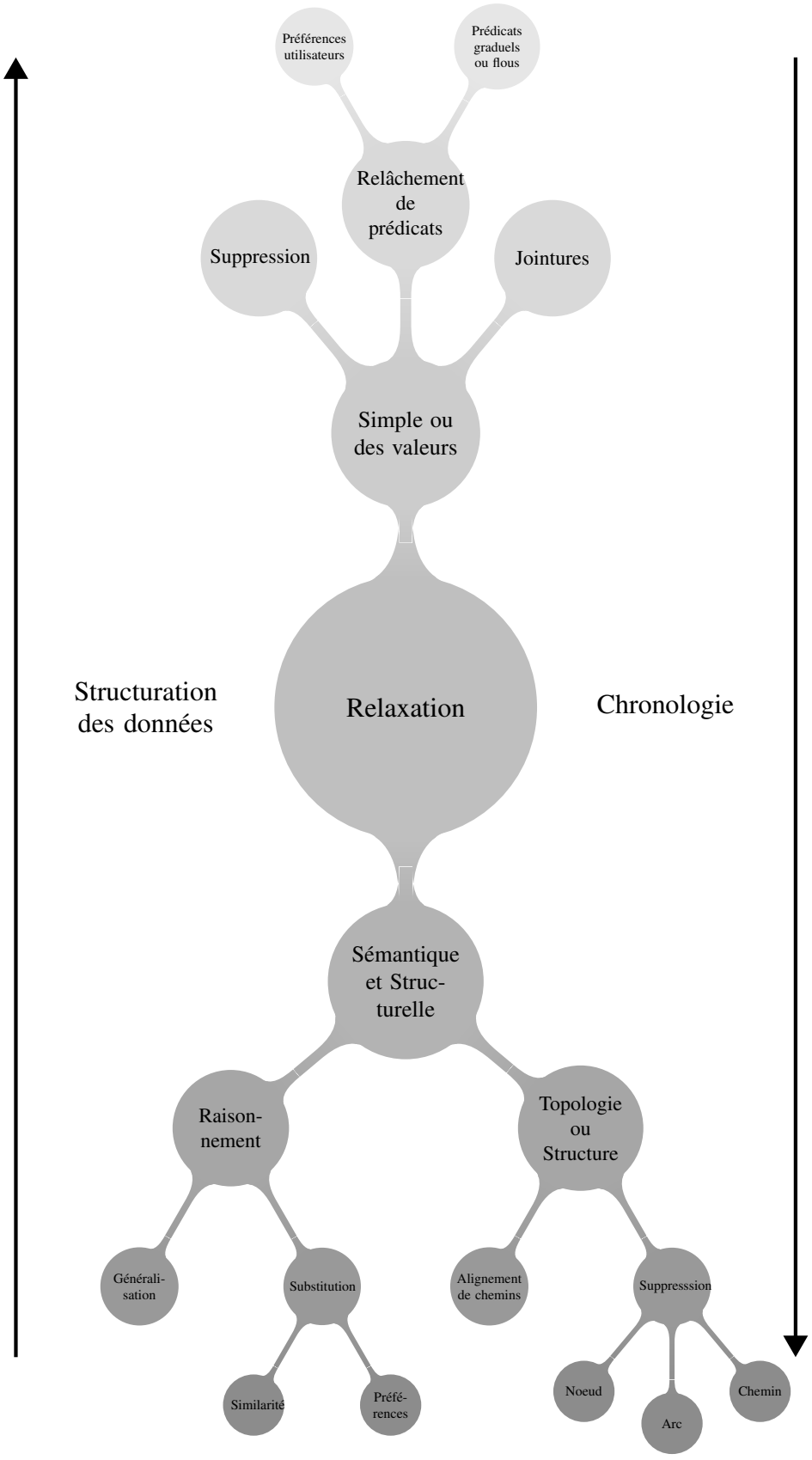


FIGURE 3.13 – Techniques de relaxation

Chapitre 4

Extension de SPARQL avec des opérateurs de relaxation

Sommaire

Résumé : Ce chapitre propose un ensemble d'opérateurs permettant de relaxer une partie précise des requêtes SPARQL suivant les besoins des utilisateurs. Nous proposons ainsi trois opérateurs de relaxation, la généralisation (GEN), la substitution (SIB) et le relâchement de valeur (PRED). GEN généralise un concept de la requête, il exploite pour cela la relation d'héritage entre les concepts et les relations. SIB substitue un concept par un autre, en utilisant les données, leurs structures et des mesures de similarités. PRED, relâche les valeurs des littéraux, il est fondé sur des techniques issues de la théorie des ensembles flous pour les valeurs numériques telles que les prédicats graduels. Nous implémentons également des combinaisons de ces opérateurs qui peuvent être exploitées dans un processus de relaxation. Nous avons aussi développé dans ce chapitre l'assistant QaRS, permettant l'édition graphique des requêtes par les utilisateurs, l'auto complétion et la suggestion de certaines contraintes de la requête. Cet éditeur permet notamment aux utilisateurs de spécifier les opérateurs de relaxation à utiliser en cas de réponses insatisfaisantes.

Introduction

Dans le chapitre précédent nous avons présenté des techniques de relaxation proposées dans la littérature. Elles diffèrent selon l'environnement d'implémentation, les transformations apportées aux requêtes et les objectifs qu'elles permettent d'atteindre. Certains travaux ont proposé une intégration de ces techniques dans les langages de requêtes sous forme d'opérateurs afin de les rendre directement accessibles aux utilisateurs. L'objectif de ces travaux est de permettre aux utilisateurs de décrire eux-mêmes la relaxation à réaliser en cas de réponses insatisfaisantes. Le langage SQLf (SQL fuzzy) [178] est un exemple d'intégration de la relaxation dans le langage de requête SQL dans le cadre des bases de données relationnelles. SQLf intègre les prédicats graduels, basés sur la logique floue, dans SQL. Il permet ainsi aux utilisateurs, en plus de décrire leurs besoins, de définir également la nature de la relaxation qu'ils souhaitent mettre en œuvre, pendant la formulation de la requête.

Dans le contexte des données RDF, des travaux ont proposé une extension du langage SPARQL en intégrant des opérateurs de relaxation. Nous avons par exemple, le langage iSPARQL (imprecise SPARQL) [226] qui permet une évaluation flexible des requêtes en utilisant des mesures de similarité sans transformer la requête. Calì et al. [19] et Frosini et al. [15] ont également proposé l'extension SPARQL^{AR} de SPARQL. Cette extension intègre les opérateurs de relaxation RELAX et APPROX qui transforment les requêtes avec respectivement des règles de raisonnement et l'alignement des chemins du graphe des données. Néanmoins, comme nous l'avons montré dans le chapitre précédent, cette extension possède des limites telles que la difficulté pour les utilisateurs de choisir la règle de relaxation à utiliser ou la partie de la requête à relaxer.

Comme première étape pour remédier à ces limites, nous proposons dans ce chapitre une intégration de *techniques de relaxation* dans le langage SPARQL sous forme d'opérateurs permettant de contrôler précisément le processus de relaxation. Ces opérateurs permettent ainsi de décrire la relaxation en précisant la partie de la requête à relaxer et la technique de relaxation à utiliser. Les techniques de relaxation implémentées sont celles qui se trouvent aux feuilles de l'arbre des techniques de relaxation de la figure 3.13 du chapitre précédent. Nous qualifions ces techniques d'*élémentaires* parce qu'elles ne peuvent plus être décomposées en d'autres techniques de relaxation. Les opérateurs de relaxation que nous proposons sont nommés *GEN*, *SIB* et *PRED*. Dans ce chapitre, nous présentons également des techniques permettant de combiner ces opérateurs de relaxation. Les combinaisons d'opérateurs permettent de décrire des processus de relaxation selon les besoins des utilisateurs. Afin d'aider les utilisateurs dans l'utilisation du framework de relaxation formé par les opérateurs de relaxation et de combinaison, nous avons implémenté l'assistant d'édition de requêtes SPARQL *QueryAndRelax* (QaRS). QaRS propose une interface intuitive d'édition graphique de requêtes SPARQL. Il aide également les utilisateurs dans la formulation de la requête en utilisant des techniques de suggestion, telles que l'auto-complétion basée sur une ontologie. La principale innovation de QaRS est la construction graphique de requêtes relaxées, c'est-à-dire des requêtes qui contiennent un opérateur ou une combinaison d'opérateurs de relaxation précédents.

Nous présentons notre framework de relaxation de requête SPARQL et l'assistant QaRS dans les cinq sections suivantes. La section 4.1 présente le contexte et les motivations qui sont à l'origine de ces travaux. La section 4.2 présente la spécification des opérateurs de relaxation et, la section 4.3 leur intégration syntaxique et sémantique dans SPARQL. Enfin, les sections 4.4 et 4.5 présentent respectivement l'assistant d'édition de requête QaRS et les résultats de l'évaluation du framework que nous avons réalisée, ainsi que le contexte de cette évaluation.

4.1 Besoin d'opérateurs de relaxation

Les techniques de relaxation transforment les requêtes dans le but d'élargir leurs champs de recherche afin de retourner des réponses alternatives aux utilisateurs. Les processus de relaxation utilisent ces techniques de façon itérative jusqu'à l'obtention de réponses alternatives. Dans l'arborescence de la figure 3.13 formée par les techniques de relaxation, nous retrouvons aux feuilles les techniques de relaxation que nous qualifions d'*élémentaires*. Ces techniques relaxent une partie précise des requêtes avec une transformation bien définie et donc désirée par les utilisateurs qui les choisissent. Nous avons observé que les processus et techniques de relaxation de l'arbre de la figure 3.13 sont tous définis à partir des techniques de relaxation élémentaires. Par conséquent, la définition des techniques de relaxation élémentaires permet de définir d'autres techniques. Dans cette section, nous montrons que les techniques et processus de relaxation utilisent, implicitement ou explicitement, une ou plusieurs relaxations élémentaires de requête. Nous identifions également les critères à respecter afin de pouvoir décrire des processus de relaxation avec des opérateurs intégrés dans un langage d'interrogation de données.

4.1.1 Typologie des processus de relaxation

Il existe plusieurs familles de processus de relaxation de requêtes RDF, parmi lesquelles :

- *la famille des processus de relaxation basés sur la requête* [221, 17] : ces processus de relaxation élargissent les contraintes des requêtes tout en restant le plus proche possible des besoins initialement exprimés par les utilisateurs ;
- *la famille des processus de relaxation basés sur les utilisateurs*[20] [18] : ces processus transforment les requêtes en d'autres requêtes en exploitant les préférences des utilisateurs afin de mieux satisfaire ces derniers ;
- *la famille des processus de relaxation basés sur le domaine des données* [224, 225] : ces processus transforment les requêtes en fonction du schéma des données interrogées, c'est-à-dire en fonction des *concepts* et des relations entre ces *concepts* qui schématisent les données interrogées.

Le terme *concept* désigne, dans ce chapitre, une classe ou une propriété du schéma des données.

L'analyse approfondie des processus de relaxation nous a permis d'identifier les différentes transformations de requêtes utilisées de façon récurrente dans ces familles de processus. Nous avons identifié :

- *la généralisation d'un concept de la requête* : elle s'appuie sur les règles de raisonnement proposées par Hurtado et al. [217]. Elle substitue un concept de la requête par un autre qui le subsume dans la hiérarchie des concepts ;
- *la substitution de concept* : cette transformation change un concept de la requête par d'autres concepts sémantiquement proches selon les préférences utilisateurs [18] et/ou des mesures de similarité [225] ;
- *l'affaiblissement des valeurs* : il s'opère sur des ensembles de données munis d'une relation d'ordre. Il permet de relâcher les valeurs des données de proche en proche selon la relation d'ordre considérée. Par exemple, les ensembles flous sont utilisés pour le relâchement des valeurs numériques [179] ;
- *la suppression d'une constante d'un patron de triplet dans la requête* : cette suppression réduit le pouvoir contraignant du patron de triplet. Elle est réalisée par la substitution d'une constante par une nouvelle variable [196] ;
- *la suppression des jointures* : elle supprime une variable de jointure entre deux patrons de triplet. Elle est réalisée en renommant la variable dans l'un des patrons de triplet [196] ;

20. Elle diffère de celle guidée par l'utilisateur : le raffinement par l'utilisateur, présenté au chapitre 2, section 2.4

— *la suppression d'un patron de triplet* : elle supprime un patron de triplet de la requête [18]. Ce patron de triplet peut également être rendu optionnel, par exemple à l'aide de la clause SPARQL `OPTIONAL`.

Les processus de relaxation existants utilisent ces transformations. L'intégration de ces transformations dans un langage d'interrogation pour des données RDF, tel que SPARQL, permet de rendre accessible la relaxation aux utilisateurs et donc facilite l'implémentation et la mise en œuvre des processus de relaxation.

4.1.2 Description de la relaxation via les langages de requêtes

Les processus de relaxation sont, en général, implémentés par deux modules. Le premier module est responsable de la transformation de la requête. Il contient l'ensemble ou seulement une partie des différentes opérations de transformations que nous avons présentées. Le second module contrôle la relaxation, et s'assure que ce processus se déroule jusqu'à l'obtention de réponses alternatives qui soient satisfaisantes pour l'utilisateur. Ce module contrôle itérativement les transformations des requêtes via des paramètres qui dépendent de l'objectif et du processus de relaxation. L'analyse des processus existants et des opérateurs intégrés dans des langages de requête permet d'identifier cinq critères principaux qui doivent être remplis [232, 233] pour réaliser une extension du langage SPARQL avec des techniques de relaxation.

Critère 1

> ***R_1 : définir un ensemble d'opérateurs de relaxation diversifiés.*** *Au vu de la diversité des techniques de relaxation existantes, les opérateurs de relaxation intégrés dans un langage de requête doivent être diversifiés et précis, c'est-à-dire associés à des paramètres de contrôle permettant sa réutilisation dans différents processus de relaxation. Dans la figure 3.13, la précison d'un opérateur est liée à sa profondeur dans l'arbre et la diversité des opérateurs est liée à la largeur de l'arbre.*

Critère 2

> ***R_2 : définir des outils de composition des opérateurs de relaxation.*** *La composition ou combinaison des opérateurs pour décrire un processus de relaxation doit être possible. Ainsi, le langage doit disposer d'outils permettant de rendre possible cette combinaison.*

Critère 3

> ***R_3 : définir une fonction pour le classement des réponses et des requêtes relaxées.*** *Comme nous l'avons vu, certains processus utilisent des mesures pour classer les réponses alternatives. L'extension du langage de requête doit permettre, à l'image de l'extension iSPARQL, de spécifier une mesure pour classer les requêtes relaxées et/ou les réponses alternatives retournées par ces requêtes.*

Critère 4

> ***R_4 : contrôler l'exécution de la relaxation.*** *Comme nous l'avons vu précédemment, le processus de relaxation intègre un module de contrôle de son fonctionnement. L'extension du langage de requête doit disposer d'outils pour contrôler la relaxation qui est de nature itérative. Ces outils définissent les règles d'évolution de la relaxation et ses conditions d'arrêt, qui correspondent très souvent au nombre de réponses alternatives trouvées, on parle alors de TOP-K réponses.*

Critère 5

> ***R_5 : intégrer l'extension du langage de requêtes dans un système de gestion des données RDF et permettre des optimisations de la relaxation.*** *L'extension du langage de requêtes doit être implémentée ou utilisée dans des BD-RDF. Étant donné le volume important des données RDF gérées par ces systèmes, les techniques d'optimisation doivent être associées à l'extension pour réduire le temps d'exécution du processus de relaxation.*

Le tableau 4.1 analyse les principaux travaux sur l'extension du langage de requêtes SPARQL avec des techniques de relaxation, suivant ces cinq critères.

	Critères					*Appli−cation*
	R_1	R_2	R_3	R_4	R_5	
Hurtado et al.[217], 2008	Raisonnement sur les ontologies	Operateur SPARQL : RELAX	Utilise des mesures de similarité	TOP-K (non paramétrable)	-[21]	-
Dolog et al.[18], 2009	Substitution de concepts	-	-	Préférences utilisateurs	Testé sur Sesame	-
Hurtado et al.[16], 2009	Approximation de chemin dans les requêtes	-	Utilise la distance d'édition (alignement des chemins)	TOP-K (non paramétrable)	-	-
Elbassuoni et al.[225], 2011	Modèles statistiques	-	Basé sur le contenu	Orienté domaine	Testé sur des données réelles	-
Huang et al.[17], 2012	Raisonnement sur les ontologies	-	Utilise des mesures de similarité et le contenu	TOP-K (non paramétrable)	Testé sur Jena (LUBM)	-
Hogan et al.[224], 2012	Substitution de concepts	-	Basé sur le contenu	Orienté domaine	Generic framework for EADS	-
De Virgilio et al.[221], 2013	Substitution de concepts et de chemins	-	Utilise la distance d'édition	Orienté domaine	Prototype SAMA testé sur LUBM	-
Cali et al.[19], 2014	Raisonnement RDFS et alignement de chemins	RELAX et APPROX	Utilise des mesures de similarité	TOP-K (non paramétrable)	SPARQL^{AR}	-

TABLE 4.1 – Comparaison des travaux existants

Ce tableau résume également leur évolution dans le temps et réalise une comparaison de ces différentes propositions d'extension du langage SPARQL pour la relaxation des requêtes. Nous observons par exemple, que Hurtado et al. [217] relaxent les requêtes SPARQL en utilisant le raisonnement et ceci est implémenté dans l'opérateur *RELAX*, ce qui permet de remplir les critères R_1 et R_2. De plus,

21. Critère non respecté

ils définissent des mesures de similarité pour ordonner les réponses alternatives et contrôler la relaxation afin d'obtenir le TOP-K de réponses les plus similaires, ce qui est conforme aux critères R_3 et R_4. Néanmoins, ce contrôle n'est pas paramétrable par les utilisateurs : il est prédéfini dans l'opérateur *RELAX*. Ainsi, les utilisateurs ne peuvent pas spécifier, par exemple, une valeur de similarité limite comme dans iSPARQL ou privilégier une transformation par rapport à une autre. De leur côté, Dolog et al. [18] proposent la substitution de concepts. Bien que cette approche soit paramétrable indirectement par les utilisateurs via leurs préférences, elle ne permet pas d'ordonner les réponses alternatives puisqu'aucune mesure de comparaison des préférences n'est définie. Ainsi, la contribution de Dolog et al. ne remplit que les critères R_1, R_4 et R_5. D'autres contributions utilisent les valeurs contenues dans la base de données afin d'ordonner les réponses alternatives, c'est la cas par exemple de Hogan et al. [224].

En conclusion, nous observons que la plupart des contributions répondent au premier critère R_1. Cependant, dans ces contributions, les opérateurs de relaxation sont implémentés comme des fonctionnalités du système et ne peuvent pas, par conséquent, être réutilisés. Cette implémentation empêche également toute combinaison de ces opérateurs pour décrire d'autres relaxations. Pour cette raison, certaines de ces contributions ne répondent pas au deuxième critère R_2. Nous remarquons également, qu'à l'exception des travaux de Calì et al. [19], les travaux existants proposent l'implémentation en dur des opérateurs de relaxation dans les BD-RDF et non pas dans le langage de requêtes [221, 18, 225].

Seules les contributions de Calì et al. [19] respectent l'ensemble des critères que nous avons identifiés. Mais, les opérateurs proposés ne laissent pas le choix aux utilisateurs sur, d'une part, la technique de relaxation à utiliser et, d'autre part, la partie de la requête à relaxer : ces opérateurs ne permettent pas de contrôler précisément la relaxation. Nous présentons dans les sections suivantes des opérateurs de relaxation paramétrables que nous avons proposés dans [232, 233] et leur intégration dans SPARQL. L'un des avantages de cette intégration est la réutilisation de ces opérateurs dans différents BD-RDF et processus de relaxation.

Dans la suite de ce chapitre, sauf mention contraire, nous utiliserons le graphe de données 3.10 utilisé dans le chapitre précédent pour illustrer les techniques de relaxation sur les données RDF.

4.2 Opérateurs de relaxation

Cette section propose des opérateurs de relaxation et une implémentation de ces derniers. Chaque opérateur proposé est évalué suivant les critères 3 et 4 de la section précédente. Les trois autres critères sont évalués dans la section suivante pour l'ensemble des opérateurs. Nous nous sommes basés sur les transformations élémentaires des requêtes, partagées par les processus de relaxation pour concevoir ces opérateurs de relaxation. Dans le but de réaliser dans un premier temps une preuve de concept, nous avons limité notre étude aux transformations suivantes :

- *le relâchement de valeur* : cette transformation est utilisée pour définir l'opérateur *PRED* ;
- *la substitution de concept ou prédicat* : transformation utilisée par l'opérateur *SIB* ;
- *la généralisation de concept ou prédicat* : transformation utilisée par l'opérateur *GEN*.

4.2.1 Relâchement : PRED

L'opérateur PRED utilise les principes des prédicats graduels pour relaxer les valeurs des littéraux dans une requête SPARQL. Cet opérateur relâche une contrainte en élargissant l'ensemble des valeurs valides pour cette contrainte. Pour le cas des contraintes numériques, que nous avons implémentées, l'extension

de l'intervalle [a, b] se fait par extension de son *support* $b - a$. Supposons, par exemple, qu'un utilisateur recherche *des professeurs (FullProfessor) ayant un âge compris entre 25 et 30 ans*. L'application de l'opérateur PRED sur cet intervalle permet de relâcher cette contrainte pour obtenir l'intervalle [20, 35]. Pour l'implémentation de l'opérateur PRED, ainsi défini, les questions suivantes se posent.

1. Comment contrôler l'extension du support de l'intervalle ?
2. Comment classer les réponses trouvées dans les extensions du support entre elles et par rapport à celles trouvées dans le support original ?

Les réponses à ces questions permettrons à l'opérateur PRED de remplir les critères 3 et 4 respectivement.

4.2.1.1 Critère 3 : Classement des réponses alternatives avec PRED

Les prédicats graduels utilisés par l'opérateur PRED sont modélisés au moyen d'ensembles flous. Dans cette sous-section, nous utilisons la définition formelle d'un ensemble flou de Zadeh [234]. Un ensemble flou F sur un domaine U est un ensemble muni d'une fonction d'appartenance $\mu_F : U \rightarrow [0, 1]$ tel que $\forall a \in U, \mu_F(a)$ soit le degré d'appartenance de a à F. Pour un élément $a \in U$, $\mu_F(a) = 0$ implique que $a \notin F$ et réciproquement $\mu_F(a) = 1$ dénote l'appartenance totale de a à F. Lorsque $0 < \mu_F(a) < 1$ on parle d'appartenance partielle de a à F. PRED utilise la fonction μ pour classer les réponses alternatives : celles ayant une valeur proche de 1 sont plus pertinentes que celles ayant un degré proche de 0. Cette modélisation divise un ensemble flou en deux sous-ensembles : *le core* et le *support* de l'ensemble flou.

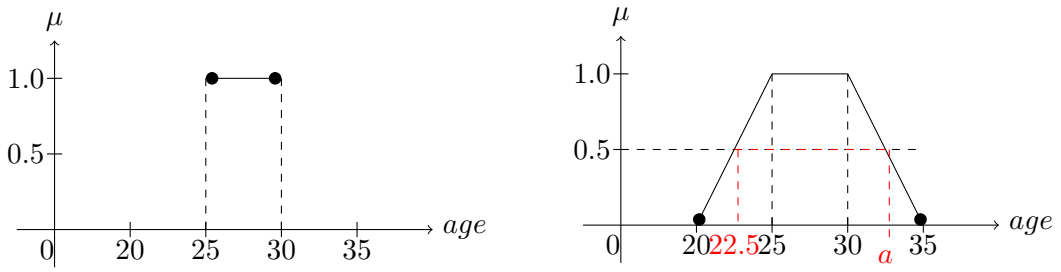
Définition 3

> *Le core d'un ensemble flou F noté $C(F)$ est l'ensemble des éléments ayant un degré d'appartenance égal à 1 :* $C(F) = \{a \in U \mid \mu_F(a) = 1\}$.

Définition 4

> *Le support d'un ensemble flou F noté $S(F)$ est l'ensemble des éléments ayant un degré d'appartenance strictement supérieur à 0 :* $S(F) = \{a \in U \mid \mu_F(a) > 0\}$.

Ainsi, un intervalle flou est noté (α, a, b, β) où $[a, b]$ est le *core* et $[\alpha, \beta]$ *le support*. Reprenons notre exemple sur la recherche *des professeurs (FullProfessor) ayant un âge compris entre 25 et 30 ans*. Une relaxation de $[25, 30]$ peut être l'ensemble flou $F = (20, 25, 30, 35)$. Dans cette relaxation, le *core* reste l'ensemble initial et le *support* est $[20, 30]$ qui est une extension de cet ensemble. La détermination du support est présentée dans la sous-section 4.2.1.2.



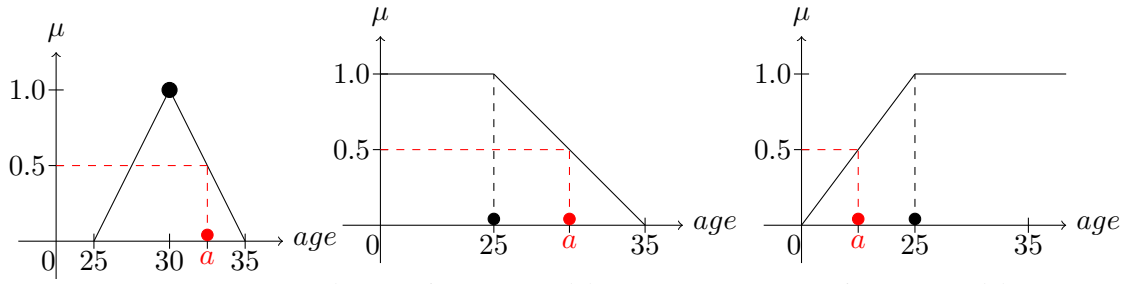
(a) Ensemble classique, $C(F)$

(b) Ensemble flou avec $\mu(x) \in [0, 1]$

FIGURE 4.1 – Ensemble flou

La figure 4.1a montre la représentation graphique de l'ensemble classique qui correspond au *core* de l'ensemble flou. La figure 4.1b présente graphiquement l'ensemble flou, son *support* et son *core*. Sur

ces figures, le degré d'appartenance se trouve sur l'axe des ordonnées. Ainsi, pour ordonner une réponse alternative a d'une requête relaxée avec l'opérateur PRED, nous évaluons $\mu_F(a)$ et le comparons aux degrés d'appartenance des réponses déjà trouvées. Nous appelons la fonction μ_F le degré de satisfaction d'une réponse d'une requête relaxée avec l'opérateur PRED. Par exemple, sur la figure 4.1b, nous avons $\mu_F(22.5) = 0.5$ ce qui traduit que le degré de satisfaction de l'âge 22.5 d'un *professeur* est de 0.5. Il existe différentes courbes du degré de satisfaction : trapézoïdale, $(20, 25, 30, 35)$, comme dans la figure 4.1b, triangulaire, $(25, 30, 30, 35)$, (figure 4.2a), trapézoïdale ouvert à gauche, $(-\infty, 25, 25, 35)$, (figure 4.2b) ou à droite, $(0, 25, 25, +\infty)$, (figure 4.2c). Le calcul du degré de satisfaction $\mu_F(a)$ varie en fonction de l'ensemble flou F. Cependant, dans tout les cas $\forall\ a \in C(F),\ \mu_F(a) = 1$.



(a) Fonction triangulaire

(b) Fonction trapézoïdale ouverte à gauche

(c) Fonction trapézoïdale ouverte à droite

FIGURE 4.2 – Différentes fonctions d'appartenance

1er **Cas (fonction trapézoïdale ouverte à gauche)** : $C(F) =]-\infty, \beta]$, figure 4.2b, c'est le cas d'une inégalité. Le support est de la forme $S(F) =]-\infty, \beta_1]$ avec $\beta < \beta_1$. La formule de calcul de $\mu_F(a)$ est la suivante [22]:

$$\mu_F(a) = \begin{cases} \forall\ a \leq \beta,\ \mu_F(a) = 1 \\ \forall\ a \in [\beta, \beta_1],\ \mu_F(a) = \dfrac{\beta_1 - a}{\beta_1 - \beta} \\ \forall\ a \geq \beta_1,\ \mu_F(a) = 0 \end{cases} \tag{4.1}$$

2^{ieme} **Cas (fonction trapézoïdale ouverte à droite)** : $C(F) = [\alpha, +\infty[$, figure 4.2c, c'est l'inégalité contraire de la précédente. Ici, le support est de la forme $S(F) = [\alpha_1, +\infty[$ avec $\alpha_1 < \alpha$. La formule de calcul de $\mu_F(a)$ est :

$$\mu_F(a) = \begin{cases} \forall\ a \geq \alpha,\ \mu_F(a) = 1 \\ \forall\ a \in [\alpha_1, \alpha],\ \mu_F(a) = 1 - \dfrac{\alpha - a}{\alpha - \alpha_1} \\ \forall\ a \leq \alpha_1,\ \mu_F(a) = 0 \end{cases} \tag{4.2}$$

3^{ieme} **Cas (fonction trapézoïdale)** : $C(F) = [\alpha, \beta]$, figure 4.1b, c'est la combinaison des deux précédents cas. Deux inégalités sont appliquées sur une même variable et donc le *core* devient un intervalle. Le support devient $S(F) = [\alpha_1, \beta_1]$ avec $\alpha_1 < \alpha$ et $\beta_1 > \beta$ et $\mu_F(a)$:

22. Théorème des accroissements finis : $\dfrac{\mu_F(\beta_1) - \mu_F(\beta)}{\beta_1 - \beta} = \dfrac{\mu_F(\beta_1) - \mu_F(a)}{\beta_1 - a}$ avec $(\beta) = 1$ et $\mu_F(\beta_1) = 1$

$$\mu_F(a) = \begin{cases} \forall\, a \in [\alpha, \beta],\ \mu_F(a) = 1 \\ \forall\, a \in [\alpha_1, \alpha],\ \mu_F(a) = 1 - \dfrac{\alpha - a}{\alpha - \alpha_1} \\ \forall\, a \leq \alpha_1,\ \mu_F(a) = 0 \\ \forall\, a \in [\beta, \beta_1],\ \mu_F(a) = \dfrac{\beta_1 - a}{\beta_1 - \beta} \\ \forall\, a \geq \beta_1,\ \mu_F(a) = 0 \end{cases} \tag{4.3}$$

4^{ieme} **Cas (fonction triangulaire)** : $C(F) = \alpha$, figure 4.2a, c'est le cas d'une égalité. Le *core* est réduit à une valeur et le support est $S(F) = [\alpha_1, \beta_1]$. $\mu_F(a)$ se calcul comme suit :

$$\mu_F(a) = \begin{cases} \forall\, a = \alpha,\ \mu_F(a) = 1 \\ \forall\, a \in [\alpha_1, \alpha],\ \mu_F(a) = 1 - \dfrac{\alpha - a}{\alpha - \alpha_1} \\ \forall\, a \leq \alpha_1,\ \mu_F(a) = 0 \\ \forall\, a \in [\alpha, \beta_1],\ \mu_F(a) = \dfrac{\beta_1 - a}{\beta_1 - \alpha} \\ \forall\, a \geq \beta_1,\ \mu_F(a) = 0 \end{cases} \tag{4.4}$$

Pour les types de données numériques, nous pouvons donc calculer le degré de satisfaction de chaque réponse par rapport à la valeur alternative de la variable relaxée et à la contrainte sur cette variable.

4.2.1.2 Critère 4 : Contrôle de la relaxation avec PRED

Comme nous le mentionnons précédemment, le contrôle de la relaxation avec l'opérateur PRED passe par le calcul du support $S(F)$ de l'ensemble flou F modélisant le prédicat graduel. Plus précisément, il correspond au calcul des bornes α et/ou β du support de relaxation d'un ensemble flou (α, a, b, β) relaxant l'intervalle initial $[a, b]$. Évidemment, le calcul du support de l'ensemble flou dépendra du type d'intervalle initial, qui peut par exemple avoir la forme $[-\infty, b]$ ou encore $[a, +\infty]$.

Nous utilisons pour ce calcul les travaux de Hadjali et al. [235] sur la relation de proximité entre deux valeurs numériques. Dans ces travaux, ils considèrent la relation de proximité $Cl(x, y)$ entre deux valeurs numériques x et y et montre qu'elle est symétrique :

$$Cl(x, y) = Cl(y, x) \tag{4.5}$$

Afin de modéliser la proximité entre deux valeurs numériques, x et y, il est possible d'utiliser la soustraction, $x - y$, ou la division, $\frac{x}{y}$. Hadjali et al. utilise la division $\frac{x}{y}$, qu'ils comparent à 1 pour évaluer la proximité entre x et y. Ainsi, ils considèrent l'intervalle $M_0 = [1, 1]$ et le paramètre de tolérance $\Gamma_M(\frac{x}{y})$ entre x et y qui permet une extension de M_0 en M. La proximité $Cl(x, y)$ est donc définie comme suit :

$$Cl(x, y) = \Gamma_M(\frac{x}{y}) \tag{4.6}$$

L'équation (4.6) peut se lire ainsi : *deux nombres sont "proches" si leur rapport est "proche" de 1*, qui est représenté par M_0. $\Gamma_M(\frac{x}{y})$ est une fonction qui retourne une valeur dans l'intervalle $[0, 1]$, qui est utilisée pour étendre le support de M_0 pour obtenir un intervalle flou M, tel que :

— $\Gamma_M(\frac{x}{x}) = \Gamma_M(1) = 1$, car x est proche de x ;

— si $\frac{x}{y} < 0$ alors $\Gamma_M(\frac{x}{y}) = 0$ car deux nombres de signes opposés ne peuvent pas être proches.

De plus, $Cl(x, y)$ est symétrique et $Cl(x, y) = \Gamma_M(\frac{x}{y})$ donc $\Gamma_M(\frac{x}{y})$ est également symétrique, d'où :

$$\Gamma_M(\frac{x}{y}) = \Gamma_M(\frac{y}{x}),\ en\ posant\ t = \frac{x}{y}\ nous\ obtenons\ \Gamma_M(t) = \Gamma_M(\frac{1}{t}) \tag{4.7}$$

Hadjali et al. [235] démontrent que cette propriété est vérifiée pour $t \in S(M) = [1 - \epsilon, \frac{1}{1-\epsilon}]$ avec $\epsilon \in [0, 1]$. Ainsi, le paramètre ϵ permet de définir le support $S(M)$ de M, qui lui même permet de définir la proximité entre x et y, et nous avons ainsi l'intervalle flou $M = (1 - \epsilon, 1, 1, \frac{1}{1-\epsilon})$. Ils démontrent également que la proximité est meilleure entre x et y si $S(M) \subseteq V = [\frac{(\sqrt{5}-1)}{2}, \frac{(\sqrt{5}+1)}{2}]$, l'intervalle V est appelé *intervalle de validité*. Ainsi, en fixant le paramètre ϵ nous obtenons un intervalle M de tolérance qui est meilleur lorsqu'il est inclus dans V : ϵ est la tolérance de la relaxation.

Nous utilisons ensuite M pour relaxer les prédicats de la façon suivante. Considérons un prédicat P, défini sur le domaine U, auquel est associé l'ensemble $I_P = (a, a, b, b)$, suivant la notation trapézoïdale de $[a, b]$, et un prédicat P' qui est une relaxation de P auquel est associé l'ensemble flou $I_{P'} = (\alpha, a, b, \beta)$. Hadjali et al. montrent que $P' = P \otimes M$ où $\otimes$ représente le produit entre deux ensembles flous, cet opérateur a été défini par Dubois et Prade. [236].

Preuve : Considérons le degré d'appartenance d'un élément $x \in U$ à I_P : $\mu_P(x) = \begin{cases} 1\ si\ x \in [a, b] \\ 0\ sinon \end{cases}$.

Le degré d'appartenance d'un élément $x \in U$ à $I_{P'}$ est formellement défini comme suit [235] :

$$\begin{aligned} \forall\, x \in U, \mu_{P'}(x) &= \sup_{y \in U}\ \min(\mu_P(y), Cl(x, y)) \\ &= \sup_{y \in U}\ \min(\mu_P(y), \Gamma_M(y/x)) \\ &= \sup_{y \in U}\ \min(\mu_P(y), \Gamma_M(x/y)) \\ &= \mu_{P \otimes M}(x) \end{aligned} \tag{4.8}$$

Cette équation (4.8) est obtenue via le principe d'extension détaillé par Dubois et Prade. [236]. Ainsi, si $x \in I_P$ alors $\mu_{P'}(x) = 1$. En effet, pour $x \in I_P$ la valeur maximale de $\min(\mu_P(y), \Gamma_M(y/x))$ est obtenue pour $x = y$ avec $\mu_P(x) = 1$ (car $x \in I_P$, $\Gamma_M(x/x) = 1$, $\mu_P(x) \in [0, 1]$ et $\Gamma_M(y/x) \in [0, 1]$) d'où $\mu_{P'}(x) = 1$ pour $x \in I_P$. De même, si $x \notin I_P$ alors $\mu_{P'}(x) = \sup_{y \in I_P}\ \Gamma_M(y/x)$. En effet, lorsque y prend les valeurs dans U deux cas de figure peuvent apparaitre. Considérons, d'une part, $y \in I_P$ alors $\mu_P(y) = 1$ et donc $\min(\mu_P(y), \Gamma_M(y/x)) = \Gamma_M(y/x)$. Considérons, d'autre part, $y \notin I_P$ alors $\mu_P(y) = 0$ et donc $\min(0, \Gamma_M(y/x)) = 0$. Par conséquent :

$$\begin{aligned} \sup_{y \in U}\ \min(\mu_P(y), \Gamma_M(y/x)) &= \sup(\sup_{y \in I_P} \Gamma_M(y/x), \sup_{y \in U - I_P}(0)) \\ &= \sup(\sup_{y \in I_P} \Gamma_M(y/x)) \\ &= \sup_{y \in I_P} \Gamma_M(y/x) \end{aligned}$$

Avec la formule $P' = P \otimes M$ nous pouvons ainsi transformer un prédicat P (resp. un intervalle I_P) en un prédicat relaxé P' (resp. un intervalle flou $I_{P'}$), avec $I_{P'} = (a * (1 - \epsilon), a, b, b * (\frac{1}{1-\epsilon})$.

Exemple 1. Reprenons notre exemple précédent sur le prédicat âge associé à l'intervalle flou $I_p = (25, 25, 30, 30)$. Lorsque nous considérons $\epsilon = 0.1$, nous obtenons $M = (0.9, 1, 1, 1.11)$ et nous avons bien $S(M) \subseteq V \approx [0.61, 1.61]$. Avec cette valeur de ϵ, nous obtenons $I_{P'} = (22.5, 25, 30, 33.33)$.

La valeur de tolérance ϵ fixe l'intervalle M et permet ainsi l'extension du prédicat et donc le contrôle de la relaxation. Cette transformation peut également être utilisée de façon itérative avec la formule : $P^n = P \otimes M^n$. De plus, le degré d'appartenance $\mu_{P'}$ est exploité pour ordonner des réponses alternatives.

4.2.2 Substitution : SIB

L'opérateur SIB permet de substituer un concept présent dans la requête par un autre concept de l'ontologie RDF. La substitution pour les concepts de l'ontologie est semblable au relâchement de prédicat qui substitue un prédicat P par un prédicat P'. À la place du degré de satisfaction pour les prédicats, SIB utilise des mesures de similarité pour comparer les concepts substitués. L'opérateur SIB satisfait également les critères de contrôle de la relaxation et de l'ordonnancement des réponses alternatives.

4.2.2.1 Critère 3 : Classement des réponses alternatives avec SIB

SIB ordonne les réponses alternatives en se basant sur la similarité entre la classe substituée c et la classe qui la substitue c'. Cette similarité est notée $sim(c, c')$. Ainsi, considérons deux classes c' et c" utilisées pour substituer c dans une requête SPARQL. Si $sim(c, c') \geq sim(c, c")$ alors les réponses alternatives obtenues par la substitution de c par c' sont plus pertinentes que celles obtenues par la substitution de c par c". C'est ainsi que sont ordonnées les réponses alternatives retournées au moyen de l'opérateur SIB. Pour notre implémentation, nous avons utilisé les mêmes formules (4.9) que Huang et al. [17], pour calculer la similarité entre deux concepts, c et c' pour les classes, et, p et p' pour les propriétés.

$$
\begin{aligned}
sim(c, c') &= \frac{IC(msca(c, c'))}{IC(c) + IC(c') - IC(msca(c, c'))} \\
sim(p, p') &= \frac{IC(msca(p, p'))}{IC(p) + IC(p') - IC(msca(p, p'))}
\end{aligned}
\tag{4.9}
$$

où $msca(c, c')$ est le plus proche ancêtre commun à c et c' dans la hiérarchie et différent de c et c', $IC(c) = -log(Pr(c))$ et $Pr(c) = \frac{|c|}{|\mathbb{D}|}$ la probabilité qu'une instance de l'ensemble des données D soit une instance de la classe c. Cette mesure a été adoptée comme mesure par défaut car elle prend en considération dans son calcul à la fois la distance dans le graphe entre les concepts (*distance-based*) et la quantité d'information contenue dans les différentes classes (*information-based*).

Exemple 2. Reprenons notre requête exemple de recherche *des professeurs (FullProfessor) ayant un âge compris entre 25 et 30 ans* dans le graphe de données 3.10. Pour substituer $c = FullProfessor$ par $c' = AssociateProfessor$ nous avons : $msca(c, c') = Professor$,
$IC(c) = -log(2/7)$, $IC(c') = -log(2/7)$ et $IC(msca(c, c')) = -log(5/7)$. Ainsi,

$$
sim(c, c') = \frac{-log(5/7)}{-log(2/7) - log(2/7) + log(5/7)} \approx 0.1550
$$

Les réponses alternatives fournies par la classe $AssociateProfessor$ ont donc une similarité de 0.1550 par rapport aux réponses attendues par les utilisateurs. Nous pouvons observer que ces deux classes ont une similarité faible, même si elles sont pourtant proches selon la topologie du graphe de données. Cette similarité est faible car son calcul intègre le nombre d'instances (quantité d'information) de chaque classe et de la super-classe directe de ces deux classes.

4.2.2.2 Critère 4 : Contrôle de la relaxation avec SIB

L'opérateur SIB contrôle la relaxation de deux façons différentes.

Utilisation des classes sœurs : Par défaut, l'opérateur SIB remplace une classe c utilisée dans une requête par ses classes sœurs dans l'ordre de similarité. Notons $C_S(c)$ l'ensemble des classes sœurs de c, $C_S(c)$ est défini formellement comme suit : $C_S(c) = \{\ c' \mid directSuperClass(c') \cap$

$directSuperClass(c) \neq \emptyset$ }, où $directSuperClass(c)$ est une fonction qui permet de retrouver la ou les super-classe(s) directe(s) de c. Une classe s est dite super-classe d'une classe c si le triplet *(c, rdfs:subClassOf, s)* existe dans la base de données RDF sans avoir été déduit à l'aide de règles de raisonnement. Une fois l'ensemble $C_S(c)$ déterminé, la substitution se fait entre c et c', $c' \in C_S(c)$, jusqu'à l'obtention des TOP-K réponses alternatives.

Exemple 3. Pour notre exemple précédent où $c = FullProfessor$ nous avons :
$C_S(c) = \{AssociateProfessor, AssistantProfessor\}$, avec pour $c' = AssociateProfessor$ $sim(c, c') \approx 0.1550$ et pour $c' = AssistantProfessor$ $sim(c, c') = 0.1175$.

Utilisation des préférences utilisateurs : l'opérateur SIB peut également être utilisé en remplaçant les classes sœurs de c, $C_S(c)$, par un ensemble de classes défini par l'utilisateur suivant ses préférences sur les données. Dans ce cas, l'utilisateur doit avoir une connaissance de l'ontologie et de ses concepts, ce qui n'est pas toujours le cas.

Quel que soit le mode de fonctionnement de l'opérateur SIB, sur la base des classes sœurs ou des préférences utilisateurs, la substitution est réalisée en tenant compte de la similarité entre les concepts substitués. L'opérateur s'arrête lorsqu'il obtient les TOP-K réponses alternatives pour les utilisateurs. SIB peut également être utilisé pour les propriétés.

4.2.3 Généralisation : GEN

L'opérateur GEN est une variante de SIB qui substitue un concept par les concepts qui le subsument. Nous parlons de *généralisation* ou de *relaxation verticale* de concepts (classes ou propriétés). La relaxation d'une classe c avec l'opérateur GEN, calcule l'ensemble des super-classes $S_C(c)$ à partir de l'ontologie. Comme dans le cas des classes sœurs, les super-classes sont classées au moyen de la mesure de similarité donnée par la formule (4.9). Dans la plupart des cas, cet ordre correspond à l'ordre dans la hiérarchie de l'ontologie. Dans ce type de relaxation, comme dans SIB, si nous considérons une classe c relaxée en c', les propriétés $p_1, p_2, ..., p_n$ définies sur c, *(p_i, rdfs:domain, c)*, et utilisées dans la requête sont transformées de la manière suivante :

- si la propriété p_i est également définie sur c', c'est-à-dire si nous avons *(p_i, rdfs:domain, c')*, alors elle est conservée dans la requête ;
- s'il existe une propriété p'_i tel que *(p'_i, rdfs:domain, c')* et *(p_i, rdfs:subProperty, p'_i)*, alors p_i est substituée par p'_i ;
- si aucune des conditions précédentes n'est vérifiée, la propriété est substituée par une variable, autrement dit, elle est supprimée.

De même, pour un ensemble de propriétés $p_1, p_2, ..., p_n$ utilisées dans la requête et ayant pour codomaine c, *(p_i, rdfs:range, c)*, les transformations sont similaires :

- si la propriété p_i a également pour codomaine c', c'est-à-dire nous avons *(p_i, rdfs:range, c')* alors elle est conservée dans la requête ;
- s'il existe une propriété p'_i tel que *(p'_i, rdfs:range, c')* et *(p_i, rdfs:subProperty, p'_i)*, alors p_i est substituée par p'_i ;
- si aucune des conditions précédentes n'est vérifiée, la propriété est substituée par une variable, elle est donc supprimée.

Le même raisonnement est utilisé pour la généralisation des propriétés. Les classes utilisées dans la requête et liées par *domaine* ou *codomaine* à une propriété généralisée sont soit conservées, soit généralisées, soit supprimées par substitution par une variable. Comme pour l'opérateur SIB, la généralisation GEN utilise la mesure de similarité de la formule (4.9) pour ordonner les réponses alternatives. Dans le cas des suppressions de concepts, classes ou propriétés, nous adoptons la mesure suivante : $sim(c, v) = 0$

et $sim(p, v) = 0$ où c est une classe, p une propriété et v une variable, distincte des autres variables utilisées dans la requête. GEN, comme SIB, répond ainsi aux critères 3 et 4 définis dans la section 4.1.2.

Exemple 4. Reprenons une fois de plus notre exemple précédent sur le graphe de données 3.10. Pour $c = FullProfessor$ nous avons $S_C(c) = \{Professor, Employee, Person\}$. Pour toutes ces classes la propriété age est définie donc elle sera conservée. Pour calculer la similarité, si $c' = Professor$, alors $msca(c, c') = Employee$; si $c' = Employee$, alors $msca(c, c') = Person$ et, si $c' = Person$, alors $msca(c, c') = Person$ car il s'agit de la racine de la hiérarchie.

Maintenant que nos opérateurs sont définis et présentés, nous pouvons passer à l'intégration de ces opérateurs dans le langage SPARQL. Cette intégration permet également l'évaluation de notre proposition suivant les critères 1, 2 et 5.

4.3 Intégration des opérateurs dans SPARQL

Dans cette section, nous spécifions les clauses correspondantes aux opérateurs de relaxation précédents avec leurs syntaxes associées dans le langage SPARQL. Cette extension de SPARQL définit une mesure de classement des réponses en s'appuyant sur le degré de satisfaction et les mesures de similarité présentées dans les sections précédentes.

4.3.1 Clauses et syntaxe

Nous étendons le langage SPARQL avec, principalement, l'ajout d'une clause `APPROX` dans laquelle la relaxation est décrite. La syntaxe associée à cette nouvelle clause est la suivante :

$$\langle \textbf{approx clause} \rangle ::= \texttt{APPROX} \ \langle \textbf{approx expr} \rangle \ [\texttt{TOP} \ \langle \textbf{Integer} \rangle] \quad (4.10)$$

$$\langle \textbf{approx expr} \rangle ::= \langle \textbf{approx term} \rangle \mid \langle \textbf{approx expr} \rangle \ \texttt{OR} \ \langle \textbf{approx term} \rangle \quad (4.11)$$

$$\langle \textbf{approx term} \rangle ::= \langle \textbf{approx op} \rangle \mid \langle \textbf{approx term} \rangle \ \texttt{AND} \ \langle \textbf{approx op} \rangle \quad (4.12)$$

$$\langle \textbf{approx op} \rangle ::= \langle \textbf{pred operator} \rangle \mid \langle \textbf{sib operator} \rangle \mid \langle \textbf{gen operator} \rangle \quad (4.13)$$

$$\langle \textbf{pred operator} \rangle ::= \texttt{PRED} \ (\ \langle \textbf{var} \rangle \ [, \langle \textbf{real} \rangle] \ [, \langle \textbf{interval} \rangle] \ ) \quad (4.14)$$

$$\langle \textbf{sib operator} \rangle ::= \texttt{SIB} \ (\ \langle \textbf{uri} \rangle \ [, [\langle \textbf{uri} \rangle^*]] \ ) \quad (4.15)$$

$$\langle \textbf{gen operator} \rangle ::= \texttt{GEN} \ (\ \langle \textbf{uri} \rangle \ [, \langle \textbf{integer} \rangle] \ ) \quad (4.16)$$

Dans cette extension, nous incluons également la syntaxe des différents opérateurs de relaxation que nous avons présentés dans la section précédente. Nous présentons en détail la syntaxe de chaque opérateur.

PRED : L'expression (4.14) présente la syntaxe de l'opérateur PRED. Dans cette formule, la variable **var** désigne le ou les prédicats à relaxer, le nombre réel $real$ est la tolérance à appliquer pour relaxer **var**, correspondant à ϵ dans la formule de M, et l'intervalle, $interval$, passé en paramètre permet de déterminer l'intervalle de validité, qui comme nous l'avons vu précédemment devra toujours contenir le support $S(M)$ de l'intervalle flou M qui permet d'étendre le prédicat. Les deux derniers paramètres de cet opérateur sont facultatifs du fait que les utilisateurs peuvent ne pas avoir une connaissance suffisante des données pour les initialiser. La valeur par défaut de ces paramètres sont 0.1 pour ϵ et $[\frac{(\sqrt{5}-1)}{2}, \frac{(\sqrt{5}+1)}{2}]$ pour l'intervalle de validité est V de Hadjali et al. [235], présenté dans la sous-section 4.2.1.2.

Exemple 5. Dans la requête ci-contre, l'utilisateur désire relaxer la propriété age. Toutes les conditions sur la variable $?age$ sont réunies pour donner le prédicat $25 \leq age \leq 35$. La valeur $\epsilon = 0.1$ est utilisée pour déterminer l'intervalle M. L'utilisateur définit l'intervalle de validité : $[0.4, 1.7]$. Le calcul de la première relaxation du prédicat age se fait ainsi : $(25, 25, 35, 35) \otimes (0.9, 1, 1, 1.11) = (22.5, 25, 35, 38.85)$, et les conditions deviennent $22.5 \leq age \leq 38.85$. Le degré de satisfaction des réponses alternatives est calculé pour les ordonner.

```
SELECT  ?pr ?age
WHERE   {?prrdf:type FullProfessor. (t1)
         ?pr age ?age. (t2)
         ?pr salaire ?s. (t3)
         FILTER (25 ≤?age AND ?age ≤ 35
         AND ?s < 3500).
         #Bloc de relaxation
         APPROX PRED (?age, 0.1, [0.4, 1.7])}
```

SIB : La syntaxe de l'opérateur SIB est décrite par la formule (4.15). Dans cette dernière, la première **uri** est l'URI du concept à substituer et les **uri** suivantes sont celles des concepts de substitution. Si les utilisateurs ne donnent aucune URI de substitution, par défaut, l'opérateur SIB utilise les concepts frères du concept à relaxer.

Exemple 6. Dans l'exemple ci-contre, la classe $FullProfessor$ sera substituée par les classes $AssociateProfessor$ et $AssistantProfessor$ successivement. Dans le cas de plusieurs occurrences de $FullProfessor$ dans la requête, elles sont toutes substituées par la nouvelle classe pour obtenir la requête relaxée. SIB permet de relaxer des classes ou des propriétés. Le codomaine de *rdf:type* est vérifié pour les deux autres classes.

```
SELECT  ?pr ?age
WHERE   {?pr rdf:type FullProfessor. (t1)
         ?pr age ?age. (t2)
         ?pr salaire ?s. (t3)
         FILTER (25 ≤?age AND ?age ≤ 35
         AND ?s < 3500).
         #Bloc de relaxation
         APPROX SIB (FullProfessor,
         [AssociateProfessor, AssistantProfessor])}
```

GEN L'opérateur de généralisation GEN respecte la syntaxe de la formule (4.16). Dans cette syntaxe, l'URI **uri** désigne le concept à généraliser et l'entier le niveau de généralisation dans la hiérarchie. Ce niveau de généralisation prend comme référence le concept à généraliser. Le niveau de relaxation par défaut de cet opérateur est le niveau de la racine de l'ontologie. Dans l'exemple suivant, la requête est généralisée jusqu'au troisième ancêtre de $FullProfessor$. Il s'agit du chemin suivant dans le graphe : $FullProfessor\ (0) \rightarrow Professor\ (1) \rightarrow Employee\ (2) \rightarrow Person\ (3)$.

```
SELECT  ?pr ?age
WHERE   {
         ?pr rdf:type FullProfessor. (t1)
         ?pr age ?age. (t2)
         ?pr salaire ?s. (t3)
         FILTER (25 ≤?age AND ?age ≤ 35 AND ?s < 3500).
         #Bloc de relaxation
         APPROX GEN (FullProfessor, 3)}
```

L'utilisation d'une combinaison d'opérateurs de relaxation donne lieu à une interprétation précise. Cette interprétation est liée à la sémantique de ces opérateurs, elle même dépendante de la définition des opérateurs présentées dans la section précédente. La section suivante présente la sémantique liée aux opérateurs de relaxation et aux combinaisons valides de ces opérateurs de relaxation.

4.3.2 Sémantique des opérateurs de relaxation et leurs combinaison

Les opérateurs de relaxation ou leurs combinaisons transforment les requêtes utilisateurs, notées Q, en requêtes dites relaxées notées Q'. Ces transformations se font de façon itérative et incrémentale. Dans cette section, nous présentons comment les instructions de relaxation sont interprétées lors de la transformation de la requête initiale Q en requête relaxée Q' pour retrouver des réponses alternatives. Nous définissons la similarité entre la requête initiale et la requête relaxée $Sim(Q, Q')$ qui est calculée à partir des similarités des opérateurs de relaxation utilisés pour obtenir Q'.

PRED : $P \times Real \times Interval \longrightarrow P'$. L'opérateur `PRED` transforme une requête Q avec un prédicat P en une requête relaxée Q' dans lequel le prédicat P est substitué par P' calculé avec la formule $P' = P \otimes M$. Dans la signature de l'opérateur $PRED$, P est une variable sur laquelle est définie des contraintes que nous considérons que nous appelons par prédicat P. Dans cette formule, la contrainte sur P est un intervalle $[a, b]$ dont nous considérons la forme trapézoïdale des ensembles flous suivante $P = (a, a, b, b)$. Le calcul de P' donne :
$P' = (a, a, b, b) \otimes (1 - \epsilon, 1, 1, \frac{1}{1-\epsilon}) = (a(1-\epsilon), a, b, b * \frac{1}{1-\epsilon})$.

De manière générale, la relaxation d'un prédicat flou $P^{(i+1)} = P^i \otimes M$ avec $P^i = (a_i, a, b, b_i)$ nous donne $P^{(i+1)} = (a_i, a, b, b_i) \otimes (1 - \epsilon, 1, 1, \frac{1}{1-\epsilon}) = (a_{(i+1)}, a, b, b_{(i+1)})$ avec $a_{(i+1)} = a_i(1-\epsilon)$ et $b_{(i+1)} = b_i * \frac{1}{1-\epsilon}$. C'est ainsi que le processus itératif de relaxation est défini pour l'opérateur `PRED`, chaque itération est appelée *étape de relaxation*, et le degré de satisfaction est calculé comme nous l'avons présenté plus haut. La figure 4.3 représente le forme graphique des prédicats sous forme trapézoïdale. Pour les cas particuliers où a ou b est $+/-\ \infty$ le calcul se fait en remplaçant a ou b par cette valeur et donc associe à l'une des bornes une valeur infinie.

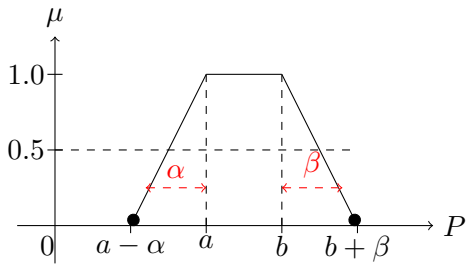


FIGURE 4.3 – Exemple de relaxation de prédicat

SIB : $uri \times List \prec uri \succ \longrightarrow uri_i$. `SIB` transforme une requête Q contenant une classe c correspondant à l'URI uri en requête relaxée Q_i contenant une classe c_i, correspondant à la $i^{ième}$ classe de la liste $List \prec uri \succ$. Chaque substitution de c par c_i constitue une étape de relaxation de l'opérateur `SIB`. Les similarités sont calculées avec la mesure de la formule (4.9), $Sim(Q, Q') = sim(c, c')$.

GEN : $uri \times Integer \longrightarrow uri$. L'opérateur `GEN` transforme une requête Q en requête relaxée Q' dans laquelle la classe c avec l'URI uri est substituée par des classes, c', obtenues par généralisation itérative de c. Le nombre d'itérations de la généralisation est égale au paramètre $Integer$ de l'opérateur `GEN`. Chaque itération, donc généralisation, est une étape de la relaxation. Comme pour le cas de l'opérateur `SIB`, la similarité de l'opérateur `GEN` est calculée avec la mesure de la formule (4.9) et nous avons $Sim(Q, Q') = sim(c, c')$.

Les opérateurs de relaxation peuvent être combinés entre eux avec l'opérateur `AND`. Dans ce cas, l'agrégation des mesures de similarité se fait via l'opérateur d'agrégation min. Nous avons choisi cet opérateur d'agrégation pour ses propriétés algébriques et afin d'adopter une attitude pessimiste qui représente le pire cas. L'interprétation de cette opérateur se fait de la manière suivante :

AND : La conjonction des opérateurs de relaxation dépend de l'opérateur de relaxation. Étant donné que l'opérateur `PRED` relaxe des prédicats et que les opérateurs `GEN` et `SIB` relaxent les classes et propriétés, ces deux catégories d'opérateurs peuvent s'exécuter simultanément. Pour la conjonction des opérateurs de ces deux catégories nous distinguons deux cas : les deux opérateurs relaxent le même paramètre ou deux paramètres différents.

Paramètres identiques Prenons le cas des opérateurs `GEN` et `SIB`. Considérons Q avec la classe c et Q' la requête relaxée obtenue avec la combinaison d'opérateurs de relaxation suivante $\texttt{GEN}(c, i)$ `AND` $\texttt{SIB}(c, [c_1,\ c_2, ...,\ c_n])$. Dans ce cas c est substituée par la classe la plus similaire entre $c_1,\ c_2, ...,\ c_n$ et les super-classes de c jusqu'à la $i^{ième}$ généralisation. Pour la conjonction entre deux opérateurs `GEN` sur la même classe, le paramètre i considéré est le plus grand. Pour l'opérateur `SIB` les listes des classes de substitution sont fusionnées en calculant l'union des deux classes.

Paramètres différents Dans ce cas, les relaxations sont réalisées indépendamment et les mesures de similarité sont agrégées avec l'opérateur d'agrégation min. Considérons Q avec les classe c' et c" et Q' une requête relaxée de Q basée sur la conjonction d'opérateurs suivante : $\texttt{GEN}(c', i)$ `AND` $\texttt{SIB}(c", [c_1,\ c_2, ...,\ c_n])$. Dans ce cas les opérateurs `GEN` et `SIB` s'exécutent normalement et les requêtes relaxées sont classées sur la base de la mesure $Sim(Q, Q') = min(sim(c', c_g), sim(c", c_i))$ où c_g est une généralisation de c'.

4.4 QaRS : Un outil de relaxation des requêtes SPARQL

Comme nous l'avons vu au chapitre 2, l'édition des requêtes est une phase importante dans le processus d'exécution d'une requête. Dans cette phase, les utilisateurs expriment leurs besoins dans le langage de requêtes et, pour les raisons que nous avons présentées au chapitre 2, les utilisateurs ont besoin d'être assistés afin d'avoir en retour des réponses satisfaisantes. Avec l'intégration des opérateurs de relaxation dans le langages de requêtes, ce besoin d'assistance reste et devient même plus important du fait de la complexité du langage et de l'exigence d'une certaine connaissance des données interrogées. Pour assister les utilisateurs dans la relaxation des requêtes, nous avons conçu et développé le système d'interrogation et de relaxation graphique de requêtes *Query and Relax (QaRS)* [237].

4.4.1 Fonctionnalités de QaRS

Le système QaRS possède trois principales fonctionnalités.

1. Assistance graphique des utilisateurs dans la formulation de requêtes SPARQL.
2. Assistance des utilisateurs dans la relaxation de requêtes SPARQL.
3. Explication de l'échec des requêtes pour celles qui échouent, et du processus de relaxation pour celles qui sont relaxées.

Dans cette section nous présentons les deux premières fonctionnalités. La troisième fonctionnalité est présentée dans le chapitre suivant traitant de l'échec des requêtes SPARQL.

4.4.1.1 Édition graphique des requêtes SPARQL

QaRS assiste les utilisateurs dans l'édition de leurs requêtes via une interface graphique intuitive. QaRS permet également la visualisation de l'ontologie et la navigation au sein de cette dernière. La figure 4.4

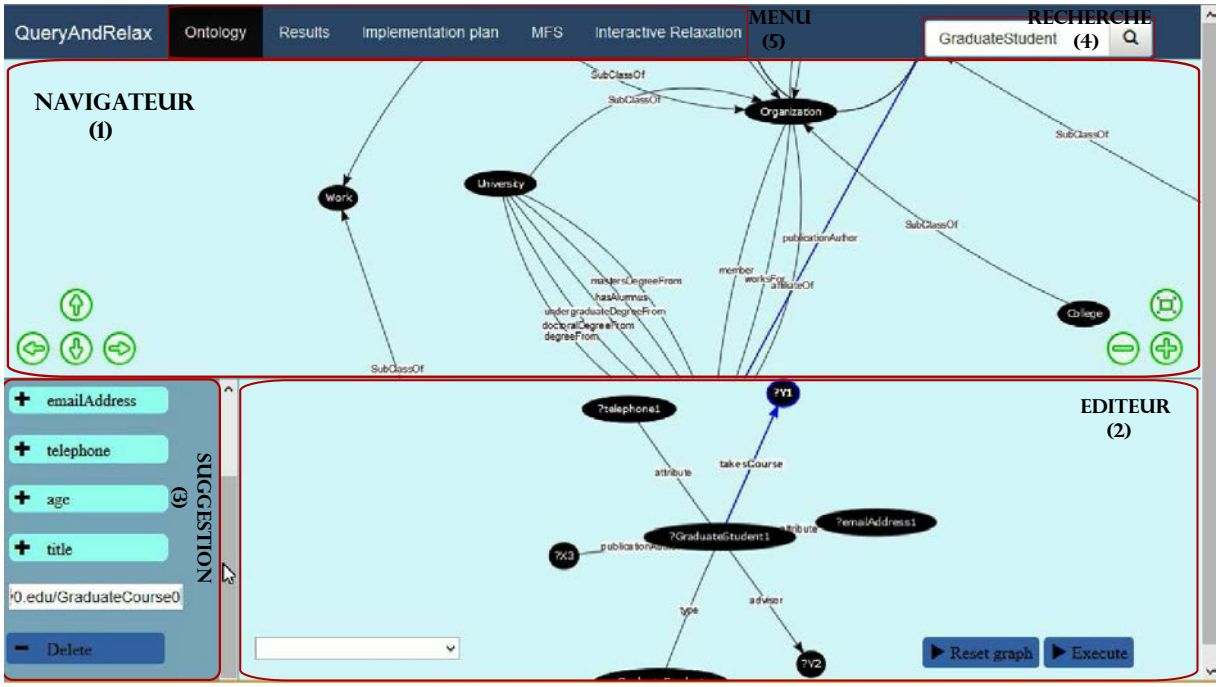


FIGURE 4.4 – Fenêtre de conception graphique de la requête

présente l'interface de QaRS pour l'édition graphique d'une requête. Nous pouvons observer trois panneaux, celui du dessus (1) est le panneau de navigation dans l'ontologie. Le panneau en bas à droite (2) est celui de l'édition graphique des requêtes. Et enfin, celui en bas à gauche (3) est le panneau de suggestion de concepts pour l'édition de la requête.

L'utilisateur peut ainsi naviguer au sein de l'ontologie à la recherche de concepts pour construire sa requête et éviter des erreurs liées aux URI des concepts. Vu que les ontologies sont généralement larges, car contenant un nombre important de concepts et de relations liés à un domaine, QaRS offre aux utilisateurs la possibilité de réaliser une recherche de concepts ou de relations à partir de leur label. Le label du concept recherché est saisi dans le champ de texte (4), en haut et à droite du panneau de navigation. De plus, grâce aux techniques d'auto-complétion déjà intégrées dans les outils de développement que nous avons utilisés, QaRS suggère aux utilisateurs des concepts et relations pertinents par rapport au nom en cours de saisie. La sélection par l'utilisateur d'un concept ou d'une relation crée un zoom sur l'élément sélectionné afin de permettre de distinguer les éléments et relations voisins.

L'éditeur graphique de requêtes, fortement inspirée des travaux de Smart et al. [95], possède trois principales opérations.

— Cliquer et copier un concept ou une relation de l'ontologie du panneau de navigation vers le panneau d'édition de requêtes. L'ajout de la classe c dans ce panneau crée un patron de triplet de la forme suivante $(?v_i\ rdf:type\ c)$ et celui d'une propriété p donne le patron $(?v_i\ p\ ?v_j)$.
— Mettre en relation les patrons de triplet définis précédemment en identifiant les variables partagées. Cette action est réalisée par la fusion de deux variables en une seule en cliquant sur une des

variables et en la déplaçant vers l'autre, ce qui indique que les deux variables sont identiques.

— Ajouter les opérateurs tels que `FILTER`, `OPTIONAL` et/ou `UNION` [23] en sélectionnant le composant de la requête sur lequel l'opérateur est appliqué, suivi d'un clic droit pour choisir l'opérateur utilisé. Les opérateurs sont graphiquement identifiés par *le libellé* contenu dans la forme géométrique.

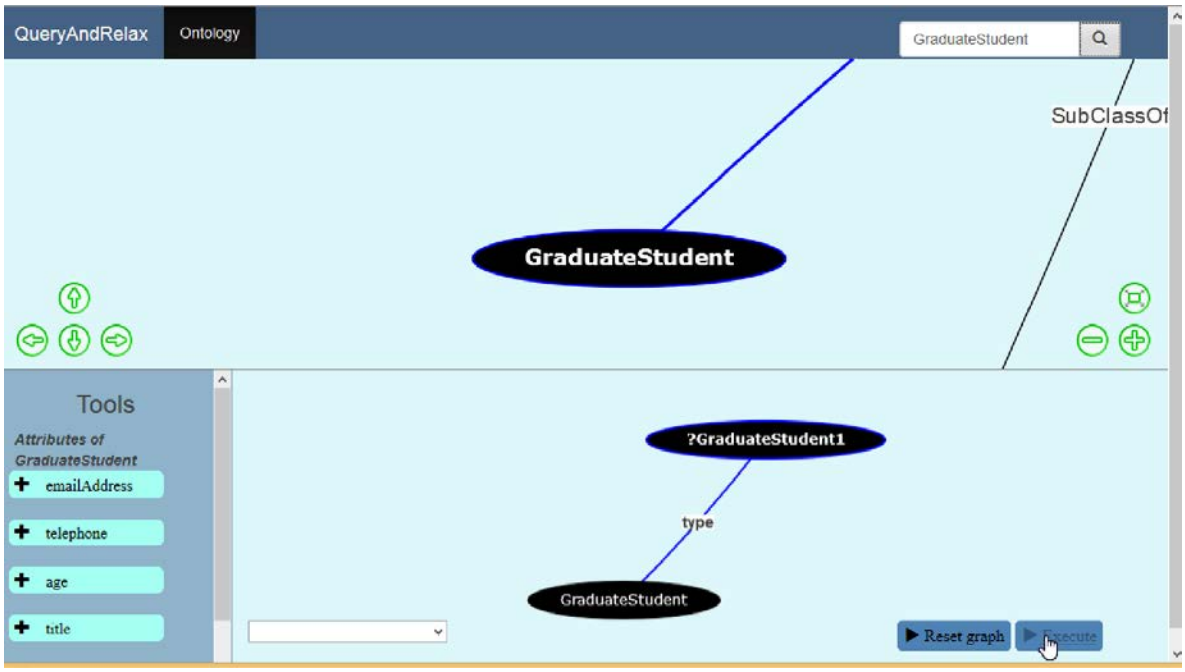


FIGURE 4.5 – Construction d'une requête SPARQL

La figure 4.5 montre la construction d'une requête dans le système QaRS. Dans les figures 4.4 et 4.5 nous observons que le panneau de suggestion (3) contient des éléments. Dans la figure 4.4, il s'agit de suggestions pour la suite de l'édition de la requête à partir de la variable $?Y1$. QaRS suggère ainsi l'ajout de patrons de triplet de la forme $(?Y1,\ P,\ ?Y3)$ où P peut-être l'une des quatre propriétés suggérées. QaRS suggère également à l'utilisateur de changer la variable $?Y1$ en instance $GraduateCourse0$. Cette suggestion s'appuie sur l'analyse de l'ontologie et de la partie de la requête déjà éditée. L'utilisateur pourra, s'il le souhaite, saisir un nom d'instance à la place de celle suggérée par QaRS qui vérifiera la cohérence de la nouvelle requête. Dans la figure 4.5, où le concept sélectionné est présent dans l'ontologie, le panneau de suggestion contient les propriétés de la classe sélectionnée.

4.4.1.2 Relaxation des requêtes dans QaRS

Parmi les fonctionnalités disponibles dans la barre de menu de QaRS (figure 4.4), nous avons : (i) *Ontology* pour la navigation et l'interrogation de l'ontologie et des données, (ii) *Results* pour l'affichage des réponses d'une requête relaxée ou non, (iii) *MFS* pour la recherche des causes d'échec d'une requête qui ne retourne aucune réponse, (iv) *interactive relaxation* pour relaxer une requête en utilisant les opérateurs de relaxation et (v) *implementation plan* pour présenter à l'utilisateur la séquence des requêtes relaxées exécutées afin d'obtenir des réponses alternatives, notamment dans le cas d'une relaxation automatique.

23. Dans cette version initiale de QaRS nous avons considéré uniquement ces trois opérateurs. Les autres seront intégrés avec l'évolution de QaRS.

Dans ce chapitre nous présentons la relaxation automatique et la relaxation avec des opérateurs de relaxation. La relaxation exploitant les MFS est présentée dans les chapitres suivants.

Notons tout d'abord que pendant l'édition de la requête, QaRS vérifie si cette dernière peut retourner des réponses ou non, autrement dit si elle est consistante. Cette vérification est réalisée en continu et diffère suivant les modifications effectuées sur la requête. Pendant l'opération de jointure entre deux variables, par exemple, la vérification est faite sur les ensembles (domaines, codomaines) auxquels appartient chaque variable, déduit en utilisant l'ontologie. Dans l'exemple 7, le patron de triplet t_3 implique que la variable $?X$ appartient au domaine de $teachnigAssistantOf$ et $?Y$ à son codomaine. Mais, d'après l'ontologie le domaine de la propriété $teachingAssistantOf$ est $GraduateStudent$, donc $?X$ doit être une instance de $GraduateStudent$ et ne peut donc pas être une instance de $AssistantProfessor$, comme l'exige le patron de triplet t_4. Le dernier patron de triplet t_4 introduit donc une inconsistance dans la requête de l'exemple 7.

Exemple 7.
`SELECT` $?X$ $?Y$ $?Z$
`WHERE` { $?Z$ $teacherOf$ $?Y$. (t_1)
$?Y$ *rdf:type* $UnderGraduateCourse$. (t_2)
$?X$ $teachingAssistantOf$ $?Y$. (t_3)
$?X$ *rdf:type* $AssistantProfessor$. (t_4)}

Exemple 8.
`SELECT` $?X$ $?Y$
`WHERE` {
$?X$ *rdf:type* $GraduateStudent$. (t_1)
$?X$ $teachingAssistantOf$ $?Y$. (t_2)
$?Y$ *rdf:type* $GraduateCourse$. (t_3)}

La requête de l'exemple 8 est également inconsistante. Cette requête reste consistante lorsque les deux premiers patrons de triplet sont ajoutés. Mais le dernier patron de triplet la rend inconsistante car le codomaine de $teachnigAssistantOf$ est $UnderGraduateCourse$. Donc $?Y$ ne peut-être une instance de $GraduateCourse$. QaRS signale l'existence des inconsistances dans la requête aux utilisateurs.
QaRS permet également d'utiliser les opérateurs de relaxation pour les requêtes qui ne retournent pas du tout ou pas assez de réponses. Les utilisateurs peuvent ainsi ajouter les opérateurs `GEN`, `SIB` et `PRED` avec les paramètres adéquats. Pour cela, l'utilisateur clique sur le paramètre à relaxer et sélectionne dans le panneau de suggestion l'opérateur de relaxation à utiliser. Les opérateurs de relaxation disponibles dans le panneau de suggestion sont ceux qui sont compatibles avec le paramètre sélectionné. La figure 4.6 montre l'édition d'une requête relaxée avec QaRS.

Sur la figure 4.6, nous voyons que les opérateurs `GEN` et `SIB` peuvent être utilisés pour relaxer la classe $UnderGraduateCourse$. L'utilisateur a choisi l'opérateur `SIB` sans spécifier d'autres paramètres, donc ce sont les classes sœurs de la classe $UnderGraduateCourse$ qui seront utilisées pour la substitution. Cette relaxation correspond à une *stratégie manuelle de relaxation* : ce sont les utilisateurs qui choisissent eux-mêmes les parties de la requête à relaxer et les paramètres de relaxation.

QaRS propose également une stratégie de relaxation automatique. Dans cette stratégie l'utilisateur précise uniquement le nombre minimal K de réponses attendues. Si la requête retourne moins de K réponses, le processus de relaxation automatique est lancé. Ce processus consiste à calculer les relaxations possibles d'une requête et à les ordonner en fonction de leur similarité par rapport à la requête initiale. Ces requêtes relaxées, générées avec les opérateurs de relaxation, sont exécutées de la plus similaire à la moins similaire. Cette stratégie est une implémentation du processus de relaxation proposé par Huang et al. [17] qui, en plus de la généralisation, utilise également les opérateurs `SIB` et `PRED`.

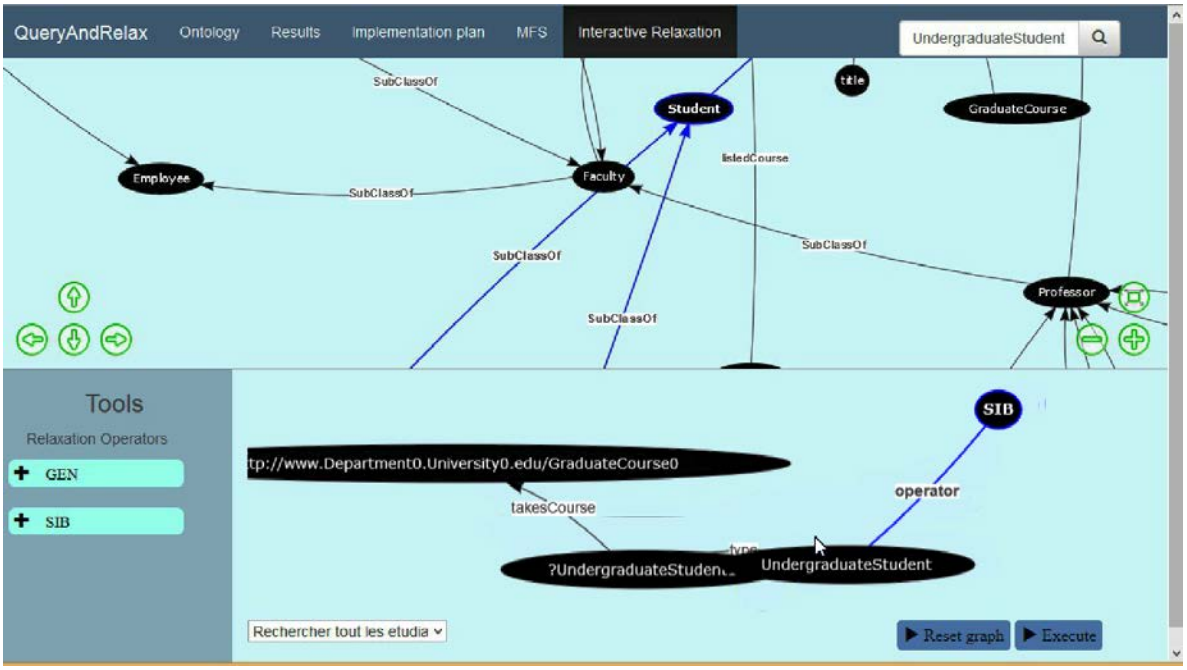


FIGURE 4.6 – Construction d'une requête relaxée

4.4.2 Architecture de QaRS

L'architecture du système QaRS est illustrée par la figure 4.7. Cette architecture peut être divisée en deux grandes parties : la partie responsable de l'édition et de l'analyse de la requête et la partie chargée de la relaxation des requêtes SPARQL dans QaRS. Cette partie constitue la contribution principale du framework que nous proposons par rapport à ceux existants sur la construction et l'exécution des requêtes SPARQL. Plus précisément, les différents modules du système QaRS sont :

L'éditeur graphique de requête : ce module est responsable de la gestion des composants graphiques pour la création des requêtes. Il assiste les utilisateurs dans la formulation des requêtes.

L'analyseur de requête SPARQL : ce module vérifie, pendant sa construction, si une requête est consistante. Sinon, il détecte les éléments responsables de l'inconsistance et alerte l'utilisateur.

L'extension de l'exécuteur de requête SPARQL : ce module une extension du module d'exécution standard des requêtes SPARQL. Il lance la relaxation si la requête échoue et si les utilisateurs ont spécifié la relaxation à réaliser dans ce cas.

L'interpréteur des opérateurs de relaxation : ce module interprète les trois opérateurs de relaxation que nous avons proposés. Il génère ainsi les requêtes relaxées à partir de la requête utilisateur.

Le module de relaxation automatique de requête : ce module implémente le processus de relaxation automatique en utilisant les opérateurs de relaxation.

Le module de recherche des causes d'échecs d'une requête SPARQL : ce module recherche les causes d'échecs d'une requête SPARQL qui échoue, présentées dans le chapitre suivant.

Le module de classement des réponses alternatives : ce module ordonne les réponses alternatives en fonction des mesures de similarité vues précédemment.

La section suivante présente les évaluations réalisées afin d'analyser le comportement des différents opérateurs de relaxation que nous avons implémentés.

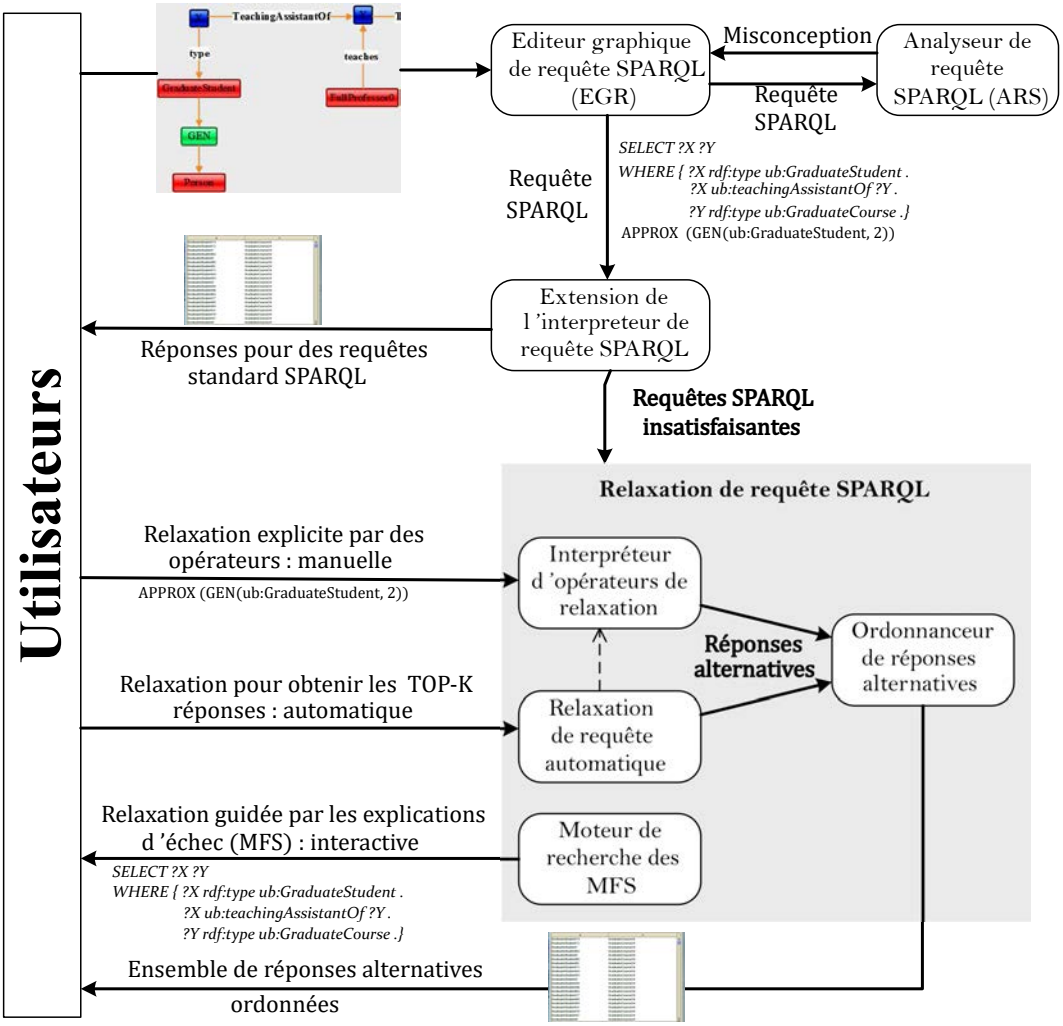


FIGURE 4.7 – Architecture de QaRS

4.5 Expérimentations : contexte et résultats

Le comportement et les performances des opérateurs de relaxation `GEN`, `SIB` et `PRED` ont été évalués à travers un ensemble d'expérimentations que nous avons menées. Dans cette dernière section nous présentons le contexte de ces expérimentations et l'analyse des résultats.

4.5.1 Contexte expérimental

Pour les expérimentations, nous avons implémenté les opérateurs de relaxation dans le BD-RDF OntoDB/OntoQL. Dans OntoDB [238], l'ontologie et les instances sont stockées en utilisant une table par classe. OntoDB utilise ainsi la représentation horizontale que nous avons présentée dans le chapitre 1. Nous avons étendu les langages d'interrogation SPARQL et OntoQL [28] avec les opérateurs de relaxation `GEN`, `SIB` et `PRED`. Pour ces expérimentations, nous avons utilisé un jeu de données réelles de plus 500 000 logements (Hôtels, Motels, etc.) appelé *HotelBase*, dont le schéma de l'ontologie est illustré figure 4.8.
Les tableaux 4.2 et 4.3 donnent la repartition des instances entre les différentes classes de l'ontologie des

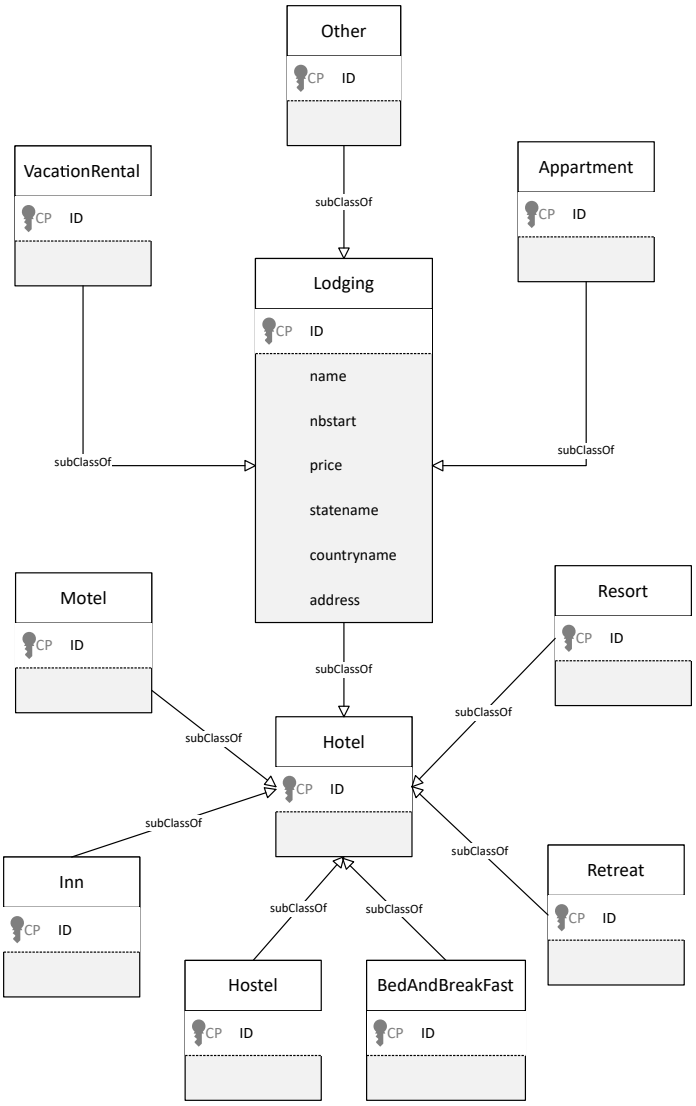


FIGURE 4.8 – Ontologie de logement

logements *HotelBase*. Nous avons relaxé 11 requêtes (Annexe A) sur ces données avec nos opérateurs de relaxation. Ces requêtes portent sur différentes propriétés et classes de l'ontologie afin de pouvoir décrire le comportement des opérateurs dans tous les cas possibles, y compris des propriétés non renseignées dans une classe mais renseignées dans une autre. Pour chaque requête, nous avons mesuré le temps d'exécution, noté $Cost\_Time$, de la requête initiale et de ses relaxations. Nous avons également mesuré la variation de la similarité entre la requête initiale et les requêtes relaxées pendant le processus de relaxation, sachant que les paramètres sont incrémentés durant le processus. C'est le cas, par exemple, de la généralisation $\texttt{GEN}(\texttt{c}, \texttt{i})$ dont le niveau de relaxation i est incrémenté. Nous avons également évalué, pour chaque relaxation, la taille des données parmi lesquelles sont recherchées les réponses alternatives pour la requête. Cette taille correspond au nombre d'instances de la classe sur laquelle porte la requête. Ces expériences ont été réalisées sur un PC de bureau Core-i7 avec 8GB de RAM et le système d'exploitation Windows 7 Pro 64Bits. Nous précisons également que pour ces expérimentations, nous avons utilisé les paramètres par défaut pour les opérateurs SIB et $PRED$. Pour l'opérateur PRED, nous avons $\epsilon = 0.1$ et l'intervalle de validité par défaut V identifié dans la section 4.2.1. Ensuite, pour l'opérateur

Classe	Lodging	Hotel	Vacation Rental	Other	Apartment
Nombre d'instances	473165	426357	2863	17630	26315

TABLE 4.2 – Nombre d'instances par classe

Classe	Motel	Inn	B&B	Hostel	Retreat	Resort
Nombre d'instances	8853	6560	18452	7186	539	14410

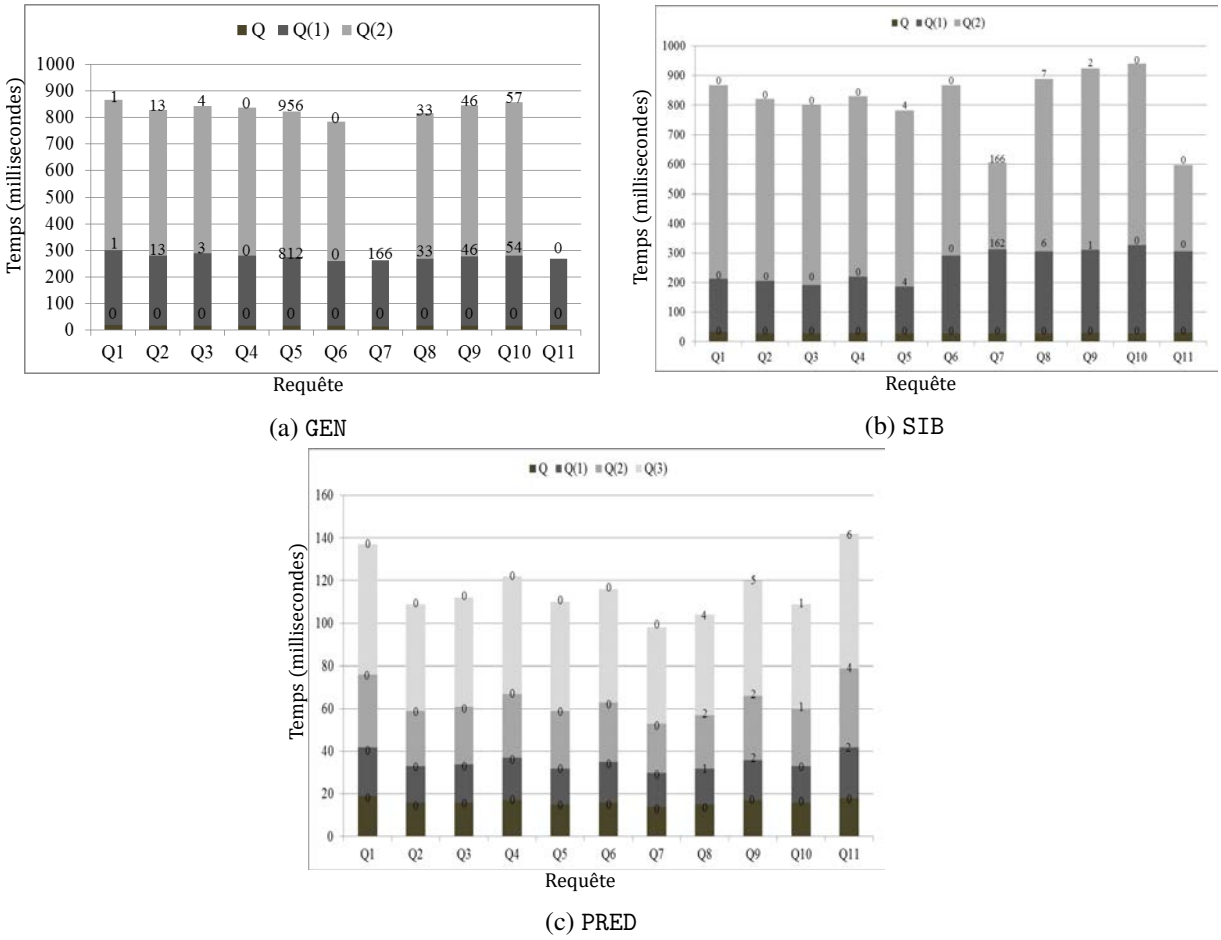
TABLE 4.3 – Nombre d'instances par classe (suite)

SIB, ce sont les concepts frères du concept à relaxer qui seront utilisés comme paramètres. Pour l'opérateur GEN, le paramètre utilisé est 2, ce qui signifie que nous généralisons une classe donnée avec ses super-classes de deux niveaux. Ce choix a été fait par rapport à l'ontologie utilisée où la profondeur maximale d'une classe de la hiérarchie est deux.

4.5.2 Résultats expérimentaux

Temps d'exécution : les figures 4.9a, 4.9b et 4.9c présentent le temps d'exécution aux différents étapes de relaxation des trois opérateurs de relaxation. Ces étapes de relaxation, notées $Q(i)$, signifient que la requête provenant de la $i^{ième}$ relaxation a été exécutée. Ainsi, le temps affiché pour Q_i est le temps total de la requête initiale et des relaxations qui ont suivies, chaque relaxation ayant une couleur pour distinguer sa durée d'exécution dans la durée totale du processus. Les chiffres sur les diagrammes 4.9a, 4.9b et 4.9c donne le nombre de réponses alternatives cumulées entre les étapes de relaxation $Q(i)$.
L'analyse du diagramme 4.9c montre que pour l'opérateur PRED, à chaque étape de la relaxation de la requête, le temps d'exécution de Q' est quasiment égale à celui de Q. En effet, nous pouvons constater que pour cet opérateur, la relation $Cost\_Time(Q_i) \approx Cost\_Time(Q) \times (i+1)\ ;\ i > 0$ est toujours vérifiée. Ce qui signifie que les requêtes exécutées à toutes les étapes de relaxation ont globalement un coût identique. Ce résultat devient plus compréhensible lorsque nous analysons le nombre d'instances sur lesquelles le prédicat relaxé est appliqué pour filtrer des réponses. En effet, cette taille reste la même entre les différentes étapes de la relaxation, figure 4.10c, car elle ne dépend pas du prédicat mais de la condition sur la classe d'où proviennent les instances. De ce fait, le coût lié au filtrage reste le même. A la différence de l'opérateur PRED, le nombre d'instances considérées, pour retourner les réponses alternatives, avec les opérateurs GEN et SIB change fortement à travers les étapes de la relaxation comme le montre les diagrammes 4.10a et 4.10b. La conséquence de cette variation, croissante, est un coût de plus en plus élevé des requêtes relaxées exécutées, illustré par les diagrammes 4.9a et 4.9b.

Sur les diagrammes 4.9a et 4.9b, nous pouvons observer qu'une requête relaxée $Q(i)$ peut coûter beaucoup plus de temps que la requête $Q(i-1)$. Ce rapport est lié à la variation du nombre d'instances filtrées qui est elle-même liée au nombre de classes et au nombre d'instances de chaque classe. Comme les tableaux 4.2 et 4.3 le montrent, une classe telle que $Motel$ possède 8853 instances alors que sa super-classe $Hotel$ en a environ 50 fois plus. La relaxation $Motel \longrightarrow Hotel$ entraine donc une requête relaxée beaucoup plus coûteuse que la requête initiale. Plus la répartition des données est hétérogène, plus la différence de taille entre les classes est grande et donc plus les coûts en temps de la relaxation avec les opérateurs GEN et SIB sont potentiellement élevés. Par contre, le coût en temps de l'opérateur PRED ne dépend pas de cette répartition des instances entre les classes, mais de la distribution des valeurs possibles d'une propriétés parmi les instances d'une classe. Remarquons que sur les requêtes Q_7 et Q_{11} qui portaient sur des classes de profondeur 1, la deuxième étape de relaxation n'a pas pu être réalisée.



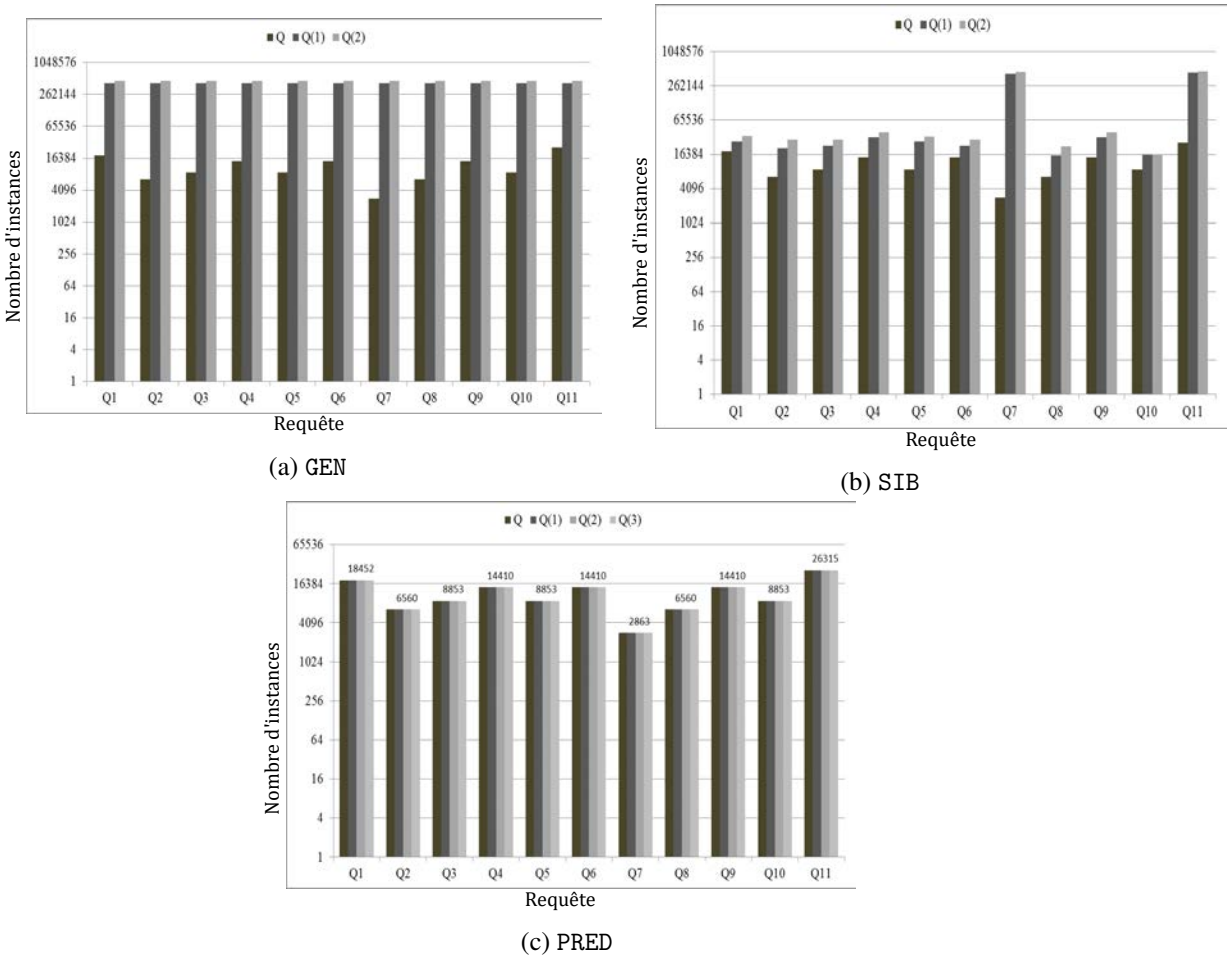
(a) `GEN` (b) `SIB`

(c) `PRED`

FIGURE 4.9 – Temps d'exécution de la relaxation et nombre des réponses alternatives retournées

Les réponses alternatives : si le temps d'exécution est un indicateur pertinent, la relaxation a, avant tout, pour objectif de retourner le TOP-K des réponses alternatives aux utilisateurs. En observant le nombre de réponses alternatives cumulées pour chaque requête et chaque opérateur sur les figures 4.9a, 4.9b et 4.9c, nous constatons que l'opérateur `GEN` est celui qui retourne le plus grand nombre de réponses alternatives entre les étapes de relaxation. Prenons le cas de la requête 5, la requête initiale ne retourne aucune réponse, la première généralisation retourne 812 réponses et la deuxième 956 réponses, soit 144 nouvelles. A l'autre extrémité, c'est-à-dire l'opérateur retournant le moins de réponses alternatives, nous retrouvons `PRED` qui, même après trois étapes de relaxation, peut ne retourner aucune réponse alternative comme le montre la figure 4.9c. Pour cet opérateur, seules les requêtes 8, 9, 10 et 11 permettent d'obtenir une augmentation mineure du nombre de réponses alternatives entre les étapes de relaxation. Le nombre de réponses alternatives est un des critères d'évaluation du bénéfice de la relaxation. Pour évaluer la pertinence des ces résultats, nous analysons à présent la variation de similarité comme estimation de leur qualité.

La variation de la similarité : la pertinence des réponses alternatives est évaluée avec les mesures de similarité présentées précédemment. Pour ce faire nous avons évalué, à chaque étape de relaxation, la similarité entre les requêtes relaxées et la requête initiale pour chaque opérateur. Ces relevés sont synthétisés dans les diagrammes 4.11a, 4.11b et 4.11c. Ces diagrammes montrent clairement que pour les opérateurs `GEN` et `SIB` la similarité entre la requête initiale et la première relaxation chute brutalement,



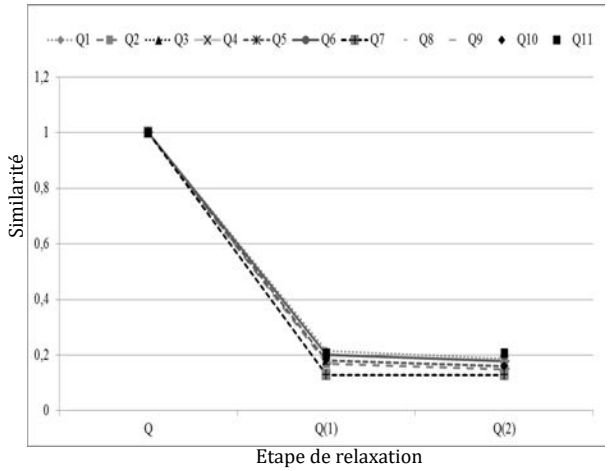
(a) GEN

(b) SIB

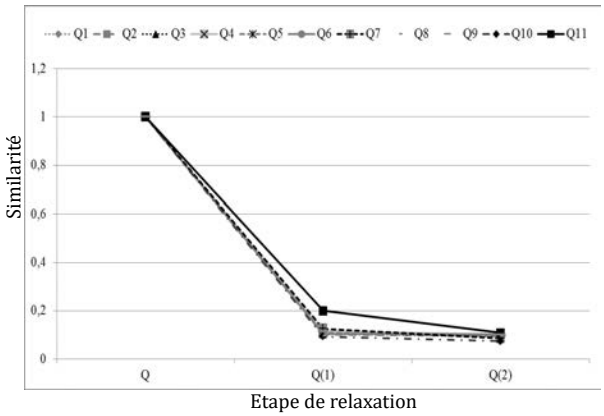
(c) PRED

FIGURE 4.10 – Nombre d'instances filtrées pour chaque opérateur de relaxation

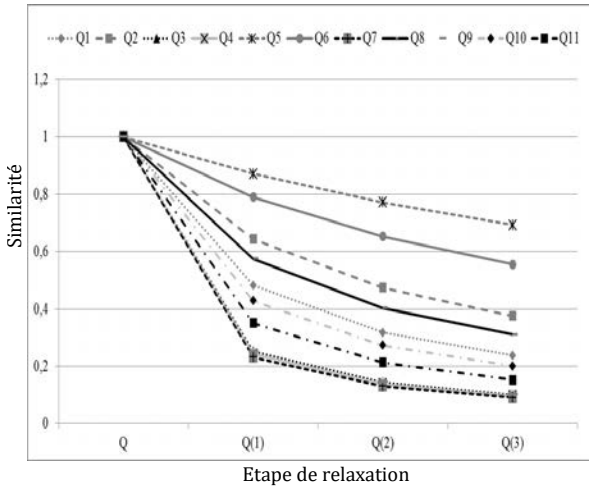
de 1 à 0.2, et décroit ensuite moins brutalement. Ce n'est pas le cas pour l'opérateur PRED pour qui la similarité connait une décroissance plus progressive. Cette différence de comportement trouve son explication dans au moins l'une des causes suivante. Premièrement, la formule de calcul de la similarité qui diffère entre l'opérateur PRED et les opérateurs GEN et SIB. Deuxièmement, la nature des paramètres qui entrent en jeu dans le calcul de ces mesures de similarité. Pour la similarité de l'opérateur PRED, seule la valeur est prise en compte dans le calcul, de même que dans la relaxation. Or, pour les opérateurs GEN et SIB, le schéma des classes et leurs contenus sont pris en considération. Vu la répartition des instances montrée dans les tableaux 4.2 et 4.3, un changement de classe entraine souvent une baisse significative de la similarité de la requête relaxée par rapport à la requête initiale. Et troisièmement, *le pas de relaxation* pour l'opérateur PRED est fixé à une valeur faible de manière à réaliser un relaxation avec tes itérations fines. Par contre, pour les opérateurs GEN et SIB la contrainte structurelle vient s'ajouter au moment de fixer les paramètres des opérateurs et de relaxer la requête. Il en ressort donc que la grande différence dans la similarité, et même dans l'approche de relaxation, de l'opérateur PRED d'un côté et les opérateurs GEN et SIB de l'autre est la considération de l'organisation hiérarchique des classes et de leurs statistiques de répartition d'instances. Ainsi, plus les données sont hétérogènes, c'est-à-dire qu'elles présentent une diversité de classes et d'instances de classe, plus la similarité entre les étapes de relaxation des opérateurs GEN et SIB décroît rapidement.



(a) GEN



(b) SIB



(c) PRED

FIGURE 4.11 – Variation de la similarité durant le relaxation pour chaque opérateur de relaxation

Conclusion

Avec le volume de données à traiter de plus en plus important et leur forte hétérogénéité, l'intégration d'opérateurs de relaxation dans les langages d'interrogation devient de plus en plus nécessaire. Le langage SPARQL avec sa clause `OPTIONAL` intégrait déjà la relaxation par suppression en permettant aux utilisateurs de rendre des contraintes optionnelle. Cependant avec la multitude de techniques et processus de relaxation que nous avons présentés dans le chapitre précédent, il devenait important de trouver un ensemble d'opérateurs qui permettrait d'implémenter à partir du langage de requêtes un processus de relaxation plus riche. C'est dans cette optique que s'inscrivent les travaux présentés dans ce chapitre.

Ainsi, nous avons identifié cinq critères à remplir pour étendre un langage de requêtes afin d'y intégrer la relaxation. Le premier critère est la définition d'un ensemble d'opérateurs de relaxation élémentaires. Le deuxième critère est la définition d'opérateurs de composition de ces opérateurs élémentaires afin de les combiner et d'implémenter les processus de relaxation existants. La définition de fonctions d'ordonnancement des réponses alternatives est le troisième critère que nous avons identifié. Le quatrième critère est la capacité de contrôler le processus de relaxation et le cinquième critère est l'intégration de

cet ensemble d'outils dans un langage de requêtes afin de les rendre accessible.

En nous appuyant sur l'arbre des techniques de relaxation de la figure 3.13, nous avons donc proposé les opérateurs `PRED`, `GEN` et `SIB` comme opérateurs élémentaires de relaxation, avec l'opérateur de combinaison `AND`. L'opérateur `PRED` permet d'étendre un prédicat numérique afin qu'un plus grand nombre de valeurs soient considérées comme valides. Les opérateurs `GEN` et `SIB` exploitent le schéma des données afin de substituer une classe par une de ses super-classes, `GEN`, ou par une autre classe sémantiquement proche de cette classe (`SIB`). Afin d'évaluer la proximité sémantique entre les requêtes, nous avons associé à ces opérateurs des mesures de similarité et des techniques pour agréger ces mesures de similarité. Nous avons également défini l'intégration de ces opérateurs dans SPARQL. Ce travail est une preuve de concept sur l'intégration d'opérateurs de relaxation dans les langages de requêtes des BD-RDF afin d'accroître la coopérativité des langages d'interrogation des données RDF. Des extensions peuvent être apportées à ce travail notamment au niveau de la relaxation des littéraux non numériques qui ne sont pas pris en compte actuellement par l'opérateur `PRED`.

Côté mise en œuvre, nous avons étendu les langages d'interrogation SPARQL et OntoQL avec les opérateurs de relaxation `PRED`, `GEN` et `SIB`. Nous avons implémenté le système QaRS (Query and Relax System) pour assister, à l'aide d'interfaces graphiques intuitives, les utilisateurs dans la construction et la relaxation de leurs requêtes SPARQL. Nous avons également mené des expérimentations sur les opérateurs de relaxation pour les caractériser. Pour ces expérimentations, nous avons utilisé la BD-RDF OntoDB et le jeu de données réelles *HotelBase*. Nous avons évalué le temps d'exécution d'un opérateur de relaxation, l'ensemble des réponses alternatives et la variation de la similarité entre les requêtes relaxées et la requête initiale. Il ressort de cette évaluation que bien que l'opérateur `PRED` soit celui qui possède un coût en temps minimal et une variation de similarité moins importante entre les étapes de relaxation, il est également celui qui retourne le moins de réponses alternatives en comparaison avec les opérateurs `GEN` et `SIB`.

Comme nous l'avons observé dans les expérimentations, la relaxation proposée par nos opérateurs ne garantit pas toujours de retrouver des réponses alternatives. C'est la cas notamment pour la requête 7 qui, pour une relaxation avec les opérateurs `GEN` et `SIB`, retourne des réponses alternatives, tandis que la relaxation avec l'opérateur `PRED` ne retourne aucune réponse. Ainsi, pour une relaxation efficace basée sur ces opérateurs il est nécessaire de déterminer dans un premier temps la partie de la requête à relaxer. Suivant la nature des patrons des triplets ainsi déterminés, il est possible de déduire l'opérateur de relaxation à utiliser. Pour le cas particulier des requêtes qui retournent un ensemble vide de réponse, il s'agit de chercher les causes de l'échec de ces requêtes afin de les réparer et donc de relaxer ces requêtes. Dans le chapitre suivant, nous proposons deux approches permettant de trouver les causes d'échecs des requêtes conjonctives SPARQL qui retournent un ensemble vide de réponses.

Chapitre 5

Recherche des MFS et XSS dans les requêtes SPARQL

Sommaire

Résumé : Les opérateurs de relaxation, présentés dans le chapitre précédent, s'appliquent sur une partie de la requête à relaxer. Identifier la partie de la requête sur laquelle appliquer les opérateurs de relaxation est important afin d'obtenir des réponses alternatives. Dans le cas des requêtes SPARQL qui retournent un ensemble vide de réponse, nous proposons de relaxer uniquement les parties responsables de l'échec de ces requêtes : l*es causes d'échec* encore appelées *Minimal Failing Subqueries* (MFS). La notion duale au MFS est celle de *MaXimal Succeeding Subquery* (XSS) qui correspond aux sous-requêtes qui réussissent avec un nombre maximal de patrons de triplets. Dans ce chapitre, nous présentons deux approches de recherche des MFS et des XSS. Les MFS sont les parties de la requête à relaxer afin d'obtenir des

réponses alternatives, tandis que les XSS sont des sous-requêtes relaxées obtenues par suppression de patrons de triplet.

Introduction

Les opérateurs `GEN`, `SIB` et `PRED`, présentés dans le chapitre précédent, permettent de relaxer une requête SPARQL. Ces opérateurs s'appliquent sur une ou plusieurs parties de la requête à relaxer. Cette requête est définie en termes de patrons de triplets. Les résultats du processus de relaxation dépendent fortement de l'identification des parties de la requête sur lesquelles les opérateurs sont appliqués. Par exemple, lors des expérimentations présentées dans le chapitre précédent, nous avons fait le constat suivant : la même requête relaxée sur deux parties différentes retournait, dans un cas, des réponses alternatives et, dans l'autre, aucune réponse. Il apparaît clairement que le choix des patrons de triplet sur lesquels appliquer des opérateurs de relaxation a un impact considérable sur le déroulement du processus de relaxation. D'où la nécessité de choisir les parties de la requête dont la relaxation garantit l'obtention de réponses alternatives. Pour cela, il faut déjà être capable d'identifier les causes d'échec la requête.

Pour le cas des requêtes qui échouent, ce chapitre propose ainsi l'utilisation des causes de cet échec pour identifier les parties de la requête sur lesquelles les opérateurs de relaxation doivent être appliqués. L'intérêt de cette approche est que la réparation des causes d'échec, dans une requête qui retourne un ensemble vide de réponse, est nécessaire pour retrouver des réponses alternatives. Rappelons que parmi les catégories de *requêtes insatisfaisantes* présentées aux chapitres 2 nous avons :

1. les requêtes qui retournent un ensemble vide de réponse, ce qui peut frustrer les utilisateurs ;
2. les requêtes qui retournent un ensemble de réponses attendues mais incomplètes ;
3. les requêtes qui retournent un ensemble de réponses inattendues pour les utilisateurs.

Ce chapitre se focalise sur les requêtes insatisfaisantes de la première catégorie c'est-à-dire celles qui retournent un ensemble vide de réponse.

Pour retrouver les causes d'échec des requêtes SPARQL qui échouent, nous développons dans ce chapitre deux approches qui permettent la recherche simultanée des causes d'échecs, encore appelées MFS pour *Minimal Failing Subqueries*, et les plus grandes sous-requêtes qui réussissent, encore appelées XSS pour *(maXimal Succeeding Subqueries)*. Nous utilisons les MFS, d'une part, pour l'explication de l'échec des requêtes et, d'autre part, pour réaliser une relaxation précise et incrémentale via l'utilisation des MFS comme paramètres d'opérateurs de relaxation. Les XSS sont des relaxations de la requête contenant un nombre maximal de contraintes de la requête utilisateur, les autres contraintes ayant été supprimées ou rendues optionnelles. L'utilisateur peut directement les exécuter pour trouver des réponses alternatives. Comme nous l'avons présenté au chapitre 1, le langage SPARQL propose plusieurs éléments pour la définitions de requêtes. Les requêtes SPARQL considérées dans ce chapitre sont celles décrites uniquement avec un ensemble de patrons de triplet : les patrons de graphe. Nous précisons que ce type de requête bien que limité est sont fortement utilisées pour interroger les ontologies [11].

L'objectif de ce chapitre est donc d'apporter des solutions au problème de la recherche des MFS et XSS dans une requête SPARQL, constituée uniquement de patrons de graphe, qui échoue. Dans la section 5.1, nous rappelons quelques notions du langage SPARQL et présentons les définitions des MFS et XSS adaptées au contexte des bases de données RDF. Ensuite, nous proposons, dans un premier temps, l'approche *Lattice-Based Approach* (LBA) qui recherche les MFS et les XSS d'une requête SPARQL qui échoue. Cette approche est présentée dans la section 5.2 et s'appuie sur les travaux de Godfrey [200] pour les bases de données relationnelles. Nous développons également cinq techniques d'optimisation de l'approche LBA, qui sont présentées dans la section 5.3. Dans un deuxième temps, nous proposons, dans la section 5.4, la deuxième approche de recherche des MFS et XSS : l'approche Matrix-Based Approach (*MBA*) basée sur les travaux de Jannach [206] dans le contexte des systèmes de recommandation. Et enfin,

dans la section 5.5, nous décrivons l'implémentation de LBA dans le système QaRS et l'évaluation de nos contributions par rapport aux algorithmes existants.

5.1 Préliminaires : rappels et définitions

Dans cette section, nous rappelons les définitions formelles des concepts RDF et SPARQL, déjà présentées au chapitre 1, dont nous aurons besoin dans la suite. Comme dans le chapitre 1, nous réutilisons pour cela les notations et définitions de Perez et al. [77].

Un *triplet* RDF est de la forme (sujet, prédicat, objet) $\in (U \cup B) \times U \times (U \cup B \cup L)$ où U est l'ensemble d'URI, B est l'ensemble des nœuds blancs et L est l'ensemble des littéraux. L'union $U \cup B \cup L$, notée $\mathbb{C}$, est l'ensemble des données concrètes RDF. Un *patron de triplet* RDF t est un triplet de la forme (sujet, prédicat, objet) $\in (U \cup B \cup V) \times (U \cup V) \times (U \cup B \cup V \cup L)$, où V est un ensemble de variables disjointes des ensembles U, B et L. Nous notons $var(t)$ l'ensemble des variables du patron de triplet t.

Nous considérons les *requêtes* SPARQL définies comme des conjonctions de patrons de triplet, c'est-à-dire des *patrons de graphes* : $Q = t_1 \wedge \cdots \wedge t_n$. Le nombre de patrons de triplet de la requête Q est noté $|Q|$. Un *mapping* μ de V vers $\mathbb{C}$ est une fonction partielle $\mu : V \rightarrow \mathbb{C}$. Pour un patron de triplet t, nous notons $\mu(t)$ le triplet obtenu en remplaçant les variables de t selon μ. Le domaine de μ, $dom(\mu)$, est le sous-ensemble de V sur lequel μ est défini. Deux mappings μ_1 et μ_2 sont *compatibles* si pour tout $x \in dom(\mu_1) \cap dom(\mu_2)$, la condition $\mu_1(x) = \mu_2(x)$ est satisfaite ; autrement dit, lorsque $\mu_1 \cup \mu_2$ est aussi un mapping. Soit Ω_1 et Ω_2 deux ensembles de mappings, la *jointure* de Ω_1 et Ω_2 est définie par : $\Omega_1 \bowtie \Omega_2 = \{\ \mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2\ sont\ compatibles\ \}$.

Soit $\mathbb{D}$ une base de données RDF et t un patron de triplet, l'évaluation de t sur $\mathbb{D}$, notée $[[t]]_{\mathbb{D}}$, est définie par : $[[t]]_{\mathbb{D}} = \{\mu \mid dom(\mu) = var(t) \wedge \mu(t) \in \mathbb{D}\}$. Soit Q une requête, l'évaluation de Q sur $\mathbb{D}$ est définie par : $[[Q]]_{\mathbb{D}} = [[t_1]]_D \bowtie \cdots \bowtie [[t_n]]_D$. Ainsi, une requête *échoue* si et seulement si elle retourne *un ensemble vide de réponse*, c'est-à-dire $[[Q]]_{\mathbb{D}} = \emptyset$. Par opposition, une requête *réussit* si et seulement si elle retourne *au moins une réponse*, c'est-à-dire $[[Q]]_{\mathbb{D}} \neq \emptyset$. L'évaluation d'une requête peut être faite suivant les différents modes de raisonnement (*entailment regime*) définis dans la spécification SPARQL. Dans ce chapitre, nous utilisons le mode de raisonnement simple (*simple entailment regime*), où la sémantique du schéma RDFS ou OWL n'est pas utilisée. Cependant, nos algorithmes peuvent être utilisés avec tous les modes de raisonnement.

Étant donnée une requête $Q = t_1 \wedge \cdots \wedge t_n$, une requête $Q' = t_i \wedge \cdots \wedge t_j$ est une *sous-requête* de Q, $Q' \subseteq Q$, si et seulement si $\{t_i, \cdots, t_j\} \subseteq \{t_1, \cdots, t_n\}$. Si $\{t_i, \cdots, t_j\} \subset \{t_1, \cdots, t_n\}$, Q' est une *sous-requête directe* de Q ($Q' \subset Q$). Si une sous-requête Q' de Q échoue, alors la requête Q échoue.

Définition 5

Une Minimal Failing Subquery (MFS) Q^* d'une requête Q est une sous-requête de Q qui échoue et dont toutes les sous-requêtes réussissent, c'est-à-dire :

$$\begin{cases} [[Q^*]]_{\mathbb{D}} = \emptyset \\ \nexists\ Q' \subset Q^*\ telle\ que\ [[Q']]_{\mathbb{D}} = \emptyset \end{cases} \tag{5.1}$$

Nous notons $mfs(Q)$ l'ensemble de toutes les MFS d'une requête Q.

Chaque MFS est une cause d'échec de la requête Q.

Définition 6

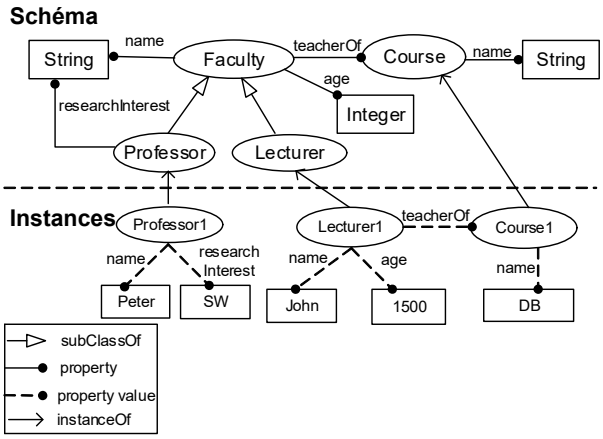
Une Maximal Succeeding Subquery (XSS) $Q^{\#}$ d'une requête Q est une sous-requête de Q de taille maximale qui réussit et dont toutes les sous-requêtes de Q qui contiennent $Q^{\#}$ échouent c'est-à-dire :

$$\begin{cases} [[Q^{\#}]]_{\mathbb{D}} \neq \emptyset \\ \nexists\, Q' \subseteq Q \; telle \; que \; Q^{\#} \subset Q' \wedge [[Q']]_{\mathbb{D}} \neq \emptyset \end{cases} \tag{5.2}$$

Nous notons $xss(Q)$ l'ensemble de toutes les XSS d'une requête Q.

Chaque XSS est une sous-requête maximale (en termes de nombre de patrons de triplet) qui réussit.

Nous développons, dans la suite, deux approches pour déterminer les ensembles $mfs(Q)$ et $xss(Q)$ dans le contexte des données RDF. Pour illustrer nos algorithmes dans ce chapitre, nous utiliserons les données représentées ci-dessous à la figure 5.1, inspirées du banc d'essai LUBM.



(a) Graphe de données

TRIPLES			
Subject	**Predicate**	**Object**	
Faculty	rdf:type	rdfs:Class	Schéma
Professor	rdf:type	rdfs:Class	
Professor	rdfs:subClassOf	Faculty	
name	rdf:type	rdf:Property	
name	rdfs:range	xsd:String	
…	…	…	
Professor1	rdf:type	Professor	Instances
Professor1	name	Peter	
Professor1	researchInterest	SW	
Lecturer1	rdf:type	Lecturer	
…	…	…	

(b) Table de triplets

FIGURE 5.1 – Schéma et données utilisés pour les exemples

5.2 Approche LBA pour la recherche des MFS et XSS

L'approche naïve de recherche de toutes les MFS et XSS d'une requête SPARQL Q consiste, dans une première étape, à énumérer l'ensemble des sous-requêtes de Q. Dans la seconde étape, les propriétés 5.1 et 5.2 sont vérifiées pour chaque sous-requête afin de déterminer si elle est MFS, XSS ou ni l'un ni l'autre. Cette approche explore le treillis formé par les sous-requêtes de Q, qui correspond à l'ensemble des parties de Q dont la taille est 2^n, où n est le nombre de patrons de triplet de la requête.

Contrairement à l'approche naïve, notre objectif dans l'approche LBA est d'explorer un minimum d'éléments de ce treillis en s'appuyant sur les propriétés des MFS et des XSS. L'approche LBA détermine les ensembles $mfs(Q)$ et $xss(Q)$ en utilisant les trois étapes suivantes.

1. Rechercher une MFS de Q et l'ajouter dans $mfs(Q)$.
2. Calculer les *XSS potentielles*, c'est-à-dire les sous-requêtes maximales qui n'incluent pas la MFS trouvée précédemment.
3. Exécuter les XSS potentielles ; si elles réussissent, ce sont des XSS et elles sont ajoutées dans $xss(Q)$; sinon, ce processus est appliqué récursivement sur les XSS potentielles qui échouent.

5.2.1 Recherche d'une MFS

Cette étape est réalisée avec l'algorithme *a_mel_fast* proposé par Gordfrey [200]. Cet algorithme est basé sur la proposition suivante.

Proposition 1

Soit $Q = t_1 \wedge ... \wedge t_n$ une requête qui échoue et $Q_i = Q - t_i$ une sous-requête directe de Q.
Si $[[Q]]_{\mathbb{D}} = \emptyset$ et $[[Q_i]]_{\mathbb{D}} \neq \emptyset$ alors chaque MFS de Q contient t_i .

Preuve [200] : Supposons qu'il existe une MFS Q^* de Q telle que $t_i \notin Q^*$ alors $Q^* \in Q - t_i$ et donc $Q^* \in Q_i$. Étant donné que toute requête contenant une MFS échoue, nous déduisons l'implication suivante : $Q^* \in Q_i \Rightarrow [[Q_i]]_{\mathbb{D}} = \emptyset$ ce qui est contraire à l'hypothèse de départ qui est $[[Q_i]]_{\mathbb{D}} \neq \emptyset$. Par conséquent, si $[[Q_i]]_{\mathbb{D}} \neq \emptyset$ alors il n'existe pas de MFS Q^* de Q telle que $t_i \notin Q^*$.

Nous utilisons cette propriété dans l'algorithme 1 pour trouver une MFS en n étapes. La complexité de cet algorithme est ainsi en $\mathcal{O}(n)$.

Algorithm 1: Rechercher une MFS d'une requête SPARQL qui échoue

FindAnMFS(Q, $\mathbb{D}$)
inputs : Une requête $Q = t_1 \wedge ... \wedge t_n$ qui échoue ; une BD-RDF $\mathbb{D}$
output: Une MFS de Q notée Q^*
1 $Q^* \leftarrow \emptyset$;
2 $Q' \leftarrow Q$;
3 **foreach** *patron de triplets* $t_i \in Q$ **do**
4 $Q' \leftarrow Q' - t_i$;
5 **if** $[[Q' \wedge Q^*]]_{\mathbb{D}} \neq \emptyset$ **then**
6 $Q^* \leftarrow Q^* \wedge t_i$;
7 **return** Q^*;

Pour trouver une MFS de Q, l'algorithme 1 procède comme suit. Il traite itérativement chaque patron de triplet. Pour un patron de triplet t_i traité, l'algorithme commence par l'enlever de Q. La sous-requête directe résultante est notée Q' (ligne 4). Si $[[Q']]_{\mathbb{D}}$ n'est pas vide, t_i appartient à la MFS recherchée (grâce à la proposition 1) et donc il est ajouté au résultat Q^* (ligne 6). Sinon, Q' possède au moins une MFS qui ne contient pas t_i, c'est cette MFS qui est recherchée par la suite. L'algorithme 1 poursuit en itérant sur un autre patron de triplet de Q pour trouver les autres patrons de triplet de la MFS Q^* en vérifiant l'échec ou la réussite de $Q' \wedge Q^*$, car Q^* contient les patrons de triplet appartenant à toutes les MFS. Ce processus s'arrête lorsque tous les patrons de triplet de Q ont été traités.

Pour illustrer cet algorithme, considérons la requête $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$ qui recherche *"les enseignants de bases de données dont les recherches portent sur le Web sémantique"* dans les données de la figure 5.1. Nous considérons également que chaque instance ne peut avoir qu'un seul type (mono-instanciation).

```
SELECT  ?p ?age
 WHERE { ?p age ?age. (t1)
         ?p rdf:type Lecturer. (t2)
         ?p researchInterest "SW". (t3)
         ?p teacherOf "DB" (t4)}
```

Cette requête échoue car le domaine de $researchInterest$ dans le graphe 5.1a est *Professor* et non *Lecturer* (un type par instance). De plus, si l'utilisateur opérait le changement de *Lecturer* en *Professor*, cette requête échouerait encore car aucun triplet de la figure 5.1b ne définit une valeur pour la propriété age. Ainsi, nous pouvons identifier deux MFS dans cette requête sur les données de la figure 5.1 :

1. t_1, les âges des personnes ne sont pas connus dans la BD-RDF ;
2. $t_2 \wedge t_3$, un *lecturer* n'a pas de *researchInterest* dans la BD-RDF.

Cette requête a également deux XSS suivantes :

1. $t_2 \wedge t_4$, il existe des *lecturers* qui enseignent un cours de *DB*;
2. $t_3 \wedge t_4$, il existe des personnes qui enseignent *DB* et sont intéressées par la recherche en *SW*.

La figure 5.2 montre une exécution possible de l'algorithme 1 pour trouver la MFS $t_2 \wedge t_3$ de la requête Q de notre exemple. L'exécution commence par le retrait du patron de triplet t_1 de Q, aboutissant à la sous-requête $Q' = t_2 \wedge t_3 \wedge t_4$. Comme cette sous-requête retourne un ensemble vide, l'algorithme recherche une des MFS encore présente dans $t_2 \wedge t_3 \wedge t_4$. L'algorithme 1 retire ensuite le patron de triplet t_2 de cette sous-requête. La requête résultante $t_3 \wedge t_4$ réussit, donc t_2 fait partie des MFS de Q' et donc de Q^*. Pour t_3, il teste la requête $t_2 \wedge t_4$, $(Q' \wedge Q^*)$ qui réussit et donc t_3 est ajouté à Q^*. Pour t_4, la sous-requête $t_2 \wedge t_3$ échoue et donc t_4 n'appartient pas à Q^*. Comme tous les patrons de triplet de Q ont été traités, l'algorithme s'arrête et retourne la MFS $Q^* = t_2 \wedge t_3$.

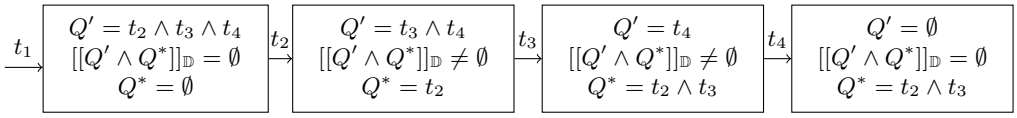


FIGURE 5.2 – Exemple d'exécution de l'algorithme 1 qui trouve la MFS $t_2 \wedge t_3$ de Q

La figure 5.3 présente une autre exécution possible de l'algorithme 1 qui cette fois retrouve la MFS t_1. Dans cette exécution, l'algorithme 1 commence par traiter le patron de triplet t_3 en opérant son retrait de Q. Comme la requête résultante $Q' = t_1 \wedge t_2 \wedge t_4$ ne retourne aucune réponse, t_3 n'appartient pas à la MFS Q^* que nous recherchons. A sa prochaine itération, l'algorithme 1 traite t_1. La requête $t_2 \wedge t_4$, résultante de la suppression de t_1, réussit, et donc t_1 appartient à Q^*. Pour les deux dernières itérations, sur t_2 et t_4 respectivement, la requête $Q' \wedge Q^*$ échoue donc aucun autre patron de triplet n'est inséré dans Q^*. Par conséquent, à la fin de l'exécution de l'algorithme 1, la MFS $Q^* = t_1$ est retournée.

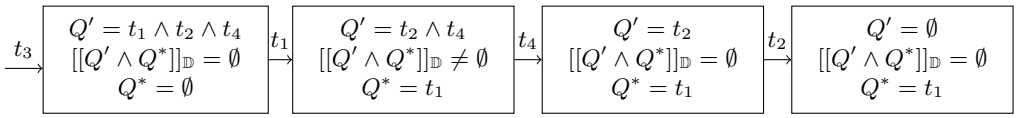


FIGURE 5.3 – Exemple d'exécution de l'algorithme 1 qui trouve la MFS t_1 de Q

5.2.2 Calcul des XSS potentielles

D'après la définition même des MFS, toutes les requêtes qui incluent la MFS Q^* trouvée dans l'étape précédente retournent un ensemble vide de réponse. Par conséquent, les sous-requêtes de Q qui incluent Q^* ne peuvent être ni MFS, ni XSS de Q. Ainsi nous pouvons réduire l'espace de recherche en supprimant ces requêtes. Parmi les sous-requêtes ne contenant pas Q^*, nous considérons celles qui ne sont pas incluses dans d'autres sous-requêtes : les sous-requêtes de tailles maximales. L'exploration du treillis des sous-requêtes continue ainsi avec uniquement ces plus larges sous-requêtes de Q qui ne contiennent pas

Q^*. Si ces sous-requêtes réussissent, elles sont des XSS de Q. Pour cette raison, nous les appelons *XSS potentielles* et notons $pxss(Q, Q^*)$ cet ensemble de requêtes. Cet ensemble peut être calculé comme suit.

$$pxss(Q, Q^*) = \begin{cases} \emptyset, & \text{si } |Q| = 1. \\ \{Q - t_i \mid t_i \in Q^*\}, & \text{sinon.} \end{cases} \tag{5.3}$$

En effet, pour chaque patron de triplet t_i de Q^*, une sous-requête de la forme $Q_m \leftarrow Q - t_i$ n'inclut pas Q^* et a une taille maximale puisque $|Q_m| = |Q| - 1$.

La figure 5.4 illustre $pxss(Q, Q^*)$ de notre exemple $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$, pour la MFS $Q^* = t_2 \wedge t_3$ sur le treillis des sous-requêtes de Q. Les sous-requêtes maximales de Q qui n'incluent pas $t_2 \wedge t_3$ sont $t_1 \wedge t_2 \wedge t_4$ et $t_1 \wedge t_3 \wedge t_4$, requêtes en fond rouge vif. De même, la sous-requête maximale qui ne contient pas t_1 est $t_2 \wedge t_3 \wedge t_4$. Nous présentons aussi sur cette figure les MFS et XSS de Q.

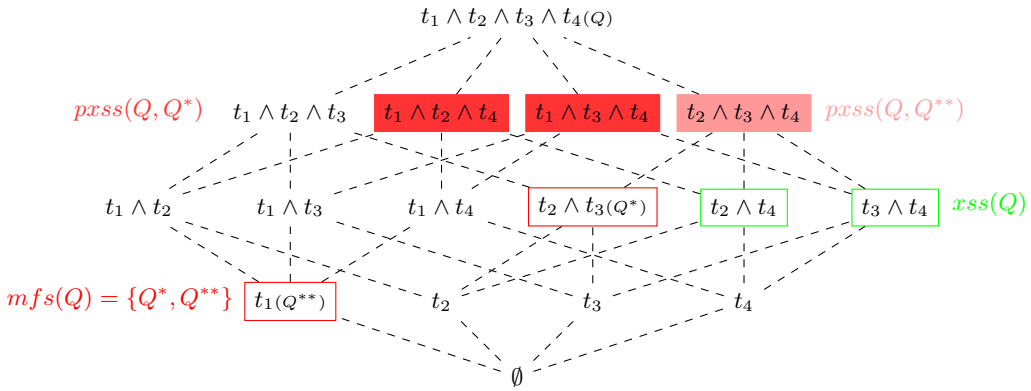


FIGURE 5.4 – Le treillis des sous-requêtes de Q avec ses MFS et XSS

5.2.3 Recherche de toutes les MFS et XSS

Comme la proposition suivante le montre, si une requête SPARQL Q possède une seule MFS (ce qui inclus le cas où Q est sa propre MFS), les XSS de cette requête sont les XSS potentielles.

Proposition 2

Si Q possède une seule MFS Q^*, alors $xss(Q) = pxss(Q, Q^*)$.

Preuve : Par définition, les requêtes contenues dans $pxss(Q, Q^*)$ sont maximales. Supposons qu'il existe une requête $Q' \in pxss(Q, Q^*)$ telle que $[[Q']]_{\mathbb{D}} = \emptyset$. D'après notre hypothèse de départ, Q ne possède qu'une seule MFS donc cette MFS doit être incluse dans Q', $Q^* \subseteq Q'$. Ce résultat est contradictoire avec la définition de $pxss(Q, Q^*)$ qui veut que tous ses éléments ne contiennent pas Q^*. Par conséquent, il n'existe pas de requête $Q' \in pxss(Q, Q^*)$ telle que $[[Q]]_{\mathbb{D}} = \emptyset$ et donc toutes les requêtes de $pxss(Q, Q^*)$ réussissent, d'où la conclusion $xss(Q) = pxss(Q, Q^*)$.

Considérons maintenant le cas général. Afin de retrouver toutes les MFS et XSS d'une requête Q, l'approche LBA procède selon l'algorithme 2. Pour chaque requête $Q' \in pxss(Q, Q^*)$ nous avons deux cas possibles :

Si $[[Q']]_{\mathbb{D}} \neq \emptyset$ **:** alors Q' est une XSS de Q, car Q' est maximale, par construction des XSS potentielles, et n'échoue pas, donc $xss(Q) \leftarrow xss(Q) \cup Q'$ (algorithme 2, ligne 6).

Sinon, c'est-à-dire si $[[Q']]_{\mathbb{D}} = \emptyset$ **:** Q' possède au moins une MFS, qui est également MFS de Q et différente de Q^*, par construction de $pxss(Q, Q^*)$. Cette nouvelle MFS est retrouvée avec l'algorithme *FindAnMFS* (algorithme 2, ligne 8) et ce processus est appliqué itérativement sur chaque requête de $pxss(Q, Q^*)$ qui échoue.

Cependant, comme les différentes requêtes de $pxss(Q, Q^*)$ peuvent contenir la même MFS, ce processus pourrait identifier une MFS plusieurs fois et ainsi être inefficace. L'algorithme 2 permet de remédier à ce problème en calculant incrémentalement les XSS potentielles qui ne contiennent pas de MFS précédemment identifiées (lignes 9-11). En effet, lorsqu'une seconde MFS Q^{**} est identifiée, cet algorithme itère sur les XSS potentielles trouvées précédemment ($pxss$) qui contiennent Q^{**}. Pour éviter de trouver à nouveau cette MFS, chaque requête Q'' ainsi parcourue est remplacée par ses sous-requêtes les plus larges qui ne contiennent pas Q^{**} (c'est-à-dire, $pxss(Q'', Q^{**})$) et ne sont pas des sous-requêtes d'une requête $pxss$ ou d'une requête incluse dans $xss(Q)$ (sinon, elles ne sont pas les plus larges XSS potentielles de Q, vu qu'il en existe une qui les contient et réussit).

Algorithm 2: Rechercher toutes les MFS et XSS d'une requête Q

LBA(Q, $\mathbb{D}$)

inputs : Une requête $Q = t_1 \wedge ... \wedge t_n$ qui échoue ; une BD-RDF $\mathbb{D}$

outputs: Les MFS et XSS de Q

1 $Q^* \leftarrow FindAnMFS(Q, \mathbb{D})$;
2 $pxss \leftarrow pxss(Q, Q^*)$;
3 $mfs(Q) \leftarrow \{Q^*\}$;
4 $xss(Q) \leftarrow \emptyset$;
5 **while** $pxss \neq \emptyset$ **do**
6 $Q' \leftarrow pxss.element()$; // choisit un élement de $pxss$
7 **if** $[[Q']]_{\mathbb{D}} \neq \emptyset$ **then** // Q' est une XSS
8 $xss(Q) \leftarrow xss(Q) \cup \{Q'\}$;
9 $pxss \leftarrow pxss - \{Q'\}$;
10 **else** // Q' contient une MFS
11 $Q^{**} \leftarrow FindAnMFS(Q', \mathbb{D})$;
12 $mfs(Q) \leftarrow mfs(Q) \cup \{Q^{**}\}$;
13 **foreach** $Q'' \in pxss$ *telle que* $Q^{**} \subseteq Q''$ **do** // évite de trouver à nouveau Q^{**}
14 $pxss \leftarrow pxss - \{Q''\}$;
15 $pxss \leftarrow pxss \cup \{Q_j \in pxss(Q'', Q^{**}) \mid \nexists Q_k \in pxss \cup xss(Q) : Q_j \subseteq Q_k\}$;
16 **return** $\{mfs(Q), xss(Q)\}$;

La figure 5.5 montre deux exemples d'exécution de l'algorithme 2 pour calculer les MFS et XSS de notre requête exemple : $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$. Dans la figure 5.5 (a), la MFS $Q^* = t_2 \wedge t_3$ est trouvée et les XSS potentielles correspondantes calculées ($pxss(Q, Q^*) = \{t_1 \wedge t_3 \wedge t_4,\ t_1 \wedge t_2 \wedge t_4\}$). L'algorithme exécute ensuite la requête $t_1 \wedge t_3 \wedge t_4$. Comme un ensemble vide de réponse est obtenu, l'algorithme 1 est appliqué sur cette requête pour trouver la seconde MFS $Q^{**} = t_1$. Les deux XSS potentielles contiennent cette MFS et ainsi elles sont remplacées par leurs plus larges sous-requêtes qui ne contiennent pas Q^{**}, c'est-à-dire, $t_3 \wedge t_4$ et $t_2 \wedge t_4$. En exécutant ces deux requêtes, l'algorithme trouve qu'elles sont effectivement des XSS. La liste $pxss$ étant vide, l'algorithme s'arrête et retourne ces deux XSS et les MFS trouvées précédemment (voir figure 5.4).

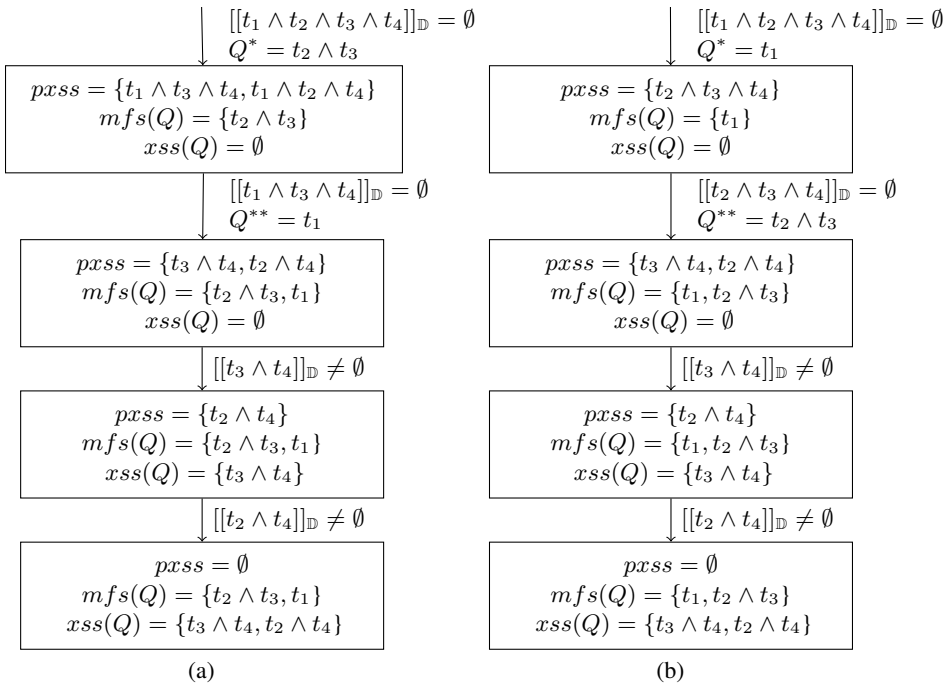


FIGURE 5.5 – Un exemple d'exécution de l'algorithme 2 pour trouver les MFS et XSS de Q

la figure 5.5 (b) présente une exécution alternative de l'algorithme 2 dans laquelle la première MFS retrouvée est t_1. La XSS potentielle correspondante dans ce cas est $t_2 \wedge t_3 \wedge t_4$. Cette requête retourne un ensemble vide de réponse, nous lançons la recherche de la deuxième MFS dans cette sous-requête et nous trouvons $t_2 \wedge t_3$. La $pxss$ $t_2 \wedge t_3 \wedge t_4$ est ensuite remplacée par ses XSS potentielles ne contenant pas $t_2 \wedge t_3$ qui sont $t_2 \wedge t_4$ et $t_3 \wedge t_4$. A partir de ce niveau, l'exécution continue de la même manière que dans l'exemple présenté à la figure 5.5 (a).

Proposition 3

L'algorithme LBA est correcte et complet, c'est-à-dire il retourne exactement toutes les MFS et XSS d'une requête Q qui échoue.

Preuve : Cette proposition est démontrée par récurrence sur la taille de Q. Si $|Q| = 1$, l'algorithme 2 retourne Q comme MFS [24] et un ensemble vide de XSS. Supposons maintenant que LBA retourne des ensembles correctes de MFS et XSS pour les requêtes de taille n et considérons la requête Q de taille $|Q| = n+1$. LBA recherche une MFS Q^* de Q en utilisant l'algorithme 1 et calcule les XSS potentielles correspondantes, $pxss(Q, Q^*)$. Le calcul des $pxss$ garantit que les XSS et les autres MFS de Q sous dans les treillis de sous-requête ayant comme racine une sous-requête Q' contenue dans $pxss(Q, Q^*)$, qui sont les plus grandes sous-requêtes de Q qui ne contiennent pas Q^*. LBA commence l'exploration de chaque treillis par l'exécution de sa racine Q'. Si Q' réussit, alors c'est une XSS de Q et toutes ses sous-requêtes réussissent et ne peuvent donc pas contenir de MFS. Sinon, Q' échoue, alors Q' contient une ou plusieurs MFS qui sont également MFS de Q. À partir de la définition des XSS potentielles nous savons que $Q' \in pxss(Q, Q^*)$ et donc $|Q'| = |Q| - 1 = n$. De plus, d'après notre hypothèse de récurrence, LBA retourne les ensembles correctes de MFS et XSS pour des requêtes de taille n, par conséquent LBA calcule correctement $mfs(Q')$ et $xss(Q')$. Comme toute MFS de Q' est MFS de Q, LBA retournera

24. Afin que toutes les sous-requêtes d'une MFS réussissent, nous considérons $[[\emptyset]]_{\mathbb{D}} \neq \emptyset$.

l'ensemble des MFS de Q qui est l'union des MFS des sous-requêtes Q', $Q' \in pxss(Q, Q^*)$, dans lequel on ajoute la première MFS Q^*. Contrairement aux MFS, toutes les XSS de Q' ne sont pas nécessairement XSS de Q. Nous devons ainsi montrer que seules les XSS de Q contenues dans $xss(Q')$ seront retournées par LBA. Considérons Q_p une XSS de Q', c'est-à-dire que $Q_p \in xss(Q')$. Si Q_p n'est pas une XSS de Q alors il existe une requête dans $pxss(Q, Q^*)$ qui contient une XSS de Q plus grande que Q_p. LBA n'insère donc pas Q_p dans la liste des $pxss$ et par conséquent Q_p ne sera pas retournée comme XSS de Q. Maintenant, si Q_p est une XSS de Q, nous avons deux cas. Dans le premier cas, Q_p n'est incluse dans aucune requête parmi les $pxss$ de Q, LBA l'ajoute comme une nouvelle $pxss$ et, par conséquent, elle sera retournée comme XSS. Dans le second cas, Q_p est incluse dans une requête Q_x contenue dans l'ensemble des $pxss$ de Q. LBA n'ajoute pas Q_p dans l'ensemble des $pxss$ de Q mais LBA va la rechercher parmi les XSS de Q_x au moment de la recherche des XSS de ce dernier qui se passe correctement d'après l'hypothèse de récurrence. Ce processus peut se répéter plusieurs fois jusqu'à ce que Q_p ne soit plus incluse dans aucune autre $pxss$. Ce qui arrivera inévitablement, dans le pire des cas lorsque l'ensemble des $pxss$ sera vide, LBA retournera alors l'ensemble correct des XSS de Q, $xss(Q)$.

Nous démontrons également la proposition suivante qui définit la complexité de LBA dans le pire cas :

Proposition 4

L'algorithme LBA a une complexité de $\mathcal{O}(\sqrt{n} * 2^n)$ dans le pire des cas, où n est la taille de la requête qui échoue et sous l'hypothèse que chaque exécution d'une requête coûte une unité de temps.

Preuve : Dans le pire des cas, il y a un nombre exponentiel de MFS. C'est le cas si toutes les sous-requêtes de taille $n/2$ sont des MFS, avec n la taille de la requête (en supposant que n est pair, sans perte de généralité). En effet, dans ce cas, le nombre de MFS est égal à : $\binom{n}{n/2}$ (choix de $n/2$ élément parmi n). Les XSS sont nécessairement les sous-requêtes directes de ces MFS. Par conséquent, le nombre de XSS est : $\binom{n}{(n/2)-1}$. Pour rechercher une MFS, LBA exécute au plus n requêtes. Par contre, pour retrouver une XSS, il suffit d'exécuter la XSS potentielle et de vérifier qu'elle retourne bien des réponses. Ainsi, la complexité de LBA dans le pire cas est : $\mathcal{O}(n * \binom{n}{n/2} + \binom{n}{(n/2)-1})$. En utilisant l'approximation de Stirling pour la fonction factorielle [240], nous avons : $\binom{n}{n/2} \approx 2^n * \sqrt{2/\pi n}$ et par conséquent $\mathcal{O}(n * \binom{n}{n/2} + \binom{n}{(n/2)-1}) = \mathcal{O}(\sqrt{n} * 2^n)$.

Puisque Godfrey [200] a montré que le problème de recherche de toutes les MFS d'une requête est NP-HARD, ce résultat n'est pas surprenant. Malgré cette complexité, nos expérimentations montrent qu'en pratique l'algorithme LBA a un temps de réponse acceptable vu que n, qui représente le nombre de patrons de triplets de la requête, est généralement faible (≤ 15 dans l'étude faite par Arias et al. [14]). Nous proposons également, dans la suite, différentes heuristiques pour optimiser l'algorithme LBA et améliorer ses performances.

5.3 Optimisation de l'approche LBA

Comme nous l'avons vu dans l'évaluation de la complexité de LBA, l'unité de temps est le temps d'exécution d'une requête. Donc, pour optimiser l'approche LBA, nous avons deux possibilités : (i) réduire le temps d'exécution d'une requête ou (ii) réduire le nombre de requêtes à exécuter et/ou à tester. L'exploration de ces deux possibilités nous a permis d'implémenter cinq techniques d'optimisation.

5.3.1 Réduction du nombre de requêtes

Pour réduire le nombre de requêtes exécutées dans la recherche des MFS et XSS avec l'approche LBA nous avons implémenté deux techniques d'optimisation : (i) l'utilisation du cache pour éviter l'exécution des requêtes déjà exécutées et celle des requêtes dont le résultat peut-être déduit grâce aux propriétés relatives aux MFS et XSS et (ii) l'identification des sous-requêtes inconsistantes, mais correctes en SPARQL, qui retournent obligatoirement un ensemble vide de réponse.

5.3.1.1 Utilisation d'un cache

Lorsque l'algorithme *FindAnMFS* recherche une deuxième MFS, il peut être amené à ré-exécuter des sous-requêtes déjà exécutées lors de la recherche de la première MFS, Q^*, de Q. Cette redondance se reproduit au fil des itérations de l'algorithme 2 pour trouver l'ensemble des MFS. Pendant ces itérations, l'algorithme 2 exécute l'algorithme 1 qui exécute plusieurs sous-requête, obtenues par suppression itérative des patrons de triplet, afin de retrouver une MFS. Pour éviter une redondance d'exécution de ces sous-requêtes, nous proposons de stocker les requêtes exécutées et leur résultat (une valeur booléenne qui indique si la requête réussit ou échoue) dans un cache de requête. Ainsi avant d'exécuter une requête nous vérifions d'abord si elle n'a pas déjà été exécutée, autrement dit si elle est dans le *cache de requête*, ce qui permet ainsi d'éviter des exécutions redondantes des requêtes. De plus, les propriétés entre sous-requêtes nous permettent de déduire si une requête échoue ou pas sans être obligé de l'exécuter, ce qui permet de réduire d'avantage le nombre de requêtes exécutées. Les propriétés utilisées sont les suivantes :

1. Si une requête Q réussit, alors toutes ses sous-requêtes réussissent.
2. Si une requête Q échoue alors toutes ses super-requêtes échouent.

Avec ces deux propriétés, nous ne vérifions pas seulement si la requête a déjà été exécutée, nous vérifions également si l'une des deux propriétés est vraie afin de déduire si la requête échoue ou pas sans avoir à l'exécuter. Ainsi, quand une requête Q doit être exécutée, nous vérifions d'abord si elle est une sous-requête d'une requête du cache qui réussit. Si c'est le cas nous déduisons qu'elle réussit (propriété 1). Sinon, si elle est super-requête d'une requête du cache qui échoue, elle échoue également (propriété 2). Nous notons que ce dernier cas ne peut pas arriver, car notre algorithme a été construit pour élaguer ces requêtes. En, effet, celui-ci est conçu pour supprimer de l'espace de recherche les requêtes qui contiennent une MFS et donc qui échouent. Nous verrons cependant une optimisation dans la section 5.3.2.1 qui conduit à exécuter des requêtes qui entrent dans ce cas.

Pour la représentation du cache, nous proposons les deux structures de données suivantes.

Une liste : le parcours est séquentiel et pour chaque requête de la liste nous vérifions si elle n'est pas identique à la requête à exécuter ou si cette dernière n'est pas une sous-requête ou une super-requête d'une requête de la liste. Si la requête n'est pas dans la liste, elle est insérée en queue.

Un arbre ternaire : Afin de faciliter la vérification des propriétés ci-dessus nous avons eu l'idée de représenter le cache sous forme d'un arbre ternaire. Chaque nœud Q a trois fils tels que :

$$\begin{cases} Q.child[1] = \{Q' \; tel \; que \; Q' \subset Q\} \\ Q.child[2] = \{Q' \; tel \; que \; Q \subset Q'\} \\ Q.child[3] = \{Q' \; tel \; que \; Q' \neq Q \wedge Q' \not\subset Q \wedge Q \not\subset Q'\}. \end{cases} \tag{5.4}$$

L'exécution d'une requête passe par le parcours de l'arbre. Si la nouvelle requête est l'un des deux premiers fils d'un nœud de l'arbre, alors le résultat est déduit à partir du nœud parent. L'insertion dans cet arbre ternaire est similaire à l'insertion dans un arbre binaire.

5.3.1.2 Détection des sous-requêtes inconsistantes

Pour cette optimisation, nous considérons que chaque instance de l'ontologie ne peut avoir qu'une seule classe comme type, c'est la mono-instanciation. Nous considérons comme requêtes inconsistantes toutes requêtes contenant une fausse supposition au niveau du schéma de l'ontologie. Cette inconsistance ontologique a pour conséquence l'échec de la requête sans avoir besoin de rechercher une quelconque instance vérifiant cette requête. Par exemple, la requête suivante :

$$\texttt{SELECT } ?p \texttt{ WHERE } \{?p\ rdf : type\ Lecturer.\ ?p\ researchInterest\ "SW"\}$$

échoue car le domaine du prédicat $researchInterest$ est $Professor$ et non $Lecturer$. Donc, implicitement, aucune n'instance ne pourra vérifier cette requête.

La recherche des sous-requêtes inconsistantes permet de trouver une sous-requête qui échoue et donc qui contient au moins une MFS. Cette MFS pourra être identifiée avec l'algorithme *FindAnMFS* qui sera appliqué à la requête inconsistante au lieu de la requête utilisateur et donc exécutera moins de requêtes.

5.3.2 Réduction du temps d'exécution des requêtes

Ces techniques d'optimisation utilisent des propriétés des requêtes pour réduire leur temps d'exécution. Nous proposons trois techniques : la détection des produits cartésiens, l'ordonnancement des XSS potentielles et l'ordonnancement des patrons de triplet de la requête initiale.

5.3.2.1 Détection des produits cartésiens

La détection des produits cartésiens est une technique d'optimisation qui permet de ne pas les exécuter car ils demandent généralement un temps important de calcul. En effet, la suppression des patrons de triplet, dans l'algorithme *FindAnMFS* ou dans le calcul de $pxss(Q, Q^*)$, peut conduire à des produits cartésiens (des patrons de triplet non liés par des variables communes). Un exemple est la recherche des MFS dans une requête SPARQL ayant la forme suivant $(?x_1, P_1, ?x_2)(?x_2, P_2, ?x_3)...(?x_{n-1}, P_{n-1}, ?x_n)$ (requête *en chaîne*). Dans ce cas, la suppression d'un patron de triplet autre que ceux des extrémités, conduit inévitablement à un produit cartésien. Le coût d'exécution des produits cartésiens pouvant être très important, nous proposons, pour l'éviter, d'exécuter séparément les sous-requêtes disjointes de ces produits cartésiens. Si une des sous-requêtes disjointes échoue alors le produit cartésien échoue sinon, il retournera au moins une réponse. Pour identifier les sous-requêtes disjointes dans un produit cartésien nous utilisons l'algorithme de recherche des composants connexes d'un graphe non-orienté. Nos expérimentations nous ont permis de constater que cette optimisation est particulièrement adaptée pour les requêtes SPARQL en chaîne.

5.3.2.2 Ordonnancement des XSS potentielles

Dans l'algorithme 2, après le calcul de $pxss(Q, Q^*)$, aucun ordre n'est précisé pour la recherche des MFS dans ces $pxss$. Or, comme nous l'avons vu, l'algorithme de recherche d'une MFS (*FindAnMFS, algorithme 1*) a une complexité de $\mathcal{O}(n)$, n étant la taille des requêtes. Ainsi, l'algorithme *FindAnMFS*, pour être efficace, doit être exécutée avec des requêtes ayant un nombre aussi petit que possible de patrons de triplet. Pour optimiser les exécutions de l'algorithme *FindAnMFS*, nous proposons donc d'ordonner, à chaque itération, les requêtes de la liste $pxss$ des plus petites aux plus grandes en terme de nombre de patrons de triplet. En utilisant cet ordre, les plus grandes requêtes de $pxss$ seront, très probablement, remplacées par de plus petites après la découverte, dans une autre $pxss$, d'une MFS qu'elles contiennent.

Avec cette optimisation, nous évitons le plus possible la recherche d'une MFS dans une $pxss$ de grande taille permettant ainsi aux exécutions de *FindAnMFS* d'être plus rapides.

5.3.2.3 Ordonnancement des patrons de triplet

L'algorithme *FindAnMFS* ne spécifie pas l'ordre dans lequel les patrons de triplet doivent être supprimés pour rechercher une MFS. Nous avons observé, de façon empirique, que la suppression des patrons de triplet dans l'ordre dans lequel ils sont traités dans la requête initiale, augmente la probabilité d'utiliser le cache de la BD-RDF. Notre hypothèse est que dans cet ordre, les sous-requêtes exécutées tendent à être syntaxiquement proches de la requête initiale, ce qui peut aider l'optimiseur de la BD-RDF à détecter les données partagées.

Les techniques présentées précédemment réduisent soit le nombre de requêtes exécutées, soit la durée d'exécution d'une requête. Dans la recherche d'une approche, qui réduirait le plus possible le nombre de requêtes exécutées, nous avons implémenté l'approche MBA de recherche des MFS et XSS d'une requête Q.

5.4 Approche MBA pour la recherche des MFS et XSS

Dans l'approche LBA, la taille de l'espace de recherche (c'est-à-dire, le treillis des sous-requêtes) augmente de façon exponentielle en fonction du nombre de patrons de triplet de la requête initiale. Pour éviter ce problème, nous nous sommes inspirés des travaux de Jannach [206] proposés dans le contexte des systèmes de recommandation. L'idée est de calculer une matrice, que nous appelons la *matrice de relaxation*, en utilisant n requêtes, où n est la taille de la requête initiale. Les MFS et XSS de la requête initiale peuvent alors être obtenues à partir de la matrice de relaxation sans exécuter de requêtes supplémentaires. Par rapport à l'approche de Jannach, la principale difficulté est de calculer la matrice de relaxation. Comme nous allons le voir dans ce qui suit, cela demande également de considérer les spécificités du langage SPARQL par rapport à SQL.

5.4.1 Matrice de relaxation d'une requête

Soit $Q = t_1 \wedge ... \wedge t_n$ une requête RDF, les *solutions potentielles* de Q sont les mappings (tels que définis en section 5.1) qui satisfont un patron de triplet de Q ou une combinaison de ces patrons de triplet. Cet ensemble de solutions potentielles de Q est noté $ps(Q, \mathbb{D})$ et est défini formellement par $ps(Q, \mathbb{D}) = \{\ \mu \mid \exists\{i, ..., j\} \subset \{1, ..., n\} : \mu \in [[t_i]]_{\mathbb{D}} \bowtie \cdots \bowtie [[t_j]]_{\mathbb{D}}\}$. La matrice relaxée d'une requête RDF est alors définie comme suit.

Définition 7

La matrice de relaxation M d'une requête $Q = t_1 \wedge ... \wedge t_n$ sur une BD-RDF $\mathbb{D}$ est une table à deux dimensions avec comme lignes les mappings $\mu \in ps(Q, \mathbb{D})$ et comme colonnes les patrons de triplet $t_i \in Q$. Pour un mapping $\mu \in ps(Q, \mathbb{D})$ et un patron de triplet $t_i \in Q$, $M[\mu][t_i] = 1 \Leftrightarrow \mu(t_i) \in \mathbb{D}$ sinon $M[\mu][t_i] = 0$.

La figure 5.6(c) présente la matrice de relaxation de la requête SPARQL Q donnée à la figure 5.6(b) lorsqu'elle est exécutée sur l'ensemble des données RDF de la figure 5.6(a). Chaque ligne de cette matrice est un mapping qui satisfait au moins un patron de triplet de Q. Par exemple, la première ligne correspond au mapping μ :$?p \rightarrow p_1$, qui satisfait le patron de triplet t_1 de Q. Un mapping μ a la valeur 1 pour la colonne t_i, si μ satisfait t_i. Par conséquent, la cellule de la matrice de relaxation située à la première

Triplets		
s	**p**	**o**
p_1	type	Professor
p_1	teacherOf	c_1
p_2	type	Professor
p_2	teacherOf	c_2
c_2	name	AI
p_3	type	Lecturer
p_3	teacherOf	c_3
c_3	name	DB
c_4	name	DB

(a) Triplets RDF

```
SELECT   ?p ?c
WHERE    { ?p rdf : type Professor. (t1)
           ?p teacherOf ?c. (t2)
           ?c name "DB". (t3)}
```

(b) La requête Q

?p	?c	t_1	t_2	t_3	
p_1	null	1	0	0	
p_2	null	1	0	0	
p_1	c_1	1	1	0	*
p_2	c_2	1	1	0	*
p_3	c_3	0	1	1	*
p_1	c_3	1	0	1	*
p_2	c_3	1	0	1	*
p_1	c_4	1	0	1	*
p_2	c_4	1	0	1	*
null	c_3	0	0	1	
null	c_4	0	0	1	

(c) Matrice de relaxation de Q

$xss(Q) = \{t_1 \wedge t_2, t_2 \wedge t_3, t_1 \wedge t_3\}$
$mfs(Q) = \{t_1 \wedge t_2 \wedge t_3\}$

(d) Les MFS et XSS de Q

FIGURE 5.6 – Illustration de l'approche MBA (Matrix-Based Approach)

ligne et à la colonne t_1 prend la valeur 1 car p_1 est de type $Professor$ dans le graphe RDF considéré à la figure 5.6(a).

5.4.2 Calcul de la matrice de relaxation d'une requête

Comme nous l'avons vu au chapitre 1 et rappelé dans les préliminaires de ce chapitre, l'évaluation d'une requête SPARQL consiste à rechercher les mappings qui satisfont tous les patrons de triplet en utilisant l'opérateur de jointure. La matrice de relaxation contient des mappings qui satisfont *au moins un patron de triplet*. Intuitivement, nous pouvons penser à l'utilisation de *l'opérateur de jointure externe* pour calculer les mappings qui satisfont tous les patrons de triplet à partir de celles qui ne satisfont qu'un seul patron de triplet. L'opérateur de jointure externe est défini de la façon suivante.

Définition 8

Soient Ω_1 et Ω_2 deux ensembles de mappings, la différence de Ω_1 et Ω_2 est définie par : $\Omega_1 - \Omega_2 = \{\mu_1 \in \Omega_1 \mid \forall \mu_2 \in \Omega_2, \mu_1 et \mu_2$ *ne sont pas compatibles* $\}$. *La jointure externe de Ω_1 et Ω_2 est définie par :* $\Omega_1 \mathbin{⟕} \Omega_2 = (\Omega_1 \bowtie \Omega_2) \cup (\Omega_1 - \Omega_2) \cup (\Omega_2 - \Omega_1)$.

Cependant, comme l'avait observé Galindo-Legaria [13] dans le contexte des bases de données relationnelles, l'opérateur de jointure externe élimine des tables qui lui servent d'opérandes les mappings qui satisfont la jointure interne. Par exemple, la figure 5.7 présente le résultat de l'opérateur de jointure externe entre des patrons de triplet de la requête Q de la figure 5.6(b). Comme nous pouvons l'observer à la figure 5.7, l'opération $[[t_1]]_{\mathbb{D}} \mathbin{⟕} [[t_2]]_{\mathbb{D}}$ ne conserve pas le mapping $\mu_1 : ?p \rightarrow p_1$ de l'ensemble des mappings du patron du triplet t_1, $[[t_1]]_{\mathbb{D}}$. En effet, ce mapping est compatible avec le mapping $\mu_2 : ?p \rightarrow p_1, ?c \rightarrow c_1$ et par conséquent ce n'est pas un élément de $[[t_1]]_{\mathbb{D}} - [[t_2]]_{\mathbb{D}}$. Conserver le mapping μ_1 est particulièrement important dans le contexte de SPARQL. En effet, contrairement à SQL, les jointures internes et externes *ne rejettent pas* les valeurs nulles dans SPARQL [77]. Un prédicat p rejette les valeurs nulles s'il est évalué à faux (ou non défini) lorsqu'une valeur nulle est utilisée pour p. Étant donné que SPARQL ne rejette pas les valeurs nulles, le mapping $\mu_1 : ?p \rightarrow p_1$ est compatible avec le mapping $?c \rightarrow c_3$ dans le figure 5.7. L'union de ces deux mappings est $?p \rightarrow p_1, ?c \rightarrow c_3$, qui

est un élément de $ps(Q, \mathbb{D})$ qui n'est pas obtenu avec l'opérateur de jointure externe. Afin de retrouver tous les éléments de $ps(Q, \mathbb{D})$ dans le contexte des requêtes SPARQL, nous définissons une extension de l'opérateur de jointure, appelée *jointure étendue*.

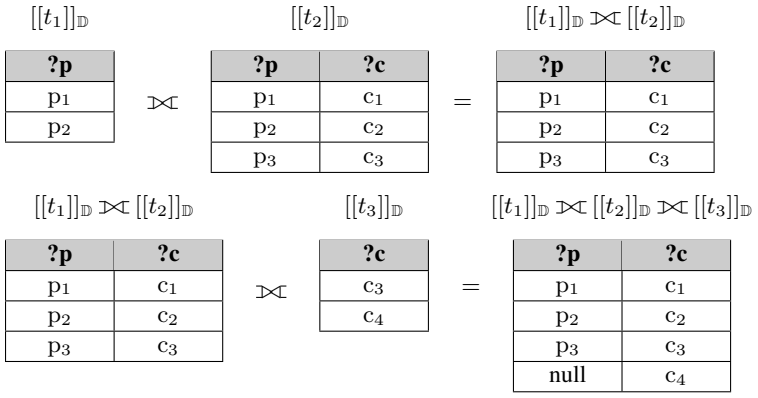


FIGURE 5.7 – Opérateur de jointure externe entre des ensembles de mappings

Définition 9

Considérons Ω_1 et Ω_2 deux ensembles de mappings, la jointure étendue de Ω_1 et Ω_2 est définie par : $\Omega_1 \leftouter^* \Omega_2 = \Omega_1 \cup (\Omega_1 \bowtie \Omega_2) \cup \Omega_2$ [25].

Proposition 5

L'ensemble des solutions potentielles de Q peut être calculé avec l'opérateur de jointure étendue : $ps(Q, \mathbb{D}) = [[t_1]]_{\mathbb{D}} \leftouter^* \cdots \leftouter^* [[t_n]]_{\mathbb{D}}$.

Preuve : En se basant sur la distributivité de l'opérateur de *jointure* sur l'*union* [77], nous montrons que :

$$\begin{aligned}[[t_1]]_{\mathbb{D}} \leftouter^* [[t_2]]_{\mathbb{D}} \leftouter^* [[t_3]]_{\mathbb{D}} = &[[t_1]]_{\mathbb{D}} \cup [[t_2]]_{\mathbb{D}} \cup [[t_3]]_{\mathbb{D}} \cup ([[t_1]]_{\mathbb{D}} \bowtie [[t_2]]_{\mathbb{D}}) \cup \\ &([[t_2]]_{\mathbb{D}} \bowtie [[t_3]]_{\mathbb{D}}) \cup ([[t_1]]_{\mathbb{D}} \bowtie [[t_3]]_{\mathbb{D}}) \cup \\ &([[t_1]]_{\mathbb{D}} \bowtie [[t_2]]_{\mathbb{D}} \bowtie [[t_3]]_{\mathbb{D}})\end{aligned}$$

Par conséquent, l'expression $[[t_1]]_{\mathbb{D}} \leftouter^* [[t_2]]_{\mathbb{D}} \leftouter^* [[t_3]]_{\mathbb{D}}$ calcule les mappings qui satisfont t_1 ou t_2 ou t_3 ou une combinaison de ces patrons de triplet. Ce résultat peut être généralisé à n patrons de triplet et par conséquent, l'expression $[[t_1]]_{\mathbb{D}} \leftouter^* \cdots \leftouter^* [[t_n]]_{\mathbb{D}}$ calcule les mappings qui satisfont un patron de triplet de Q ou une conjonction de ces patrons de triplet (sous-requêtes de Q), c'est-à-dire, l'ensemble $ps(Q, \mathbb{D})$.

L'algorithme 3 est basé sur la définition précédente. Il calcule la matrice de relaxation en utilisant une boucle imbriquée. Chaque patron de triplet t_i est évalué sur $\mathbb{D}$ pour obtenir l'ensemble des mappings $[[t_i]]_{\mathbb{D}}$ (ligne 3). Ensuite, chaque mapping μ de cet ensemble est comparé avec chaque mapping μ' dans la matrice (ligne 5). Si, μ et μ' sont compatibles, le mapping $\mu' \cup \mu$ est inséré dans la matrice avec la valeur 1 affectée à la colonne t_i et pour les autres colonnes, les valeurs des colonnes correspondantes à la ligne μ' sont recopiées. Si ce mapping existe déjà dans la matrice, la ligne correspondante est mise à jour en affectant la valeur 1 à la colonne t_i (ligne 6-12). Afin de respecter la sémantique de la jointure

25. Vu que la sémantique de SPARQL ne rejette pas les valeurs nulles, à la différence de l'algèbre relationnelle, cette expression n'est pas équivalente à : $(\Omega_1 \cup \Omega_2) \bowtie (\Omega_1 \cup \Omega_2)$.

Algorithm 3: Calcul de la matrice de relaxation d'une requête SPARQL Q

```
ComputeMatrix(Q, 𝔻)
    inputs : Requête qui échoue Q = t1 ∧ ... ∧ tn; BD-RDF 𝔻
    output: La matrice de relaxation M
1   M ← ∅;
2   foreach patron de triplet ti ∈ Q do
3       foreach μ ∈ [[ti]]𝔻 do
4           isInserted ← false;
5           foreach μ' ∈ M do
6               if μ et μ' sont compatibles then
7                   if (μ' ∪ μ) ∉ M then
8                       M ← M ∪ {μ' ∪ μ};
9                       M[μ' ∪ μ][tk] ← M[μ'][tk] for k ∈ 1 ··· n ∧ k ≠ i;
10                  M[μ' ∪ μ][ti] ← 1;
11                  if (μ ∪ μ') = μ then
12                      isInserted ← true;
13          if not isInserted then
14              M ← M ∪ {μ};
15              M[μ][tk] ← 1 if k = i, else 0; (k ∈ 1 ··· n)
16  return M;
```

étendue, la mapping μ est ajouté (si ce n'est pas déjà fait) dans la matrice avec la colonne t_i à 1 et les autres colonnes à 0 (ligne 13-15).

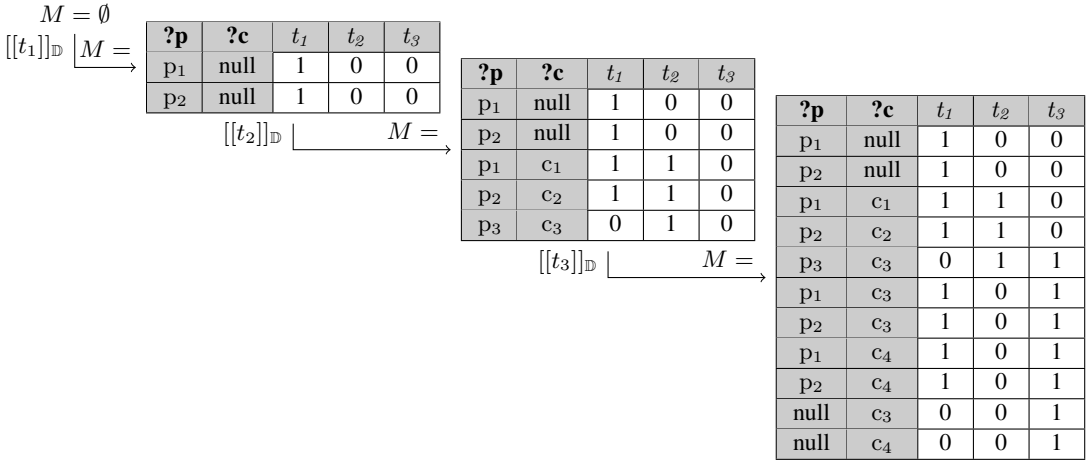


FIGURE 5.8 – Exécution de l'algorithme 3 qui calcule la matrice de relaxation

Un exemple d'exécution de l'algorithme 3 pour la requête SPARQL de la figure 5.6(b) est présenté à la figure 5.8. L'algorithme exécute le patron de triplet t_1 de la requête Q. Chaque résultat est inséré dans la matrice de relaxation avec la valeur 1 affectée à la colonne t_1 et 0 pour les autres colonnes. Ensuite, l'algorithme exécute le patron de triplet t_2. Parmi les mappings de t_2, deux d'entre eux sont compatibles avec les mappings déjà dans la matrice de relaxation M. Ils sont insérés dans la matrice M

avec les valeurs des colonnes t_1 et t_2 à 1. Le mapping $?p \rightarrow p_3, ?c \rightarrow c_3$ n'est compatible avec aucun des mappings de M. Il est ajouté dans la matrice avec la valeur 1 pour la colonne t_2 et 0 pour les autres. Le même processus est appliqué pour le patron de triplet t_3 afin de calculer la matrice de relaxation finale.

L'algorithme 3 n'exécute que n requêtes, n étant le nombre de patrons de triplet de la requête. Cependant, nos expérimentations sur le benchmark LUBM [21] montrent que cet algorithme peut, malgré cela, avoir un coût important en temps étant donnée la taille importante de la matrice de relaxation, ce qui arrive lorsque les requêtes sont composées de patrons de triplet peu sélectifs. De plus, les sous-requêtes de la requête initiale peuvent contenir un produit cartésien, ce qui aura pour conséquence un coût très important de calcul de la matrice de relaxation et également une importante augmentation de la taille de cette matrice. Afin d'améliorer les performances de cet algorithme, nous proposons une spécialisation de l'algorithme pour les requêtes *en étoile* [26] que nous retrouvons fréquemment dans les fichiers logs des BD-RDF réelles [14].

5.4.3 Optimisation du calcul de la matrice de relaxation

Dans cette section, nous proposons deux approches pour calculer la matrice de relaxation des requêtes SPARQL en étoile. La première approche, nommée NQ, est indépendante de l'implémentation de la BD-RDF. La seconde, nommée $1Q$, nécessite une BD-RDF implémentée comme une table de triplet dans une base de données relationnelle. Afin d'illustrer ces approches dans cette section, nous considérons la requête SPARQL de l'exemple à la figure 5.9(b).

Triplets		
s	**p**	**o**
p_1	type	Professor
p_1	researchInterest	SW
p_1	teacherOf	DB
p_2	type	Lecturer
p_2	teacherOf	DB
p_3	teacherOf	DB
p_4	researchInterest	SW
p_5	type	Lecturer
p_5	teacherOf	AI
p_6	type	Professor
p_6	researchInterest	NLP
p_6	teacherOf	WS
p_7	type	Professor
p_7	name	John
p_7	researchInterest	NLP

(a) Triplets RDF

```
SELECT ?p ?a WHERE {
?p age ?a                          (t1)
?p type Lecturer                   (t2)
?p researchInterest "SW"           (t3)
?p teacherOf "DB" }                (t4)
```

(b) La requête Q

?p	t_1	t_2	t_3	t_4	
p_1	0	0	1	1	*
p_2	0	1	0	1	*
p_3	0	0	0	1	
p_4	0	0	1	0	
p_5	0	1	0	0	

(c) Matrice de relaxation de Q

$xss(Q) = \{t_2 \wedge t_4, t_3 \wedge t_4\}$
$mfs(Q) = \{t_1, t_2 \wedge t_3\}$

(d) Les MFS et XSS de Q

FIGURE 5.9 – Illustration de l'approche MBA

5.4.3.1 Approche NQ

Le calcul de la matrice de relaxation pour les requêtes en étoiles est plus simple que dans le cas général, où l'algorithme 3 est utilisé pour tous les types de requête. Il existe plusieurs raisons à cette simplification. La première raison est que les sous-requêtes d'une requête étoile ne peuvent pas contenir un produit

26. Ce sont des requêtes dont tous les patrons de triplets ont la même variable comme sujet, par exemple la requête de la figure 5.9(b).

cartésien. La seconde est qu'une seule variable, notée x, est utilisée pour réaliser la jointure entre tous les patrons de triplet. Cette dernière propriété nous permet de ne sauvegarder que les valeurs de la variable de jointure x (c'est-à-dire la restriction de la fonction μ à la variable x notée $\mu_{|\{x\}}$) dans la matrice de relaxation. Les valeurs de la variable de jointure qui satisfont un patron de triplet t sont évaluées de la manière suivante : $[[t]]_{\mathbb{D}} = \{\mu_{|\{x\}} \mid dom(\mu) = var(t) \wedge \mu(t) \in \mathbb{D}\}$ et la matrice de relaxation est calculée en utilisant la proposition suivante.

Proposition 6

L'ensemble des potentielles réponses d'une requête SPARQL en étoile peut être calculé en utilisant une jointure externe de la manière suivante : $ps(Q, \mathbb{D}) = [[t_1]]_{\mathbb{D}} \mathbin{\text{⟗}} \cdots \mathbin{\text{⟗}} [[t_n]]_{\mathbb{D}}$.

Preuve : Considérons t_1 et t_2, deux patrons de triplet d'une requête en étoile. Si $\mu_1 \in [[t_1]]_{\mathbb{D}}$ et $\mu_2 \in [[t_2]]_{\mathbb{D}}$, μ_1 et μ_2 sont compatibles s'ils ont la même valeur pour la variable de jointure, c'est-à-dire, $[[t_1]]_{\mathbb{D}} \bowtie [[t_2]]_{\mathbb{D}} = [[t_1]]_{\mathbb{D}} \cap [[t_2]]_{\mathbb{D}}$. En utilisant cette propriété et les règles de la théorie des ensembles, nous pouvons déduire que :

$$
\begin{aligned}
[[t_1]]_{\mathbb{D}} \mathbin{\text{⟗}} [[t_2]]_{\mathbb{D}} &= ([[t_1]]_{\mathbb{D}} \cap [[t_2]]_{\mathbb{D}}) \cup ([[t_1]]_{\mathbb{D}} - [[t_2]]_{\mathbb{D}}) \cup ([[t_2]]_{\mathbb{D}} - [[t_1]]_{\mathbb{D}}) \\
&= ([[t_1]]_{\mathbb{D}} \cap [[t_2]]_{\mathbb{D}}) \cup [[t_1]]_{\mathbb{D}} \cup [[t_2]]_{\mathbb{D}} \\
&= [[t_1]]_{\mathbb{D}} \mathbin{\text{⟗}}^* [[t_2]]_{\mathbb{D}}
\end{aligned}
$$

Par conséquent, dans le cas des requêtes en étoile, la jointure externe est équivalente à la jointure externe étendue et nous avons montré dans la proposition 5 que l'opérateur de jointure externe étendue calcule la matrice de relaxation d'une requête.

Algorithm 4: Calcul de la matrice pour les requêtes en étoile (NQ)

ComputeMatrixStarQueryNQ(Q, $\mathbb{D}$)
  inputs : Une requête en étoile $Q = t_1 \wedge ... \wedge t_n$ qui échoue, avec x comme variable de jointure;
  Une BD-RDF $\mathbb{D}$
  output: La matrice de relaxation M
1 $M \leftarrow \emptyset$;
2 **foreach** *patron de triplets* $t_i \in Q$ **do**
3   **foreach** $\mu \in [[t_i]]_{\mathbb{D}}$ **do**
4   **if** $\mu_{|\{x\}} \notin M$ **then**
5       $M \leftarrow M \cup \{\mu_{|\{x\}}\}$;
6       $M[\mu_{|\{x\}}][t_k] \leftarrow 0$ pour $k \in 1 \cdots n \wedge k \neq i$;
7     $M[\mu_{|\{x\}}][t_i] \leftarrow 1$;
8 **return** M;

L'algorithme 4, appelé NQ, utilise la proposition précédente pour calculer la matrice de relaxation d'une requête en étoile. Cet algorithme exécute une requête pour chaque patron de triplet t_i (ligne 2). Pour chaque mapping μ résultant, si μ n'est pas dans la matrice de relaxation, la valeur de la variable de jointure est ajoutée à la matrice de relaxation avec la valeur 1 pour la colonne t_i et 0 pour les autres colonnes (lignes 4-6). Par contre, si le mapping μ est déjà dans la matrice de relaxation, alors la ligne correspondante à ce mapping dans la matrice est mise à jour en affectant la valeur 1 à la colonne t_i (ligne 7).

Une illustration de l'exécution de l'algorithme 4 pour notre requête exemple (voir figure 5.9(b)) est présentée à la figure 5.10. L'exécution du patron de triplet t_1 retourne un ensemble vide de réponse et donc la matrice de relaxation reste vide. Celle du patron de triplet t_2 retourne deux résultats qui sont ajoutés comme nouvelles lignes de la matrice de relaxation avec la valeur 1 pour la colonne t_2 et 0 pour les autres. Le même processus est appliqué pour le patron de triplet t_3, ce qui ajoute deux nouvelles lignes dans la matrice de relaxation. Enfin, le patron de triplet t_4 est exécuté. Il retourne les mappings $?p \rightarrow p_1$, $?p \rightarrow p_2$ et $?p \rightarrow p_3$. Comme les deux premiers sont déja présents dans la matrice de relaxation, les lignes correspondantes sont mises à jour avec la valeur 1 à la colonne t_4. Le mapping $?p \rightarrow p_3$ est inséré comme une nouvelle ligne de la matrice de relaxation avec une valeur de 1 uniquement pour la colonne t_4 et 0 pour les autres colonnes.

$M = \emptyset$

$[[t_1]]_{\mathbb{D}}$ $M = \emptyset$

$[[t_2]]_{\mathbb{D}}$ $M =$

?p	t_1	t_2	t_3	t_4
p2	0	1	0	0
p5	0	1	0	0

$[[t_3]]_{\mathbb{D}}$ $M =$

?p	t_1	t_2	t_3	t_4
p2	0	1	0	0
p5	0	1	0	0
p1	0	0	1	0
p4	0	0	1	0

$[[t_4]]_{\mathbb{D}}$ $M =$

?p	t_1	t_2	t_3	t_4
p2	0	1	0	1
p5	0	1	0	0
p1	0	0	1	1
p4	0	0	1	0
p3	0	0	0	1

FIGURE 5.10 – Un exemple d'exécution de l'algorithme 4

5.4.3.2 Approche 1Q

L'algorithme NQ est utilisé pour n'importe quelle BD-RDF (implémentée sur des bases de données relationnelles ou pas). Si nous considérons les données RDF stockées dans une table de triplet $T(s, p, o)$ dans une base de données relationnelle, nous pouvons utiliser une seule requête pour calculer la matrice de relaxation qui sera la traduction en SQL de l'expression : $[[t_1]]_{\mathbb{D}} \ltimes \cdots \ltimes [[t_n]]_{\mathbb{D}}$. Inspiré par les travaux de Cyganiak [241] sur la traduction des requêtes SPARQL en SQL, nous utilisons la jointure externe de SQL pour calculer ces expressions et la fonction $coalesce$ [27] pour prendre en compte les valeurs nulles. Nous utilisons également l'opérateur $case$ pour vérifier si un mapping est solution d'un patron de triplet ou pas et ainsi affecter la valeur appropriée dans la matrice (1 si c'est une solution, 0 sinon).

Illustrons maintenant cette approche avec la requête Q de la figure 5.9(b). Nous notons t_i la requête ou la vue qui calcule les valeurs de la variable de jointure pour le patron de triplet t_i. Par exemple, t_1 est la requête suivante : `select distinct s from t where p='age'`. En utilisant cette notation, la requête SQL utilisée pour calculer la matrice de relaxation de notre requête exemple est :

```
SELECT coalesce(t1.s , t2.s, t3.s, t4.s),
       case when t1.s is null then 0 else 1 end as t1,
       case when t2.s is null then 0 else 1 end as t2,
       case when t3.s is null then 0 else 1 end as t3,
       case when t4.s is null then 0 else 1 end as t4
```

27. La fonction $coalesce$ retourne les premières expressions non-nulles dans la liste des paramètres.

```
FROM t1 full outer join t2 on t1.s = t2.s
       full outer join t3 on coalesce(t1.s , t2.s) = t3.s
       full outer join t4 on coalesce(t1.s , t2.s, t3.s) = t4.s
```

Cette approche, nommée $1Q$, a deux principaux avantages : elle utilise une seule requête pour calculer la matrice de relaxation et les SGBDR choisissent l'algorithme adéquat pour l'opérateur de jointure. Pour toutes ces raisons, l'algorithme $1Q$ est plus rapide que l'algorithme précédent (environ 25% plus rapide que l'approche générale dans nos expérimentations). Elle possède néanmoins l'inconvénient de n'être possible que pour un petit nombre de BD-RDF étant donné qu'un nombre important de BD-RDF n'utilisent pas un système de stockage basé sur les bases de données relationnelles.

5.4.4 Recherche des MFS et XSS en utilisant la matrice de relaxation

En se permettant un abus de notation, nous notons $xss(\mu)$ la sous-requête maximale de Q qui retourne un mapping $\mu \in ps(Q, \mathbb{D})$. Elle peut être directement obtenue à partir de la matrice de relaxation : $xss(\mu) = \{t_i \wedge ... \wedge t_j \mid \forall t_k \in \{t_i, ..., t_j\} : \ M[\mu][t_k] = 1\}$. La recherche des XSS de la requête Q est réalisée par les deux étapes suivantes.

1. Calculer le *skyline* SKY de la matrice de relaxation : $SKY(M) = \{ \mu \in ps(Q, \mathbb{D}) \mid \nexists \mu' \in ps(Q, \mathbb{D}) : \mu \prec \mu'\}$ où $\mu \prec \mu'$ si (i) sur chaque patron de triplet t_i, $M[\mu][t_i] \leq M[\mu'][t_i]$ et (ii) sur au moins un patron de triplet t_j, $M[\mu][t_j] < M[\mu'][t_j]$. Cette étape peut etre réalisée en utilisant un des algorithmes définis dans la littérature pour calculer efficacement le skyline d'une table relationnelle (dans ce contexte un état de l'art est proposé par Hose et al. [242]). Dans la figure 5.9(c), les lignes qui composent le skyline de la matrice de relaxation sont marquées avec le caractère $*$.
2. Retourner les sous-requêtes maximales distinctes de Q qui récupèrent un élément du skyline : $xss(Q) = \{xss(\mu) \mid \mu \in SKY(M)\}$. Chacune de ces sous-requêtes est une XSS. Les XSS de notre exemple sont présentées à la figure 5.9(d). Elles sont en gris dans la matrice de relaxation.

Pour le calcul des MFS, nous avons observé que la matrice de relaxation fournit un moyen de savoir si une requête retourne un résultat vide ou non. Il suffit pour cela de calculer l'intersection des colonnes de la matrice qui correspondent aux patrons de triplet de la requête. Si la colonne qui en résulte est uniquement composée de 0, alors la requête retourne un ensemble vide et inversement. Or, dans l'approche LBA, les sous-requêtes sont exécutées sur la BD-RDF pour vérifier si elles retournent ou pas des réponses. Ainsi, au lieu d'exécuter ces sous-requêtes, nous pouvons utiliser la matrice de relaxation. Par conséquent, la matrice de relaxation calculée dans l'approche MBA peut être utilisée comme index pour accroître les performances de l'approche LBA. Même si ce dernier explorera toujours un espace de recherche qui augmente de façon exponentielle avec le nombre de patrons de triplet de la requête initiale, cette exploration ne nécessitera plus de requêtes sur la BD-RDF une fois que la matrice de relaxation aura été calculée.

5.5 Implémentation et expérimentations

Nous avons réalisé deux implémentations de la recherche des MFS et XSS dans les requêtes SPARQL. Nous avons, d'une part, intégré l'approche LBA dans le système QaRS et, d'autre part, implémenté un prototype pour l'évaluation des différentes approches LBA et MBA, en comparaison avec les algorithmes naïfs et de l'état de l'art. Dans cette section, nous présentons ces deux implémentations et nous analysons également les résultats de l'évaluation de nos approches.

5.5.1 Implémentation dans QaRS

Dans l'architecture du système QaRS, présentée dans le chapitre précédent, le module de recherche des causes d'échecs d'une requête SPARQL implémente l'algorithme LBA pour trouver les MFS d'une requête qui échoue. Ce module assure la fonctionnalité d'explication de l'échec de la requête et suggère aux utilisateurs les patrons de triplet sur lesquels les opérateurs de relaxation doivent être utilisés. En plus de suggérer les patrons de triplet à relaxer dans une requête qui échoue, QaRS suggère également les XSS aux utilisateurs. Les XSS sont proposées aux utilisateurs comme des relaxations par suppression de certains patrons de triplet. S'ils ont l'avantage de conserver le plus grand nombre possible de patrons de triplet, leur limite est qu'ils réalisent une relaxation extrême (suppression) et pas incrémentale.

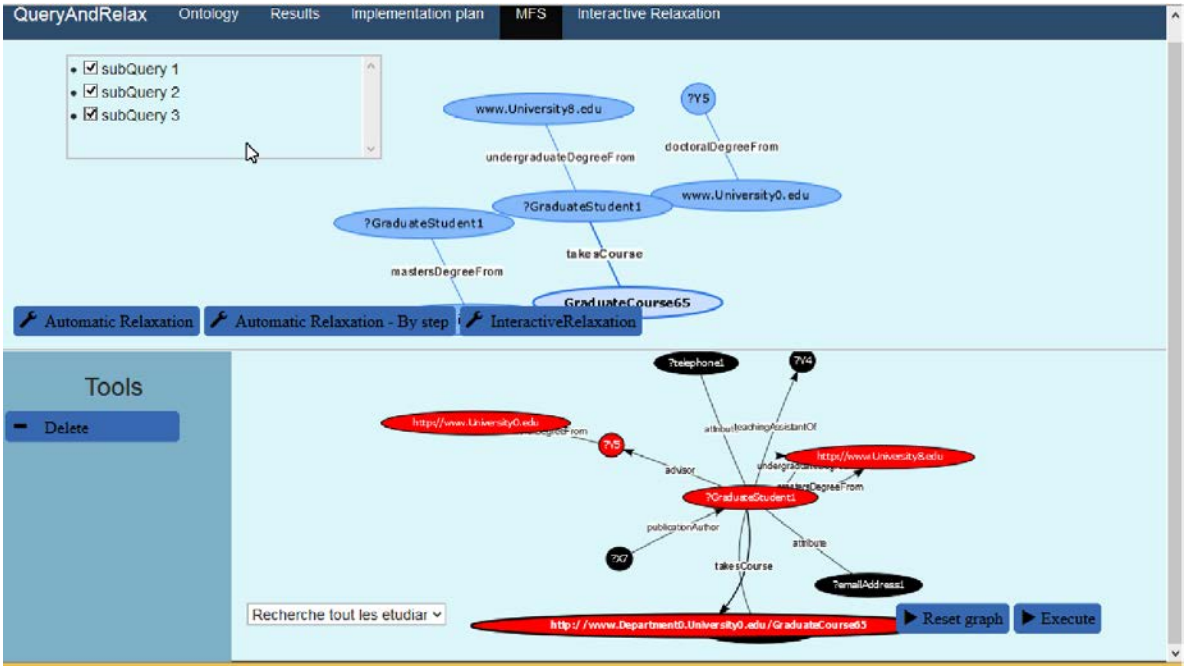


FIGURE 5.11 – Recherche des MFSs dans QaRS

La figure 5.11 présente l'interface de QaRS en mode recherche de MFS. Le panneau du bas présente la requête initiale et celui du haut une MFS (qui est une sous-requête de la requête initiale) sélectionnée par l'utilisateur parmi toutes les MFS trouvées. Une fois la MFS sélectionnée dans le panneau du haut, les nœuds contenus dans cette MFS prennent un fond rouge dans la requête initiale. Comme nous pouvons le voir les relaxations proposées par QaRS sont la relaxation automatique, l'usage des opérateurs de relaxation sur les nœuds des MFS choisis par l'utilisateur (relaxation interactive) ou la suppression de patrons de triplet (XSS).

5.5.2 Protocole de l'expérimentation

Nous avons également implémenté un prototype d'évaluation des algorithmes LBA, LBA-OPT (LBA avec les techniques d'optimisations vues dans la section 5.3), MBA et ISHAMEL de Godfrey [200] adapté au contexte RDF. Ce prototype d'expérimentation est implémenté en JAVA 1.8 64 bits et est exécuté sur les BD-RDF Jena TDB (version 1.0.1) et Sesame Native Store (version 2.8.4). Ces deux BD-RDF ont été choisis pour leur popularité dans le domaine du Web sémantique, l'utilisation d'une autre BD-RDF est possible et nécessite uniquement d'implémenter une interface qui définit des opérations

basiques telles que la création d'une connexion ou l'exécution d'une requête SPARQL. Notre implémentation est disponible à l'adresse : `http://www.lias-lab.fr/forge/projects/qars`.

Nos expérimentations ont été réalisées sur un serveur Ubuntu 4.04.02 LTS 64 bits avec un CPU Intel XEON E5-2630 v3 @2.4Ghz et 32GO RAM. Nous reportons ici uniquement les expérimentations réalisées sur Jena TDB, car les résultats obtenus avec Sesame étaient similaires. Les temps d'exécution reportés sont une moyenne des durées de 5 exécutions consécutives de chaque algorithme . Avant d'enregistrer la première mesure, l'algorithme est lancé une fois. Pour les requêtes ayant des temps d'exécution très importants ou requérant un espace mémoire supérieur à la mémoire de la machine virtuelle Java, les indicateurs de performance ne sont pas représentés.

Comme dans les précédents travaux sur la relaxation de requêtes RDF [243, 17], nous avons utilisé cinq jeux de données générés avec le banc d'essai LUBM [21]. Ces jeux de données, décrits dans le tableau 5.1, contiennent de 13 millions (LUBM100) à 130 millions (LUBM1K) triplets RDF.

	LUBM100	LUBM250	LUBM500	LUBM750	LUBM1K
Nombre de patrons	13M	30M	65M	90M	130M
Nombre d'instances	2M	5M	11M	16M	22M

TABLE 5.1 – Caractéristiques des jeux de données

Comme le banc d'essai LUBM contient principalement des requêtes qui réussissent, nous avons généré au hasard un ensemble de 225 requêtes qui échouent. Les caractéristiques de ces requêtes ont été définies à partir de l'étude proposée par Arias Gallego et al. [14] sur les requêtes SPARQL utilisées sur des données réelles. D'après cette étude, les requêtes SPARQL des utilisateurs sur des données réelles possèdent généralement entre 1 et 15 patrons de triplet et sont des requêtes en *étoile*, en *chaîne* ou *composite*. Nous rappelons que :

- une requête en étoile est caractérisée par la jointure entre les sujets, $sujet - sujet$, des différents patrons de triplet de la requête, la variable de jointure est le sujet des patrons de triplet ;
- une requête en chaîne est caractérisée par une jointure entre l'objet d'un patron de triplet et le sujet d'un autre, $objet - sujet$. La variable de jointure est objet dans l'un et sujet dans l'autre ;
- une requête composite est caractérisée par un usage aléatoire des jointures $sujet-sujet$, $objet-objet$, $sujet - objet$ ou $objet - sujet$, avec au moins deux types différents dans la requête.

Nous avons généré cinq requêtes pour un type de requête et un nombre de patrons de triplet donnés. Par exemple, nous générons cinq requêtes étoile avec un patron de triplet ou encore cinq requêtes composites avec 15 patrons de triplet. Pour chaque type de requête nous avons ainsi 75 (5*15) requêtes, cinq par nombre de patrons de triplet. Nous n'avons pas généré des requêtes avec des variables en position de $predicat$ dans les patrons de triplet car ces requêtes ne sont pas fréquentes en pratique [14]. le tableau 5.2 montre un exemple des trois types de requêtes générés [28]

5.5.3 Temps de calcul de la matrice de relaxation dans MBA

L'approche MBA est basée sur la matrice de relaxation. Pour l'implémentation de la structure de données utilisée pour la matrice de relaxation, nous avons exploité les similitudes existantes entre la matrice de relaxation et les index bitmap utilisés dans les bases de données relationnelles. Nous avons donc défini la matrice de relaxation comme un ensemble de bitmaps, un bitmap par colonne de la matrice de relaxation. Nous avons utilisé la bibliothèque Roaring bitamp [244] comme implémentation du type bitmap,

28. Pour des besoins de lisibilité, nous avons tronqué les URI.

Étoile	SELECT $*$ WHERE {$?X$ *rdf:type ub:FullProfessor* . $?X$ *ub:age* $?Y1$. $?X$ *ub:memberOf* $?Y2$. $?X$ *ub:doctoralDegreeFrom* <*University911*> . }
Chaîne	SELECT $*$ WHERE {$?Y1$ *ub:softwareDocumentation* $?Y2$. $?Y2$ *ub:publicationAuthor* $?Y3$. $?Y3$ *ub:headOf* $?Y4$. $?Y4$ *ub:affiliateOf* <*FullProfessor4*> . }
Composite	SELECT $*$ WHERE <*Dept8*> *ub:subOrganizationOf* $?Y1$. $?Y1$ *ub:subOrganizationOf* <*University7*> . $?Y2$ *ub:headOf* $?Y1$. $?Y3$ *ub:affiliateOf* $?Y2$. }

TABLE 5.2 – Exemples de requêtes générées avec quatre patrons de triplet

qui permet d'obtenir un bon taux de compression. Comme nous pouvons le voir sur le tableau 5.3, l'implémentation de la matrice de relaxation avec la bibliothèque Roaring bitamp permet d'avoir une matrice de relaxation de faible taille (de l'ordre du méga-octet) même si elle possède un nombre important de lignes. La matrice de relaxation a, par exemple, une taille de 1MO pour 1.7 millions de lignes. Le tableau 5.3 contient uniquement les indicateurs pour les requêtes en étoile car, les autres types de requête exigent énormément de ressources à cause des produits cartésiens. MBA n'est donc pas adéquat pour ces types de requêtes.

Pour le temps de calcul de la matrice de relaxation, nous comparons l'algorithme 4 (NQ) avec l'algorithme $1Q$. Étant donné que cette dernière requière des données RDF stockées dans une table de triplet implémentée dans une base données relationnelle, nous avons réalisé cette évaluation sur Oracle 12c, avec notre propre implémentation de la représentation horizontale.

	3 PT	5 PT	8 PT	10 PT	13 PT	15 PT
Temps de calcul avec NQ (sec)	2.36	8.9	17.3	25.5	29.3	36.2
Temps de calcul avec 1Q (sec)	2.36	9	15.5	19.6	23.6	29
Taille (KO)	80	320	480	620	620	1290
Nombre de lignes (Millier)	249	911	1326	1532	1532	1738

TABLE 5.3 – Propriété de la matrice de relaxation sur LUBM100 (PT = Patrons de triplet)

Les résultats de cette évaluation, présentés dans le tableau 5.3, montrent que la technique $1Q$ est approximativement 25% plus rapide que l'algorithme NQ. Cependant, cette optimisation reste valable uniquement pour les BD-RDF utilisant une table de triplets, puisque la technique $1Q$ ne peut être implémentée dans des BD-RDF telles que Jena TDB. Malgré cette optimisation, le temps de calcul de la matrice reste quand même important, de l'ordre de 30 secondes pour une requête avec 15 patrons de triplet. L'approche MBA reste néanmoins intéressante car la matrice de relaxation peut-être pré-calculée pour des requêtes qui échouent et sont utilisées très fréquemment. Ces requêtes peuvent être identifiées dans *des fichiers log* de requêtes par exemple.

5.5.4 Évaluation du temps de recherche des MFS et XSS

Nous avons testé les performances (en termes de temps de réponse) des algorithmes suivants (lorsque la taille de la requête et la taille du jeu de données augmentent) :

- LBA : l'algorithme présenté en section 5.2.
- LBA-OPT : il s'agit de l'algorithme LBA optimisé avec les techniques présentées dans la section 5.3 (le cache est implémenté sous forme de liste).
- MBA+M : cet algorithme calcule la matrice relaxée puis calcule les XSS et MFS de la requête en utilisant l'algorithme LBA qui s'appuie sur cette matrice pour ne pas exécuter de requêtes.
- MBA-M : même que MBA+M mais sans le calcul de la matrice.
- DFS : un algorithme naïf de recherche en profondeur du treillis des sous-requêtes. Pour chaque sous-requête parcourue qui échoue (respectivement qui réussie), nous testons si c'est une MFS (respectivement une XSS). Cet algorithme exécute chaque sous-requête une fois (au total $2^n - 2$ requêtes où n est le nombre de patrons de triplets de la requête).
- ISHMAEL : l'algorithme proposé dans [200] que nous avons modifié pour retourner à la fois les XSS et les MFS d'une requête RDF.

Nous présentons ces évaluations pour chaque type de requête : *étoile*, *chaîne* et *composite*. L'approche MBA n'est évaluée que dans le cas des requêtes en étoile, puisqu'elle ne passe pas à l'échelle pour les autres types de requêtes.

5.5.4.1 Résultats dans le cas des requêtes en étoile

Dans la figure 5.12, nous présentons le temps de réponse des algorithmes testés en fonction de la taille des requêtes (en nombre de patrons de triplets) et dans la figure 5.13 en fonction de la taille du jeu de données. Ces graphiques sont présentés en échelle logarithmique pour des raisons de lisibilité. La table 5.4 indique le nombre de requêtes exécutées pour chaque algorithme (sauf pour les algorithmes basés sur MBA qui n'utilisent que n requêtes, où n est la taille de la requête).

		3 PT	5 PT	8 PT	10 PT	13 PT	15 PT
LBA	#Requêtes exécutées	5	14	37	67	117	175
LBA-OPT	#Requêtes exécutées	4	11	30	57	103	153
	#Requêtes dans cache	1	4	7	10	14	22
DFS	#Requêtes exécutées	6	30	254	1022	8190	32766
ISHMAEL	#Requêtes exécutées	9	27	76	138	260	393

TABLE 5.4 – Nombre de requêtes exécutées sur LUBM100 (PT = Patrons de Triplets)

Comme le montrent nos résultats, un algorithme qui exécute toutes les sous-requêtes tel que DFS ne peut être utilisé que pour des requêtes comportant peu de patrons de triplets. Pour les requêtes plus larges, le nombre de requêtes exécutées augmente de manière exponentielle et ainsi la performance de DFS décroit très rapidement. Dans ce cas, l'exploration avec élagage proposé par les algorithmes LBA et ISHMAEL est plus efficace. Leur temps de réponse est en dessous d'une seconde pour des requêtes qui ont moins de 12 patrons de triplets sur LUBM100. Pour les requêtes plus larges, LBA a des meilleurs performances qu'ISHMAEL. LBA retourne le résultat en environ 2 secondes tandis qu'ISHMAEL le fait en environ 10 secondes. Nous avons identifié que cela vient du processus de calcul des XSS potentielles qui permet une exécution d'un nombre moindre de requêtes pour LBA en comparaison avec ISHMAEL (voir table 5.4). Comme la figure 5.13 le montre, le temps de réponse de ces algorithmes augmente linéairement avec la taille du jeu de données.

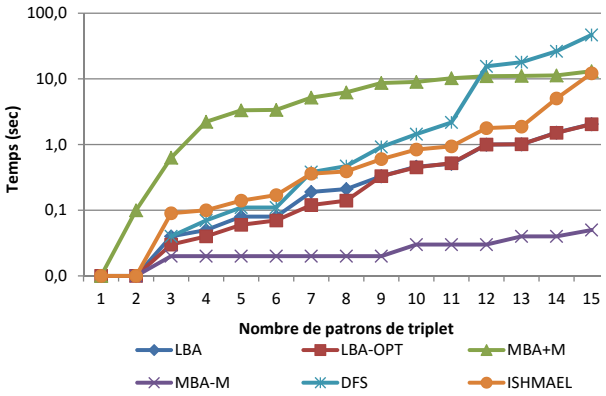


FIGURE 5.12 – Temps de réponse vs taille des requêtes, LUBM100

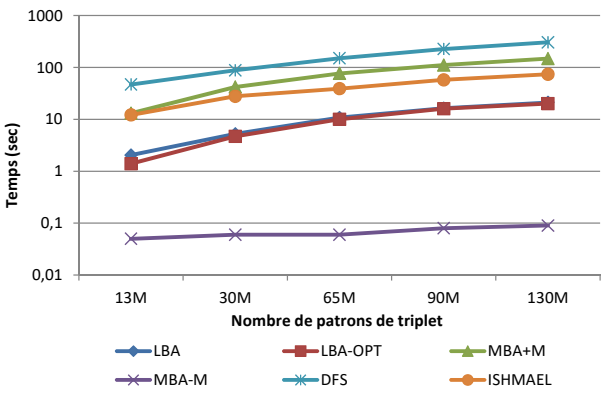


FIGURE 5.13 – Temps de réponse vs taille du jeu de données, requête 15PT

Concernant l'approche MBA-M, elle a le meilleur temps de réponse qui est de quelques millisecondes même pour les requêtes avec 15 patrons de triplets sur LUBM1K. Ceci vient du fait que cet algorithme doit simplement calculer l'intersection de colonnes de la matrice au lieu d'exécuter des requêtes. Cependant cet algorithme repose sur l'hypothèse forte que la matrice a été pré-calculée, c'est-à-dire que la requête a dû être identifiée comme une requête qui échoue (par exemple, en utilisant l'historique des requêtes exécutées). Si la matrice est calculée au cours de l'exécution de cet algorithme (MBA+M), ce temps de réponse est beaucoup plus important car le calcul de la matrice est coûteux (environ 30 secondes pour 15 patrons de triplets).

Enfin, nous observons que l'algorithme LBA et LAB-OPT ont des temps d'exécution proches. Ce résultat vient du fait qu'une requête en étoile ne peut pas produire un produit cartésien quels que soient les patrons de triplet supprimés. L'analyse de l'optimisation par l'utilisation du cache, nous montre que LBA-OPT exécute approximativement 10% de requêtes de moins que LBA. Ce pourcentage n'impacte pas le temps d'exécution car ces 10% de requêtes non exécutées par LBA-OPT sont des requêtes de petites tailles, ayant des temps d'exécution insignifiant par rapport aux requêtes de grandes tailles exécutées par les deux algorithmes pendant la recherche des MFS et XSS. Si les techniques d'optimisation n'apportent pas de gain de temps dans ce cas, elles ont un plus grand impact pour les autres types de requêtes, comme nous le montrons par la suite.

5.5.4.2 Résultats dans le cas des requêtes en chaîne

Les figures 5.14 et 5.15 et le tableau 5.5 présentent les résultats que nous avons obtenus pour les requêtes SPARQL en chaîne. Dans ce cas, les sous-requêtes provenant de la suppression d'un patron de triplet autre que l'une des extrémités de la requête en chaîne conduit inévitablement à un produit cartésien. Ces produits cartésiens en plus de rendre l'approche MBA difficilement exploitable, à cause de la taille de la matrice, rendent également l'approche naïve DFS inexploitable pour des requêtes de plus de 6 patrons de triplet car à ce niveau le processus prend plus de trois minutes pour LUBM100.

Pour les requêtes en chaîne nous constatons, sur la figure 5.14, que l'approche LBA a des temps d'exécution meilleurs que DFS, car il exécute 75% moins de requête que cet algorithme, et ISHMAEL , car il exécute 25% moins de requêtes. Mais, l'algorithme LBA exécute toujours des produits cartésiens très

		3 PT	5 PT	8 PT	10 PT	13 PT	15 PT
LBA	#Requêtes exécutées	5	14	33	50	75	100
LBA-OPT	#Requêtes exécutées	4	10	24	36	58	75
	#Requêtes dans cache	1	7	22	37	57	88
DFS	#Requêtes exécutées	6	30	254	1022	8190	32766
ISHMAEL	#Requêtes exécutées	11	31	74	113	169	225

TABLE 5.5 – Nombre de requêtes exécutées sur LUBM100 (PT = Patrons de Triplets)

coûteux en temps. Par conséquent, le moyen le plus sûr et le moins coûteux pour rechercher les MFS et les XSS dans une requête en chaîne est d'utiliser l'algorithme LBA-OPT, qui décompose les produits cartésiens en ces différentes composantes connexes, au lieu de les exécuter.

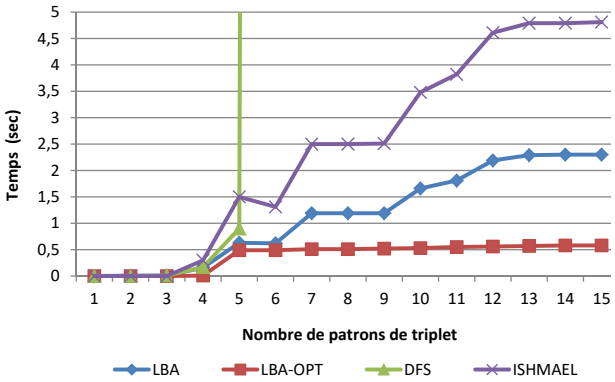


FIGURE 5.14 – Temps de réponse vs taille des requêtes, LUBM100

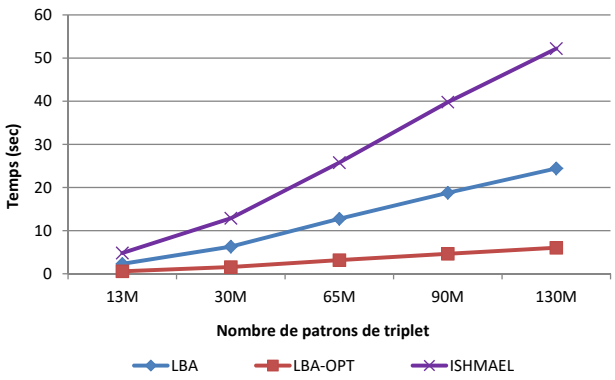


FIGURE 5.15 – Temps de réponse vs taille du jeu de données, requête 15 PT

Dans nos expérimentations, LBA-OPT est environ 67% plus rapide que LBA. Ce résultat de LBA-OPT s'explique par le fait qu'il utilise énormément le cache de requête (cf. table 5.5) et n'exécute pas les produits cartésiens. En effet, puisque la décomposition des produits cartésiens conduit à l'utilisation fréquente du cache de requête, LBA-OPT exécute approximativement 30% de requêtes de moins que LBA. Le temps de réponse de l'algorithme LBA-OPT est de moins d'une seconde pour toutes les requêtes exécutées sur le jeu de données LUBM100. La figure 5.15 montre que le temps d'exécution des algorithmes croît linéairement avec la taille des données, comme pour les requêtes en étoile. Par conséquent, ce temps de réponse est cinq fois plus grand pour les plus grandes requêtes exécutées sur LUBM1K.

5.5.4.3 Résultats dans le cas des requêtes composites

Les résultats des expérimentations menées sur les requêtes composites sont présentés dans le tableau 5.6 et les figures 5.16 et 5.17. Comme les requêtes composites sont une forme hybride intégrant à la fois les requêtes en étoile et en chaîne, la probabilité de rencontrer des produits cartésiens est moins importante que dans le cas des requêtes en chaîne. Cette probabilité impacte les courbes des temps d'exécution des algorithmes comme le montre la figure 5.16. Si MBA reste inexploitable et DFS toujours le moins performant, l'algorithme LBA-OPT est cette fois seulement 37% plus rapide que LBA (en moyenne) et 60% plus rapide que ISHMAEL. Même si LBA-OPT est moins performant que dans le cas des requêtes en

chaîne, il reste néanmoins celui qui donne les meilleurs résultats.

		3 PT	5 PT	8 PT	10 PT	13 PT	15 PT
LBA	#Requêtes exécutées	6	15	35	51	90	118
LBA-OPT	#Requêtes exécutées	5	11	26	38	67	86
	#Requêtes dans cache	1	5	17	28	49	82
DFS	#Requêtes exécutées	6	30	254	1022	8190	32766
ISHMAEL	#Requêtes exécutées	12	29	76	116	212	285

TABLE 5.6 – Nombre de requêtes exécutées sur LUBM100 (PT = Patrons de Triplets)

Le fait que LBA-OPT, soit moins efficace avec les requêtes composites en comparaison avec les requêtes en étoile a pour cause l'utilisation dans les requêtes composites de jointures différentes de celles de types *objet-sujet*[29] présentes dans les requêtes en chaîne. En particulier, nous avons observé que l'exécution d'une jointure *objet-objet* est particulièrement lente sur Jena TDB. Comme pour les cas précédents, la figure 5.17 montre que le temps d'exécution est linéaire par rapport à la taille des données interrogées.

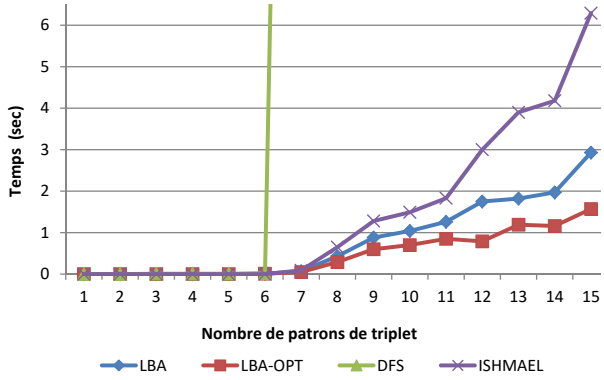


FIGURE 5.16 – Temps de réponse vs taille des requêtes, LUBM100

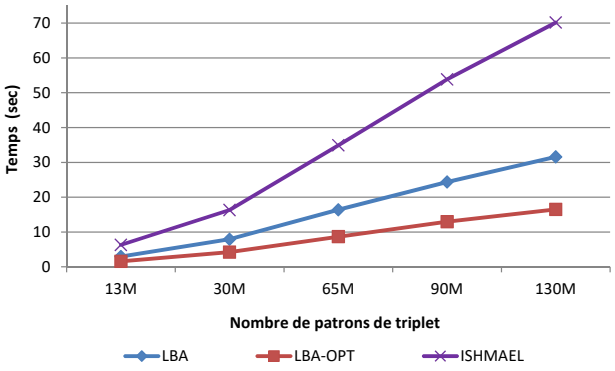


FIGURE 5.17 – Temps de réponse vs taille du jeu de données, requête 15 PT

Comme perspectives a ces expérimentations, nous envisageons d'évaluer ces algorithmes sur des données réelles (par exemple, DBPedia) avec des requêtes utilisateurs extraites des journaux de requêtes.

Conclusion

Ce chapitre avait pour objectif d'aider à l'identification des patrons de triplet sur lesquels appliquer les opérateurs que nous avons proposés dans le chapitre précédent. L'idée développée dans ce chapitre est de rechercher les causes d'échec des requêtes qui ne retournent aucune réponse. Pour le faire, nous avons proposé deux approches : LBA et MBA.

L'approche LBA s'inspire des travaux de Godfrey [200] dans les bases de données relationnelles. A la différence de Godfrey [200], LBA permet de rechercher en même temps les causes d'échec, nommées

29. l'objet du patron de triplet t_i est une variable égale au sujet du patron de triplet t_{i+1}

MFS, et les plus grandes sous-requêtes retournant au moins une réponse, nommées XSS. Pour faire cela, l'approche LBA recherche les MFS et XSS dans le treillis des sous-requêtes d'une requête qui échoue. Pour améliorer les performances de LBA, nous avons proposé cinq techniques d'optimisation qui permettent de réduire le nombre de requêtes exécutées dans LBA et/ou le temps d'exécution de ces requêtes.

La seconde approche proposée, MBA, s'inspire des travaux de Jannach [206] dans les systèmes de recommandation. Par rapport au travaux de Jannach, le contexte RDF nous a amené à développer des algorithmes de calcul de la matrice de relaxation qui s'appuient sur les particularités du langage SPARQL par rapport à SQL (plus précisément le traitement des valeurs nulles). L'approche MBA est intéressante car elle n'exécute que n requêtes, les n patrons de triplet composant la requête. Les réponses de chacune de ces exécutions sont insérées dans la matrice de relaxation suivant l'algorithme 3 pour tout type de requêtes. Mais la taille de la matrice générée pour stocker ces réponses potentielles pouvant être très importante suivant la requête, nous avons proposé une optimisation de l'approche MBA pour les requêtes en étoile uniquement, l'algorithme 4. Et, lorsque ces requêtes portent sur des données RDF stockées dans des bases de données relationnelles, nous avons proposé la construction de la matrice de relaxation en exécutant seulement une requête SQL. Ces optimisations permettent d'obtenir de meilleurs résultats.

Les expérimentations menées sur les approches LBA et MBA montrent l'intérêt de ces deux approches en comparaison avec une recherche en profondeur dans le treillis de sous-requête ou une adaptation de l'algorithme d'ISHMAEL de Godfrey. L'analyse des résultats de ces expérimentations nous ont permis de constater que l'algorithme LBA et sa version optimisée donnent de meilleurs performances que les algorithmes existants. Avec ces expérimentations, nous avons également pu déterminer les contextes dans lesquels l'approche MBA peut être exploitée efficacement, notamment en pré-calculant la matrice.

Pour relaxer une requête, une fois les patrons de triplet responsables de l'échec de la requête identifiées nous devons choisir parmi les patrons de triplet composant les MFS, ceux qui seront relaxés pour devenir moins sélectifs. Une fois les patrons de triplet choisis, nous pouvons appliquer les opérateurs sur le *sujet*, le *prédicat* ou l'*objet* suivant leur type. Par exemple, si le sujet ou l'objet est un littéral, nous utilisons l'opérateur `PRED`, si c'est une classe ou une propriété, ce sera `SIB` ou `GEN`.

Étant donné que chaque patron de triplet peut être relaxé de plusieurs façons différentes, un très grand nombre de requêtes relaxées peut être obtenu. Ce nombre important de requêtes relaxées rend très fastidieuse, pour les utilisateurs, la recherche d'une relaxation qui retournera des réponses satisfaisantes. Pour cette raison, certains travaux proposent d'automatiser cette tâche. Dans le chapitre 3 nous avons ainsi présenté l'approche de Huang et al. [243, 17] qui consiste à ordonner ces requêtes relaxées en utilisant une mesure de similarité, puis à les exécuter dans cet ordre jusqu'à trouver le nombre de réponses désirées par l'utilisateur. Dans le cas des requêtes qui échouent, le problème de cette approche est qu'elle relaxe une requête sans connaître les raisons de l'échec de cette requête, c'est-à-dire sans identifier les MFS de cette requête. Tant que toutes les MFS n'auront pas été réparées, les requêtes relaxées retourneront toujours un ensemble vide de réponse. Ainsi, cette approche risque d'exécuter un grand nombre de requêtes relaxées avant de retourner une réponse. Aussi, nous avons développé de nouvelles stratégies de relaxation qui sont guidées par les MFS de la requête. Ces stratégies sont présentées dans le prochain chapitre.

Chapitre 6

Stratégies de relaxation guidées par les causes d'échec

Sommaire

Résumé : Les opérateurs de relaxation produisent un nombre important de requêtes relaxées qui sont ensuite exécutées de la plus similaire à la requête utilisateur à la moins similaire : c'est le principe habituel des processus existants de relaxation. L'inconvénient de cette approche est qu'il arrive qu'un grand nombre de requêtes relaxées échouent avant de trouver des réponses pour les utilisateurs. Nous avons constaté que ces échecs surviennent lorsque ces requêtes relaxées contiennent certaines MFS de la requête initiale. Dans ce chapitre, nous développons des stratégies de relaxation qui utilisent les MFS de la requête initiale pour *guider* le processus de relaxation. En effet, nous démontrons que la relaxation d'une requête qui échoue ne pourra retourner des réponses alternatives que si toutes les MFS sont réparées. Donc, parmi les requêtes relaxées les plus similaires, il serait plus efficace de ne considérer que celles qui relaxent toutes les MFS. Ces stratégies nous permettent ainsi d'éviter d'exécuter des requêtes relaxées non pertinentes (c'est-à-dire, qui échouent) à la différence des processus existants qui exécutent toutes les relaxations dans le pire des cas. Expérimentalement, nous montrons dans ce chapitre que les

stratégies que nous proposons permettent de retourner plus rapidement des réponses alternatives aux utilisateurs.

Introduction

Dans le chapitre 4, nous avons présenté un ensemble d'opérateurs de relaxation qui relaxent une partie de la requête initiale et génèrent une nouvelle requête, dite relaxée. Sachant que la relaxation peut être appliquée sur différentes parties de la requête initiale et par différents opérateurs de relaxation, il est possible de générer un très grand nombre de requêtes relaxées. Il se pose alors la question de l'exploration de cet ensemble de requêtes relaxées et le choix de celle qui sera exécutée pour retourner des réponses alternatives à un instant donné.

Au regard du nombre important de requêtes relaxées possibles, le choix de la requête relaxée à exécuter est une tâche très fastidieuse pour des utilisateurs finaux. Des processus de relaxation automatiques, permettant de répondre à ce problème, ont été proposés. Ils utilisent des mesures de similarité entre la requête initiale et les requêtes relaxées pour déterminer l'ordre d'exécution des requêtes relaxées générées. Par exemple, Huang et al. [243, 17] proposent l'algorithme *Best-First Search* (BFS) qui génère toutes les relaxations possibles d'une requête utilisateur et les exécute ensuite par ordre de similarité, de la plus similaire à la moins similaire. BFS génère les requêtes relaxées de façon itérative, pour chaque type de relaxation considéré. Il les exécute ensuite dans l'ordre de similarité jusqu'à l'obtention d'au moins K meilleures réponses alternatives où K est fixé par l'utilisateur. Ce processus de relaxation de Huang et al. [17] est particulièrement adapté pour des requêtes utilisateurs qui retournent déjà des réponses mais dont le nombre est insuffisant pour les utilisateurs (requêtes à réponses insuffisantes). En effet, ce processus retournera des réponses pour chaque requête relaxée exécutée, vu que toutes les relaxations retourneront des réponses alternatives puisque ce sont des relaxations d'une requête initiale qui retourne des réponses. Par contre, dans le cas des requêtes qui échouent, l'algorithme BFS risque d'exécuter également des requêtes relaxées qui échouent car contenant des MFS de la requête utilisateur. En conséquence, le coût du processus de relaxation peut être très important, comme nous le montrerons expérimentalement.

La relaxation d'une requête qui échoue ne pourra retourner des réponses alternatives que si elle ne contient plus aucune MFS de la requête initiale. En effet, nous avons montré dans le chapitre 5 que toute requête qui contient une MFS échoue. Dans les approches guidées uniquement par la similarité, la seule condition vérifiée est la similarité maximale de la requête relaxée à exécuter avec la requête initiale. Cette condition est insuffisante pour déterminer si la requête relaxée échouera ou pas. En effet, il arrive souvent que la relaxation opérée ne relaxe pas toutes les MFS et donc que la requête relaxée obtenue échoue, puisqu'elle contient au moins une MFS. Dans ce cas, il n'est pas nécessaire de l'exécuter même si elle est la plus similaire à la requête initiale. Pour éviter ces exécutions inutiles, nous proposons dans ce chapitre des stratégies de relaxation qui optimisent l'exploration des requêtes relaxées en se basant à la fois sur leur similarité avec la requête initiale et sur les MFS de ces requêtes. Nous proposons ainsi une stratégie basée sur la MFS qui vérifie si la requête la plus similaire à la requête initiale relaxe toutes les MFS. Si c'est le cas, elle est exécutée ; sinon, elle ne l'est pas et nous passons à la prochaine requête relaxée la plus similaire. Nous développons également d'autres stratégies qui s'appuient non seulement sur les MFS de la requête initiale mais aussi sur celles des requêtes relaxées qui échouent malgré la relaxation de toutes les MFS de la requête initiale.

Afin de permettre l'évaluation des stratégies que nous proposons et leur comparaison au processus de relaxation existant BFS, nous avons réutilisé les règles de relaxation et les mesures de similarités utilisées par Huang et al. [17]. Dans la section 6.1, nous présentons ce modèle de relaxation. Ensuite, la section 6.2 décrit les structures de données utilisées pour le processus de relaxation. Dans la section 6.3, nous développons la stratégie de relaxation basée sur les MFS de la requête initiale nommée *MFS-Based Search* (MBS). L'exploitation des propriétés des MFS nous a permis de développer, dans la section 6.4,

des optimisations de la stratégie MBS. La section 6.5 clôture ce chapitre en décrivant notre implémentation des stratégies de relaxation proposées, l'environnement d'expérimentation utilisé et l'analyse des résultats expérimentaux obtenus.

6.1 Modèle de relaxation considéré

Dans cette section, nous présentons le modèle de relaxation utilisé par Huang et al. [17]. Nous avons choisi d'utiliser ce modèle à la place des opérateurs de relaxation que nous avons présentés au chapitre 4 afin de pouvoir comparer nos contributions à celles déjà existantes, notamment celle de Huang et al. [17]. Pour présenter ce modèle de relaxation, nous utilisons les définitions, les fonctions et les notations présentées dans les préliminaires du chapitre précédent.

6.1.1 Règles de relaxation

Huang et al. [17] définissent *une règle de relaxation* comme une transformation d'un patron de triplet en un nouveau patron de triplet, dit relaxé. Nous considérons les trois règles de relaxation suivantes.

Règle de relaxation de classe (R1) : elle relaxe un patron de triplet en transformant une classe par une de ses super-classes. Elle est formellement décrite par la règle suivante, dans laquelle sc représente la propriété RDFS $subClassOf$: $(s,\ type,\ c_1) \Rightarrow (s,\ type,\ c_2)$ si $(c_1,\ sc,\ c_2)$.

Règle de relaxation de propriété (R2) : cette règle transforme un patron de triplet en remplaçant une propriété par une de ses super-propriétés. Elle est formalisée de la façon suivante : $(s,\ p_1,\ o) \Rightarrow (s,\ p_2,\ o)$ si $(p_1,\ sp,\ p_2)$, où sp représente la propriété RDFS $subPropertyOf$.

Règle de supression de constante (R3) : cette règle transforme un patron de triplet t en un autre patron de triplet t', $t \Rightarrow t'$. Cette transformation se fait par le changement d'une constante du patron de triplet t en une nouvelle variable v qui n'appartient ni au patron de triplet, $v \notin var(t)$, ni à la requête, $v \notin var(Q)$.

Nous remarquons que les règles R1 et R2 sont des cas particuliers de notre opérateur `GEN` sur les classes et les propriétés. Nous définissons maintenant les mesures de similarité associées à ces règles.

6.1.2 Mesures de similarité

Considérons deux patrons de triplet $t = (s, p, o)$ et $t' = (s', p', o')$ tel que $t' = R(t)$ où R est l'une des règles R1, R2 et R3 appliquée à t. La similarité entre le patron de triplet t et sa relaxation t' est définie comme suit [17] :

$$Sim(t, t') = \frac{1}{3} * Sim(s, s') + \frac{1}{3} * Sim(p, p') + \frac{1}{3} * Sim(o, o') \tag{6.1}$$

La similarité entre les sujets, les prédicats ou les objets dépend de la règle de relaxation utilisée dans $t' = R(t)$ pour obtenir t' en fonction de t. Suivant la règle de relaxation utilisée, la similarité entre ces éléments se calcule comme suit :

R1 : pour une super-classe c' de c, nous avons $Sim(c, c') = \dfrac{IC(c')}{IC(c)}$ où $IC(c) = -log(Pr(c))$ et $Pr(c) = \dfrac{|c|}{|\mathbb{D}|}$; $|c|$ est le nombre d'instances de la classe c et $|\mathbb{D}|$ est le nombre d'instances dans dans la BD-RDF $\mathbb{D}$.

R2 : pour une super-propriété p' de p, nous avons $Sim(p,p') = \frac{IC(p')}{IC(p)}$ où $IC(p) = -log(Pr(p))$ et $Pr(p) = \frac{|Triplets(p)|}{|Triplets(\mathbb{D})|}$; $|Triplets(p)|$ est le nombre de triplets contenant p comme prédicat dans $\mathbb{D}$ et $|Triplets(\mathbb{D})|$ est le nombre de triplets contenus dans la BD-RDF $\mathbb{D}$.

R3 : Pour une constante $const$ remplacée par une variable v nous avons : $Sim(const, v) = 0$.

Ainsi, pour chaque règle appliquée sur un patron de triplet, nous savons calculer la similarité entre le patron de triplet initial et le patron de triplet relaxé. La notation $t \prec t'$ décrit de façon formelle que t' est obtenu par relaxation de t en utilisant l'une des règles $R1$, $R2$ et $R3$.

Dans la figure 6.1, nous donnons un exemple de relaxation de requête sur les données de la figure 6.1(a). Dans cette exemple, le patron de triplet t_3 de la requête Q, figure 6.1(b), est relaxé avec la règle R3 pour donner le patron de triplet relaxé $t_3^{(1)}$ de la requête relaxée Q', figure 6.1(c). $t_3^{(1)}$ signifie que le patron de triplet t_3 a été relaxé une fois.

Triples		
subject	**predicate**	**object**
s_1	type	Lecturer
s_1	teacherOf	SW
s_1	age	45
s_2	type	Lecturer
s_2	nationality	US
s_2	age	46
s_3	type	FullProfessor
s_3	teacherOf	DB
s_3	age	46

(a) Triplets RDF

```
SELECT ?p ?n WHERE {
?p type Lecturer      (t1)
?p nationality ?n     (t2)
?p teacherOf "SW"     (t3)
?p age 46 }           (t4)
```

(b) Requête Q

```
SELECT ?p ?n WHERE {
?p type Lecturer      (t1)
?p nationality ?n     (t2)
?p teacherOf ?c       (t3(1))
?p age 46 }           (t4)
```

(c) Requête relaxée Q'

FIGURE 6.1 – Exemple de relaxation de la requête Q

Considérons maintenant une requête $Q = t_1 \wedge \cdots \wedge t_n$ et sa relaxation $Q' = t'_1 \wedge \cdots \wedge t'_n$. La similarité $Sim(Q, Q')$ entre Q et Q' est calculée avec la formule :

$$Sim(Q, Q') = \prod_{i=1}^{n} Sim(t_i, t'_i) \tag{6.2}$$

Nous disons que la requête relaxée Q' est obtenue par relaxation de Q en utilisant l'une des règles $R1$, $R2$ et $R3$ et nous notons $Q \prec Q'$, si et seulement si :

— pour chaque patron de triplet t_i, $t_i \preceq t_{i'}$ ($t_i = t_{i'}$ ou $t_i \prec t_{i'}$) ;

— pour au moins un patron de triplet t_j, $t_j \prec t_{j'}$.

Par exemple, dans la figure 6.1, la requête Q' est une requête relaxée de Q : $Q \prec Q'$.

Une réponse μ de Q' est dite *alternative ou approximative* si et seulement si $\mu \in [[Q']]_{\mathbb{D}}$ et $\mu \notin [[Q]]_{\mathbb{D}}$. Une réponse alternative μ peut être retournée par plusieurs requêtes relaxées de Q. Ainsi, il peut exister deux requêtes relaxées de Q notées Q' et Q" telles que $\mu \in [[Q']]_{\mathbb{D}} \wedge \mu \in [[Q"]]_{\mathbb{D}}$. Pour classer une réponse alternative μ, nous utilisons la requête relaxée qui retourne μ et qui possède la plus grande similarité avec la requête initiale. Nous définissons ainsi le score de μ par rapport à Q comme $Score(\mu, Q) = \{max(Sim(Q, Q'))|Q \prec Q' \wedge \mu \in [[Q']]_D\}$ pour classer la réponse alternative μ parmi toutes les réponses alternatives de Q.

6.2 Structures de données utilisées pour la relaxation

Dans le modèle de relaxation que nous avons présenté dans la section précédente, nous constatons que la relaxation d'un patron de triplet peut générer plusieurs patrons de triplet relaxés. Afin de pouvoir les traiter, ces patrons de triplet relaxés sont gérés via des structures de données que nous présentons dans cette section. Il en est de même pour les requêtes relaxées.

6.2.1 Relaxation des patrons de triplet

Pour tout patron de triplet t, nous pouvons appliquer l'une des règles $R1$, $R2$ et $R3$, et même plusieurs de ces règles, sur un patron de triplet t. De plus, ce processus de relaxation est récursif, c'est-à-dire qu'une relaxation de t peut être relaxée à son tour selon le même processus. Toutes les relaxations de t sont stockées dans une liste ordonnée par similarité décroissante par rapport au patron de triplet initial t. En tête de cette liste, nous trouvons le patron de triplet initial noté $t^{(0)}$. Nous notons $t^{(i)}$ la i^{ieme} meilleure relaxation de t dans la liste des patrons de triplet relaxés. Par conséquent, pour deux relaxations $t^{(i)}$ et $t^{(j)}$ de t, si $i < j$ alors $Sim(t^{(i)}, t) \geq Sim(t^{(j)}, t)$. Nous notons $nbRel(t)$ le nombre de tous les patrons de triplet relaxés à partir de t.

Exemple 1. Dans la représentation de l'ontologie LUBM schématisée dans le chapitre 1, nous pouvons observer que la classe $FullProfessor$ est une sous-classe de $Professor$. L'application récursive, des règles $R1$, $R2$ et $R3$ sur le patron de triplet $(?X,\ type,\ FullProfessor)$ donne le graphe de la figure 6.2. Sur ce graphe nous constatons que $\forall i, j; i < j \Rightarrow Sim(t^{(i)}, t) \geq Sim(t^{(j)}, t)$. Prenons la requête relaxée $t^{(1)} = (?X,\ type,\ Professor)$ avec $Sim(t^{(1)}, t) = 0.9$ et la relaxation $t^{(2)} = (?X,\ ?Y,\ FullProfessor)$ avec $Sim(t^{(2)}, t) = \frac{2}{3}$ nous avons bien $Sim(t^{(1)}, t) \geq Sim(t^{(2)}, t)$.

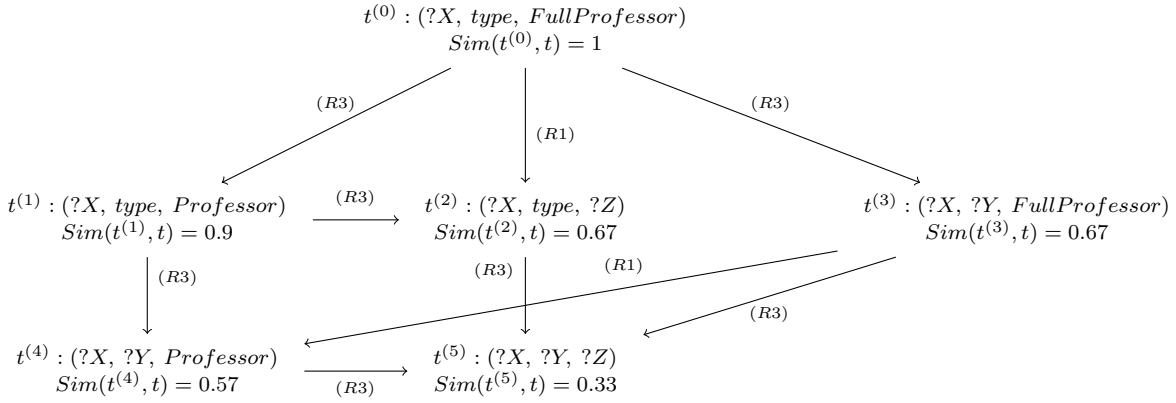


FIGURE 6.2 – Relaxation d'un patron de triplet

.

Précisons que la relation $Sim(t^{(i)}, t) \geq Sim(t^{(j)}, t)$ si elle implique que $i \leq j$, elle n'implique pas forcement que $t^{(i)} \prec t^{(j)}$. Par exemple, sur la figure 6.2, considérons les patrons de triplet de $t^{(2)}$ et $t^{(3)}$. Pour ces patrons de triplet nous avons bien $Sim(t^{(2)}, t) \geq Sim(t^{(3)}, t)$ mais pas $t^{(2)} \prec t^{(3)}$.

Soit t un patron de triplet et $ApplyRules(t)$ une fonction qui retourne tous les patrons de triplet relaxées de t résultants de l'application des trois règles de relaxation $R1$, $R2$ et $R3$ considérées. L'algorithme 5 calcule les patrons de triplet relaxés de t ordonnés par similarité avec t. La figure 6.3 montre un exemple d'exécution de cet algorithme sur le patron de triplet $(?X,\ type,\ FullProfessor)$ en utilisant les notations $t^{(i)}$ utilisées dans la figure 6.2.

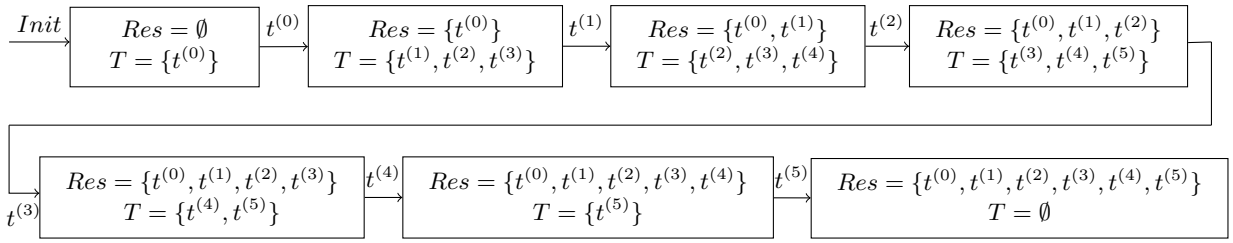


FIGURE 6.3 – Exemple d'exécution de l'algorithme 5 sur un patron de triplet t

Initialement, le résultat est vide et la liste T contient le patron de triplet initial $t^{(0)}$. Dans l'itération suivante, celui-ci est retiré de T et ajouté à Res. Les patrons de triplet résultants de la fonction $ApplyRules$ appliquée sur $t^{(0)}$, c'est-à-dire $t^{(1)}$, $t^{(2)}$ et $t^{(3)}$ sont ajoutés à T. Ils ont automatiquement triés par ordre de similarité décroissant vu la structure de données utilisée (file par priorité, *priority queue*). L'algorithme poursuit en retirant $t^{(1)}$ de T et en l'ajoutant dans Res. Le nouveau patron de triplet généré à partir de $t^{(1)}$, c'est-à-dire $t^{(4)}$ ($t^{(2)}$ est déjà dans T) est ajouté à T. L'algorithme poursuit ainsi jusqu'à ce que $T = \emptyset$, retournant alors le résultat $Res = \{t^{(0)}, t^{(1)}, t^{(2)}, t^{(3)}, t^{(4)}, t^{(5)}\}$.

Algorithm 5: Calcul des relaxations de t ordonnées par similarité

Relax(t)
input : Un patron de triplet t ;
output: Liste des patrons de triplet relaxés de t : $t^{(0)} \cdots t^{(n)}$;
1 $T \leftarrow \emptyset$; $Res \leftarrow \emptyset$; `// Res: liste des` $t^{(i)}$ `ordonnée par similarité décroissante`
2 $T.enqueue(t)$; `// T: file de priorité des` $t^{(i)}$`, ordonnée par similarité décroissante`
3 **while** $T \neq \emptyset$ **do**
4 $t_i = T.dequeue()$;
5 $Res.enqueue(t_i)$;
6 **foreach** *patron de triplet* $t_j \in ApplyRules(t_i)$ **do**
7 **if** $t_j \notin T$ **then**
8 $T.enqueue(t_j)$;
9 **return** Res;

6.2.2 Relaxation des patrons de graphes

Maintenant que nous avons présenté les structures de données considérées pour la relaxation des patrons de triplet, nous considérons dans cette section le cas des patrons de graphe. Considérons une requête initiale $Q = t_1^{(0)} \wedge \cdots \wedge t_n^{(0)}$. Les requêtes relaxées de Q sont exprimées de la manière suivante :
$Q_i = t_1^{(i_1)} \wedge \cdots \wedge t_n^{(i_n)}$ où il existe au moins un entier $i_k \in [i_1, i_n]$ tel que $i_k > 0$, c'est-à-dire Q_i doit relaxer au moins un patron de triplet de Q. En nous inspirant des travaux de Huang et al. [17] [30], nous avons organisé l'ensemble des requêtes relaxées de Q sous forme d'un graphe appelé *graphe de relaxations*. Ce graphe est construit comme suit :

30. le graphe de relaxation que nous proposons n'est pas équivalent à celui proposé par Huang et al. [17]. En effet, un arc entre deux requêtes Q_1 et Q_2 ne signifie pas nécessairement que $Q_1 \preceq Q_2$. Cette propriété simplifie le calcul des fils d'un nœud du graphe.

— la requête initiale $Q_0 = t_1^{(0)} \wedge \cdots \wedge t_n^{(0)}$ est la racine du graphe de relaxation ;
— un nœud $Q_i = t_1^{(i_1)} \wedge \cdots \wedge t_n^{(i_n)}$ a pour fils toutes les requêtes $Q_j = t_1^{(j_1)} \wedge \cdots \wedge t_n^{(j_n)}$ tel qu'il existe un seul patron de triplet t_l avec $j_l = i_l + 1$, pour tous les autres patrons de triplet t_e, $e \neq l$, nous avons $j_e = i_e$. Ainsi, Q_i a au plus n nœuds fils et chacun de ces nœuds fils est obtenu par substitution d'un patron de triplet relaxé $t_k^{(i_k)}$ par la prochaine meilleure relaxation de t_k qui est $t_k^{(i_k+1)}$ si $i_k + 1 \leq nbRel(t_k)$.

Ainsi, nous avons $Sim(Q, Q_i) \geq Sim(Q, Q_j)$ pour toute requête Q_j dont il existe un arc de Q_i vers Q_j dans le graphe de relaxation. Cette propriété est assurée par le processus de construction du graphe.

Preuve : $Q_i = t_1^{(i_1)} \wedge \cdots \wedge t_n^{(i_n)} \Rightarrow Sim(Q, Q_i) = \prod_{k=1}^{n} Sim(t_k^{(0)}, t_k^{(i_k)})$, de même nous avons $Q_j = t_1^{(j_1)} \wedge \cdots \wedge t_n^{(j_n)} \Rightarrow Sim(Q, Q_j) = \prod_{k=1}^{n} Sim(t_k^{(0)}, t_k^{(j_k)})$. Il existe un arc de Q_i vers Q_j dans le graphe de relaxation : donc Q_j est un nœud fils de Q_i, par construction du graphe, il existe donc un unique patron de triplet t_l avec $j_l = i_l + 1$ et pour tout autre patron de triplet t_k nous avons $j_k = i_k$. Par conséquent : $Sim(Q, Q_j) = \prod_{k=1}^{n} Sim(t_k^{(0)}, t_k^{(j_k)}) = Sim(t_l^{(0)}, t_l^{(j_l)}) * \prod_{k=1,k\neq l}^{n} Sim(t_k^{(0)}, t_k^{(j_k)})$ et d'où $\frac{Sim(Q, Q_i)}{Sim(Q, Q_j)} = \frac{Sim(t_l^{(0)}, t_l^{(i_l)})}{Sim(t_l^{(0)}, t_l^{(j_l)})}$ car pour tout $t_k \neq t_l$, $j_k = i_k$ donc $Sim(t_k^{(0)}, t_k^{(i_k)}) = Sim(t_k^{(0)}, t_k^{(j_k)})$.

De plus, $j_l = i_l + 1$ donc $i_l < j_l$ ce qui implique $Sim(t_l^{(0)}, t_l^{(i_l)}) \geq Sim(t_l^{(0)}, t_l^{(j_l)})$ d'après la propriété sur la relaxation des patrons de triplet présentée précédemment. $Sim(t_l^{(0)}, t_l^{(i_l)}) \geq Sim(t_l^{(0)}, t_l^{(j_l)}) \Rightarrow \frac{Sim(t_l^{(0)}, t_l^{(i_l)})}{Sim(t_l^{(0)}, t_l^{(j_l)})} \geq 1 \Rightarrow \frac{Sim(Q, Q_i)}{Sim(Q, Q_j)} \geq 1$ et finalement $Sim(Q, Q_i) \geq Sim(Q, Q_j)$.

La figure 6.4 donne un exemple du graphe de relaxations pour la requête $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$ de l'exemple de la figure 6.1. Pour réduire la taille du graphe, nous avons considéré que pour tous les patrons de triplet t_i, $nbRel(t_i) = 1$.

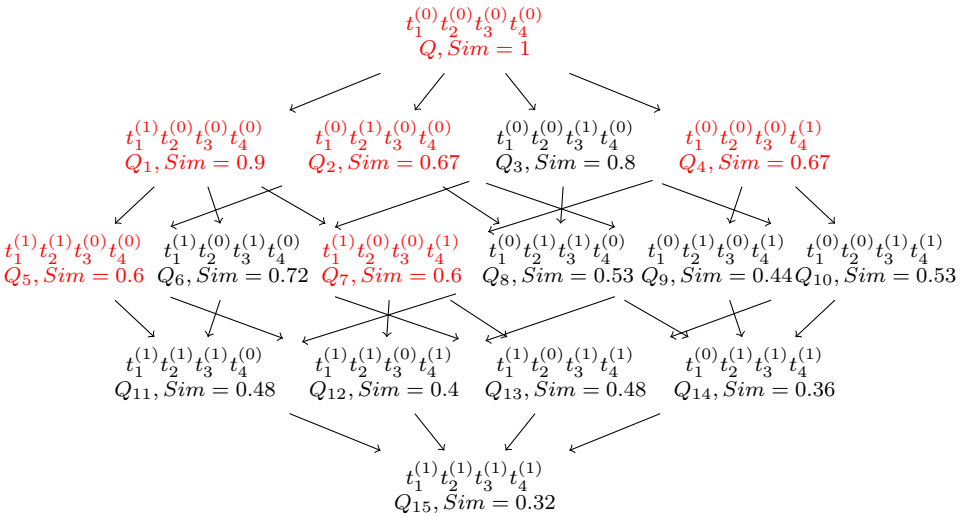


FIGURE 6.4 – Exemple de graphe de relaxations

Dans nos stratégies de relaxation présentées dans la suite, ce graphe est construit de façon incrémentale et représenté par une liste ordonnée de requêtes relaxées. Au moment de l'exécution de la meilleure relaxation, (c'est-à-dire la plus similaire), ses fils sont générés et insérés dans la liste suivant leur similarité par rapport à la requête initiale. Comme pour les patrons de triplet, une requête relaxée possède égale-

ment un niveau de relaxation qui est le nombre minimal de relaxation à opérer sur la requête initiale pour obtenir la requête relaxée.

Ce graphe possède différents niveaux de relaxation suivant la longueur des chemins de la racine à une requête relaxée. Au niveau de relaxation h nous retrouvons les requêtes relaxées $Q' = t_1^{(i_1)} \wedge \cdots \wedge t_n^{(i_n)}$ telles que $\sum_{k=1}^{n} i_k = h$. Notons également que le nombre de relaxation possible de $Q_0 = t_1^{(0)} \wedge \cdots \wedge t_n^{(0)}$, c'est-à-dire le nombre de nœuds du graphe de relaxation, est : $\prod_{i=1}^{n}(nbRel(t_i) + 1)$.

Dans le chapitre 3, nous avons présenté l'algorithme BFS proposé par Huang et al. [17] qui parcourt les requêtes relaxées en allant de la plus à la moins similaire à la requête initiale, jusqu'à l'obtention des K meilleures (TOP-K) réponses alternatives. Nous avons également vu que Huang et al. [17] s'appuient sur un graphe de relaxations dont le notre est inspiré, ayant la même taille mais pas les mêmes propriétés. L'application de l'algorithme BFS sur l'exemple du graphe de la figure 6.4 exécute les requêtes relaxées dans l'ordre suivant : Q_1, Q_3, Q_6, Q_2, Q_4, Q_5, Q_7, Q_8, Q_{10}, Q_{11}, Q_{13}, Q_9, Q_{12}, Q_{14}, Q_{15}. Dans le meilleur des cas, l'exécution de la première relaxation permet d'obtenir les TOP-K réponses alternatives. Dans le pire des cas, l'algorithme doit exécuter toutes les relaxations avant d'obtenir éventuellement les TOP-K réponses alternatives. De plus, dans cet algorithme, certaines requêtes relaxées peuvent également échouer alors que cet échec aurait pu être identifié grâce au MFS. Dans la section suivante, nous présentons notre stratégie de relaxation qui exploite ainsi les MFS d'une requête pour optimiser son processus de relaxation.

6.3 Stratégie de relaxation basée sur les MFS : MBS

La stratégie de relaxation *MFS-Based Search* (MBS) utilise les MFS d'une requête SPARQL qui échoue afin de déterminer les requêtes relaxées qu'il est inutile d'exécuter parce qu'elles échouent forcement. En effet, la relaxation d'une requête qui échoue peut générer des requêtes relaxées qui continuent d'échouer. Si nous identifions ces requêtes relaxées qui échouent, alors nous pourrons les ignorer et ainsi accélérer le processus de relaxation. Pour identifier les requêtes relaxées qui échouent nous utilisons la proposition suivante, conséquence de la définition des MFS :

Proposition 7

Soit Q_i une requête relaxée de Q. Si Q_i ne relaxe pas toutes les MFS de Q alors Q_i échoue.

Preuve : Supposons qu'il existe une MFS de Q, notée Q^*, dont aucun patron de triplet n'a été relaxé dans la relaxation Q_i. Nous avons par conséquent $Q^* \subset Q_i$. Or, d'après la définition d'une MFS, toute requête contenant une MFS échoue, donc Q_i échoue.

La présence d'une MFS de la requête initiale dans une requête relaxée suffit donc à être sûr que cette requête relaxée échouera à son tour. Ainsi, lors de l'exploration du graphe de relaxations, comme celui de la figure 6.4, nous vérifions si toutes les MFS de Q ont été relaxées au moins une fois avant d'exécuter une requête relaxée Q_i ; sinon, elle n'est pas exécutée. Cette stratégie est décrite par l'algorithme 6.

L'algorithme 6 utilise la liste ordonnée par similarité RQ pour classer les requêtes relaxées Q_i qui seront exécutées durant le processus de relaxation. La première requête insérée dans la liste est la requête initiale Q avec pour similarité 1 (ligne 6). Cette requête est marquée comme *échouée* à son insertion dans RQ (ligne 4), il n'est donc pas nécessaire de l'exécuter. Ensuite, l'algorithme 6 parcourt les requêtes relaxées Q_i de Q dans RQ en commençant par la plus similaire par rapport à la requête initiale (ligne 5). Pour chaque relaxation Q_i, si elle échoue (ligne 7), (c'est-à-dire qu'elle est marquée comme échouée), alors

il n'est pas nécessaire de l'exécuter. Sinon, elle est exécutée et les réponses alternatives retournées sont ajoutées à l'ensemble Res des réponses déjà trouvées (ligne 8). Une fois cette étape passée, les nœuds fils de la requête relaxée Q_i traitée, exécutée ou pas, sont calculées (ligne 9) en substituant itérativement chaque patron de triplet $t_k^{(i_k)} \in Q_i$ par sa relaxation suivante, c'est-à-dire $t_k^{(i_k+1)}$. Pour chaque nœud fils ainsi calculé nous vérifions s'il n'a pas déjà été exploré, c'est-à-dire marqué comme *inséré* (ligne 11), si ce n'est pas le cas, il est inséré dans la liste RQ et marqué comme inséré (lignes 12 et 13). Ensuite nous vérifions si le nœud fils contient au moins une MFS de Q (ligne 14). Si tel est le cas, il échouera et est donc marqué comme échoué (ligne 15). Ainsi, la requête relaxée correspondant au nœud fils ne sera pas exécutée dans les prochaines itérations mais sera juste utilisée pour calculer d'autres requêtes relaxées. La stratégie MBS optimise ainsi la relaxation par réduction du nombre de requêtes exécutées.

Algorithm 6: Stratégie de relaxation basée sur les MFS : MBS

Relax(Q**,** $mfs(Q)$**,** $\mathbb{D}$**,** k**)**
- **inputs :** Q : requête qui échoue ;
  $mfs(Q)$: ensemble des MFS de Q ;
  $\mathbb{D}$: BD-RDF ;
  k : nombre de réponses attendues ;
- **output:** Res : ensemble des TOP-k réponses alternatives de la requête Q;

1 $Res \leftarrow \emptyset$;
2 $RQ \leftarrow \emptyset$; `// Requêtes relaxées ordonnées par similarité`
3 $RQ.enqueue(Q)$;
4 marqué Q comme échouée ;
5 **while** $RQ \neq \emptyset \wedge |Res| < k$ **do**
6 $Q_i = RQ.dequeue()$;
7 **if** Q_i *n'est pas marquée comme échouée* **then**
8 $Res \leftarrow Res \cup [[Q_i]]_{\mathbb{D}}$;
9 **foreach** *patron de triplet* $t_k^{(i_k)} \in Q_i$ *tel que* $i_k < nbRel(t_k)$ **do**
10 $Q_c \leftarrow t_1^{(i_1)} \wedge \cdots \wedge t_k^{(i_{k+1})} \wedge \cdots \wedge t_n^{(i_n)}$; `// un nœud fils de` Q_i
11 **if** Q_c *n'est pas marquée comme insérée* **then** `// donc pas explorée`
12 $RQ.enqueue(Q_c)$;
13 marqué Q_c comme insérée;
14 **if** $\exists Q^* \in mfs(Q)$ *tel que* $Q^* \subseteq Q_c$ **then**
15 marqué Q_c comme échouée;
16 **return** Res;

Exemple 2. Prenons l'exemple de la requête $Q = t_1 \wedge t_2 \wedge t_3 \wedge t_4$ de la figure 6.5(b). La figure 6.5(c) montre les deux MFS de cette requête pour le jeu données de la figure 6.5(a), il s'agit de $Q^* = t_2 \wedge t_3$ et $Q^{**} = t_3 \wedge t_4$. Lorsque nous construisons le graphe de relaxations de cette requête en considérant que chaque patron de triplet ne peut être relaxé qu'une seule fois, nous obtenons le graphe de la figure 6.4. Dans ce graphe les requêtes relaxées en rouge sont celles qui échouent forcément car elles contiennent Q^* ou Q^{**}. Comme illustration, considérons la requête relaxée Q_2. La relaxation de t_2 en $t_2^{(1)}$ a permis de relaxer la MFS Q^*. Par contre, la MFS $Q^{**} = t_3 \wedge t_4$ est restée, quant à elle, inchangée dans Q_2. En conséquence, elle sera responsable de l'échec de Q_2 qui ne sera pas exécutée par MBS. De la même manière, la stratégie MBS permet d'ignorer les requêtes Q_1, Q_2, Q_4, Q_5, Q_7 pendant le

Triples		
subject	**predicate**	**object**
s_1	type	Lecturer
s_1	teacherOf	SW
s_1	age	45
s_2	type	Lecturer
s_2	nationality	US
s_2	age	46
s_3	type	FullProfessor
s_3	teacherOf	DB
s_3	age	46

(a) Triplets RDF

```
SELECT ?p ?n WHERE {
?p type Lecturer      (t1)
?p nationality ?n     (t2)
?p teacherOf "SW"     (t3)
?p age 46 }           (t4)
```

(b) Requête Q

$mfs(Q) = \{t_2 \wedge t_3, t_3 \wedge t_4\}$
$xss(Q) = \{t_1 \wedge t_2 \wedge t_4,\ t_1 \wedge t_3\}$

(c) MFS et XSS de Q

FIGURE 6.5 – Paramètre d'exécution de la stratégie de relaxation MBS

processus de relaxation de Q. MBS exécute donc, pour cet exemple, les requêtes relaxées suivantes : $Q_3, Q_6, Q_8, Q_9, Q_{10}, Q_{11}, Q_{13}, Q_{14}, Q_{12}, Q_{15}$.

6.4 Optimisation de l'algorithme MBS

Dans la stratégie MBS, une requête est exécutée si et seulement si elle relaxe toutes les MFS de la requête initiale. Une MFS est considérée comme relaxée si au moins un de ses patrons de triplet est relaxée. Mais, dans certains cas, cette condition est insuffisante. En effet, relaxer une requête n'est pas équivalent à *réparer* une requête. Dans nos travaux, *la relaxation est le relâchement des contraintes tandis que la réparation est une relaxation qui conduit à la découverte de réponses alternatives*. Donc, lorsque nous relaxons une requête, nous n'avons aucune garantie que cette relaxation sera suffisante pour être une réparation. Ce constat reste vrai même si toutes les MFS ont été relaxées. La relaxation d'une MFS ne garantit ainsi pas sa réparation. De manière générale, nous avons identifié trois possibilités pour qu'une requête relaxée échoue.

1. La requête relaxée contient au moins une MFS non relaxée de la requête initiale, donc cette MFS n'est pas réparée et cause l'échec de la requête relaxée. Nous gérons ce cas dans la stratégie MBS.
2. La requête relaxée relaxe toutes les MFS mais il existe au moins une MFS dont la relaxation n'est pas suffisante pour la réparer : la requête relaxée échoue. Nous gérons ce cas dans la stratégie O-MBS, section 6.4.1.
3. La requête relaxée échoue ayant néanmoins réparée toutes les MFS de la requête initiale. Il existe donc une ou plusieurs nouvelles MFS. Nous traitons ce cas dans la stratégie F-MBS, section 6.4.2.

6.4.1 Vérification de la réparation des MFS : O-MBS

La stratégie Optimized-MFS-Based Search (O-MBS), comme MBS, exécute les requêtes relaxées qui relaxent toutes les MFS de la requête initiale. Mais, pour celles qui échouent, O-MBS recherche, parmi les MFS relaxées, celles qui n'ont pas été réparées. Ces MFS relaxées, mais pas réparées, sont ajoutées à l'ensemble des MFS découvertes pendant le processus de relaxation et, désormais, toute autre requête relaxée contenant une de ces MFS ne sera pas exécutée. Ainsi O-MBS élague l'espace de recherche non seulement avec les MFS de la requête initiale mais aussi avec celles des requêtes relaxées qui échouent.

L'intuition sous-jacente à O-MBS est que les MFS de la requête initiale Q peuvent permettre de trouver les MFS d'une requête relaxée de Q. Formellement, considérons M_Q une MFS de Q ($M_Q \in mfs(Q)$),

et Q_i une requête relaxée de Q. Nous notons $M_Q^{\uparrow Q_i}$ la relaxation de M_Q, c'est-à-dire la sous-requête qui correspond à M_Q dans Q_i après la relaxation. Ainsi, si $M_Q = M_Q^{\uparrow Q_i}$ alors cela veut dire que M_Q n'a pas été relaxée. Par extension, nous notons $mfs^{\uparrow Q_i}(Q)$ l'ensemble des MFS relaxées de Q dans Q_i. Par exemple, pour $Q_0 = t_1^{(0)} \wedge t_2^{(0)} \wedge t_3^{(0)} \wedge t_4^{(0)}$ et sa requête relaxée $Q_3 = t_1^{(0)} \wedge t_2^{(0)} \wedge t_3^{(1)} \wedge t_4^{(0)}$ des figures 6.1 (b) et (c), nous avons $mfs(Q) = \{t_2 \wedge t_3, t_3 \wedge t_4\}$ (figure 6.5 (c)) et donc $mfs^{\uparrow Q_3}(Q) = \{t_2^{(0)} \wedge t_3^{(1)}, t_3^{(1)} \wedge t_4^{(0)}\}$. En exécutant la sous-requête $t_3^{(1)} \wedge t_4^{(0)}$ de Q_3, sur l'ensemble des données RDF de la figure 6.5(a), nous obtenons la réponse s_3, donc $t_3^{(1)} \wedge t_4^{(0)}$ n'échoue pas et n'est pas une MFS de Q_3 ($t_3^{(1)} \wedge t_4^{(0)} \notin mfs(Q_3)$). Par contre, les sous-requêtes $t_2^{(0)} \wedge t_3^{(1)}$ et $t_1^{(0)} \wedge t_3^{(1)} \wedge t_4^{(0)}$ échouent, par conséquent elles appartiennent à l'ensemble $mfs(Q_3)$.

L'exemple précédent montre que la relation suivante n'est pas toujours vraie : $mfs(Q_i) \subseteq mfs^{\uparrow Q_i}(Q)$. Cependant, comme nous le prouvons ci-dessous, chaque MFS de Q_i contient une requête de $mfs^{\uparrow Q_i}(Q)$.

Proposition 8

Pour toute MFS M_{Q_i} de Q_i il existe une MFS M_Q de Q telle que $M_Q^{\uparrow Q_i} \subseteq M_{Q_i}$.

Preuve : Considérons M_{Q_i} une MFS de Q_i. Par définition M_{Q_i} échoue donc $[[M_{Q_i}]] = \emptyset$. Considérons $M_{Q_i}^{\uparrow Q}$ qui est la sous-requête de Q correspondant à M_{Q_i} avant la relaxation. Étant donné que M_{Q_i} est une relaxation de $M_{Q_i}^{\uparrow Q}$ alors $[[M_{Q_i}^{\uparrow Q}]] \subseteq [[M_{Q_i}]]$. Puisque la relaxation élargit l'ensemble des réponses possibles et que $[[M_{Q_i}]] = \emptyset$, nous avons $[[M_{Q_i}^{\uparrow Q}]] = \emptyset$. Comme $M_{Q_i}^{\uparrow Q}$ échoue, elle contient une MFS M_Q de Q. Puisque $M_Q \subseteq M_{Q_i}^{\uparrow Q}$, nous avons $M_Q^{\uparrow Q_i} \subseteq M_{Q_i}$.

La propriété précédente montre que, si une MFS Q^* de Q a été réparée dans une requête relaxée Q_i, Q_i peut tout de même contenir une MFS qui inclut Q^*. Identifier cette MFS est nécessaire pour ne pas exécuter Q_i qui échoue forcement. Cependant, le nombre de requêtes qui incluent Q^* est d'un ordre exponentiel par rapport à la taille de la requête. C'est pourquoi O-MBS se contente d'identifier les MFS de Q qui n'ont pas été réparées dans Q_i. En effet, leur identification est simple puisqu'il suffit de les exécuter pour voir si elles échouent encore dans Q_i. Ainsi, O-MBS ne connaitra pas forcément toutes les MFS de Q_i, mais uniquement celles qui n'ont pas été réparées.

O-MBS étend l'algorithme MBS (algorithme 6) de la manière suivante (l'algorithme complet est fourni dans l'annexe B, algorithme 9). Pour chaque requête relaxée Q_i explorée dans le graphe de relaxation, chaque requête $M_{Q_i} \in mfs^{\uparrow Q_i}(Q)$ est exécutée. Si la requête M_{Q_i} échoue, M_{Q_i} est une MFS de M_{Q_i} et donc, toutes les requêtes qui contiennent M_{Q_i} peuvent être supprimées du graphe de relaxation (grâce à la propostion 8). Pour optimiser ce processus, les MFS découvertes de chaque requête Q_i explorée sont enregistrées. Elles sont notées $dmfs(Q_i)$. Lorsqu'une requête Q_1 est explorée, la stratégie O-MBS n'exécute que les MFS dans $dmfs(Q_0)$, où Q_0 est la dernière requête explorée telle que $Q_0 \prec Q_1$. En effet, il n'est pas nécessaire d'exécuter les requêtes MFS qui ont déjà été réparées précédemment par Q_0.

Reprenons l'exemple $Q_0 = t_1^{(0)} \wedge t_2^{(0)} \wedge t_3^{(0)} \wedge t_4^{(0)}$ de la figure 6.5 avec $mfs(Q) = \{t_2 \wedge t_3, t_3 \wedge t_4\}$ et son graphe de relaxation à la figure 6.6. Après la suppression des requêtes contenant une des MFS de Q (en rouge dans la figure 6.6), la requête la plus similaire est Q_3 qui relaxe les deux MFS, donc elle n'est pas marquée comme échouée. Cependant, son exécution retourne toujours un ensemble vide de réponse. La vérification des sous-requêtes de $mfs^{\uparrow Q_3}(Q)$ montre, comme nous l'avons indiqué précédemment, que Q_3 répare la MFS $t_3^{(0)} \wedge t_4^{(0)}$ de Q car $t_3^{(1)} \wedge t_4^{(0)}$ retourne s_3. Par contre, $t_2^{(0)} \wedge t_3^{(1)}$ échoue toujours et donc $dmfs(Q_3) = \{t_2^{(0)} \wedge t_3^{(1)}\}$. Cette nouvelle MFS permet de supprimer les requêtes Q_6, Q_{10} et

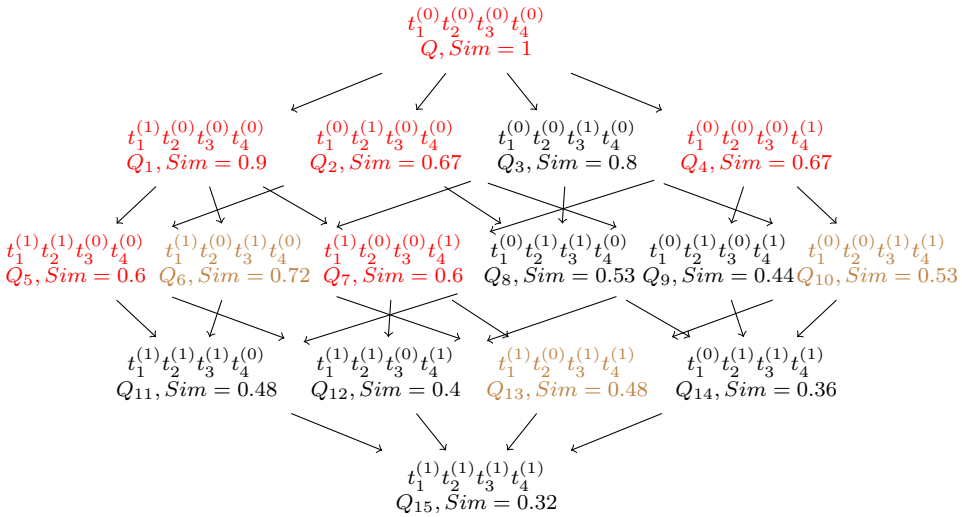


FIGURE 6.6 – Graphe de relaxation exploré avec O-MBS

Q_{13} (en marron dans la figure 6.6). Si aucune autres MFS n'est découverte par la suite, alors O-MBS exécutera les requêtes rélaxées dans l'ordre suivant : $Q_3, Q_8, Q_9, Q_{11}, Q_{14}, Q_{12}, Q_{15}$.

6.4.2 Recherche de toutes les MFS : Full-MBS

La stratégie Full-MBS (F-MBS), à la différence de O-MBS, recherche toutes les nouvelles MFS contenues dans une requête relaxée Q_i de Q qui échoue, et non pas seulement les MFS de la requête initiale qui ne sont pas réparées dans Q_i (comme O-MBS). F-MBS recherche ainsi dans Q_i toutes les nouvelles MFS, c'est-à-dire aussi bien celles qui n'ont pas été réparées que celles qui contiennent les MFS réparées. Pour être efficace, la stratégie F-MBS doit permettre d'obtenir un bon compromis entre le coût de recherche des nouvelles MFS et le nombre de requêtes relaxées que ces nouvelles MFS permettront d'ignorer. F-MBS est basée sur deux corollaires dérivés directement de la proposition 8.

Corollaire 1

> *Si toutes les requêtes* $M_Q^{\uparrow Q_i} \in mfs^{\uparrow Q_i}(Q)$ *échouent alors* $mfs(Q_i) = mfs^{\uparrow Q_i}(Q)$.

Preuve : si toutes les requêtes $M_Q^{\uparrow Q_i} \in mfs(Q_i)$ échouent, ceci veut dire qu'aucune MFS n'a été réparée par Q_i et elles sont donc aussi MFS de Q_i. Q_i ne peut pas avoir d'autres MFS car si une telle MFS existait, elle inclurait une des requêtes de $mfs^{\uparrow Q_i}(Q)$ (proposition 8) qui est une MFS (contradiction, une MFS doit être minimale).

Corollaire 2

> *Chaque MFS* M_{Q_i} *de* Q_i *contient les patrons de triplet qui sont partagés par les MFS relaxées* $mfs^{\uparrow Q_i}(Q)$.

Preuve : D'après la proposition 8, chaque MFS de Q_i contient au moins une MFS de Q projetée dans Q_i, $M_Q^{\uparrow Q_i} \subseteq M_{Q_i}$. Donc si les MFS de Q ont des patrons de triplet en commun, ces patrons de triplet appartiennent forcément à chaque MFS de Q_i (en les projetant sur chacune de ces MFS).

En s'appuyant sur ces deux corolaires, la stratégie de relaxation F-MBS étend O-MBS de la manière suivante (l'algorithme complet est donné en annexe B, algorithme 10). Pour chaque requête relaxée Q_i,

F-MBS exécute toutes les MFS de $mfs^{\uparrow Q_i}(Q_0)$, où Q_0 est la dernière requête relaxée explorée telle que $Q_0 \prec Q_i$. Si toutes ces requêtes échouent, alors $mfs(Q_i) = mfs^{\uparrow Q_i}(Q_0)$ (grâce au corrolaire 1). Sinon, F-MBS exécute une version optimisée de l'algorithme LBA (décrit au chapitre 5) pour trouver les MFS de Q_i. Comme dans les stratégies précédentes, les requêtes relaxées qui incluent au moins une des MFS identifiées de Q_i sont supprimées du graphe de relaxations. La version optimisée de LBA utilisée pour rechercher les MFS dans Q_i est l'algorithme 8. Il diffère de l'original, présentée au chapitre 5, en deux principaux points :

1. en considérant le corollaire 2, nous savons que toutes les MFS de Q_i contiennent les patrons de triplet Q_s partagés par les MFS de Q. L'algorithme FindAnMFS peut alors être remplacé par l'agorithme 7 qui prend en paramètre Q_s. Nous constatons que le coût de FindAnMFS n'est plus de $\mathcal{O}(n)$ mais de $\mathcal{O}(|Q - Q_s|)$ qui est dans le pire des cas égale à $\mathcal{O}(n)$ et dans le meilleur $\mathcal{O}(1)$;
2. au moment de la recherche des MFS dans l'instruction 6, l'ensemble $dmfs(Q_i)$ contient déjà des MFS trouvées pendant l'exécution de l'instruction 5. Nous recherchons donc les XSS potentielles de Q_i qui ne contiennent aucune MFS de $dmfs(Q_i)$ dans une première étape. Nous pouvons voir cet étape dans la version optimisée de LBA, algorithme 8 (ligne 3 à 6). Cet algorithme initialise préalablement $mfs(Q_i)$ avec $dmfs(Q_i)$ (ligne 1). Après ces deux étapes l'algorithme exécuté reste le même que l'original.

Dans le pire des cas, lorsqu'à la ligne 14 $dmfs(Q_i) = \emptyset$ (toutes les MFS relaxées ont été réparées) et lorsque toutes les MFS sont disjointes $Q_s = \emptyset$, l'exécution de l'algorithme LBA optimisé est identique à celle de l'original, donc le coût est le même. Dans le meilleur des cas, $dmfs(Q_i) = mfs^{\uparrow Q_i}(Q) - M_{Q_i}$ (une seule MFS a été réparée M_{Q_i}) et les MFS diffèrent d'un seul patron de triplet. Dans ce cas, l'algorithme LBA optimisé exécute une requête pour retrouver chaque nouveau patron de triplet de M_{Q_i}, ce qui fait au maximum n exécutions.

Si nous reprenons l'exemple précédent dans lequel la requête relaxée Q_3 relaxait la MFS $t_2^{(0)} \wedge t_3^{(1)}$ sans la réparer donc $dmfs(Q_3) = \{t_2^{(0)} \wedge t_3^{(1)}\}$. Q_3 réparait la MFS relaxée $t_3^{(1)} \wedge t_4^{(0)}$. Nous recherchons donc les nouvelles MFS avec l'algorithme 8 qui prend en paramètre $Q_3 = t_1^{(0)} \wedge t_2^{(0)} \wedge t_3^{(1)} \wedge t_4^{(0)}$, $dmfs(Q_3) = \{t_2^{(0)} \wedge t_3^{(1)}\}$, $Q_s = t_3^{(1)}$ et $\mathbb{D}$. La recherche des premières $pxss$ à partir de $dmfs(Q_3)$ nous donne $pxss(Q_3, t_2^{(0)} \wedge t_3^{(1)}) = \{t_1^{(0)} \wedge t_3^{(1)} \wedge t_4^{(0)}\ \ t_1^{(0)} \wedge t_2^{(0)} \wedge t_4^{(0)}\}$. Ensuite nous recherchons les MFS dans ces deux $pxss$. $t_1^{(0)} \wedge t_2^{(0)} \wedge t_4^{(0)}$ est une XSS de Q donc il ne contient pas de MFS. Par contre, $t_1^{(0)} \wedge t_3^{(1)} \wedge t_4^{(0)}$ échoue et aucune de ses sous-requêtes de la pxss $t_1^{(0)} \wedge t_3^{(1)} \wedge t_4^{(0)}$ échoue. Donc $t_1^{(0)} \wedge t_3^{(1)} \wedge t_4^{(0)}$ est une nouvelle MFS. Par conséquent $mfs(Q_3) = \{t_2^{(0)} \wedge t_3^{(1)}, t_1^{(0)} \wedge t_3^{(1)} \wedge t_4^{(0)}\}$. En supposant qu'aucune autre MFS ne sera trouvée par la suite, la stratégie F-MBS exécutera les requêtes relaxées dans l'ordre suivant : $Q_3, Q_9, Q_{11}, Q_{14}, Q_{12}, Q_{15}$. F-MBS aura ainsi permis la suppression de Q_8 par rapport à O-MBS.

6.5 Implémentation et expérimentations

Dans le but d'évaluer et de comparer nos trois stratégies à la stratégie existante de Huang et al. [17], nous avons implémenté ces quatre stratégies. Nous avons également conduit un ensemble d'expérimentations sur ces stratégies en relevant à chaque fois des indicateurs de performances et d'évaluation. Dans cette section, nous présentons l'environnement d'expérimentation, les indicateurs de performances utilisés et l'analyse des résultats obtenus.

Algorithm 7: Trouver une MFS d'une requete relaxée qui échoue

OptimizedFindAnMFS(Q_i, Q_s, $\mathbb{D}$)
inputs : Q_i : requête relaxée qui échoue ;
Q_s : sous-requête commune à toutes les MFSs relaxées ;
$\mathbb{D}$: BD RDF
output: ${Q_i}^*$: une MFS de Q_i
1 ${Q_i}^* \leftarrow Q_s$;
2 $Q' \leftarrow Q_i - Q_s$;
3 **foreach** *patron de triplets* $t_i \in Q_i - Q_s$ **do**
4 $Q' \leftarrow Q' - t_i$;
5 **if** $[[Q' \wedge {Q_i}^*]]_{\mathbb{D}} \neq \emptyset$ **then**
6 ${Q_i}^* \leftarrow {Q_i}^* \wedge t_i$;
7 **return** ${Q_i}^*$;

Algorithm 8: Trouver les MFS et XSS d'une requête relaxée Q_i qui échoue

OptimizedLBA(Q_i, $dmfs(Q_i)$, Q_s, $\mathbb{D}$)
inputs : Q_i : une requête relaxée qui échoue ;
$dmfs$: l'ensemble des MFSs de Q_i déjà trouvées ;
Q_s : sous-requête commune à toutes les MFSs relaxées ;
$\mathbb{D}$: BD RDF
outputs: $mfs(Q_i)$: MFS de Q_i
$xss(Q_i)$: XSS de Q_i
1 $mfs(Q_i) \leftarrow dmfs(Q_i)$;
2 $xss(Q_i) \leftarrow \emptyset$;
3 $pxss \leftarrow pxss(Q_i, mfs_1)$; `// pxss da la première MFS`
4 $mfs(Q_i) \leftarrow mfs(Q_i) - \{mfs_1\}$; `// suppression da la première MFS`
5 **foreach** ${Q_i}^* \in mfs(Q_i)$ **do** `// pxss de` ${Q_i}^*$
6 **foreach** $Q' \in pxss$ *telle que* ${Q_i}^* \subseteq Q'$ **do** `// évite de trouver encore` ${Q_i}^*$
7 $pxss \leftarrow pxss - \{Q'\}$;
8 $pxss \leftarrow pxss \cup \{Q_j \in pxss(Q', {Q_i}^*) \mid \nexists Q_k \in pxss \cup xss(Q_i) : Q_j \subseteq Q_k\}$;
9 **while** $pxss \neq \emptyset$ **do**
10 $Q' \leftarrow pxss.element()$; `// choisit un élement de` $pxss$
11 **if** $[[Q']]_{\mathbb{D}} \neq \emptyset$ **then** `//` Q' `est une XSS`
12 $xss(Q_i) \leftarrow xss(Q_i) \cup \{Q'\}$;
13 $pxss \leftarrow pxss - \{Q'\}$;
14 **else** `//` Q' `contient une MFS`
15 ${Q_i}^* \leftarrow OptimizedFindAnMFS(Q_i, Q_s, \mathbb{D})$;
16 $mfs(Q_i) \leftarrow mfs(Q_i) \cup \{{Q_i}^*\}$;
17 **foreach** $Q'' \in pxss$ *telle que* ${Q_i}^* \subseteq Q''$ **do** `// évite de trouver à nouveau` ${Q_i}^*$
18 $pxss \leftarrow pxss - \{Q''\}$;
19 $pxss \leftarrow pxss \cup \{Q_j \in pxss(Q'', {Q_i}^*) \mid \nexists Q_k \in pxss \cup xss(Q_i) : Q_j \subseteq Q_k\}$;
20 **return** $\{mfs(Q_i), xss(Q_i)\}$;

6.5.1 Implémentation et environnement d'expérimentation

Nous avons implémenté un prototype d'évaluation des algorithmes BFS, MBS, O-MBS et F-MBS. Ce prototype d'expérimentation a été implémenté en JAVA 1.8 64 bits au dessus des systèmes Jena TDB (version 3.0.0) et Virtuoso (version 7.2.1). Ce prototype prend comme paramètre d'entrée une requête SPARQL qui échoue et le nombre de réponses attendues par l'utilisateur. Les algorithmes MBS, O-MBS et Full-MBS utilisent le prototype de calcul des MFS d'une requête qui échoue présenté dans le chapitre 5. Notre prototype et la documentation liée à ces expérimentations sont disponibles à l'adresse : `http://www.lias-lab.fr/forge/projects/qars`.

L'environnement matériel et le protocole d'expérimentation sont les mêmes que dans le chapitre précédent. Nos expérimentations ont été réalisées sur un serveur Ubuntu 4.04.02 LTS 64 bits avec un CPU Intel XEON E5-2630 v3 @2.4Ghz et 32GO RAM. Les temps reportés sont une moyenne de 5 exécutions consécutives. Avant d'enregistrer la première mesure, l'algorithme est lancé une fois.

Comme dans le chapitre précédent, nous avons utilisé cinq jeux de données générés avec le banc d'essai LUBM [21]. Ces jeux de données, présentés dans le tableau 5.1 du chapitre 5, contiennent de 13 millions (LUBM100) à 130 millions (LUBM1K) de triplets RDF. Ces jeux de données ont été préalablement saturés, c'est-à-dire qu'ils contiennent les triplets générés avec le banc d'essai LUBM et les triplets inférés à l'aide des règles d'inférences RDFS. Afin de faciliter le calcul des mesures de similarité entre classes et propriétés, nous avons également calculé les statistiques liées aux données qu'ils contiennent pour chaque classe et chaque propriété, le nombre d'instances par classe et le nombre de triplets par propriété. Ces statistiques ont été stockées dans une structure de données en mémoire afin d'être utilisées durant le processus de relaxation suivant les différentes stratégies de relaxation évaluées. Nous avons modifié les requêtes utilisées dans [17] pour l'évaluation de la stratégie BFS afin qu'elles comprennent une variété de taille (entre 1 et 15 patrons de triplet) et de type (étoile, chaîne et composite). Les requêtes résultantes sont celles du tableau 6.1.

6.5.2 Évaluation des stratégies et analyse des résultats

Nous avons évalué les performances (en termes de temps de réponse) des algorithmes suivants avec des tailles croissantes des jeux de données.

- *Best-First Search (BFS)* : l'algorithme existant proposé par Huang et al. [17].
- *MFS-Based Search (MBS)* : l'algorithme qui n'exécute que des requêtes relaxées ne contenant aucune des MFS de la requête initiale.
- *Optimized-MBS (O-MBS)* : l'algorithme qui met à jour l'ensemble des MFS connues en vérifiant si les MFS relaxées sont réparées.
- *Full-MBS* (F-MBS) : l'algorithme qui met à jour l'ensemble des MFS connues avec toutes les MFS d'une requête relaxée qui échoue.

Sur ces algorithmes nous avons évalué les coûts en temps sur les différentes requêtes considérées qui recouvrent le spectre des requêtes rencontrées pour des données RDF réelles [14]. Nous avons également évalué l'impact de la BD-RDF afin de voir si les performances de certaines stratégies variaient anormalement en fonction de la BD-RDF. Enfin nous avons évalué l'impact de la taille des données sur les différentes stratégies.

6.5.2.1 Évaluation du temps d'exécution

Nous avons évalué les temps d'exécution des stratégies de relaxation MBS, O-MBS et F-MBS et nous les avons comparé à ceux de notre implémentation de la stratégie BFS. Nous présentons, dans cette section,

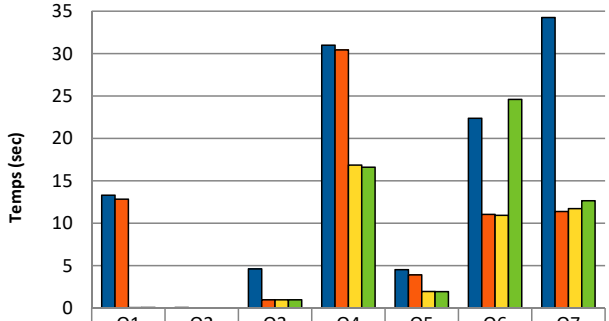
$Q1$ (1 MFS)	SELECT $*$ WHERE { $?X$ *rdf:type ub:FullProfessor* . $?X$ *title 'Dr'* . }
$Q2$ (3 MFS)	SELECT $*$ WHERE { *UndergraduateStudent33 advisor* $?Y1$. $?Y1$ *doctoralDegreeFrom* $?Y2$. $?Y2$ *hasAlumnus* $?Y3$. $?Y3$ *title* $?Y4$ }
$Q3$ (4 MFS)	SELECT $*$ WHERE { $?X$ *type FullProfessor* . $?X$ publicationAuthor $?Y1$. $?X$ worksFor $?Y2$. $?Y3$ advisor $?X$. $?X$ title $?Y4$ }
$Q4$ (3 MFS)	SELECT $*$ WHERE { $?X$ *type UndergraduateStudent* . $?X$ *memberOf* $?Y1$. $?X$ *mastersDegreeFrom University822* . $?X$ *emailAddress* $?Y2$. $?X$ *advisor FullProfessor0* . $?X$ *takesCourse* $?Y3$. $?X$ *name* $?Y4$ }
$Q5$ (3 MFS)	SELECT $*$ WHERE { $?X$ *type FullProfessor* . $?X$ *doctoralDegreeFrom* $?Y1$. $?X$ *memberOf* $?Y2$. $?X$ *headOf* $?Y1$. $?X$ *title* $?Y3$. $?X$ *officeNumber* $?Y4$. $?X$ *researchInterest* $?Y5$. $?Y6$ *advisor* $?X$. $?Y6$ *name* $?Y7$ }
$Q6$ (4 MFS)	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 Department2* . $?X$ *undergraduateDegreeFrom* $?Y9$ }
$Q7$ (4 MFS)	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* $?Y12$ }

TABLE 6.1 – Requêtes d'évaluation

les résultats obtenus sur le jeu de données LUBM100 stocké dans la BD-RDF JenaTDB. Le nombre de réponses alternatives attendues, passé en paramètre des différentes stratégies, est fixé à 50. Les figures 6.7 et 6.8 montrent respectivement les coûts en temps d'exécution et en nombre de requêtes exécutées des stratégies de relaxation pour chaque requête testée. Pour les requêtes basées sur les MFS, MBS, O-MBS et F-MBS, le temps d'exécution inclut le temps de calcul des MFS.

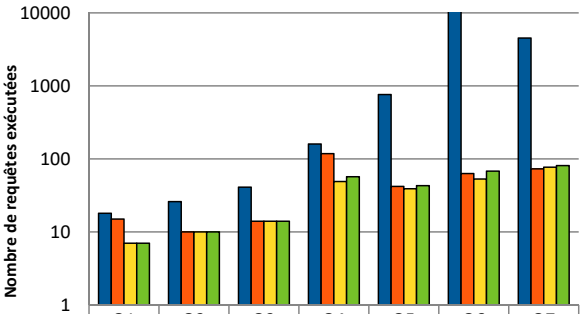
En étudiant les résultats obtenus, nous constatons que BFS exécute plus de requêtes relaxées que nos algorithmes basés sur les MFS. Cette différence croît avec le nombre de patrons de triplet de la requête, comme nous pouvons le voir sur la figure 6.8. Pour les requêtes 6 et 7 qui possèdent respectivement 12 et 15 patrons de triplet (tableau 6.1) et 4 MFS, BFS doit explorer un très grand nombre de requêtes avant de pouvoir enfin réparer les MFS, 13809 pour Q_6 et 4520 pour Q_7. Pour ces deux requêtes la différence entre le nombre de requêtes relaxées exécutées par BFS et par nos stratégies de relaxation est très grande, ce qui a pour conséquence une différence tout aussi grande entre les temps d'exécution. Cependant, cette proportionnalité n'est pas toujours respectée. Par exemple, pour la requête Q_2 où BFS exécute 26 requêtes tandis que nos stratégies n'en exécutent que 10, nous avons une différence entre les temps d'exécution très faible. Cette différence de temps d'exécution moins importante dans Q_2 est liée au

nombre de patrons de triplet de Q_2 qui est de 4, ce qui a pour conséquence un faible coût d'exécution des requêtes relaxées. Cependant, en moyenne, l'algorithme O-MBS est plus de deux fois plus performant que BFS (le temps moyen est de 18 secondes avec BFS contre 7 secondes pour O-MBS).



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
BFS	13,3	0,06	4,61	30,99	4,52	22,37	34,25
MBS	12,83	0,01	0,97	30,43	3,91	11,04	11,38
O-MBS	0,05	0,01	0,97	16,85	1,95	10,93	11,72
F-MBS	0,06	0,01	0,97	16,60	1,93	24,60	12,65

FIGURE 6.7 – Coût en temps sur Jena TDB



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
BFS	18	26	41	160	761	13809	4520
MBS	15	10	14	118	42	63	73
O-MBS	7	10	14	49	39	53	77
F-MBS	7	10	14	57	43	68	81

FIGURE 6.8 – Coût en nombre de requêtes

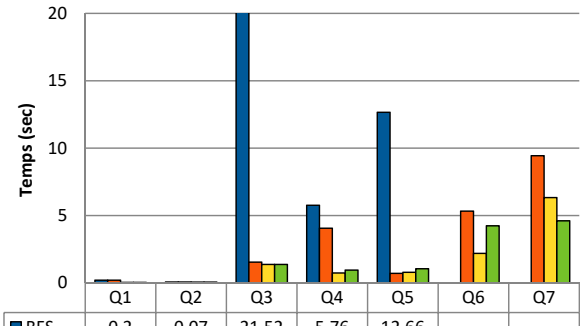
Considérons maintenant les stratégies que nous avons proposées. D'après les figures 6.7 et 6.8, il découle clairement que la stratégie O-MBS est plus performante que les stratégies MBS et F-MBS dans la majorité des cas. L'algorithme MBS rivalise avec O-MBS en exécutant le même nombre de requêtes relaxées pour Q_2 et Q_3 et en exécutant même moins de requêtes relaxées pour Q_7. Mais nous pouvons observer que, dans tous ces cas, la différence entre le temps d'exécution de MBS et O-MBS est très faible. Par contre pour les requêtes telles que Q_4, Q_5 et Q_6, O-MBS est plus performant que MBS et permet d'avoir des gains en temps allant d'une seconde à 15 secondes. Prenons la requête Q_1 pour illustrer la réduction des requêtes relaxées exécutées par O-MBS. $Q1$ possède une MFS : ($?X$ *title 'Dr'*) car la propriété *title* n'est pas utilisée dans le jeu de données. MBS et O-MBS commencent par relaxer ce patron de triplet en utilisant la suppression de constante et obtiennent ($?X$ *title* $?Y$). Ensuite les deux algorithmes diffèrent dans leur stratégie. O-MBS vérifie si cette relaxation répare la requête et trouve que ce n'est pas le cas ; donc ce patron de triplet devient la deuxième MFS découverte. Ainsi, toutes les requêtes relaxées qui contiendront ce patron de triplet ne seront pas exécutées, notamment toutes celles qui relaxent uniquement le patron de triplet ($?X$ *rdf:type ub:FullProfessor*). De ce son côté, vu que MBS utilise uniquement les MFS de la requête initiale $Q1$, ($?X$ *title 'Dr'*). Il exécute toutes les requêtes relaxées qui ne contiennent pas cette MFS, donc même celles qui contiennent le patron de triplet ($?X$ *title* $?Y$) qui causera leur échec. Par conséquent, MBS exécutera plus de requêtes que O-MBS et le temps d'exécution sera significativement plus long.

Concernant la stratégie F-MBS, elle exécute un nombre égal ou supérieur de requêtes par rapport à la stratégie O-MBS. Ce résultat s'explique par le fait que, pour ces requêtes, le nombre de requêtes exécutées pour rechercher les nouvelles MFS, avec la version optimisée de LBA, est plus grand que le nombre de requêtes relaxées que ces MFS permettent d'ignorer. Considérons, par exemple, la requête Q_7. Alors que l'algorithme O-MBS exécute 25 requêtes relaxées et 52 requêtes pour le calcul des MFS, l'algorithme F-MBS exécute 7 requêtes relaxées et 74 requêtes pour la recherche des MFS. Dans certains cas, F-MBS exécute également plus de requêtes que MBS. C'est notamment le cas pour les requêtes Q_6 et Q_7. Cette différence sur le nombre de requêtes exécutées impacte forcément le coût en temps de la stratégie F-MBS par rapport aux deux autres. En effet, O-MBS est généralement plus performante que

F-MBS, et pour des requêtes telles que Q_6, la différence entre le nombre de requêtes exécutées a un impact important sur le temps d'exécution.

6.5.2.2 Impact de la BD-RDF

Afin d'être sûr que les tendances observées précédemment étaient liées uniquement à la conception de nos stratégies et non à la BD-RDF, nous avons réalisé une deuxième expérimentation. Elle consiste à évaluer les coûts en temps des différentes stratégies mais, cette fois-ci, avec la BD-RDF Virtuoso. La figure 6.9 présente les temps d'exécution des différents algorithmes exécutés dans les mêmes conditions que dans la précédente expérimentation, excepté que la BD-RDF n'est plus Jena TDB mais Virtuoso.



	Q1	Q2	Q3	Q4	Q5	Q6	Q7
BFS	0,2	0,07	21,52	5,76	12,66	-	-
MBS	0,20	0,07	1,54	4,06	0,71	5,33	9,44
O-MBS	0,04	0,06	1,37	0,74	0,79	2,19	6,33
F-MBS	0,04	0,06	1,37	0,95	1,05	4,24	4,61

FIGURE 6.9 – Coût en temps sur Virtuoso

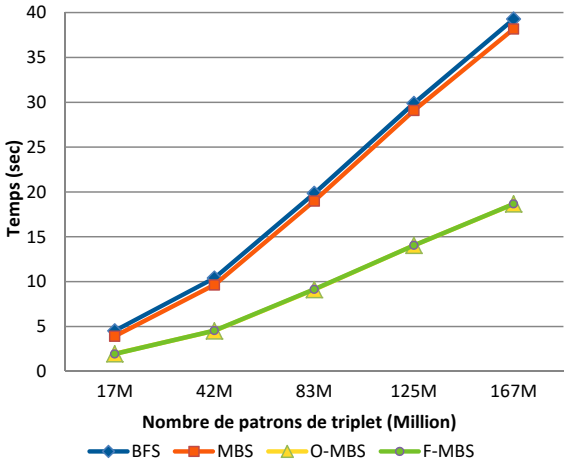


FIGURE 6.10 – Temps Vs Taille (log)

Cette expérience nous permet de confirmer les résultats précédents. Nous observons sur la figure 6.9 que les stratégies basées sur les MFS sont plus performantes que BFS, comme nous l'avions déjà constaté précédemment. L'exécution de BFS pour les requêtes Q_6 et Q_7 sur Virtuoso a dû être arrêtée car le temps d'exécution excédait une heure. C'est pour cette raison que ces temps d'exécution ne sont pas indiqués dans la figure 6.9. Même si nous obtenons des tendances identiques à celles trouvées avec les expérimentations sur Jena TDB, nous constatons néanmoins que les temps d'exécution des stratégies diffèrent de manière importante sur Virtuoso par rapport à ceux obtenus sur Jena TDB. Par exemple, les temps d'exécution des stratégies pour les requêtes Q_1 et Q_4 sont meilleurs sur Virtuoso que sur Jena TDB. Par contre, pour le reste des requêtes nous retrouvons des meilleurs temps d'exécution sur Jena TDB par rapport à Virtuoso. Ces résultats confirment bien qu'aucune BD-RDF ne donne de meilleurs temps de réponse pour tous les types de requêtes comme l'ont montré des expérimentations avec quelques bancs d'essais (par exemple, Bizer et al. [245]).

6.5.2.3 Impact de la taille des données

Dans cette dernière expérimentation, nous analysons l'impact de la taille des données sur le temps d'exécution des différentes stratégies. Pour le faire, nous avons évalué les stratégies sur des jeux de données de taille croissante allant de 17M à 167M de triplets RDF stockés dans Jena TDB. Le nombre de réponses alternatives attendues est à nouveau $K = 50$. La figure 6.10 montre l'évolution des temps d'exécution pour la requête Q_5.

Cette expérimentation montre que les temps d'exécution des différentes stratégies pour Q_5 augmentent linéairement avec la taille des données. Ce résultat est identique pour les autres requêtes. Ces résultats se comprennent étant donné que LUBM génère des données avec une distribution similaire des instances et proportionnelle à la taille des données générées. La conséquence de cette distribution similaire est que nos requêtes ont exactement le même nombre de MFS pour les différents jeux de données et les requêtes relaxées ont la même similarité par rapport à la requête initiale. Ainsi, les stratégies de relaxation exécutent les mêmes requêtes relaxées dans le même ordre. Le seul facteur qui impacte alors le temps d'exécution des stratégies est la taille des données qui engendre un coût supérieur pour chaque requête. La taille des données influence ainsi de la même façon toutes les stratégies de relaxation évaluées d'où la croissance linéaire du temps d'exécution par rapport à la taille des données.

Conclusion

Dans ce chapitre nous avons développé des stratégies pour relaxer des requêtes qui retournent des ensembles vides de résultats. Ces stratégies se démarquent de celles existantes par l'utilisation des informations sur les causes d'échecs des requêtes à différents niveaux, de la requête initiale aux requêtes relaxées. Nous avons proposé trois stratégies : MBS, O-MBS et F-MBS. MBS calcule, à l'aide de l'algorithme LBA du chapitre 5, les causes d'échecs de la requête initiale. Avec ces MFS, MBS peut identifier les requêtes relaxées qui échouent : celles qui contiennent une de ces MFS. En évitant d'exécuter des requêtes relaxées qui ne renverront aucune réponse alternative, MBS obtient des meilleurs résultats par rapport à l'approche existante qui les exécutent toutes. La stratégie O-MBS va plus loin que MBS. Lorqu'une requête relaxée ne contient pas une MFS, ce qui veut dire que la MFS a été relaxée, alors O-MBS vérifie si cette relaxation répare la MFS en exécutant la MFS relaxée. Si la MFS relaxée ne retourne aucune réponse alternative, alors elle est ajoutée à l'ensemble des MFS. Ainsi, O-MBS augmente les MFS trouvées pendant le processus de relaxation. Cette augmentation du nombre de MFS trouvées permet de réduire à nouveau le nombre de requêtes relaxées exécutées. La stratégie F-MBS ne se limite pas à la vérification de la réparation des MFS mais cherche toutes les MFS dans une requête relaxée qui échoue. Nous savons, en effet, que si une requête relaxée échoue, alors elle contient au moins une MFS. De plus, nous avons montré que les nouvelles MFS des requêtes relaxées contiennent les MFS réparées de la requête initiale. Afin de rechercher ces nouvelles MFS, nous avons implémenté une variante de l'algorithme LBA qui prend en compte les propriétés des MFS dans les requêtes relaxées et est dédié à la recherche des MFS des requêtes relaxées dans la stratégie F-MBS.

Nous avons évalué nos stratégies de relaxation et les avons comparées avec la stratégie existante BFS. Le principal résultat de ces expérimentations est que nos stratégies, guidées par les MFS, sont plus performantes que BFS pour relaxer les requêtes qui échouent. En moyenne, la stratégie O-MBS est plus de deux fois plus rapide que BFS. Nous avons pu également constater que O-MBS est la meilleure stratégie car elle permet d'obtenir un bon compromis entre le nombre de requêtes exécutées pour la recherche des MFS et le nombre de requêtes que ces MFS permettent d'ignorer (ce qui permet d'accélérer le processus de relaxation). Ainsi, O-MBS permet d'ignorer plus de requêtes que MBS tout en exécutant moins de requêtes pour trouver de nouvelles MFS que F-MBS.

Conclusion et perspectives

Conclusion

Suite à la création du concept de Web sémantique, des langages tels que RDF, RDFS et OWL ont été conçus et normalisés par le W3C. Ces langages ont permis la création d'ontologies qui définissent formellement des entités sur un domaine d'étude ainsi que les liens entre ces entités. Un challenge important est, à présent, de faciliter l'exploitation de ces ontologies. Les travaux présentés dans ce mémoire se situent dans ce contexte et visent, plus particulièrement, à aider les utilisateurs lorsque leurs requêtes, posées sur des ontologies, retournent un résultat vide, qui ne les satisfont pas. Pour résoudre ce problème, nous nous sommes intéressés à la technique de relaxation de requête qui affaiblit les conditions d'une requête afin de retourner des résultats alternatifs à l'utilisateur.

Nos travaux ont visé à répondre à deux principales questions : 1) comment relaxer les requêtes SPARQL ? 2) quelles parties d'une requête SPARQL doivent être relaxées ? L'étude de la littérature nous a permis de constater que, même si des opérateurs de relaxation avaient été proposés pour SPARQL, ils ne permettaient pas de relaxer finement la requête (c'est-à-dire sans trop affaiblir les conditions de la requête) tout en contrôlant le processus de relaxation. Concernant les parties d'une requête SPARQL à relaxer, nous avons observé qu'elles pouvaient être identifiées à l'aide de la notion de MFS, qui avait été introduite dans le contexte des bases de données relationnelles, afin d'identifier les causes d'échec d'une requête. Cependant, cette notion n'avait jamais été utilisée dans le contexte RDF, ce qui nécessitait une adaptation non triviale des techniques existantes. Ce constat nous a ainsi amené à développer plusieurs contributions regroupées dans un framework. Ces contributions sont détaillées ci-dessous.

Proposition d'opérateurs de relaxation pour des requêtes SPARQL

Nous avons proposé de mettre à la disposition des utilisateurs, via le langage d'interrogation SPARQL, un ensemble d'opérateurs de relaxation constitué des opérateurs `GEN`, `SIB` et `PRED`. Nous avons également développé des opérateurs de conjonctions pour combiner ces opérateurs. Ces derniers relaxent, de façon incrémentale, des parties précises de la requête et contrôlent la relaxation avec des mesures de similarité. Nous avons également implémenté l'outil QaRS qui permet la construction des requêtes SPARQL via des composants graphiques. Il propose une interface permettant aux utilisateurs de formuler, intuitivement, la relaxation à opérer sur les requêtes. Les relaxations étant définies sur certaines parties des requêtes, QaRS permet également aux utilisateurs de relaxer les parties responsables de l'échec de la requête.

Recherche des MFS et XSS d'une requête SPARQL

Les notions de MFS et XSS ont été introduites dans le contexte relationnel pour identifier respectivement les requêtes qui échouent avec un nombre minimal de conditions de la requête utilisateur (causes d'échec d'une requête) et les requêtes qui réussissent avec un nombre maximal de conditions de la requête utilisateur. Deux approches principales avaient été développées dans le contexte relationnel pour identifier ces MFS et XSS. Nous avons étendu et adapté ces deux approches au contexte RDF en proposant les approches LBA et MBA. LBA consiste en une exploration du treillis des sous-requêtes de la requête utilisateur. Cet espace de recherche étant large (exponentiel par rapport à la taille de la requête), LBA tire profit des propriétés des MFS et XSS pour élaguer cet espace de recherche. MBA est basée sur le calcul d'une matrice qui enregistre, pour chaque solution potentielle (c'est-à-dire, qui satisfait au moins un patron de triplets de la requête utilisateur), l'ensemble des patrons de triplets qu'elle satisfait. Une fois cette matrice calculée, les XSS de la requête sont obtenues, sans aucun accès à la base de données, en calculant le skyline de la matrice. Pour trouver les MFS, la matrice peut être utilisée par l'approche LBA pour éviter d'exécuter des requêtes sur la base de données.

Nous avons implémenté ces deux approches et évalué leur performance sur plusieurs jeux de données générés avec le benchmark LUBM. Alors qu'un algorithme de recherche en profondeur dans le treillis des sous-requêtes de la requête utilisateur ne passe pas à l'échelle pour des requêtes ayant plus de cinq patrons de triplets, l'approche LBA optimisée avec les heuristiques que nous avons proposées, calcule les MFS et XSS d'une requête qui possède 15 patrons de triplets en quelques secondes. Les résultats obtenus montrent aussi que l'approche LBA donne de meilleurs performances que MBA. Nous avons observé que le coût élevé de l'approche MBA est dû à la construction de la matrice. Une fois la matrice construite, le coût de recherche des MFS et XSS est très inférieur aux autres approches. L'approche MBA est donc intéressante si la matrice est calculée à l'avance ce qui est possible pour les requêtes régulièrement exécutées qui échouent.

Relaxation guidée par les causes d'échecs

Les XSS calculées avec les approches LBA et MBA, sont des requêtes relaxées par *suppression de patrons de triplets*. Un autre type de relaxation, plus précis, consiste à *généraliser des patrons de triplets*, plutôt que de les supprimer. En étudiant les travaux de la littérature, nous avons constaté que l'approche classique pour réaliser ce type de relaxation consiste à générer un ensemble de requêtes relaxées, à les ordonner par ordre décroissant de similarité avec la requête utilisateur et à les exécuter dans cet ordre jusqu'à trouver un nombre satisfaisant de réponses. Intuitivement, nous avons pensé que, si nous connaissons les causes d'echec de la requête, nous pouvons accélérer ce processus de relaxation en évitant d'exécuter des requêtes relaxées qui ne les réparent pas. Nous avons été plus loin en considérant non seulement les MFS de la requête utilisateur mais aussi celles des requêtes relaxées qui échouent malgré la réparation des MFS de la requête utilisateur. Ceci nous a conduit à définir trois stratégies basées sur les causes d'échec pour relaxer une requête par généralisation de ses patrons de triplets.

Dans la première stratégie, nommée MBS, nous utilisons uniquement les MFS de la requête utilisateur. Ces MFS sont utilisées pour ne pas exécuter les requêtes relaxées contenant une de ces MFS, puisqu'elles contiennent une cause d'échec et donc retourneront forcément un résultat vide. La deuxième stratégie O-MBS étend la précédente en recherchant les causes d'échec des requêtes relaxées qui échouent, malgré le fait qu'elles ne contiennent pas de MFS de la requête utilisateur. Pour cela, elle vérifie si les relaxations des MFS de la requête utilisateur, opérées dans la requête relaxée, échouent. Si c'est le cas, ce sont des MFS de la requête relaxée et elles sont utilisées pour élaguer l'espace de recherche, comme pour les

MFS de la requête utilisateur. Dans O-MBS, toutes les MFS d'une requête relaxée qui échoue ne sont pas forcément identifiées. C'est par contre le cas de notre dernière stratégie, nommée F-MBS, qui s'appuie sur une variante de LBA pour identifier les MFS de la requête utilisateur mais aussi de toutes les requêtes relaxées qui échouent malgré la réparation des MFS précédemment identifiées.

Nous avons conduit des expérimentations avec le banc d'essai LUBM et deux BD-RDF pour comparer les performances de ces stratégies avec l'approche classique de l'état de l'art. Dans ces expérimentations, notre meilleure stratégie, O-MBS, est, en moyenne, plus de deux fois plus performante que l'approche classique. O-MBS est un bon compromis entre MBS, qui élague peu l'espace de recherche à cause d'un nombre trop faible de MFS identifiées et F-MBS qui élague le plus l'espace de recherche mais qui, pour cela, passe trop de temps à calculer des MFS.

Perspectives

Au terme de nos travaux, les résultats que nous avons obtenus laissent entrevoir de nombreuses perspectives. Ces perspectives portent notamment sur l'ajout de nouveaux opérateurs de relaxation à notre framework, l'exploitation des XSS pour relaxer les requêtes, la généralisation des techniques proposées pour des requêtes qui n'échouent pas mais ne retournent pas assez de réponses, l'adaptation des stratégies de relaxation guidées par les MFS proposées pour le contexte de données RDF incertaines et des requêtes fédérées SPARQL. Dans cette section, nous détaillons ces perspectives.

Extension des opérateurs de relaxation

Les opérateurs GEN, SIB et PRED proposés dans le chapitre 4 ne sont que des exemples d'opérateurs de relaxation pouvant être intégrés au langage SPARQL. Nous pensons qu'il serait intéressant d'enrichir notre framework en y intégrant de nouveaux opérateurs de relaxation et en prenant en considération toutes les fonctionnalités de SPARQL et notamment celles introduites dans sa version 1.1 (par exemple, les opérateurs de négation, d'expression de chemin ou d'agrégation). Dans cette perspective, nous envisageons aussi d'utiliser toutes les règles de raisonnement définies dans les langages RDFS et OWL. À l'image des travaux récents proposés par Yahya et al. [246], des règles utilisateur pourraient aussi être utilisées. Nous envisageons aussi d'exploiter des constructeurs spécifiques de OWL tels que `owl : sameAs` ou `owl : equivalentClass` pour définir de nouveaux opérateurs de relaxation.

Utilisation des XSS pour relaxer des requêtes SPARQL

Nous avons, dans le chapitre 6, proposé de guider le processus de relaxation par les MFS de la requête utilisateur. Nous avons montré que ceci accélère le processus de relaxation en évitant d'exécuter des requêtes relaxées. Nous pensons que les XSS, qui sont calculées avec les approches LBA et MBA proposées au chapitre 5, pourraient également être utilisées pour proposer de nouvelles stratégies de relaxation.

Plus précisément, nous avons vu au chapitre 5 qu'une XSS s'obtient par suppression de certains patrons de triplet de la requête initiale. La suppression de patrons de triplets est une relaxation extrême qui peut conduire à un ensemble pléthorique de réponses parmi lesquelles nous ne pourrions distinguer les plus pertinentes des moins. Dans cette perspective, nous proposons donc de généraliser les patrons de triplets supprimés par une XSS. Cette généralisation pourrait se faire en s'appuyant sur les structures de données utilisées dans nos stratégies basées sur les MFS. Alors que les stratégies de relaxation basées sur les MFS permettent d'élaguer un large espace de recherche (car constitué de la relaxation de tous les patrons de triplet de la requête utilisateur), les stratégies basées sur les XSS permettraient de construire directement

un espace de recherche plus réduit (car constitué uniquement de la relaxation des patrons de triplets enlevés dans une XSS). Ils seraient donc intéressant de comparer ces deux types de stratégies en termes de performance et de qualité des réponses alternatives retournées.

Généralisation de la relaxation avec les MFS et XSS

Les stratégies de relaxation guidées par les MFS que nous avons proposées ne sont pertinentes que pour des requêtes qui ne retournent aucune réponse. En effet, si la requête initiale retourne un ensemble de réponse dont la taille est inférieure à `K`, le nombre de réponses désirées par l'utilisateur, nous ne pourrons pas utiliser nos stratégies, puisqu'elles s'appuient sur les MFS de la requête (ce qui suppose que celle-ci ne retourne pas de résultat).

Pour répondre à cette limitation, notre idée est de généraliser la définition des MFS en introduisant la notion de `K-MFS`. Au lieu de considérer comme MFS une sous-requête qui ne retourne aucune réponse alors que toutes ses sous-requêtes retournent des réponses, notre idée consiste à considérer une `K-MFS` comme une sous-requête qui retourne un ensemble de réponses de taille inférieure à `K` et dont toutes ses sous-requêtes retournent un ensemble de réponses de taille supérieure ou égale à `K`. Il sera par contre nécessaire de vérifier si les propriétés des MFS sont conservées pour les `K-MFS`. Si ce n'est pas le cas, nous devrons étudier les adaptations qu'il sera nécessaire de faire aux approches LBA et MBA pour qu'elles retournent les `K-MFS` et, par analogie aux XSS, les `K-XSS`.

Par ailleurs, pour la vérification de la taille des réponses retournées par les requêtes nous pensons qu'il serait intéressant d'exploiter les travaux menés pour estimer la sélectivité des requêtes SPARQL [247, 248]. Ainsi, la recherche des `K-MFS` pourraient se faire plus rapidement en utilisant ces estimations plutôt qu'en exécutant des requêtes sur la base de données. Cependant, il sera nécessaire de voir si la précision de ces estimations est suffisante pour obtenir correctement les `K-MFS`.

Relaxation des requêtes SPARQL pour des BD-RDF avec degré de confiance

Les ontologies sont souvent construites en intégrant les données de différentes sources hétérogènes. Cette intégration soulève quelques interrogations. Qu'en est-il de la fiabilité de ces données, sont-elles certaines ou incertaines ? Qu'en est-il du degré de confiance des sources de données, sont-elles sûres ? Des travaux ont donc été menés afin d'associer un degré de confiance à des données RDF [249]. En particulier, Olaf Hartig [250] propose une version de SPARQL pour interroger des données RDF associées à un degré de confiance : le langage *tSPARQL* (*Trust SPARQL*). Il définit le modèle des données RDF avec degré de confiance comme un quadruplet $(s,\ p,\ o,\ \alpha)$ où $(s,\ p,\ o)$ est un triplet RDF et α le degré de confiance associé à ce triplet, qui est étroitement lié au degré de confiance de la source de données de provenance de ce triplet. Avec tSPARQL, l'utilisateur peut spécifier des contraintes sur le degré de confiance désiré pour chaque patron de triplet de sa requête et/ou pour l'ensemble de la requête.

Dans ce contexte, une requête peut échouer par absence de réponses ayant un degré de confiance suffisant pour l'utilisateur. Ce nouveau contexte introduit donc de nouvelles possibilités de causes d'échec. Il serait donc intéressant de voir comment les notions et approches développées dans cette thèse peuvent être adaptées à ce contexte. Même si la relaxation se fera à nouveau sur les patrons de triplet, elle devra prendre en considération le degré de confiance des réponses alternatives retournées par les requêtes relaxées. Notons aussi que l'utilisateur pourra relaxer également le degré de confiance d'un ou plusieurs patrons de triplet, ou celui de toute la requête.

Relaxation des requêtes fédérées SPARQL sur des données liées

Au lieu d'intégrer des données provenant de différentes sources, certains travaux ont proposé de lier entre elles plusieurs sources de données à l'aide de propriétés : ce sont *les données liées (Linked Data)*. Ce terme désigne l'ensemble des bonnes pratiques pour la publication et la connexion des données à travers le Web [251]. L'interrogation à travers plusieurs sources de données liées est l'un des points forts des données liées. Le langage SPARQL permet de réaliser cette interrogation à travers des services appelés *SPARQL Endpoint*. Plusieurs travaux ont proposé des techniques afin d'optimiser l'interrogation des données liées [252, 253]. Dans ses travaux, Hartig a notamment proposé le langage *LDQL* (*Linked Data Query language*) [254, 255] pour l'interrogation des données liées.

Dans le contexte des données liées, les requêtes peuvent retourner des réponses insatisfaisantes, en termes de quantité et/ou de qualité, pour plusieurs raisons. Par exemple, une source de données peut ne pas avoir de réponses correspondant au patron de graphe à exécuter sur cette source ou encore la jointure entre les ensembles de réponses provenant de deux sources de données peuvent donner un ensemble vide de réponse. En adaptant nos algorithmes dans ce contexte, nous pourrions identifier la partie de la requête et/ou la source de données qui pose problème. Dans le cas des jointures entre deux ensembles de réponses de deux sources de données, nous pourrions étudier la possibilité d'adapter les approches de relaxation de jointures de Koudas et al. [167], développées dans le contexte relationnel, pour relaxer ces jointures.

Bibliographie

[1] Berners-Lee, T., Fischetti, M.: Weaving the Web: The Original Design and Ultimate Destiny of the World Wide Web by Its Inventor. 1st edn. Harper San Francisco (1999)

[2] Gerber, A., van der Merwe, A., Barnard, A. In: A Functional Semantic Web Architecture. Springer Berlin Heidelberg, Berlin, Heidelberg (2008) 273–287

[3] Suchanek, F., Gjergji, K., Gerhard, W.: Yago: A large ontology from wikipedia and wordnet. Web Semantics: Science, Services and Agents on the World Wide Web **6**(3) (2008) 203 – 217 World Wide Web Conference 2007 Semantic Web Track.

[4] Jens, L., Robert, I., Max, J., Anja, J., Dimitris, K., Pablo N., M., Sebastian, H., Mohamed, M., Patrick, v.K., Sören, A., Christian, B.: Dbpedia - A large-scale, multilingual knowledge base extracted from wikipedia. Semantic Web **6**(2) (March 2015) 167–195 Data link: `http://dbpedia.org/snorql/`.

[5] Ley, M.: The dblp computer science bibliography: Evolution, research issues, perspectives. In: Proceedings of the 9th International Symposium on String Processing and Information Retrieval. SPIRE 2002, London, UK, UK, Springer-Verlag (2002) 1–10

[6] Dong, X., Gabrilovich, E., Heitz, G., Horn, W., Lao, N., Murphy, K., Strohmann, T., Sun, S., Zhang, W.: Knowledge vault: A web-scale approach to probabilistic knowledge fusion. In: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. KDD '14, New York, NY, USA, ACM (2014) 601–610

[7] Deshpande, O., Lamba, D.S., Tourn, M., Das, S., Subramaniam, S., Rajaraman, A., Harinarayan, V., Doan, A.: Building, maintaining, and using knowledge bases: A report from the trenches. In: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD'13). (2013) 1209–1220

[8] Dan, B., Ramanathan, V., G.: Rdf vocabulary description language 1.0: Rdf schema. W3C Recommendation (February 2004) `https://www.w3.org/TR/2004/REC-rdf-schema-20040210/#ch_introduction`.

[9] Motik, B., Cuenca Grau, B., Horrocks, I., Wu, Z., Fokoue, A., Lutz, C.: Owl 2 web ontology language profiles (second edition). W3C Recommendation (Decembre 2012) `https://www.w3.org/TR/owl2-profiles/#Feature_Overview_3`.

[10] Prud'hommeaux, E., Seaborne, A.: Sparql query language for rdf. W3C Recommendation (January 2008) `https://www.w3.org/TR/rdf-sparql-query/`.

[11] Saleem, M., Ali, I., Hogan, A., Mehmood,

Q., Ngonga Ngomo, A.C.: Lsq: The linked sparql queries dataset. In: International Semantic Web Conference (ISWC), Berlin, Heidelberg, Springer Berlin Heidelberg (October 2015)

[12] Kaplan, S.J.: Cooperative Responses from a Portable Natural Language Data Base Query System. PhD thesis, University of Pennsylvania, Philadelphia, PA, USA (1979)

[13] Galindo-Legaria, C.A.: Algebraic Optimization of Outerjoin Queries. PhD thesis, Harvard University (1992)

[14] Arias, M., Fernández, J.D., Martínez-Prieto, M.A., de la Fuente, P.: An empirical study of real-world SPARQL queries. CoRR **abs/1103.5043** (2011)

[15] Frosini, R., Calì, A., Poulovassilis, A., Wood, P.T.: Flexible query processing for sparql. Semantic Web Journal (2015) 1–31

[16] Hurtado, C.A., Poulovassilis, A., Wood, P.T. In: Ranking Approximate Answers to Semantic Web Queries. Springer Berlin Heidelberg, Berlin, Heidelberg (2009) 263–277

[17] Huang, H., Liu, C., Zhou, X.: Approximating query answering on rdf databases. World Wide Web **15**(1) (January 2012) 89–114

[18] Dolog, P., Stuckenschmidt, H., Wache, H., Diederich, J.: Relaxing rdf queries based on user and domain preferences. Journal of Intelligent Information Systems **33**(3) (December 2009) 239–260

[19] Calì, A., Frosini, R., Poulovassilis, A., Wood, P.T. In: Flexible Querying for SPARQL. Springer Berlin Heidelberg, Berlin, Heidelberg (2014) 473–490

[20] Corby, O., Dieng-Kuntz, R., Faron-Zucker, C., Gandon, F.: Searching the semantic web: Approximate query processing based on ontologies. IEEE Intelligent Systems **21**(1) (January 2006) 20–27

[21] Yuanbo, G., Zhengxiang, P., Jeff, H.: Lubm: A benchmark for owl knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web **3**(2–3) (2005) 158 – 182

[22] Manola, F., Miller, E.: RDF Primer. World Wide Web Consortium. (February 2004) `http://www.w3.org/TR/rdf-primer`.

[23] Hayes, P.: Rdf semantics. W3C Recommendation (February 2004) Latest version : `http://www.w3.org/TR/rdf-mt/`.

[24] Smith, M., Chris, W., Deborah L., M.: Owl web ontology language guide. W3C Recommendation (February 2004) `https://www.w3.org/TR/owl-guide/`.

[25] ISO13584-42: Industrial automation systems and integration – Parts library – Part 42: Description methodology: Methodology for structuring parts families. Technical report, International Standards Organization, Genève (1998)

[26] International Standards Organization, ISO: Standard data element types with associated classification scheme for electric components - part 4: IEC reference collection of standard data element types, component classes and terms.

[27] Wang, T.D., Parsia, B., Hendler, J. In: A Survey of the Web Ontology Landscape. Springer Berlin Heidelberg, Berlin, Heidelberg (2006) 682–694

[28] Jean, S.: OntoQL, un langage d'exploitation des bases de données à base ontologique. Theses, Université de Poitiers (Décembre 2007)

[29] Psyché, V., Mendes, O., Bourdeau, J.: Apport de l'ingénierie ontologique aux environnements de formation à distance. Sciences et Technologies de l'Information et de la Communication pour l'Éducation et la Formation (STICEF) **10** (2003) 89–126

[30] Thomas R., G.: Toward principles for the design of ontologies used for knowledge sharing. In: In Formla Ontology in Conceptual Analysis and Knowledge Representation, Kluwer Academic Publishers (1993)

[31] Fankam, C., Bellatreche, L., Dehainsala, H., Ait-Ameur, Y., Pierra, G.: Sisro : conception de bases de données à partir d'ontolo-

gies de domaine. Technique et science informatiques (TSI) (2009)

[32] Ramanathan, V., G., Tim, B.: Meta content framework using xml. submission to W3C (June 1997) https://www.w3.org/TR/NOTE-MCF-XML/#sec3.5.

[33] Estival, D., Nowak, C., Zschorn, A.: Towards ontology-based natural language processing. In: Proceeedings of the Workshop on NLP and XML (NLPXML-2004): RDF/RDFS and OWL in Language Technology. NLPXML '04, Stroudsburg, PA, USA, Association for Computational Linguistics (2004) 59–66

[34] Adda, M.: Intégration des connaissances ontologiques dans la fouille de motifs séquentiels avec application à la personnalisation web. Theses, Université des Sciences et Technologie de Lille - Lille I ; Université de Montréal (November 2008)

[35] Brisson, L., Collard, M.: An ontology driven data mining process. In: International Conference on Enterprise Information Systems, Barcelone, Spain (June 2008) 54–61

[36] Suchanek, F.M., Kasneci, G., Weikum, G.: Yago: A core of semantic knowledge. In: Proceedings of the 16th International Conference on World Wide Web. WWW '07, New York, NY, USA, ACM (2007) 697–706

[37] The UniProt Consortium: The universal protein resource (uniprot). Nucleic Acids Research **36** (2008) 190 – 195

[38] The Gene Ontology Consortium: Gene ontology: tool for the unification of biology. Nature Genetics **25** (May 2000) 25 – 29

[39] ISO13584-25: Industrial automation systems and integration – Parts library – Part 25: Logical resource: Logical model of supplier library with aggregate values and explicit content. Technical report, International Standards Organization, Genève (2004)

[40] W3C: Des acteurs essentiels de l'industrie joignent leurs efforts pour développer des méta-données interopérables sur le web. World Wide Web Consortium (W3C) (October 1997) https://www.w3.org/Press/RDF.

[41] Cyganiak, R., David, W., Markus, L.: Rdf 1.1 concepts and abstract syntax. W3C Recommendation (February 2014) https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/.

[42] RDF Working Group: Resource description framework (rdf) (February 2014) https://www.w3.org/2001/sw/wiki/RDF.

[43] Sowa, J.F.: From existential graphs to conceptual graphs. International Journal of Conceptual Structures and Smart Applications **1**(1) (January 2013) 39–72

[44] ter Horst, H.: Completeness, decidability and complexity of entailment for {RDF} schema and a semantic extension involving the {OWL} vocabulary. Web Semantics: Science, Services and Agents on the World Wide Web **3**(2–3) (2005) 79 – 115

[45] Hayes, P., Patel-Schneider, P.: Rdf 1.1 semantics. W3C Recommendation (February 2014) https://www.w3.org/TR/rdf11-mt/#rdf-entailment.

[46] Antoniou, G., van Harmelen, F. International Handbooks on Information Systems. In: Web Ontology Language: OWL. Springer Berlin Heidelberg (2004) 67–92

[47] Özsu, M.T.: A survey of RDF data management systems. CoRR **abs/1601.00707** (February 2016)

[48] Jean, S., Bellatreche, L., Fokou, G., Baron, M., Khouri, S.: OntoDBench: Novel Benchmarking System for Ontology-Based Databases. In: On the Move to Meaningful Internet Systems: OTM 2012: Confederated International Conferences: CoopIS, DOA-SVI, and ODBASE 2012, Rome, Italy, September 10-14, 2012. Proceedings, Part II. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 897–914

[49] Kontchakov, R., Rezk, M., Rodríguez-Muro, M., Xiao, G., Zakharyaschev, M.: Answering sparql queries over databases under owl 2 ql entailment regime. In: Proceedings of the 13th International Semantic

Web Conference - Part I. ISWC '14, New York, NY, USA, Springer-Verlag New York, Inc. (2014) 552–567

[50] Chebotko, A., Lu, S., Fotouhi, F.: Semantics preserving sparql-to-sql translation. Data Knowl. Eng. **68**(10) (October 2009) 973–1000

[51] Diego, C., Benjamin, C., Sarah, K.E., Roman, K., Davide, L., Martin, R., Mariano, R.M., Guohui, X.: Ontop: Answering sparql queries over relational databases. Semantic Web Journal **Preprint** (February 2016) 1–17 `http://ontop.inf.unibz.it/publications/`.

[52] Chong, E.I., Das, S., Eadon, G., Srinivasan, J.: An efficient sql-based rdf querying scheme. In: Proceedings of the 31st International Conference on Very Large Data Bases. VLDB '05, VLDB Endowment (2005) 1216–1227

[53] Rodriguez-Muro, M., Rezk, M., Hardi, J., Slusnys, M., Bagosi, T., Calvanese, D.: Evaluating SPARQL-to-SQL translation in Ontop. In: Proceedings of the 2nd International Workshop on OWL Reasoner Evaluation (ORE), Ulm, Germany, July 22, 2013. Volume 1015 of CEUR Workshop Proceedings., CEUR-WS.org (2013) 94–100

[54] Stephen, H., Nicholas, G.: 3store: Efficient bulk rdf storage. Event Dates: 2003-10-20 (October 2003)

[55] C. Faye, D., Cure, O., Blin, G.: A survey of rdf storage approaches. ARIMA Journal **15** (February 2012)

[56] Salvadores, M., Correndo, G., Harris, S., Gibbins, N., Shadbolt, N.: The design and implementation of minimal rdfs backward reasoning in 4store. In: The Semanic Web: Research and Applications: 8th Extended Semantic Web Conference, ESWC 2011, Heraklion, Crete, Greece, May 29 – June 2, 2011, Proceedings, Part II, Berlin, Heidelberg, Springer Berlin Heidelberg (2011) 139–153

[57] Dehainsala, H., Pierra, G., Bellatreche, L. In: OntoDB: An Ontology-Based Database for Data Intensive Applications. Springer Berlin Heidelberg, Berlin, Heidelberg (2007) 497–508

[58] Hondjack, D.: Explicitation de la sémantique dans les base de données : Base de données à base ontologique et le modèle OntoDB. Theses, Université de Poitiers (Mai 2007)

[59] Erling, O., Mikhailov, I.: Rdf support in the virtuoso dbms. In: Conference on Social Semantic Web. Volume 113 of LNI., GI (2007) 59–68

[60] Owens, A., Seaborne, A., Gibbins, N., schraefel, M.: Clustered tdb: A clustered triple store for jena. In: World Wide Web, WWW2009, Madrid, Spain, VLDB Endowment (April 2009)

[61] Neumann, T., Weikum, G.: Rdf-3x: A risc-style engine for rdf. Proceedings of the 34th International Conference on Very Large Data Bases **1**(1) (August 2008) 647–659

[62] Birte, G., Chimezie, O.: Sparql 1.1 entailment regimes. W3C Recommendation (March 2013) `https://www.w3.org/TR/2013/REC-sparql11-entailment-20130321/#entEffects`.

[63] Bursztyn, D., Goasdoué, F., Manolescu, I.: Reformulation-based query answering in rdf: Alternatives and performance. Proc. VLDB Endow. **8**(12) (August 2015) 1888–1891

[64] Goasdoué, F., Manolescu, I., Roatiş, A.: Efficient query answering against dynamic rdf databases. In: Proceedings of the 16th International Conference on Extending Database Technology. EDBT '13, New York, NY, USA, ACM (2013) 299–310

[65] Mbaiossoum Bery, L.: Conception physique des bases de données à base ontologique : le cas des vues matérialisées. Theses, Ecole Nationale Supérieure de Mécanique et d'Aerotechnique (Décembre 2014)

[66] Seaborne, A.: Rdql - a query language for rdf. submission to W3C

(January 2004) https://www.w3.org/Submission/RDQL/.

[67] Miller, L., Seaborne, A., Reggiori, A.: Three implementations of squishql, a simple rdf query language. In: Proceedings of the First International Semantic Web Conference on The Semantic Web. ISWC '02, London, UK, UK, Springer-Verlag (2002) 423–435

[68] Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., Scholl, M.: Rql: A declarative query language for rdf. In: Proceedings of the 11th International Conference on World Wide Web. WWW '02, New York, NY, USA, ACM (2002) 592–603

[69] Aduna, B., Sirma, A.L.: Chapter 6. the serql query language (revision 1.2). User Guide for Sesame (2005) online resource, http://poc.vl-e.nl/distribution/manual/sesame-1.2.3/ch06.html.

[70] Jean, S., Aït-Ameur, Y., Pierra, G.: Querying Ontology Based Database Using OntoQL (An Ontology Query Language). In: On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE, Montpellier, France, October 29 - November 3, 2006. Proceedings, Part I. Springer Berlin Heidelberg, Berlin, Heidelberg (2006) 704–721

[71] Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., Lausen, G.: RDFPath: Path Query Processing on Large RDF Graphs with MapReduce. In: The Semantic Web: ESWC 2011 Workshops: ESWC 2011 Workshops, Heraklion, Greece, May 29-30, 2011, Revised Selected Papers. Springer Berlin Heidelberg, Berlin, Heidelberg (2012) 50–64

[72] Chimezie, O.: Versa: Path-based rdf query language. O'Reilly XML.com (July 2005) online resource, http://www.xml.com/pub/a/2005/07/20/versa.html?page=1.

[73] Aimilia, M., Grigoris, K., Ta Tuan, A., Vassilis, C., Dimitris, P.: Ontology storage and querying. Foundation for Research and Technology Hellas, Institute of Computer Science, Information Systems Laboratory (July 2002) Technical ReportT, No 308.

[74] Picalausa, F., Vansummeren, S.: What are real sparql queries like? In: Proceedings of the International Workshop on Semantic Web Information Management. SWIM '11, New York, NY, USA, ACM (2011) 7:1–7:6

[75] Stadler, C., Lehmann, J., Höffner, K., Auer, S.: Linkedgeodata: A core for a web of spatial open data. Semant. web **3**(4) (October 2012) 333–354 Data link: http://linkedgeodata.org/OnlineAccess.

[76] Möller, K., Heath, T., Handschuh, S., Domingue, J.: Recipes for Semantic Web Dog Food — The ESWC and ISWC Metadata Projects. In: The Semantic Web: 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2007) 802–815 Data link: https://datahub.io/dataset/semantic-web-dog-food.

[77] Jorge, P., Marcelo, A., Claudio, G.: Semantics and complexity of SPARQL. ACM Trans. Database Syst. **34**(3) (2009)

[78] ANSI/ISO/IEC: Data Language SQL-Part 2: Foundation (SQL/Foundation) (1999)

[79] ANSI/ISO/IEC: Data Language SQL-Part 2: Foundation (SQL/Foundation)

[80] Cha, S.K., Wiederhold, G.: Kaleidoscope: A cooperative menu-guided query interface. SIGMOD Rec. **19**(2) (May 1990) 387–

[81] Mohan, L., Kashyap, R.L.: A visual query language for graphical interaction with schema-intensive databases. IEEE Transactions on Knowledge and Data Engineering **5**(5) (October 1993) 843–858

[82] Gal, A.: Cooperative responses in Deductive Databases. PhD thesis, University of Maryland, College Park (1988)

[83] Janas, J.M.: On the feasibility of informative answers. In: Advances in Data Base Theory:

Volume 1, Boston, MA, Springer US (1981) 397–414

[84] Motro, A.: Query generalization: A method for interpreting null answers. In: Proceedings from the First International Workshop on Expert Database Systems, Redwood City, CA, USA, Benjamin-Cummings Publishing Co., Inc. (1986) 597–616

[85] Motro, A.: Cooperative database systems. International Journal of Intelligent Systems **11**(10) (1996) 717–731

[86] Minker, J.: An overview of cooperative answering in databases. In: Flexible Query Answering Systems: Third International Conference, FQAS'98 Roskilde, Denmark, May 13–15, 1998 Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (1998) 282–285

[87] Wesley W., C. In: Cooperative Database Systems. John Wiley & Sons (2008)

[88] Motro, A.: A trio of database user interfaces for handling vague retrieval requests. IEEE Data Engineering Bulletin **12** (1989) 54–63

[89] Motro, A.: Baroque: A browser for relational databases. ACM Trans. Inf. Syst. **4**(2) (April 1986) 164–181

[90] Motro, A.: Vague: A user interface to relational databases that permits vague queries. ACM Transactions on Information Systems Journal (TOIS) **6**(3) (July 1988) 187–214

[91] Motro, A.: Flex: a tolerant and cooperative user interface to databases. IEEE Transactions on Knowledge and Data Engineering **2**(2) (June 1990) 231–246

[92] Harper, S.: Technology: Sql developer, building queries visually. Oracle Magazine (March 2008) `http://www.oracle.com/technetwork/issue-archive/2008/08-mar/o28sql-100636.html`.

[93] Manoj E., P., Ravi N., M., Ravikumar R., A.: Design and implementation of graphical user interface for relational database management. International Journal of Computer Science and Information Technologies (IJCSIT) **3**(3) (March 2012) 3871–3874

[94] Catarci, T., Costabile, M.F., Levialdi, S., Batini, C.: Visual query systems for databases. Journal of Visual Languages and Computing **8**(2) (April 1997) 215–260

[95] Smart, P.R., Russell, A., Braines, D., Kalfoglou, Y., Bao, J., Shadbolt, N.R.: A visual approach to semantic query design using a web-based graphical query designer. In: Proceedings of the 16th International Conference on Knowledge Engineering: Practice and Patterns. EKAW '08, Berlin, Heidelberg, Springer-Verlag (2008) 275–291

[96] Murray, N., Paton, N., Goble, C.: Kaleidoquery: A visual query language for object databases. In: Proceedings of the Working Conference on Advanced Visual Interfaces. AVI '98, New York, NY, USA, ACM (1998) 247–257

[97] Alashqur, A.M., Su, S.Y.W., Lam, H.: Oql: A query language for manipulating object-oriented databases. In: Proceedings of the 15th International Conference on Very Large Data Bases. VLDB '89, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (1989) 433–442

[98] Blau, H., Immerman, N., Jensen, D.: A visual query language for relational knowledge discovery title2:. Technical report, University of Massachusetts, Amherst, MA, USA (2001)

[99] Berger, S., Bry, F., Schaffert, S.: A visual language for web querying and reasoning. In: Principles and Practice of Semantic Web Reasoning: International Workshop, PPSWR 2003, Mumbai, India, December 8, 2003. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2003) 99–112

[100] Jethro, B., Hanno, E.: Graphical query composition and natural language processing in an rdf visualization

[101] M., E.: Xing: a visual {XML} query language. Journal of Visual Languages and Computing **14**(1) (2003) 5 – 45

[102] e Zainab, S.S., Hasnain, A., Saleem, M., Mehmood, Q., Zehra, D., Decker, S.: Fed-

viz: A visual interface for sparql queries formulation and execution. In: Workshop Visualizations and User Interfaces for Ontologies and Linked Data (VOILA) at International Semantic Web Conference. (October 2015)

[103] Haag, F., Lohmann, S., Siek, S., Ertl, T.: QueryVOWL: Visual composition of SPARQL queries. In: Proceedings of ESWC 2015 Satellite Events. Volume 9341 of LNCS., Springer (2015) 62–66

[104] Bar-Yossef, Z., Kraus, N.: Context-sensitive query auto-completion. In: Proceedings of the 20th International Conference on World Wide Web. WWW '11, New York, NY, USA, ACM (2011) 107–116

[105] Campinas, S., Perry, T.E., Ceccarelli, D., Delbru, R., Tummarello, G.: Introducing rdf graph summary with application to assisted sparql formulation. In: Proceedings of the 2012 23rd International Workshop on Database and Expert Systems Applications. DEXA '12, Washington, DC, USA, IEEE Computer Society (2012) 261–266

[106] Russell, A., Smart, P.R., Braines, D., Shadbolt, N.R.: Nitelight: A graphical tool for semantic query construction. In: Semantic Web User Interaction Workshop (SWUI 2008). (April 2008) Event Dates: 5th April 2008.

[107] Xiao, C., Qin, J., Wang, W., Ishikawa, Y., Tsuda, K., Sadakane, K.: Efficient error-tolerant query autocompletion. Proc. VLDB Endow. **6**(6) (April 2013) 373–384

[108] Jiang, J.Y., Ke, Y.Y., Chien, P.Y., Cheng, P.J.: Learning user reformulation behavior for query auto-completion. In: Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval. SIGIR '14, New York, NY, USA, ACM (2014) 445–454

[109] Meng, L.: A survey on query suggestion. International Journal of Hybrid Information Technology **7**(6) (2014) 43 – 56

[110] Nandi, A., Jagadish, H.V.: Assisted querying using instant-response interfaces. In: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data. SIGMOD '07, New York, NY, USA, ACM (2007) 1156–1158

[111] Khoussainova, N., Kwon, Y., Balazinska, M., Suciu, D.: Snipsuggest: Context-aware autocompletion for sql. Proc. VLDB Endow. **4**(1) (October 2010) 22–33

[112] Whiting, S., Jose, J.M.: Recent and robust query auto-completion. In: Proceedings of the 23rd International Conference on World Wide Web. WWW '14, New York, NY, USA, ACM (2014) 971–982

[113] Aston, Z., Amit, G., Weize, K., Hongbo, D., Anlei, D., Yi, C., Carl A., G., Jiawei, H.: adaqac: Adaptive query auto-completion via implicit negative feedback. In: Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval, Santiago, Chile, August 9-13, 2015. (2015) 143–152

[114] Li, L., Xu, G., Yang, Z., Dolog, P., Zhang, Y., Kitsuregawa, M.: An efficient approach to suggesting topically related web queries using hidden topic model. World Wide Web **16**(3) (2013) 273–297

[115] Qumsiyeh, R., Yiu-Kai, N.: Assisting web search using query suggestion based on word similarity measure and query modification patterns. World Wide Web **17**(5) (2013) 1141–1160

[116] Campinas, S.: Live SPARQL autocompletion. In: Proceedings of the ISWC 2014 Posters & Demonstrations Track a track within the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Italy, October 21, 2014. (2014) 477–480

[117] Zenz, G., Zhou, X., Minack, E., Siberski, W., Nejdl, W.: From keywords to semantic queries-incremental query construction on the semantic web. Web Semant. **7**(3) (September 2009) 166–176

[118] Zloof, M.: Query-by-example: The invocation and definition of tables and forms. In:

Proceedings of the 1st International Conference on Very Large Data Bases. VLDB '75, New York, NY, USA, ACM (1975) 1–24

[119] Ramakrisjnan, R., Gehrke, J.: QUERY-BY-EXAMPLE (QBE). In: Database Management Systems, Second Edition. 2nd edn. Osborne/McGraw-Hill, Berkeley, CA, USA (August 2002) Hardcover.

[120] Kacprzyk, J., Zadrozny, S., Ziólkowski, A.: FQUERY III +:a "human-consistent" database querying system based on fuzzy logic with linguistic quantifiers. Journal of Information Systems **14**(6) (1989) 443–453

[121] Zadrozny, S., Kacprzyk, J.: FQUERY for access: towards human consistent querying user interface. In: Proceedings of the 1996 ACM Symposium on Applied Computing, SAC'96, Philadelphia, PA, USA, February 17-19, 1996. (1996) 532–536

[122] Li, H., Chan, C.Y., Maier, D.: Query from examples: An iterative, data-driven approach to query construction. PVLDB **8**(13) (2015) 2158–2169

[123] Jayaram, N., Khan, A., Li, C., Yan, X., Elmasri, R.: Towards a query-by-example system for knowledge graphs. In: Proceedings of Workshop on GRAph Data Management Experiences and Systems. GRADES'14, New York, NY, USA, ACM (2014) 11:1–11:6

[124] Jayaram, N., Khan, A., Li, C., Yan, X., Elmasri, R.: Querying knowledge graphs by example entity tuples. CoRR **abs/1311.2100** (2013)

[125] Mottin, D., Lissandrini, M., Velegrakis, Y., Palpanas, T.: Exemplar queries: Give me an example of what you need. Proc. VLDB Endow. **7**(5) (January 2014) 365–376

[126] Mottin, D., Lissandrini, M., Velegrakis, Y., Palpanas, T.: Searching with xq: The exemplar query search engine. In: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. SIGMOD '14, New York, NY, USA, ACM (2014) 901–904

[127] Braga, D., Campi, A., Ceri, S.: Xqbe (xquery by example): A visual interface to the standard xml query language. ACM Trans. Database Syst. **30**(2) (June 2005) 398–443

[128] Benzaken, V., Castagna, G., Colazzo, D., Miachon, C.: Pattern by example: Type-driven visual programming of xml queries. In: Proceedings of the 10th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming. PPDP '08, New York, NY, USA, ACM (2008) 131–142

[129] Chapman, A., Jagadish, H.V.: Why not? In: Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data. SIGMOD '09, ACM (2009) 523–534

[130] Herschel, M., Hernández, M.A.: Explaining missing answers to SPJUA queries. Proc. VLDB Endow. **3**(1-2) (September 2010) 185–196

[131] Huang, J., Chen, T., Doan, A., Naughton, J.F.: On the provenance of non-answers to queries over extracted data. Proc. VLDB Endow. **1**(1) (August 2008) 736–747

[132] Herschel, M.: A hybrid approach to answering why-not questions on relational query results. J. Data and Information Quality **5**(3) (March 2015) 10:1–10:29

[133] Lei, C., Xin, L., Haibo, H., Christian S., J., Jianliang, X.: Answering why-not questions on spatial keyword top-k queries. In: 31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015. (2015) 279–290

[134] Ten Cate, B., Civili, C., Sherkhonov, E., Tan, W.: High-level why-not explanations using ontologies. In: Proceedings of the 34th ACM Symposium on Principles of Database Systems. PODS '15, New York, NY, USA, ACM (2015) 31–43

[135] Md. Saiful, I., Chengfei, L., Jianxin, L.: Efficient answering of why-not questions in similar graph matching. IEEE Trans. Knowl. Data Eng. **27**(10) (2015) 2672–2686

[136] Yao, S., Liu, J., Wang, M., Wei, B., Chen, X.: Anna: Answering why-not questions for sparql. In Villata, S., Pan, J.Z., Dragoni, M., eds.: International Semantic Web Conference (Posters and Demos). Volume 1486 of CEUR Workshop Proceedings., CEUR-WS.org (2015)

[137] Bidoit, N., Herschel, M., Tzompanaki, A.: EFQ: why-not answer polynomials in action. PVLDB **8**(12) (2015) 1980–1991

[138] Tran, Q.T., Chan, C.Y.: How to conquer why-not questions. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data. SIGMOD '10, New York, NY, USA, ACM (2010) 15–26

[139] Zhang, J., Han, W., Jia, Y., Zou, P., Fan, H.: A novel approach to query modification based on user's why-not question. International Journal of Computer Science Issues (IJCSI) **10**(1) (2013) 1694 – 0784

[140] Bidoit, N., Herschel, M., Tzompanaki, A.: Efficient computation of polynomial explanations of why-not questions. In: Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, CIKM 2015, Melbourne, VIC, Australia, October 19 - 23, 2015. (2015) 713–722

[141] Bidoit, N., Herschel, M., Tzompanaki, K.: Query-based why-not provenance with NedExplain. In: Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014. (2014) 145–156

[142] Tran, Q.T., Chan, C.Y., Parthasarathy, S.: Query reverse engineering. The VLDB Journal **23**(5) (October 2014) 721–746

[143] Md. Saiful, I.: On answering why and why-not questions in databases. In: IEEE 29th International Conference on Data Engineering Workshops (ICDEW). ICDEW '13, IEE (2013) 298 – 301

[144] Glavic, B., Köhler, S., Riddle, S., Ludäscher, B.: Towards constraint-based explanations for answers and non-answers. In: Proceedings of the 7th USENIX Workshop on the Theory and Practice of Provenance (TaPP). (2015)

[145] Bosc, P., Hadjali, A., Pivert, O.: Empty versus overabundant answers to flexible relational queries. Fuzzy Sets Syst. **159**(12) (June 2008) 1450–1467

[146] Vasilyeva, E., Thiele, M., Bornhövd, C., Lehner, W.: Answering "why empty?" and "why so many?" queries in graph databases. J. Comput. Syst. Sci. **82**(1) (February 2016) 3–22

[147] Mishra, C., Koudas, N.: Interactive query refinement. In: Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology. EDBT '09, New York, NY, USA, ACM (2009) 862–873

[148] Md. Saiful, I., Chengfei, L., Zhou, R.: A framework for query refinement with user feedback. Journal of Systems and Software **86**(6) (June 2013) 1580–1595

[149] Md. Saiful, Chengfei, L., Zhou, R.: On modeling query refinement by capturing user intent through feedback. In: Proceedings of the Twenty-Third Australasian Database Conference - Volume 124. ADC '12, Darlinghurst, Australia, Australia, Australian Computer Society, Inc. (2012) 11–20

[150] Stefanidis, K., Koutrika, G., Pitoura, E.: A survey on representation, composition and application of preferences in database systems. ACM Trans. Database Syst. **36**(3) (August 2011) 19:1–19:45

[151] Koutrika, G., Ioannidis, Y.: Personalized queries under a generalized preference model. In: Proceedings of the 21st International Conference on Data Engineering. ICDE '05, Washington, DC, USA, IEEE Computer Society (2005) 841–852

[152] Bing, L., Lam, W., Wong, T.L.: Using query log and social tagging to refine queries based on latent topics. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management. CIKM

’11, New York, NY, USA, ACM (2011) 583–592

[153] Paik, J.H., Pal, D., Parui, S.K.: Incremental blind feedback: An effective approach to automatic query expansion. Journal ACM Transactions on Asian Language Information Processing (TALIP) **13**(3) (October 2014) 13:1–13:22

[154] Viviani, M., Bennani, N., Egyed-Zsigmond, E.: A survey on user modeling in multi-application environments. In: Third International Conference on Advances in Human-Oriented and Personalized Mechanisms, Technologies and Services (CENTRIC), 2010. CENTRIC’10, IEEE (2010) 111–116

[155] Maja, P., Alex, P., Anton, N., Thomas, S. In: Human Computing and Machine Understanding of Human Behavior: A Survey. Springer Berlin Heidelberg

[156] Kobsa, A.: Generic user modeling systems. Journal of User Modeling and User-Adapted Interaction **11**(1) (March 2001) 46–63

[157] Li, P., Tu, M., Yen, I.L., Xia, Z.: Preference update for e-commerce applications: Model, language, and processing. Journal of Electronic Commerce Research **7**(1) (March 2007) 17–44

[158] Jelassi, M., Ben Yahia, S., Mephu Nguifo, E.: A personalized recommender system based on users’ information in folksonomies. In: Proceedings of the 22Nd International Conference on World Wide Web. WWW ’13 Companion, New York, NY, USA, ACM (2013) 1215–1224

[159] Md. Saiful, Chengfei, L., Zhou, R.: Flexiq: A flexible interactive querying framework by exploiting the skyline operator. Journal of Systems and Software **97** (2014) 97 – 117

[160] Chandramouli, K., Kliegr, T., Nemrava, J., Svatek, V., Izquierdo, E.: Query refinement and user relevance feedback for contextualized image retrieval. In: 5th International Conference on Visual Information Engineering, 2008. VIE 2008. (July 2008) 453–458

[161] Raymond, C., Ada, W., Jian, P., Yip, S., Tai, W., Yubao, L.: Efficient skyline querying with variable user preferences on nominal attributes. CoRR **abs/0710.2604** (2007)

[162] Cao, H., Qi, Y., Candan, K.S., Sapino, M.L.: Feedback-driven result ranking and query refinement for exploring semi-structured data collections. In: Proceedings of the 13th International Conference on Extending Database Technology. EDBT ’10, New York, NY, USA, ACM (2010) 3–14

[163] Dongwon, L.: Querry Relaxation for XML Model. Theses, University of California (2002)

[164] Alilaouar, A.: Contribution à l’interrogation flexible de données semi-structurées. Theses, Université Paul Sabatier de Toulouse (Octobre 2007)

[165] Chu, W.W., Yang, H., Chiang, K., Minock, M., Chow, G., Larson, C.: Cobase: A scalable and extensible cooperative information system. Journal of Intelligent Information Systems **6**(2) (1996) 223–259

[166] Fayech, I., Ounalli, H.: Expansion de requêtes sql par une ontologie de domaine. In Yahia, S.B., Petit, J.M., eds.: ECG, Revue des Nouvelles Technologies de l’Information. Volume RNTI-E-19 of Revue des Nouvelles Technologies de l’Information., Cépaduès-Éditions (2010) 495–500

[167] Koudas, N., Li, C., Tung, A.K.H., Vernica, R.: Relaxing join and selection queries. In: Proceedings of the 32Nd International Conference on Very Large Data Bases. VLDB ’06, VLDB Endowment (2006) 199–210

[168] Harispe, S.: Knowledge-based Semantic Measures: From Theory to Applications. Theses, Université Montpellier 2 (Avril 2014)

[169] Bernstein, A., Kaufmann, E., Kiefer, C., Bürki, C.: SimPack: A Generic Java Library for Similiarity Measures in Ontologies. Technical report, Department of Informatics, University of Zurich (2005)

[170] Harispe, S., Ranwez, S., Janaqi, S., Montmain, J.: The semantic measures library and toolkit: fast computation of semantic similarity and relatedness using biomedical ontologies. Bioinformatics **30**(5) (2014) 740–742

[171] Francesco, R.: Travel recommender systems. IEEE Intelligent Systems (November 2002) 55–57

[172] Borrás, J., Moreno, A., Valls, A.: Review: Intelligent tourism recommender systems: A survey. Expert Syst. Appl. **41**(16) (November 2014) 7370–7389

[173] Bobadilla, J., Ortega, F., Hernando, A., GutiéRrez, A.: Recommender systems survey. Knowledge-Based Systems **46** (July 2013) 109–132

[174] Mirzadeh, N., Ricci, F., Bansal, M.: Supporting User Query Relaxation in a Recommender System. In: E-Commerce and Web Technologies: 5th International Conference, EC-Web 2004, Zaragoza, Spain, August 31-September 3, 2004. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2004) 31–40

[175] Herzog, D., Wörndl, W.: A travel recommender system for combining multiple travel regions to a composite trip. In: Proceedings of the 1st Workshop on New Trends in Content-based Recommender Systems co-located with the 8th ACM Conference on Recommender Systems, CBRecSys@RecSys 2014, Foster City, Silicon Valley, California, USA, October 6, 2014. (2014) 42–48

[176] Moreno, A., Valls, A., Isern, D., Marin, L., BorríS, J.: Sigtur/e-destination: Ontology-based personalized recommendation of tourism and leisure activities. Eng. Appl. Artif. Intell. **26**(1) (January 2013) 633–651

[177] Motro, A.: Vague: A user interface to relational databases that permits vague queries. ACM Trans. Inf. Syst. **6**(3) (July 1988) 187–214

[178] Bosc, P., Pivert, O.: SQLf: a relational database language for fuzzy querying. IEEE Transactions on Fuzzy Systems **3**(1) (Feb 1995) 1–17

[179] Bosc, P., Rocacher, D., Liétard, L., Pivert, O.: Gradualité et imprécision dans les bases de données: ensembles flous, requêtes flexibles et interrogation de données mal connues. Technosup (Paris). Ellipses (2004)

[180] Abbaci, K.: Contribution à l'interrogation flexible et personnalisée d'objets complexes modélisés par des graphes. Theses, Université de Rennes 1 (2013)

[181] Dubois, D., Prade, H.: Using fuzzy sets in flexible querying: Why and how? In: Flexible Query Answering Systems, Boston, MA, Springer US (1997) 45–60

[182] Sugeno, M.: Theory of fuzzy integrals and its applications. Thesis, Tokyo Institute of Technology (1974)

[183] Choquet, G.: Theory of capacities. Annales de l'institut Fourier **5** (1954) 131–295

[184] Tamani, N., Liétard, L., Rocacher, D.: Bipolar SQLf: A Flexible Querying Language for Relational Databases. In: Flexible Query Answering Systems: 9th International Conference, FQAS 2011, Ghent, Belgium, October 26-28, 2011 Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2011) 472–484

[185] Dubois, D., Prade, H.: Handling Bipolarity Queries in Fuzzy Information Processing. In: Handbook of Research on Fuzzy Information Processing in Databases. (2008) 94–114

[186] Smits, G., Pivert, O., Girault, T.: PostgreSQLf : Un système d'interrogation floue. In: Journées Bases de Données Avancées (BDA'12) - Session démo. (October 2012)

[187] Galindo, J., Carrasco, R.A., Almagro, A.M.: Fuzzy quantifiers with and without arguments for databases: Definition, implementation and application to fuzzy dependencies. In: 12th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, IPMU'O8. (June 2008) 227–234

[188] Kießling, W.: Foundations of preferences in database systems. In: Proceedings of the 28th International Conference on Very Large Data Bases. VLDB '02, VLDB Endowment (2002) 311–322

[189] Kießling, W., Köstler, G.: Preference sql: Design, implementation, experiences. In: Proceedings of the 28th International Conference on Very Large Data Bases. VLDB '02, VLDB Endowment (2002) 990–1001

[190] Arvanitis, A., Koutrika, G.: Prefdb: Bringing preferences closer to the dbms. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. SIGMOD '12, New York, NY, USA, ACM (2012) 665–668

[191] Mindolin, D., Chomicki, J.: Preference elicitation in prioritized skyline queries. The VLDB Journal **20**(2) (April 2011) 157–182

[192] Chomicki, J., Ciaccia, P., Meneghetti, N.: Skyline queries, front and back. SIGMOD Rec. **42**(3) (October 2013) 6–18

[193] Belkasmi, D., Hadjali, A., Azzoune, H.: Relaxation des requêtes skyline : Une approche centrée utilisateur. In: 16ème Journées Francophones Extraction et Gestion des Connaissances, EGC 2016, 18-22 Janvier 2016, Reims, France. Volume E-30., Hermann-Éditions (2016) 357–362

[194] Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: Proceedings of the 17th International Conference on Data Engineering, Washington, DC, USA, IEEE Computer Society (2001) 421–430

[195] Hadjali, A., Pivert, O., Prade, H.: On different types of fuzzy skylines. In: Proceedings of the 19th International Conference on Foundations of Intelligent Systems. ISMIS'11, Berlin, Heidelberg, Springer-Verlag (2011) 581–591

[196] Inoue, K., Wiese, L.: Generalizing conjunctive queries for informative answers. In: Flexible Query Answering Systems: 9th International Conference, FQAS 2011, Ghent, Belgium, October 26-28, 2011 Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2011) 1–12

[197] Bakhtyar, M., Wiese, L., Inoue, K., Dang, N.: Filtering of unrelated answers in a cooperativequery answering system. In: Proceedings of the First International Conference on Advanced Data and Information Engineering, DaEng 2013, Kuala Lumpur, Malaysia, December 16-18, 2013. (2013) 461–470

[198] Mottin, D., Marascu, A., Roy, S.B., Das, G., Palpanas, T., Velegrakis, Y.: A holistic and principled approach for the empty-answer problem. The VLDB Journal (2016) 1–26

[199] Mottin, D., Marascu, A., Roy, S.B., Das, G., Palpanas, T., Velegrakis, Y.: A probabilistic optimization framework for the empty-answer problem. Proc. VLDB Endow. **6**(14) (September 2013) 1762–1773

[200] Godfrey, P.: Minimization in cooperative response to failing database queries. International Journal of Cooperative Information Systems **06**(02) (1997) 95–149

[201] Bosc, P., Hadjali, A., Pivert, O.: Incremental controlled relaxation of failing flexible queries. J. Intell. Inf. Syst. **33**(3) (December 2009) 261–283

[202] Pivert, O., Smits, G.: How to Efficiently Diagnose and Repair Fuzzy Database Queries that Fail. In: Fifty Years of Fuzzy Logic and its Applications. Volume 326 of Studies in Fuzziness and Soft Computing., Springer (2015) 499–517

[203] Gaasterland, T., Godfrey, P., Minker, J.: Relaxation as a platform for cooperative answering. Journal of Intelligent Information Systems **1**(3) (1992) 293–321

[204] Joon Ho, L., Myoung Ho, K., Yoon Joon, L.: Information retrieval based on conceptual distance in is-a hierarchies. Journal of Documentation **49**(2) (1993) 188–207

[205] Chu, W.W., Chen, Q.: Neighborhood and associative query answering. Journal of Intelligent Information Systems **1**(3) (1992) 355–382

[206] Jannach, D.: Fast computation of query relaxations for knowledge-based recommenders. AI Commun. **22**(4) (December 2009) 235–248

[207] Jannach, D.: Techniques for fast query relaxation in content-based recommender systems. In: Proceedings of the 29th Annual German Conference on Artificial Intelligence. KI'06, Berlin, Heidelberg, Springer-Verlag (2007) 49–63

[208] McSherry, D.: Completeness Criteria for Retrieval in Recommender Systems. In: Advances in Case-Based Reasoning: 8th European Conference, ECCBR 2006 Fethiye, Turkey, September 4-7, 2006 Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2006) 9–29

[209] Mcsherry, D.: Retrieval failure and recovery in recommender systems. Artificial Intelligence Review **24**(3) (2005) 319–338

[210] McSherry, D. In: Incremental Relaxation of Unsuccessful Queries. Springer Berlin Heidelberg, Berlin, Heidelberg (2004) 331–345

[211] De Calmès, M., Prade, H., Sèdes, F.: Flexible querying of semistructured data: A fuzzy-set-based approach. International Journal of Intelligent Systems **22**(7) (2007) 723–737

[212] Jiang, T., Wang, L., Zhang, K.: Alignment of trees - an alternative to tree edit. In: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching. CPM '94, London, UK, UK, Springer-Verlag (1994) 75–86

[213] Chu, W.W., Liu, S.: Coxml: Cooperative XML query answering? In: Wiley Encyclopedia of Computer Science and Engineering. (2008)

[214] Liu, J., Yan, D.L.: Answering approximate queries over xml data. IEEE Transactions on Fuzzy Systems **24**(2) (April 2016) 288–305

[215] Kanza, Y., Sagiv, Y.: Flexible queries over semistructured data. In: Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems. PODS '01, New York, USA (2001) 40–51

[216] Hurtado, C.A., Poulovassilis, A., Wood, P.T. In: A Relaxed Approach to RDF Querying. Springer Berlin Heidelberg, Berlin, Heidelberg (2006) 314–328

[217] Hurtado, C.A., Poulovassilis, A., Wood, P.T.: Query relaxation in rdf. Journal on Data Semantics X (2008) 31–61

[218] Poulovassilis, A., Wood, P.T. In: Combining Approximation and Relaxation in Semantic Web Path Queries. Springer Berlin Heidelberg, Berlin, Heidelberg (2010) 631–646

[219] Thompson, K.: Programming techniques: Regular expression search algorithm. Commun. ACM **11**(6) (June 1968) 419–422

[220] De Virgilio, R., Maccioni, A., Torlone, R.: Approximate querying of rdf graphs via path alignment. Distrib. Parallel Databases **33**(4) (December 2015) 555–581

[221] De Virgilio, R., Maccioni, A., Torlone, R.: A similarity measure for approximate querying over rdf data. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops. EDBT '13, New York, NY, USA, ACM (2013) 205–213

[222] Cheng, J., Ma, Z. M.and Yan, L. In: f-SPARQL: A Flexible Extension of SPARQL. Springer Berlin Heidelberg, Berlin, Heidelberg (2010) 487–494

[223] Ruizhe, M., Xiangyue, J., Jingwei, C., Rafal A., A.: Sparql queries on rdf with fuzzy constraints and preferences. (2016) 183–195

[224] Hogan, A., Mellotte, M., Powell, G., Stampouli, D.: Towards fuzzy query-relaxation for rdf. In Simperl, E., Cimiano, P., Polleres, A., Corcho, O., Presutti, V., eds.: The Semantic Web: Research and Applications: 9th Extended Semantic Web Conference, ESWC 2012, Heraklion, Crete, Greece, May 27-31, 2012. Proceedings, Berlin, Heidelberg, Springer Berlin Heidelberg (2012) 687–702

[225] Elbassuoni, S., Ramanath, M., Weikum, G.: Query relaxation for entity-relationship

search. In: Proceedings of the 8th Extended Semantic Web Conference on The Semanic Web: Research and Applications - Volume Part II. ESWC'11, Berlin, Heidelberg, Springer-Verlag (2011) 62–76

[226] Kiefer, C., Bernstein, A., Stocker, M.: The Fundamentals of iSPARQL: A Virtual Triple Approach for Similarity-Based Semantic Web Tasks. In: The Semantic Web: 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference, ISWC 2007 + ASWC 2007, Busan, Korea, November 11-15, 2007. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2007) 295–309

[227] Bernstein, A., Kiefer, C.: Imprecise rdql: Towards generic retrieval in ontologies using similarity joins. In: Proceedings of the 2006 ACM Symposium on Applied Computing. SAC '06, New York, NY, USA, ACM (2006) 1684–1689

[228] Kiefer, C., Bernstein, A., Lee, H.J., Klein, M., Stocker, M.: Semantic Process Retrieval with iSPARQL. In: The Semantic Web: Research and Applications: 4th European Semantic Web Conference, ESWC 2007, Innsbruck, Austria, June 3-7, 2007. Proceedings. Springer Berlin Heidelberg, Berlin, Heidelberg (2007) 609–623

[229] Corby, O., Dieng-Kuntz, R., Faron-Zucker, C., Gandon, F.: Ontology-based approximate query processing for searching the semantic web with corese. Research Report RR-5621, INRIA (2006)

[230] Huang, H., Liu, C.: Query relaxation for star queries on rdf. In: Proceedings of the 11th International Conference on Web Information Systems Engineering. WISE'10, Berlin, Heidelberg, Springer-Verlag (2010) 376–389

[231] Dolog, P., Stuckenschmidt, H., Wache, H.: Robust query processing for personalized information access on the semantic web. In: Flexible Query Answering Systems, 7th International Conference, FQAS 2006, Milan, Italy, June 7-10, 2006, Proceedings. (2006) 343–355

[232] Fokou, G., Jean, S., Hadjali, A.: Endowing semantic query languages with advanced relaxation capabilities. In: Foundations of Intelligent Systems - 21st International Symposium, ISMIS 2014, Roskilde, Denmark, June 25-27, 2014. Proceedings. (2014) 512–517

[233] Fokou, G., Jean, S., Hadjali, A.: Endowing semantic query languages with advanced relaxation capabilities. Research Report 1.0, LIAS/ISAE-ENSMA (2014)

[234] Lotfi Aliasker, Z.: Fuzzy sets. Information and Control **8**(3) (1965) 338 – 353

[235] Hadjali, A., Dubois, D., Prade, H.: Qualitative reasoning based on fuzzy relative orders of magnitude. IEEE Transactions on Fuzzy Systems **11**(1) (February 2003) 9–23

[236] Dubois, D., Prade, H., eds.: Fundamentals of Fuzzy Sets. The Handbooks of Fuzzy Sets Series. Kluwer, Boston, Mass. (2000)

[237] Fokou, G., Jean, S., Hadjali, A., Baron, M.: Qars: A user-friendly graphical tool for semantic query design and relaxation. In: Proceedings of the 18th International Conference on Extending Database Technology, EDBT 2015, Brussels, Belgium, March 23-27, 2015. (2015) 553–556

[238] Hondjack, D.: Explicitation de la sémantique dans les base de données : Base de données à base ontologique et le modèle OntoDB. Theses, Université de Poitiers (Mai 2007)

[239] Duan, S., Kementsietsidis, A., Srinivas, K., Udrea, O.: Apples and oranges: A comparison of rdf benchmarks and real rdf datasets. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data. SIGMOD '11, New York, NY, USA, ACM (2011) 145–156

[240] Mortici, C.: Ramanujan formula for the generalized stirling approximation. Applied Mathematics and Computation **217**(6) (2010) 2579 – 2585

[241] Cyganiak, R.: A relational algebra for SPARQL. HP-Labs Technical Report, HPL-2005-170, `http:`

//www.hpl.hp.com/techreports/2005/HPL-2005-170.html (2005)

[242] Hose, K., Vlachou, A.: A survey of skyline processing in highly distributed environments. The VLDB Journal **21**(3) (June 2012) 359–384

[243] Huang, H., Liu, C., Zhou, X.: Computing relaxed answers on rdf databases. In: Proceedings of the 9th International Conference on Web Information Systems Engineering. WISE '08, Berlin, Heidelberg, Springer-Verlag (2008) 163–175

[244] Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with roaring bitmaps. CoRR **abs/1402.6407** (2014)

[245] Bizer, C., Schultz, A.: The berlin SPARQL benchmark. International Journal Semantic Web Information System **5**(2) (2009) 1–24

[246] Yahya, M., Berberich, K., Ramanath, M., Weikum, G.: Exploratory querying of extended knowledge graphs. PVLDB **9**(13) (2016) 1521–1524

[247] Stocker, M., Seaborne, A., Bernstein, A., Kiefer, C., Reynolds, D.: Sparql basic graph pattern optimization using selectivity estimation. In: Proceedings of the 17th International Conference on World Wide Web. WWW '08, New York, NY, USA, ACM (2008) 595–604

[248] Neumann, T., Moerkotte, G.: Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In: Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany. (2011) 984–994

[249] Dividino, R., Sizov, S., Staab, S., Schueler, B.: Querying for provenance, trust, uncertainty and other meta knowledge in rdf. Web Semant. **7**(3) (September 2009) 204–219

[250] Hartig, O.: Querying trust in rdf data with tsparql. In: Proceedings of the 6th European Semantic Web Conference on The Semantic Web: Research and Applications. ESWC 2009 Heraklion, Berlin, Heidelberg, Springer-Verlag (2009) 5–20

[251] Bizer, C., Heath, T., Berners-Lee, T.: Linked data - the story so far. International Journal on Semantic Web and Information Systems, (IJSWIS) **5**(3) (2009) 1–22

[252] Muhammad, S.: Efficient Source Selection For SPARQL Endpoint Query Federation. Theses, Université de Leipzig (Mai 2016)

[253] B. R. Kuldeep, R., P. Sreenivasa, K.: Efficient approximate sparql querying of web of linked data. In: Proceedings of the 6th International Conference on Uncertainty Reasoning for the Semantic Web - Volume 654. URSW'10, Aachen, Germany, Germany, CEUR-WS.org (2010) 37–48

[254] Hartig, O.: Querying a Web of Linked Data: Foundations and Query Execution. Theses, Université Humboldt de Berlin (Juillet 2014)

[255] Hartig, O., Pérez, J.: LDQL: A query language for the web of linked data (extended version). CoRR **abs/1507.04614** (2015)

[256] Fokou, G., Jean, S., Hadjali, A., Baron, M.: Handling failing rdf queries: from diagnosis to relaxation. Knowledge and Information Systems (KAIS) (2016) 1–29

[257] Fokou, G., Jean, S., Hadjali, A., Baron, M.: Cooperative techniques for SPARQL query relaxation in RDF databases. In: The Semantic Web. Latest Advances and New Domains - 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Proceedings. Volume 9088 of Lecture Notes in Computer Science., Springer (2015) 237–252

[258] Fokou, G., Jean, S., Hadjali, A., Baron, M.: RDF query relaxation strategies based on failure causes. In: The Semantic Web. Latest Advances and New Domains - 13th International Conference, ESWC 2016, Heraklion, Crete, Greece, May 29 - June 2, 2016, Proceedings. Volume 9678 of Lecture Notes in Computer Science., Springer (2016) 439–454 Best Paper Award.

Annexe A

Requêtes sur HotelBase

Q_1 :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?B rdf:type BedAndBreakfast.
  ?B name ?name.
  ?B price ?price.
  ?B nbstart ?nbstart.
  ?B cityname 'Belgrade'.
FILTER (?nbstar=5 && ?price <=29)}
```

Q_2 :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?I rdf:type Inn.
  ?I name ?name.
  ?I price ?price.
  ?I nbstart ?nbstart.
  ?I cityname 'Shanghai'.
FILTER (?nbstar=5 && ?price <=15)}
```

Q_3 :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?M rdf:type Motel.
  ?M name ?name.
  ?M price ?price.
  ?M nbstart ?nbstart.
  ?M cityname 'Istanbul'.
FILTER (?nbstar=5 && ?price <=80)}
```

Q_4 :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?R rdf:type Resort.
  ?R name ?name.
  ?R price ?price.
  ?R nbstart ?nbstart.
  ?R cityname 'California'.
FILTER (?nbstar=5 && ?price <=85)}
```

Q_5 :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?M rdf:type Motel.
  ?M name ?name.
  ?M price ?price.
  ?M nbstart ?nbstart.
  ?M cityname 'Istanbul'.
FILTER ((?nbstar<=4)}
```

Q_6 :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?R rdf:type Resort.
  ?R name ?name.
  ?R price ?price.
  ?R nbstart ?nbstart.
  ?R cityname 'Istanbul'.
FILTER (?nbstar>=8)}
```

Q_7 :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?V rdf:type VacationRental.
  ?V name ?name.
  ?V price ?price.
  ?V nbstart ?nbstart.
  ?V cityname 'Istanbul'.
FILTER (?nbstar=4 && ?price <90)}
```

Q_8 :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?I rdf:type Inn.
  ?I name ?name.
  ?I price ?price.
  ?I nbstart ?nbstart.
  ?I cityname 'Shanghai'.
FILTER (?nbstar=3 && ?price <=20)}
```

Q_9 :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?R rdf:type Resort.
  ?R name ?name.
  ?R price ?price.
  ?R nbstart ?nbstart.
  ?R cityname 'Pennsylvania'.
FILTER (?nbstar=3 && ?price<80)}
```

Q_{10} :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?M rdf:type Motel.
  ?M name ?name.
  ?M price ?price.
  ?M nbstart ?nbstart.
  ?M cityname 'Nanjing'.
FILTER (?nbstar=3 && ?price>=80)}
```

Q_{11} :

```
SELECT ?name ?nbstart ?price ?cityname
WHERE {
  ?A rdf:type Apartment.
  ?A name ?name.
  ?A price ?price.
  ?A nbstart ?nbstart.
  ?A cityname 'Kolobrzeg'.
FILTER (?nbstar=4 && ?price<50)}
```

Annexe **B**

Algorithmes

Algorithm 9: O-MBS : Optimisation de la stratégie de relaxation MBS

Relax(Q**,** $mfs(Q)$**,** $\mathbb{D}$**,** k**)**

inputs : Q : requête qui échoue ; $mfs(Q)$: ensemble des MFS de Q ;
$\mathbb{D}$: BD-RDF ; k : nombre de réponses attendues ;

output: Res : ensemble des TOP-k réponses alternatives de la requête Q;

1 $Res \leftarrow \emptyset$;
2 $RQ \leftarrow \emptyset$; // Requêtes relaxées ordonnées par similarité
3 $dmfs \leftarrow mfs(Q)$;// Initialisation des MFS découvertes avec $mfs(Q)$
4 $RQ.enqueue(Q)$; marqué Q comme échouée ;
5 **while** $RQ \neq \emptyset \wedge |Res| < k$ **do**
6 $Q_i = RQ.dequeue()$;
7 **if** $\exists Q^* \in dmfs$ *tel que* $Q^* \subseteq Q_i$ **then**
8 marqué Q_i comme échouée;
9 **else** // Q_i ne contient pas des MFS découvertes ($mfs^{\uparrow Q_i}(Q) \cap dmfs = \emptyset$)
10 **if** $[[Q_i]]_{\mathbb{D}} = \emptyset$ **then** // Q_i est exécutée et échoue
11 **foreach** $M_{Q_i} \in mfs^{\uparrow Q_i}(Q)$ **do**
12 **if** $[[M_{Q_i}]]_{\mathbb{D}} = \emptyset$ **then** // M_{Q_i} échoue
13 $dmfs \leftarrow dmfs \cup \{M_{Q_i}\}$; // Ajout de M_{Q_i} dans les MFS découvertes
14 **else** // Q_i n'échoue pas
15 $Res \leftarrow Res \cup [[Q_i]]_{\mathbb{D}}$;
16 **foreach** *patron de triplet* $t_k^{(i_k)} \in Q_i$ *tel que* $i_k < nbRel(t_k)$ **do**
17 $Q_c \leftarrow t_1^{(i_1)} \wedge \cdots \wedge t_k^{(i_{k+1})} \wedge \cdots \wedge t_n^{(i_n)}$; // un nœud fils de Q_i
18 **if** Q_c *n'est pas marquée comme insérée* **then** // donc pas explorée
19 $RQ.enqueue(Q_c)$;
20 marqué Q_c comme insérée;
21 **return** Res;

Algorithm 10: F-MBS : Recherche de toutes les MFS d'une requête relaxée

Relax(Q**,** $mfs(Q)$**,** $\mathbb{D}$**,** k**)**
inputs : Q : requête qui échoue ;
$mfs(Q)$: ensemble des MFS de Q ;
$\mathbb{D}$: BD-RDF ;
k : nombre de réponses attendues ;
output: Res : ensemble des TOP-k réponses alternatives de la requête Q;

$Res \leftarrow \emptyset$;
$RQ \leftarrow \emptyset$; // Requêtes relaxées ordonnées par similarité
$dmfs \leftarrow mfs(Q)$;// Initialisation des MFS découvertes avec $mfs(Q)$
$RQ.enqueue(Q)$;
marqué Q comme échouée ;
while $RQ \neq \emptyset \wedge |Res| < k$ **do**
  $Q_i = RQ.dequeue()$;
  if $\exists Q^* \in dmfs$ *tel que* $Q^* \subseteq Q_c$ **then**
    marqué Q_c comme échouée;
  else // Q_i ne contient pas de MFS
    if $[[Q_i]]_{\mathbb{D}} = \emptyset$ **then** // Q_i est exécutée et échoue
      $dmfs(Q_i) \leftarrow \emptyset$; // Ensemble des MFS de Q_i
      foreach $M_{Q_i} \in mfs^{\uparrow Q_i}(Q)$ **do**
        if $[[M_{Q_i}]]_{\mathbb{D}} = \emptyset$ **then** // M_{Q_i} échoue, MFS non réparée
          $dmfs(Q_i) \leftarrow dmfs(Q_i) \cup \{M_{Q_i}\}$; // Ajout de M_{Q_i} dans les MFS de Q_i
      if $|dmfs(Q_i)| \neq |mfs(Q)|$ **then** // $dmfs(Q_i)$ ne contient pas toutes les MFS
        $Q_s \leftarrow \emptyset$;
        foreach $t_k^{(i_k)} \in Q_i$ *tel que* $\forall M_Q^{\uparrow Q_i} \in mfs(Q_i), t_k^{(i_k)} \in M_Q^{\uparrow Q_i}$ **do**
          $Q_s \leftarrow t_k^{(i_k)} \wedge Q_s$;
        $(newMFS(Q_i), newXSS(Q_i) \leftarrow OptimizedLBA(Q_i, dmfs(Q_i), Q_s, \mathbb{D})$;
        $dmfs(Q_i) \leftarrow dmfs(Q_i) \cup newMFS(Q_i)$;
      $dmfs \leftarrow dmfs \cup dmfs(Q_i)$; // Ajoute dans $dmfs$ toutes les MFS de Q_i.
    else // Q_i n'échoue pas
      $Res \leftarrow Res \cup [[Q_i]]_{\mathbb{D}}$;
  foreach *patron de triplet* $t_k^{(i_k)} \in Q_i$ *tel que* $i_k < nbRel(t_k)$ **do**
    $Q_c \leftarrow t_1^{(i_1)} \wedge \cdots \wedge t_k^{(i_{k+1})} \wedge \cdots \wedge t_n^{(i_n)}$; // un nœud fils de Q_i
    if Q_c *n'est pas marquée comme insérée* **then** // donc pas explorée
      $RQ.enqueue(Q_c)$;
      marqué Q_c comme insérée;
return Res;

Annexe C

Publications

C.1 Revues internationales

[256] : Géraud Fokou, Stéphane Jean, Allel Hadjali Mickaël Baron. Handling failing RDF queries: from diagnosis to relaxation. In Knowledge and Information Systems (KAIS) Journal. April 2016. Pages 1–29. Springer.

C.2 Conférences internationales

[232] : Géraud Fokou, Stéphane Jean, Allel Hadjali. Endowing Semantic Query Languages with Advanced Relaxation Capabilities. In Foundations of Intelligent Systems - Proceedings of 21st International Symposium, ISMIS, Roskilde, Denmark, June 25-27, 2014. Pages 512–517.

[237] : Géraud Fokou, Stéphane Jean, Allel Hadjali Mickaël Baron. QaRS: A User-Friendly Graphical Tool for Semantic Query Design and Relaxation. In Proceedings of the 18th International Conference on Extending Database Technology, EDBT, Brussels, Belgium, March 23-27, 2015.Pages 553–556.

[257] : Géraud Fokou, Stéphane Jean, Allel Hadjali Mickaël Baron. Cooperative Techniques for SPARQL Query Relaxation in RDF Databases. In The Semantic Web. Latest Advances and New Domains - Proceedings of the 12th European Semantic Web Conference, ESWC 2015, Portoroz, Slovenia, May 31 - June 4, 2015. Pages 237–252. Series Lecture Notes in Computer Science. Springer.

[258] : Géraud Fokou, Stéphane Jean, Allel Hadjali Mickaël Baron. RDF Query Relaxation Strategies Based on Failure Causes. In The Semantic Web. Latest Advances and New Domains - Proceedings of the 13th International Conference, ESWC, Heraklion, Crete, Greece, May 29 - June 2, 2016. Pages 439–454. Series Lecture Notes in Computer Science. Springer. Best Paper Award.

C.3 Conférences nationales

Géraud Fokou, Framework pour la relaxation des requêtes dans les bases de données du web sémantiques, Actes du 7ième Forum jeunes chercheurs, 32ième congrès INFORSID 2014, Lyon, France, 2014.

Géraud Fokou, Stéphane Jean, Allel Hadjali, Mickael Baron, Cooperative Techniques for SPARQL Query Relaxation in RDF Databases, 31ème Conférence des Bases de données Avancées : gestion de données - Principes, Technologies et Application (BDA'15), Porquerolles, Toulon, France, 2015.

Géraud Fokou, Stéphane Jean, Allel Hadjali, Mickael Baron, RDF Query Relaxation Strategies Based on Failure Causes, 32ème Conférence des Base de données Avancées : gestion de données - Principes, Technologies et Application (BDA'16), Futuroscope, Poitiers, France, 2016.

Table des figures

Liste des tableaux

Glossaire

BD
Bases de données

BD-RDF
Bases de Données RDF

BFS
Best-First Search

EADS
European Aeronautic Defence and Space

EMBL
European Molecular Biology Laboratory

F-MBS
Full MFS-Based Search

GOA
Gene Ontology Annotation

GQBE
Graph Query By Example

IBM
International Business Machines Corporation

IEC
International Electrotechnical Commission

INRIA
Institut National de Recherche en Informatique et en Automatique

ISO
International Organization for Standardization

LBA
Lattice-Based Approach

LIAS
Laboratoire d'Informatique et d'Automatique pour les Systèmes

LUBM
Lehigh University BenchMark

MBA
Matrix-Based Approach

MBS
MFS-Based Search

MFS
Minimal Failing Subquery

O-MBS
Optimized MFS-Based Search

OntoQL
Ontology Query Language

OWL
Ontology Web Language

PLIB
Parts LIBrary

QARS
Query And Relax System

QBE
Query By Example

QFE
Query From Example

QL
Query Language

RDF
Ressource Description Framework

RDFS
Ressource Description Framework Schema

RDQL
RDF Data Query Language

RIF
Rules Interchanged Format

RQL
RDF Query Language

SeRQL
Sesame RDF Query Language

SGBD
Système de Gestion des Bases de Données

SGBDR
Systèmes de Gestion des Bases de Données Relationnelles

SIB
Swiss Institute of Bioinformatics

SPARQL
Simple Protocol And RDF Query Language

SQL
Structured Query Language

URI
Universal Ressource Identifier

W3C
World Wide Web Consortium

WIMMICS
Web-Instrumented Man-Machine Interactions, Communities and Semantics

XML
eXtensible Markup Language

XQBE
XML Query By Example

XSD
XML Schema Definition Language

XSS
maXimal Succeeding Subquery

Abstract

Ontology (or Knowledge base) is a formal representation of knowledge as entities and facts related to these entities. In the past years, several ontologies have been developed in academic and industrial contexts. They are generally defined with RDF language and querying with SPARQL language. A partial knowledge of instances and schema of ontology may lead user to execute queries that result in empty answers, considered as unsatisfactory. Among cooperative querying techniques which have been developed to solve the problem of empty answers, query relaxation technique is the well-known and used. It aims at weakening the conditions expressed in the original query to return alternative answers to the user. Existing work on relaxation of SPARQL queries we suffer from many drawbacks: (1) they do not allow defining in precise way the relaxation to perform with the ability to control the relaxation process (2) they do not identify the causes of failure of the request expressed by the user and (3) they do not include interactive tools to better exploit the relaxation techniques proposed. To address these limitations, this thesis proposes an advanced framework for query relaxation SPARQL. First, this framework includes a set of relaxation operators dedicated to SPARQL queries, to incrementally relax specific parts of the user request while controlling the relevance of the alternative responses returned w.r.t. to the user needs expressed in his request. Our framework also provides both several algorithms that identify the causes of failure of the user query and queries that are successful with a maximum number of conditions initially expressed in the failing request. This information allows the user to better understand why his request fails and execute queries that return non-empty alternative results. Finally, our framework offers intelligent relaxation strategies that rely on the causes of query failure. Such strategies reduce the execution time of the relaxation process compared to the traditional approach, which executes relaxed requests, based on their similarity to the user request, until a number of satisfactory alternative results is obtained. All contributions proposed in this framework were implemented and validated by experiments and scenarios based on the tests bench LUBM. They show the interest of our contributions w.r.t. the state of the art.

Keywords: Ontologies, Database searching, Ressource Description Framework (RDF), RDF Schema, SPARQL, Query relaxation methods, Non-responses, Empty answer problem, Failing causes.

Résumé

Une ontologie (ou base de connaissances) est une représentation formelle de connaissances sous la forme d'entités et de faits sur ces entités. Ces dernières années de nombreuses ontologies ont été développées dans des contextes académiques et industriels. Elles sont généralement définies à l'aide du langage formel RDF et interrogées avec le langage de requêtes SPARQL. Une connaissance partielle du contenu et de la structure d'une ontologie peut amener les utilisateurs à exécuter des requêtes qui retournent un résultat vide de réponses, considéré comme insatisfaisant. Parmi les techniques d'interrogation coopératives développées pour résoudre ce problème se trouve la technique de relaxation de requêtes. Elle consiste à affaiblir les conditions exprimées dans les requêtes pour retourner des résultats alternatifs à l'utilisateur. En étudiant les travaux existants sur la relaxation de requêtes SPARQL nous avons constaté qu'ils présentent plusieurs limitations : (1) ils ne permettent pas de définir précisément la relaxation à effectuer tout en offrant la possibilité de contrôler le processus de relaxation (2) ils n'identifient pas les causes réelles d'échec de la requête formulée par l'utilisateur et (3) ils n'intègrent pas d'outils interactifs pour mieux exploiter les techniques de relaxation proposées. Pour répondre à ces limitations, ce travail de thèse propose un framework pour la relaxation de requêtes SPARQL. Ce framework inclut un ensemble d'opérateurs de relaxation des requêtes SPARQL permettant de relaxer incrémentalement des parties précises de la requête utilisateur tout en contrôlant la pertinence des réponses alternatives retournées par rapport aux besoins exprimés par l'utilisateur dans sa requête. Notre framework propose également plusieurs algorithmes qui identifient les causes d'échec de la requête utilisateur et les requêtes qui réussissent (c'est-à-dire, qui ont des résultats) ayant un nombre maximal de conditions de la requête initialement exprimée. Ces informations permettent à l'utilisateur de mieux comprendre pourquoi sa requête échoue et d'exécuter des requêtes qui retournent des résultats alternatifs. Enfin, notre framework propose des stratégies de relaxation qui élargissent les conditions de la requête utilisateur en s'appuyant sur les causes d'échec de celle-ci. Ces stratégies permettent de réduire le temps d'exécution du processus de relaxation par rapport à l'approche classique, qui consiste à exécuter les requêtes relaxées, en fonction de leur similarité avec la requête utilisateur, jusqu'à l'obtention d'un nombre satisfaisant de résultats alternatifs. Les contributions proposées dans ce framework ont été implémentées et validées par des scénarios et expérimentations basés sur le banc d'essai LUBM. Ils montrent l'intérêt de nos contributions par rapport à l'état de l'art.

Mots-clés: Ontologies, Base de données–Interrogation, Ressource Description Framework (RDF), RDF Schéma, SPARQL, Relaxation de requêtes, Non-réponses, Problème des réponses vides, Causes d'échec.

La plus grande peur dans le monde est celle de l'opinion des autres. A partir du moment où vous n'avez plus peur de la foule, alors vous n'êtes plus un mouton vous devenez un lion. un grand rugissement émane de votre cœur. Le rugissement de la liberté.

Osho

Les grandes choses ne se réalisent pas par la force mais par la persévérance.

Samuel Johnson

La force, c'est de pouvoir regarder la douleur en face, lui sourire et continuer malgré ses coups à se tenir debout.

Conception d'un framework pour la relaxation de requêtes SPARQL

Présentée par :

Géraud FOKOU

Directeurs de Thèse :

Allel HADJALI et Stéphane JEAN

Résumé. Une ontologie (ou base de connaissances) est une représentation formelle de connaissances sous la forme d'entités et de faits sur ces entités. Ces dernières années de nombreuses ontologies ont été développées dans des contextes académiques et industriels. Elles sont généralement définies à l'aide du langage formel RDF et interrogées avec le langage de requêtes SPARQL. Une connaissance partielle du contenu et de la structure d'une ontologie peut amener les utilisateurs à exécuter des requêtes qui retournent un résultat vide de réponses, considéré comme insatisfaisant. Parmi les techniques d'interrogation coopératives développées pour résoudre ce problème se trouve la technique de relaxation de requêtes. Elle consiste à affaiblir les conditions exprimées dans les requêtes pour retourner des résultats alternatifs à l'utilisateur. En étudiant les travaux existants sur la relaxation de requêtes SPARQL nous avons constaté qu'ils présentent plusieurs limitations : (1) ils ne permettent pas de définir précisément la relaxation à effectuer tout en offrant la possibilité de contrôler le processus de relaxation (2) ils n'identifient pas les causes réelles d'échec de la requête formulée par l'utilisateur et (3) ils n'intègrent pas d'outils interactifs pour mieux exploiter les techniques de relaxation proposées. Pour répondre à ces limitations, ce travail de thèse propose un framework pour la relaxation de requêtes SPARQL. Ce framework inclut un ensemble d'opérateurs de relaxation des requêtes SPARQL permettant de relaxer incrémentalement des parties précises de la requête utilisateur tout en contrôlant la pertinence des réponses alternatives retournées par rapport aux besoins exprimés par l'utilisateur dans sa requête. Notre framework propose également plusieurs algorithmes qui identifient les causes d'échec de la requête utilisateur et les requêtes qui réussissent (c'est-à-dire, qui ont des résultats) ayant un nombre maximal de conditions de la requête initialement exprimée. Ces informations permettent à l'utilisateur de mieux comprendre pourquoi sa requête échoue et d'exécuter des requêtes qui retournent des résultats alternatifs. Enfin, notre framework propose des stratégies de relaxation qui élargissent les conditions de la requête utilisateur en s'appuyant sur les causes d'échec de celle-ci. Ces stratégies permettent de réduire le temps d'exécution du processus de relaxation par rapport à l'approche classique, qui consiste à exécuter les requêtes relaxées, en fonction de leur similarité avec la requête utilisateur, jusqu'à l'obtention d'un nombre satisfaisant de résultats alternatifs. Les contributions proposées dans ce framework ont été implémentées et validées par des scénarios et expérimentations basés sur le banc d'essai LUBM. Ils montrent l'intérêt de nos contributions par rapport à l'état de l'art.

Mots-clés : Ontologies, Base de données--Interrogation, Ressource Description Framework (RDF), RDF Schéma, SPARQL, Relaxation de requêtes, Non-réponses, Problème des réponses vides, Causes d'échec.

Résumé. Une ontologie (ou base de connaissances) est une représentation formelle de connaissances sous la forme d'entités et de faits sur ces entités. Ces dernières années de nombreuses ontologies ont été développées dans des contextes académiques et industriels. Elles sont généralement définies à l'aide du langage formel RDF et interrogées avec le langage de requêtes SPARQL. Une connaissance partielle du contenu et de la structure d'une ontologie peut amener les utilisateurs à exécuter des requêtes qui retournent un résultat vide de réponses, considéré comme insatisfaisant. Parmi les techniques d'interrogation coopératives développées pour résoudre ce problème se trouve la technique de relaxation de requêtes. Elle consiste à affaiblir les conditions exprimées dans les requêtes pour retourner des résultats alternatifs à l'utilisateur. En étudiant les travaux existants sur la relaxation de requêtes SPARQL nous avons constaté qu'ils présentent plusieurs limitations : (1) ils ne permettent pas de définir précisément la relaxation à effectuer tout en offrant la possibilité de contrôler le processus de relaxation (2) ils n'identifient pas les causes réelles d'échec de la requête formulée par l'utilisateur et (3) ils n'intègrent pas d'outils interactifs pour mieux exploiter les techniques de relaxation proposées. Pour répondre à ces limitations, ce travail de thèse propose un framework pour la relaxation de requêtes SPARQL. Ce framework inclut un ensemble d'opérateurs de relaxation des requêtes SPARQL permettant de relaxer incrémentalement des parties précises de la requête utilisateur tout en contrôlant la pertinence des réponses alternatives retournées par rapport aux besoins exprimés par l'utilisateur dans sa requête. Notre framework propose également plusieurs algorithmes qui identifient les causes d'échec de la requête utilisateur et les requêtes qui réussissent (c'est-à-dire, qui ont des résultats) ayant un nombre maximal de conditions de la requête initialement exprimée. Ces informations permettent à l'utilisateur de mieux comprendre pourquoi sa requête échoue et d'exécuter des requêtes qui retournent des résultats alternatifs. Enfin, notre framework propose des stratégies de relaxation qui élargissent les conditions de la requête utilisateur en s'appuyant sur les causes d'échec de celle-ci. Ces stratégies permettent de réduire le temps d'exécution du processus de relaxation par rapport à l'approche classique, qui consiste à exécuter les requêtes relaxées, en fonction de leur similarité avec la requête utilisateur, jusqu'à l'obtention d'un nombre satisfaisant de résultats alternatifs. Les contributions proposées dans ce framework ont été implémentées et validées par des scénarios et expérimentations basés sur le banc d'essai LUBM. Ils montrent l'intérêt de nos contributions par rapport à l'état de l'art.

Mots-clés : Ontologies, Base de données--Interrogation, Ressource Description Framework (RDF), RDF Schéma, SPARQL, Relaxation de requêtes, Non-réponses, Problème des réponses vides, Causes d'échec.

Abstract. ontology (or Knowledge base) is a formal representation of knowledge as entities and facts related to these entities. In the past years, several ontologies have been developed in academic and industrial contexts. They are generally defined with RDF language and querying with SPARQL language. A partial knowledge of instances and schema of ontology may lead user to execute queries that result in empty answers, considered as unsatisfactory. Among cooperative querying techniques which have been developed to solve the problem of empty answers, query relaxation technique is the well-known and used. It aims at weakening the conditions expressed in the original query to return alternative answers to the user. Existing work on relaxation of SPARQL queries we suffer from many drawbacks : (1) they do not allow defining in precise way the relaxation to perform with the ability to control the relaxation process (2) they do not identify the causes of failure of the request expressed by the user and (3) they do not include interactive tools to better exploit the relaxation techniques proposed. To address these limitations, this thesis proposes an advanced framework for query relaxation SPARQL. First, this framework includes a set of relaxation operators dedicated to SPARQL queries, to incrementally relax specific parts of the user request while controlling the relevance of the alternative responses returned w.r.t. to the user needs expressed in his request. Our framework also provides both several algorithms that identify the causes of failure of the user query and queries that are successful with a maximum number of conditions initially expressed in the failing request. This information allows the user to better understand why his request fails and execute queries that return non-empty alternative results. Finally, our framework offers intelligent relaxation strategies that rely on the causes of query failure. Such strategies reduce the execution time of the relaxation process compared to the traditional approach, which executes relaxed requests, based on their similarity to the user request, until a number of satisfactory alternative results is obtained. All contributions proposed in this framework were implemented and validated by experiments and scenarios based on the tests bench LUBM. They show the interest of our contributions w.r.t. the state of the art.

Keywords : Ontologies, Database searching, Ressource Description Framework (RDF), RDF Schema, SPARQL, Query relaxation methods, Non-responses, Empty answer problem, Failing causes.