Raisonnement incrémental sur des flux de données

Jules Chevalier

LABORATOIRE HUBERT CURIEN
UMR • CNRS • 5516 • SAINT-ETIENNE

Raisonnement incrémental sur des flux de données

Thèse

Pour obtenir le grade de

Docteur de l'Université Jean Monnet de Saint-Étienne

Spécialité : informatique

Présentée et soutenue publiquement le *5 février 2016*

par

Jules Chevalier

Directrice de thèse :
Frédérique Laforest

Co-encadrants de thèse :
Christophe Gravier & Julien Subercaze

Composition du Jury :

Amel Bouzeghoub	Professeur, Télécom SudParis	*Rapporteur*
Nathalie Aussenac-Gilles	Directeur de recherche CNRS, IRIT	*Rapporteur*
François Goasdoué	Professeur, Université Rennes 1	*Examinateur*
Pierre-Antoine Champin	Maître de conférences, LIRIS	*Examinateur*
Frédérique Laforest	Professeur, Université Jean Monnet	*Directrice*
Christophe Gravier	Maître de conférences, Université Jean Monnet	*Co-encadrant*
Julien Subercaze	Chargé de recherche, Université Jean Monnet	*Co-encadrant*

Remerciements

Cette thèse a été effectuée dans un premier temps avec l'équipe Satin au sein du laboratoire Télécom Claude Chappe, puis dans l'équipe Connected Intelligence du laboratoire Hubert Curien. Elle a été financée par le projet OpenCloudware.

Pour commencer, je tiens à remercier comme il se doit mes encadrants, Frédérique, Christophe et Julien. C'est l'aide, le soutient et la confiance qu'ils m'ont accordé tout au long de cette thèse qui ont permis la réussite de cette aventure tant professionnelle qu'humaine.

Je remercie également Nathalie Aussenac-Gilles et Amel Bouzeghoub pour le temps qu'elles ont accordé à la relecture de ce manuscrit et pour les remarques pertinentes et constructives qui m'ont entre autre permis d'améliorer celui-ci. J'adresse de la même manière mes remerciements à François Goasdoué et Pierre-Antoine Champin pour leur participation au jury.

Je souhaite aussi dire un grand merci à Charline et Gisèle qui ont relu ma thèse et m'ont permis de corriger et de clarifier un grand nombre de points cruciaux de ce manuscrit.

Je remercie les locataires du bureau i012, passés et présents, pour l'ambiance qu'ils ont contribué à créer. En espérant n'oublier personne, je remercie donc Pierre-Olivier, Tanguy, Syed, Abderrahmen, Nicolas, Antoine, Mérième, Yves-Gaël, Samuel, Romain et Kévin.

Merci à tous ceux avec qui j'ai eu le plaisir de travailler à Télécom Saint Etienne, que ce soit au niveau enseignant, administratif ou technique.

Je remercie également Mehdi, Kévin et les Marions des Craftsmen pour leur accueil lorsque, privé de foyer pendant plusieurs semaines, je suis venu chercher l'asile. Je n'oublie pas Pierre-Yves, sans qui nombre de repas auraient été bien moins appréciables.

Je souhaite tout particulièrement remercier mes parents, sans qui je ne serais pas arrivé jusque là (cette affirmation allant bien évidemment au delà de sa propre évidence). Ils ont su, dès mon plus jeune âge, me pousser à travailler et à étudier, grâce à quoi je me suis lancé avec passion dans les études supérieures. Je n'oublie pas non plus le soutient que j'ai reçu de ma famille, ainsi que des personnes présentes à ma soutenance. Je remercie enfin mon parrain, qui a très largement contribué à la naissance de ma passion pour les ordinateurs.

Enfin, je souhaite adresser un grand merci à mes amis, pour les soirées, les échanges, les discussions et j'en passe, bref pour leur présence qui m'a permis d'arriver au bout de ces années tout en conservant une santé mentale acceptable. Merci à Lucie, à Mickaël, à Dimitri, à Jérémie, à Marie ainsi qu'à tous ceux sur qui j'ai pu compter, dont le soutien a été plus qu'indispensable à la réussite de ce travail et qui m'ont aidé à garder confiance jusqu'à la fin.

Pour terminer, je souhaite adresser quelques derniers remerciements. Je ne saurais clore ces remerciements sans citer la boule de poils qui m'a tenu compagnie pendant ma rédaction à domicile. Merci donc à Plume pour sa participation, puisqu'elle a, en marchant sur le clavier, contribué à 17% de la rédaction de ce manuscrit. Pour des raisons plus qu'évidentes, je remercie Bob Kane pour son œuvre et son inspiration, ainsi que tous ceux qui, par la suite, ont fait perdurer la légende à travers les décennies. De même, je remercie Russell T Davies pour son imagination hors paire, source de détente sans équivalent lorsque ma propre imagination eut été tarie après une longue rédaction. J'adresse également mes remerciements à tous ceux qui auront la patience de lire ce manuscrit (ne vous inquiétez pas, il y a aussi des images).

Table des matières

Liste des figures

Liste des tableaux

Liste des Algorithmes

I

Introduction

1 Introduction

1.1 Contexte

La quantité de données produites chaque seconde sur Internet est devenue gigantesque ces dernières années. L'Homme a créé au cours des 18 derniers mois autant de données que depuis le début de son existence. En 2015, 29 téraoctets de données sont publiés chaque seconde sur le Web [1]. Toutes ces données sont souvent brutes et représentées sous de nombreux formats.

Le Web Sémantique a pour vocation de représenter ces données de façon à ce qu'elles soient interprétables tant par une machine que par un humain. Grâce à cette formalisation, les données brutes présentes sur le Web deviennent des connaissances structurées. Cette quantité phénoménale de données peut alors être automatiquement traitée par des machines, ce qui serait impossible par des méthodes de traitement manuelles.

Une des opérations permises par cette formalisation des données du Web est le **raisonnement**. Son objectif est de rendre explicites d'éventuelles informations implicites grâce à un ensemble de déductions logiques. Par exemple, sachant qu'un chat est un félin et qu'un félin est un animal, il est logique de déduire qu'un chat est un animal. Ce genre de déductions logiques est utilisé pour appliquer le raisonnement. C'est à cette opération que nous nous intéressons dans cette thèse, plus particulièrement au raisonnement par règles d'inférence. La matérialisation correspond au stockage des connaissances implicites extraites par cette opération.

Cette thèse s'inscrit dans le projet OpenCloudware [2], financé par le Fond national pour la Société Numérique [3] et supporté par les pôles Minalogic [4],

1. http://www.planetoscope.com/
2. http://www.opencloudware.org/
3. http://www.caissedesdepots.fr/activites/investissements-davenir/le-fonds-national-pour-la-societe-numerique-fsn-services-usages-et-contenus-numeriques.html
4. http://www.minalogic.org/

Systematic [5] et SCS [6]. Il a pour objectif de proposer une plateforme pour le développement collaboratif d'applications distribuées. Dans ce contexte, nous avons pour objectif de proposer une solution évolutive de raisonnement pour diriger les prises de décisions opérées par la plateforme. Tous ces choix (instanciation ou destruction d'une nouvelle machine virtuelle, déplacement des machines virtuelles d'un serveur à un autre, choix des ressources attribuées à une machine virtuelle, etc) sont dépendants d'un grand nombre de facteurs. Le raisonnement permet de déduire les informations, implicites à ces facteurs, sur lesquelles s'appuie le système pour prendre ses décisions. Afin de prendre en compte ces facteurs en constante évolution, il est indispensable de proposer une solution capable de mettre à jour efficacement les informations implicites avec l'évolution des facteurs.

C'est dans ce contexte que le sujet de cette thèse a été proposé par l'équipe SATIN du laboratoire Télécom Claude Chappe, intégrée par la suite à l'équipe Connected Intelligence [7] du laboratoire Hubert Curien [8], plus particulièrement dans l'équipe-projet *représentation des connaissances et raisonnement*.

1.2 Motivations

De nombreux travaux ont été entrepris afin de rendre le raisonnement possible dans le cadre actuel du Web. Le défi principal est de faire passer à l'échelle ce processus pour permettre le traitement de la quantité gigantesque de données disponibles sur le Web.

Le raisonnement par lots consiste à appliquer le raisonnement sur un ensemble de données. L'objectif des solutions pour ce type de raisonnement est d'expliciter (ou d'inférer) un maximum de connaissances le plus rapidement possible. Différentes optimisations ont été proposées et la parallélisation du processus apparaît souvent comme une solution efficace.

Ce paradigme n'est cependant pas suffisant. Une fois le raisonnement par lots commencé, il est impossible de prendre en compte de nouvelles connaissances sans recommencer le processus depuis le départ. De nouvelles données étant constamment disponibles, recommencer le processus de raisonnement est trop coûteux au vu de la fréquence d'arrivée de ces connaissances et de leur nombre. Le raisonnement incrémental contribue à répondre à ce problème. Il permet de mettre à jour

5. `http://www.systematic-paris-region.org/en/get-info-topics/free-and-open-source-software`
6. `http://www.pole-scs.org/`
7. `https://connected-intelligence.univ-st-etienne.fr/`
8. `http://laboratoirehubertcurien.fr/`

la matérialisation en utilisant les connaissances explicitées précédemment. Seules les connaissances implicites apportées par les nouvelles données sont extraites, afin de prendre avantage du travail déjà effectué.

Peu de travaux permettant le raisonnement incrémental ont été publiés. Les systèmes prenant en charge ce paradigme sont rares et peinent à passer à l'échelle lorsque les mises à jour sont importantes.

En fonction des données en entrée du raisonneur et des règles d'inférence appliquées, le raisonnement peut être très complexe, voire indécidable. Le raisonnement incrémental est un processus continu, pendant lequel les connaissances matérialisées peuvent être utilisées. Il n'est donc pas garanti que le raisonnement soit terminé lors de l'utilisation des connaissances matérialisées. L'utilisateur accédant à ces données peut alors récupérer des informations partielles, dont le complément n'a pas encore été explicité. Cette contrainte est inévitable : il n'est pas possible d'attendre la fin du raisonnement, qui est continu, pour répondre à une requête. Il est donc nécessaire de proposer des méthodes maximisant les chances de répondre correctement à une requête dans cette situation.

1.3 Contributions

Dans cette thèse, nous proposons une architecture pour le raisonnement incrémental. Sa conception modulaire a été pensée pour la parallélisation du raisonnement. Chaque règle d'inférence est associée à un module indépendant. Ces modules reçoivent les données utilisables par la règle et envoient les connaissances inférées aux autres modules pour la continuation du raisonnement. Les goulots d'étranglement ont été réduits en autorisant l'exécution de plusieurs instances de la même règle en parallèle. Des espaces tampons associés à chaque règle garantissent la récupération des données envoyées aux règles, assurant que le raisonnement est complet même en cas d'arrivée massive de nouvelles données ou d'explosion de la quantité de connaissances inférées.

Cette architecture permet de raisonner aussi bien sur des flux de données que sur des sources de données statiques. Son fonctionnement interne est basé sur la gestion de données sous forme de flux transitant entre les modules de l'architecture, lui permettant nativement de prendre en entrée des flux de données. La matérialisation existante est mise à jour grâce aux données entrantes.

Nous avons travaillé sur la réduction de l'impact des doublons sur le système, qui est un problème récurrent du raisonnement. Il s'agit de connaissances générées en double lors du raisonnement, qui peuvent ralentir le fonctionnement du processus de raisonnement s'ils ne sont pas pris en charge dès leur génération. Le *triplestore*, stockant l'ensemble des connaissances envoyées au raisonneur ou infé-

rées, est accessible par tous les modules, leur permettant de supprimer au plus tôt les doublons.

La conception modulaire de notre architecture lui permet également d'être indépendante du fragment de règles d'inférence utilisé.

Notre architecture propose trois modes de fonctionnement différents :

- Le mode *classique*, qui a pour vocation d'inférer le plus rapidement possible les connaissances implicites ;
- Le mode *supervisé*, permettant d'influencer l'ordre dans lequel sont inférées les connaissances, dans le but de prioriser la génération des connaissances les plus importantes pour l'utilisateur ;
- Le mode *intuitif*, permettant de maximiser la quantité de connaissances inférées par seconde, dans le but d'augmenter les chances de répondre correctement à une requête quelconque.

Cette architecture a été implémentée au travers du raisonneur **Slider**. Il prend nativement en charge les fragments ρdf et RDFS [9] et peut être étendu à d'autres fragments plus complexes. Le code source de Slider est disponible librement sous licence Apache 2.0 sur Github [11].

Les expérimentations que nous avons menées nous ont permis de :

- tester l'impact des paramètres de l'architecture sur les performances de Slider ;
- comparer Slider à l'état de l'art des raisonneurs ;
- mesurer l'amélioration des performances apportées par le raisonnement incrémental par rapport au raisonnement par lots ;
- vérifier la priorisation des connaissances lors de l'inférence pour le mode *supervisé*.

Les informations permettant de reproduire ces expérimentations sont également disponibles sur la page Github de Slider.

9. *RDF*[10] *Schema*
11. `http://juleschevalier.github.io/slider`

1.4 Organisation du manuscrit

Après avoir introduit les notions préliminaires nécessaires pour aborder la suite du manuscrit dans le chapitre *Notions préalables*, nous détaillons les solutions existantes pour le raisonnement par lots ainsi que pour le raisonnement incrémental dans le chapitre *Solutions pour le raisonnement*.

Le chapitre *Système pour le raisonnement incrémental* détaille les caractéristiques de notre architecture pour le raisonnement incrémental ainsi que son fonctionnement. Les différents modes de fonctionnement sont également décrits dans ce chapitre.

Le chapitre *Slider : Implémentation du raisonneur incrémental* présente les structures de données utilisées dans l'implémentation de notre architecture Slider et les détails de la parallélisation du processus de raisonnement. Dans le chapitre *Expérimentations*, nous présentons les expérimentations que nous avons menées et leurs résultats.

Le chapitre *Conclusion* termine ce mémoire par un rappel des contributions et dresse quelques perspectives au travail proposé.

II

État de l'art

2 Notions préalables

Sommaire

Le *Web sémantique* est un mouvement qui a pour objectif de fournir des outils permettant d'unifier la manière dont sont représentées des données présentes à travers le Web. Cette représentation se doit de les rendre à la fois lisibles par un humain tout en étant interprétables par une machine. L'objectif est de lier et de structurer ces données pour qu'elles puissent être partagées et réutilisées entre utilisateurs et applications. Le Web sémantique introduit également un certain nombre de traitements automatiques sur ces données, comme le parcours des données liées, la vérification de la consistance des informations ou encore l'extraction de connaissances implicites.

2.1 Le Web Sémantique

Dans cette section, nous reviendrons sur un rapide historique du Web sémantique. Nous présenterons ensuite les objectifs qui ont motivé sa création, pour terminer par un aperçu des critiques dont il fait l'objet.

2.1.1 Historique

Le concept de *modèle de réseau sémantique* apparaît dès à la fin des années 1960 [9–11], introduisant l'idée de représenter des connaissances de manière structurée. Lors de la création du Web à la fin des années 80, le principe de mise en relation des documents est introduit. Par la suite, le réseau de liens hypertextes des pages Web est étendu en y ajoutant des métadonnées interprétables par une machine, leur permettant d'accéder aux pages de manière plus intelligente et d'effectuer des traitements automatiques.

Peu après sa création, le W3C [1] publie en 1997 les premières recommandations sur le Web sémantique et pose ainsi les bases de ce nouveau concept de Web. Pour devenir universels, les outils du Web sémantique se doivent d'être libres et ouverts à tous. L'agencement des briques qui composeront le Web sémantique est présenté par la figure 2.1.

Tim Berners-Lee, réputé fondateur du W3C et instigateur du Web sémantique, en présente en 1999 sa vision :

1. *World Wide Web Consortium, communauté internationale fondée en 1994, chargée de développer des standards ouverts afin d'assurer la compatibilité des différentes technologies*

> *“J’ai fait un rêve pour le Web [dans lequel les ordinateurs] deviennent capables d’analyser toutes les données sur le Web – le contenu, les liens, et les transactions entre les personnes et les ordinateurs. Un « Web Sémantique », qui devrait rendre cela possible, n’a pas encore émergé, mais quand ce sera fait, les mécanismes d’échange au jour le jour, de bureaucratie et de nos vies quotidiennes seront traités par des machines dialoguant avec d’autres machines. Les « agents intelligents » qu’on nous promet depuis longtemps vont enfin se concrétiser.”*

Traduction Wikipédia, depuis [29]

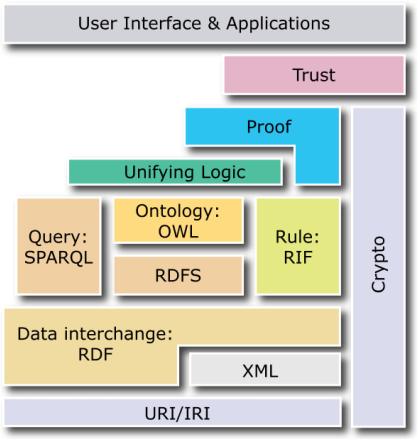

FIGURE 2.1 – Agencement des différentes technologies du Web sémantique (source : `http://www.w3.org`)

C’est en 2001 que Berners-Lee, Lassila et Hendler vulgarisent le Web sémantique [5] et en 2006 que Berners-Lee introduit la notion de *Linked Data* [70], ou *données liées*, renforçant l’importance de l’interconnexion des données.

En 2009, il lance le “Raw Data Now” [2] (ou “des données brutes maintenant”), qui incite chacun, qu’il soit politique, scientifique, professionnel ou toute autre personne, à ouvrir ses données afin de les connecter.

Les objectifs du Web sémantique, définis il y a maintenant plus de vingt ans, perdurent et dirigent toujours son évolution.

2. `http://linkedinfo.ikmemergent.net/content/tim-berners-lee-calls-raw-data-now.html`

2.1.2 Objectifs et défis

Depuis la création du Web, la quantité de données disponibles n'a cessé de croître de manière exponentielle, particulièrement ces dernières années. L'objectif premier du Web sémantique est d'étendre le Web afin de faciliter la recherche, le partage et l'agrégation de connaissances au sein de cet océan d'informations. Les opérations fastidieuses, comme la recherche d'information, seront déléguées aux machines alors capables d'interpréter les demandes des utilisateurs à un niveau sémantique. Pour cela, il est nécessaire de rendre le Web compréhensible à la fois par les humains et les machines.

Le Web sémantique apporte également une standardisation tant dans la forme de la représentation des connaissances que dans la représentation elle-même. En effet, le partage permet d'unifier les concepts définis et crée un pont entre les applications, les systèmes et les utilisateurs qui échangent dans un langage commun.

Mais derrière ces objectifs se cachent des défis conséquents que devra surmonter le Web sémantique pour atteindre son but.

Tout d'abord, les technologies du Web sémantique devront donc être capables de passer à cette échelle du Big Data [76]. La quantité de données accessibles sur le Web est devenue au fil des années plus que gigantesque. À titre d'exemple, en 2014, chaque minute plus de 3 millions de partages sont effectués sur les principaux réseaux sociaux, 72 heures de vidéos sont envoyées sur YouTube [3] et 200 millions de courriels sont transmis, le tout par 2,3 milliards d'utilisateurs [4]. Afin de traiter de tels volumes de données, un travail important est nécessaire que ce soit au niveau de la conception des algorithmes qui composeront les outils du Web sémantique, ou de l'infrastructure capable de passer à l'échelle. Ce travail représente une première barrière significative à l'application du Web sémantique. En plus de la quantité phénoménale de données à traiter, la fréquence extrêmement élevée à laquelle de nouvelles informations sont créées [13] rend le problème encore plus complexe. Il ne s'agit pas seulement de gérer la quantité déjà considérable d'informations existantes, mais également de prendre en charge celles continuellement générées, au fur et à mesure de leur arrivée. Il est indispensable de gérer ces données aussi vite qu'elles arrivent, car le flux est ininterrompu, donc tout retard dans le traitement des données se répercuterait sur la suite et pourrait devenir irrécupérable ou engendrer des pertes de données.

Un autre obstacle porte sur la difficulté de représentation des connaissances. Le Web sémantique vise en effet à décrire des concepts flous, incertains, imprécis avec potentiellement une infinité de variantes, et ce dans un langage formel, précis, exact et logique [52]. Ce langage devra donc permettre de représenter ces concepts avec le plus de précision, tout en conservant ses caractéristiques formelles, afin de pouvoir y appliquer des traitements logiques. Certains travaux permettent déjà de représenter des connaissances introduisant des probabilités [7, 21, 24]. Mais l'utilisation de ces connais-

3. `https://www.youtube.com/`
4. `https://www.domo.com/blog/2014/04/data-never-sleeps-2-0/`

sances probabilistes amène une complexité supplémentaire aux traitements qui y sont appliqués. Ceci nous ramène au défi du point précédent, concernant le passage à l'échelle.

La confiance que nous plaçons dans les informations qui nous sont accessibles, que ce soit sur internet ou n'importe quel autre média, est une problématique capitale datant de bien avant l'apparition du Web. Même une fois écartées les erreurs de saisies, les erreurs de "bonne foi" sont toujours possibles, commises sans mauvaises intentions par des personnes dont les connaissances sont erronées. Mais le plus grand problème reste la tromperie. Lorsque les données sont unifiées, il devient difficile d'évaluer la confiance que l'on peut accorder à une information. Cette confiance, bien qu'elle soit guidée par une opinion globale, est personnelle et différente pour chaque individu, comme le précise Richardson dans un article de 2003 [63]. Il est donc nécessaire de mettre en place des mécanismes permettant de déceler d'éventuelles erreurs dans les données accessibles, mais également de donner à l'utilisateur des indications pour l'aider à estimer la confiance qu'il peut accorder aux connaissances qui lui sont présentées.

Cette liste n'est bien sûr pas exhaustive, mais permet d'entrevoir la complexité du travail à accomplir pour atteindre les objectifs du Web sémantique. Certains défis n'ont été que partiellement traités, et d'autres ne seront mis au jour qu'avec l'avancement de la mise en place du Web sémantique.

2.1.3 Critiques

Le Web sémantique possède un certain nombre de détracteurs et il est important de fournir une vision équilibrée du Web sémantique dans ce chapitre. L'association des données permet de retrouver directement des informations liées à un concept grâce au réseau de liens qui compose ce concept. Cette facilité de récupération des données, prônée par les défenseurs du Web sémantique, inquiète ses détracteurs pour des raisons de sécurité et de respect de la vie privée.

La critique principale du Web sémantique concerne sa faisabilité. Le Web sémantique s'appuie sur le partage de modèles et de connaissances. Ce paradigme nécessite donc une collaboration et une entente entre les acteurs du Web. Une telle coopération semble difficile étant donné les enjeux actuels de la possession d'information et du contrôle des échanges.

En 2001, Doctorow [14] présente une liste de sept obstacles mettant en défaut la fiabilité du Web sémantique :

- Les gens mentent ;
- Les gens sont paresseux ;
- Les gens sont stupides ;
- Il est difficile de se décrire soi-même ;

- Les classifications ne sont pas neutres ;
- L'unité de mesure retenue influence les résultats ;
- Il y a plusieurs façons de décrire une même chose.

Il note également que les données ne sont pas éternellement vraies, et qu'elles ne peuvent pas intégrer de nouveaux concepts. Il en déduit donc que le Web sémantique ne pourrait pas fournir des données exactes et valides.

Le temps et la complexité nécessaires à la création et à la publication de nouvelles ressources sont, du fait de la formalisation inhérente au Web sémantique, plus chronophages et plus complexes que pour des données brutes. Ces opérations nécessitent également la maîtrise d'outils ou langages supplémentaires pour la sémantisation des données. Cette surcharge est cependant contrebalancée par la simplification des tâches de recherche et de traitement des informations amenées par la sémantique.

D'autres critiques du Web sémantique portent sur la lourdeur de la formalisation, extrêmement verbeuse et volumineuse pour le stockage de chaque donnée.

Malgré ces réserves, le Web sémantique continue d'évoluer et de plus en plus d'applications intègrent des données sémantiques. On peut citer DBPedia [27] qui a pour objectif d'extraire des données de Wikipedia [5] et de les structurer, pour ensuite les mettre à disposition sur le Web. Le vocabulaire FOAF (*Friend Of A Friend*) [73], qui permet de décrire les informations sur un individu et les relations qui le lient avec d'autres individus, est de plus en plus utilisé comme référence pour le stockage de profil. BioPortal [6] regroupe une grande quantité d'ontologies dans le domaine du biomédical afin de les rendre facilement accessibles.

5. https://fr.wikipedia.org/
6. `http://bioportal.bioontology.org/`

2.2 Représentation des connaissances

Cette section se focalise sur les différentes manières que met à disposition le Web sémantique afin de représenter les connaissances. Nous verrons dans un premier temps comment sont organisées les données, entre ontologies et bases de connaissances, avant de voir quels formats sont utilisés dans ces structures. Enfin, nous verrons les différentes opérations introduites par le Web sémantique sur les bases de connaissances.

2.2.1 Ontologies et bases de connaissances

Une ontologie regroupe l'ensemble des concepts et des relations permettant de décrire un domaine de manière formelle. Deux parties la composent : la *TBox* et la *ABox*. La `TBox` contient la terminologie de l'ontologie et définit les classes et les relations qui les lient. La `ABox`, quant à elle, contient les instances des concepts définis dans la `TBox`.

Voici un exemple d'ontologie que nous utiliserons tout au long de ce manuscrit. Dans la `TBox` nous définissons que *chat* est une sous-classe de *félin*, et que *félin* est une sous-classe de *animal* (nous reviendrons sur une définition plus formelle dans la section 2.2.2). Nous ajoutons la classe *chien* sous-classe de *canidé*, elle-même sous-classe de *animal*. Dans le `ABox`, *Garfield* et *Gromit* sont définis comme des instances respectivement des classes *chat* et *chien*. On obtient ceci :

`TBox`	`ABox`
chat est une sous-classe de *félin*	
félin est une sous-classe de *animal*	*Garfield* est un *chat*
chien est une sous-classe de *canidé*	*Gromit* est un *chien*
canidé est une sous-classe de *animal*	

Pour stocker ces informations physiquement, des *bases de connaissances* sont utilisées. Elles sont l'équivalent sémantique des bases de données. Il existe plusieurs manières de sérialiser les connaissances dans ces bases, comme nous le verrons dans la section suivante. Ces bases offrent souvent des outils de traitements des connaissances, que nous détaillerons dans la section 2.2.3.

2.2.2 Formats de représentation

Comme nous l'avons vu, le Web sémantique offre les outils permettant de représenter formellement des concepts et des relations. Différents formats permettent de modéliser cette représentation.

La logique de description [28] offre une sérialisation mathématique des connaissances. Elle permet de représenter un certain nombre de relations. La table 2.1 donne quelques exemples de relations.

Concept	Symbole	Exemple	Signification
Subsomption	$\sqsubseteq$	$Humain \sqsubseteq Animal$	Un Humain est un Animal
Intersection	$\sqcap$	$Animal \sqcap Bipède$	Un Animal Bipède
Union	$\sqcup$	$Animal \sqcup Végétal$	Un Animal ou un Végétal

TABLE 2.1 – Exemples de concepts en logique de description

La table 2.2 formalise notre ontologie en logique de description. Cette représentation est une formalisation qui n'est pas utilisée en pratique pour sérialiser des ontologies. D'autres formats sont utilisés pour représenter des connaissances et les stocker.

`TBox`		`ABox`
$félin$	$\sqsubseteq animal$	
$chat$	$\sqsubseteq félin$	$chat(Garfield)$
$canidé$	$\sqsubseteq animal$	$chien(Gromit)$
$chien$	$\sqsubseteq canidé$	

TABLE 2.2 – Exemple de formalisation d'une ontologie en logique de description

Un graphe RDF [75] est un modèle de représentation des données sémantiques. La figure 2.2 fournit le graphe RDF représentant notre exemple.

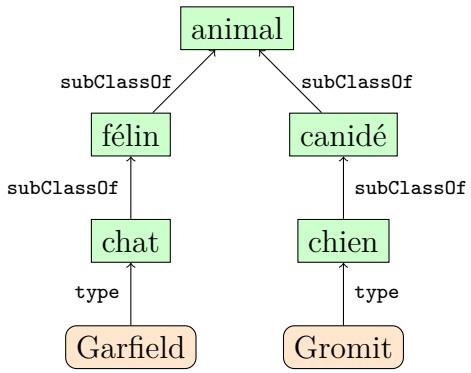

FIGURE 2.2 – Exemple de graphe RDF

L'unité de base d'un graphe RDF est le *triplet* $< sujet, prédicat, objet >$. Le `sujet` et l'`objet` sont des nœuds du graphe, tandis que l'arête qui les relie est annotée avec le `prédicat`. $< félin, subClassOf, animal >$ est un triple présent dans le graphe de la figure 2.2.

Partant de cette représentation générale, il existe de nombreuses sérialisations des graphes RDF. Les plus courantes sont RDF/XML [7] (parfois appelé RDF par abus de langage) [74], Turtle, N-Triples, N-Quads, JSON-LD, N3 (ou Notation 3). Un exemple de notation RDF/XML et son équivalent N-Triples sont donnés en Annexes B.1 et B.2.

2.2.3 Opérations sur les bases de connaissances

Du fait de la formalisation des données du Web, les machines ont la capacité d'interpréter et d'utiliser ces données structurées. Grâce à cette caractéristique, il est possible de déléguer à des machines des opérations généralement fastidieuses et répétitives. Ces opérations permettent en outre de :

- vérifier la consistance des connaissances ;
- vérifier la satisfaisabilité d'un fait ;
- déterminer toutes les classes auxquelles une instance appartient ;
- déduire des informations à partir des connaissances disponibles.

Cette dernière opération est appelée *raisonnement*. Il existe un certain nombre de raisonnements différents, mais l'idée principale reste la même : utiliser les données accessibles afin d'en déduire les faits implicites, c'est-à-dire la connaissance latente dans la base de connaissances

Ce procédé sera le sujet principal de cette thèse. La section suivante décrit plus en détail le type de raisonnement auquel nous nous intéressons.

7. *Extensible Markup Language*

2.3 Le raisonnement

La structure des données du Web sémantique permet l'exécution automatique de traitements sur ces données. Le raisonnement a pour objectif de rendre explicites des informations cachées dans les données disponibles. Il s'agit d'un processus déductif, qui ne crée pas de nouvelles connaissances, mais extrait celles latentes dans la base de connaissances.

Plusieurs méthodes permettent de raisonner sur un ensemble de connaissances. L'algorithme des tableaux [2, 51, 59], s'appuyant sur la logique de description, permet de vérifier la satisfaisabilité d'un fait. Pour cela, ce fait est explicité au maximum en utilisant une série de transformations. Une inconsistance est alors cherchée dans la version expansée du fait. Si aucune inconsistance n'est trouvée, alors le fait est considéré comme satisfaisable. L'explicitation des faits utilisés permet d'extraire des connaissances implicites, autrement dit de raisonner. Cependant, cette méthode n'est valable que pour des ontologies définies en logique de description.

Une solution plus générique est basée sur l'application de règles permettant le raisonnement. Ces règles, appelées règles d'inférence, sont définies par des prémisses à valider, entraînant une conclusion. Par exemple, pour la règle *si A est le père de B et B est le père de C, alors A est le grand-père de C*, les prémisses sont *A est le père de B* et *B est le père de C*, et *A est le grand-père de C* est la conclusion. Un ensemble de règles est appliqué en boucle sur la base de connaissances jusqu'à ce que plus aucune nouvelle conclusion ne soit extraite. Cette méthode apporte plusieurs avantages. En plus des règles standardes définies par le W3C [77], la possibilité de définir de nouvelles règles offre une généricité incomparable à cette méthode. Un grand nombre de solutions [55, 56, 62, 64, 66] utilisent les règles d'inférence pour supporter le raisonnement, et offrent pour certaines une généricité plus qu'intéressante [67]. De plus, même si la mise en pratique des règles reste réservée à des utilisateurs avertis, la définition des règles peut être faite sans connaissances sur la logique de description ou sur le Web sémantique en général.

Pour ces raisons, notre travail est axé sur le raisonnement par règles d'inférence. Cette section se focalise sur ce type de raisonnement, son fonctionnement, et ses conditions d'applications.

2.3.1 Raisonnement par règles

Une méthode populaire pour le raisonnement est basée sur l'utilisation de règles d'inférences. Ces règles définissent des conditions à remplir afin de déduire de nouveaux triples. Par exemple, on peut définir en langage naturel la règle suivante :

"*Si c_1 est une sous-classe de c_2, et que c_2 est elle-même sous-classe de c_3, alors on peut en déduire que c_1 est une sous-classe de c_3*"

Cette règle permet de définir la transitivité de la propriété `subClassOf`. Elle est formalisée sous la forme suivante :

$$\frac{c_1 \texttt{ subClassOf } c_2, \quad c_2 \texttt{ subClassOf } c_3}{c_1 \texttt{ subClassOf } c_3} \text{ (scm-sco)}$$

Pour effectuer le raisonnement, un ensemble de ces règles d'inférence est utilisé sur une ontologie afin de déduire les connaissances implicites. Les règles sont appliquées successivement jusqu'à ce qu'elles ne permettent plus de déduire de nouvelles informations.

Le W3C définit des ensembles de règles standards. On appelle *fragment* un ensemble de règles utilisé pour le raisonnement. Il existe des fragments standards eux aussi définis par le W3C. On peut citer RDFS [72], ou encore les différentes versions de OWL [8] [77]. D'autres fragments ont été définis dans la littérature, comme ρdf[60] ou OWL Horst [19].

Voici un exemple de raisonnement utilisant deux règles sur une ontologie contenant six triples. Les règles utilisées sont `scm-sco` (définie précédemment) et `cax-sco`, définie ci-dessous, qui dit que si $c1$ est une sous-classe de $c2$, et que x est de type $c1$, alors on en déduit que x est aussi de type $c2$.

$$\frac{c_1 \texttt{ subClassOf } c_2, \quad x \texttt{ type } c_1}{x \texttt{ type } c_2} \text{ (cax-sco)}$$

L'ontologie est initialement composée des triples suivants :

TBox	ABox
flin `subClassOf` *animal*	*Garfield* `type` *chat*
chat `subClassOf` *flin*	*Gromit* `type` *chien*
canide `subClassOf` *animal*	
chien `subClassOf` *canide*	

La première étape du raisonnement consiste à récupérer l'ensemble des couples de triples correspondant aux prémisses de la règle. Pour `scm-sco`, il s'agit des couples de triples `subClassOf` avec l'objet du premier égal au sujet du second. Cela donne pour cet exemple les couples suivants, suivis du triple inféré pour chaque couple. Pour les raisons de lisibilité, `subClassOf` sera noté `sc`.

chat `sc` *flin*	*flin* `sc` *animal*	$\rightarrow$	*chat* `sc` *animal*
chien `sc` *canide*	*canide* `sc` *animal*	$\rightarrow$	*chien* `sc` *animal*

Le même principe est maintenant appliqué pour la seconde règle. Les triples suivants sont obtenus.

chat `sc` *flin*	*Garfield* `type` *chat*	$\rightarrow$	*Garfield* `type` *flin*
chien `sc` *canide*	*Gromit* `type` *chien*	$\rightarrow$	*Gromit* `type` *canide*

Maintenant que chaque règle a été appliquée sur l'ontologie, les nouveaux triples sont ajoutés à l'ontologie. Les règles sont à nouveau appliquées sur les nouvelles données, jusqu'à ce qu'il n'y ait plus de nouveau triple inféré.

8. *Web Ontology Language*

Pour cette nouvelle étape, seule la règle `cax-sco` permet de déduire de nouvelles connaissances. Les couples/triples déjà inférés précédemment sont ignorés.

`cax-sco`			
félin `sc` *animal*	*Garfield* `type` *félin*	$\rightarrow$	*Garfield* `type` *animal*
canide `sc` *animal*	*Gromit* `type` *canide*	$\rightarrow$	*Gromit* `type` *animal*

Une dernière utilisation de chaque règle ne mène à aucun nouveau triple. L'inférence est donc terminée. L'ontologie, accompagnée des données implicites inférées est la suivante :

TBox	ABox
félin `subClassOf` *animal*	*Garfield* `type` *chat*
chat `subClassOf` *félin*	*Gromit* `type` *chien*
canide `subClassOf` *animal*	*Garfield* `type` *félin*
chien `subClassOf` *canidé*	*Garfield* `type` *animal*
chat `subClassOf` *animal*	*Gromit* `type` *canide*
chien `subClassOf` *animal*	*Gromit* `type` *animal*

La figure 2.3 représente le graphe RDF de notre ontologie, agrémenté des nouvelles relations en pointillés.

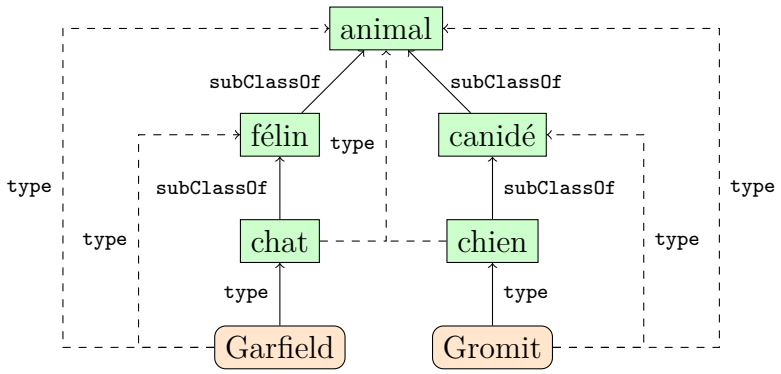

FIGURE 2.3 – Exemple de graphe RDF après application du raisonnement. Les relations en pointillés sont déduites suite au raisonnement.

2.3.2 Chaînage avant et matérialisation

Le raisonnement est un processus complexe qui peut se révéler très chronophage en fonction des données présentes dans l'ontologie et des règles utilisées pour l'inférence. Pour certains ensembles de règles, le raisonnement est indécidable. Suivant les cas, le temps d'inférence est bien trop long pour pouvoir être effectué intégralement avant de lancer une quelconque requête. Pour cette raison, il existe deux façons d'utiliser le raisonnement en fonction de la quantité de données à traiter, de l'ensemble des règles d'inférences utilisées, et du temps imparti au raisonnement.

Le chaînage avant (ou *forward chaining*) consiste à inférer l'ensemble des connaissances implicites par avance, et de les matérialiser dans la base de connaissances. C'est le paradigme utilisé dans l'exemple de la section 2.3.1. Cette méthode permet de répondre plus rapidement à une requête sur la base de connaissances dans la mesure où toutes les données implicites sont déjà matérialisées. La *matérialisation* est l'écriture des résultats du chaînage avant. Cela permet de fixer les connaissances inférées, et donc de ne pas les recalculer lorsqu'elles sont nécessaires.

Le chaînage arrière (ou *backward chaining*) répond à une requête. Pour cela, la requête est réécrite en rendant les concepts implicites explicites. Une fois cette explicitation de la requête effectuée, il est plus facile d'y répondre puisque les données nécessaires à la résolution de la requête ont été explicitées, et le travail consiste donc à retrouver ces données dans la base de connaissances. L'intérêt du chaînage arrière est de ne pas effectuer de pré-calculs et de ne pas occuper autant de mémoire que pour la matérialisation. La requête est traitée lorsqu'elle arrive.

Le chaînage avant est recommandé quand [9] :

- il y a un accès fréquent aux données, d'où l'intérêt du pré-calcul,
- les données sont relativement stables, donc il n'y a pas de nécessité de recalculer,
- les données sont difficiles à calculer, d'où un gain de temps avec le pré-calcul, et/ou
- les données sont suffisamment petites pour être efficacement stockées.

Le chaînage arrière est utilisé de préférence lorsque :

- seule une petite partie des connaissances inférées est réutilisée, donc il y a peu d'intérêt au pré-calcul,
- les réponses sont dynamiques,
- les réponses peuvent être calculées efficacement en direct, et/ou
- la réponse est trop volumineuse pour être matérialisée et stockée.

9. `http://answers.semanticweb.com/questions/3304/forward-vs-backward-chaining`

On peut mettre en place des solutions alternatives hybrides qui matérialisent les données fréquemment utilisées, statiques et suffisamment petites, et utilise le chaînage arrière pour les requêtes peu coûteuses en temps d'exécution, dynamiques et rarement utilisées.

Le paradigme qui nous intéresse dans cette thèse est le chaînage avant, couplé à la matérialisation des connaissances inférées.

2.3.3 Les fragments ρdf et RDFS

Comme nous l'avons vu, il existe un certain nombre de règles d'inférences, regroupées en fragments, définies par le W3C. ρdf et RDFS sont deux fragments très utilisés pour l'évaluation d'outils du Web sémantique. Ils présentent un compromis intéressant entre leur expressivité et leur complexité. La table 2.3 présente les règles incluses dans ces deux fragments. Les règles marquées d'un disque plein sont incluses dans le fragment correspondant, celles marquées d'un cercle vide ne le sont pas. En pratique, il existe deux versions de ces fragments, *full* et `default`. La première version contient l'ensemble des règles du fragment (y compris celles marquées d'un demi-cercle), contrairement à la seconde qui, pour des raisons de complexité, ne contient pas ces dernières.

2.3.4 Génération de doublons

Lors du processus de raisonnement, un même triple peut être inféré plusieurs fois, et ce pour deux raisons différentes :

- Une règle peut inférer plusieurs fois le même triple à partir de prémisses différentes ;
- Deux règles différentes peuvent inférer le même triple.

La gestion de ces doublons est un problème souvent abordé, comme nous le verrons dans le chapitre suivant.

Nous présentons dans cette section un exemple pour ces deux situations. Si la propagation des doublons n'est pas endiguée au plus tôt, ils seront utilisés dans la suite du raisonnement et les mêmes prémisses seront utilisées plusieurs fois pour inférer la même connaissance, générant encore plus de doublons. Le nombre de doublons dans le système augmente ainsi à chaque nouvelle application d'une règle d'inférence.

Les doublons causent donc un ralentissement du processus de raisonnement et augmentent la quantité de données à stocker afin de terminer le raisonnement. À titre d'exemple, le calcul de la fermeture transitive d'une chaîne de sous-classes de taille n du type $< 1, \texttt{sc}, 2 >< 2, \texttt{sc}, 3 >< 3, \texttt{sc}, 4 >< 4, \texttt{sc}, 5 >$ génère n^3 triples dont n doublons, soit la moitié des triples utiles à la fin de l'inférence.

#	RDFS	ρdf	Condition	Déduction
1	●	●	c_1 rdfs:subClassOf c_2 x rdf:type c_1	x rdf:type c_2
2	●	●	p rdfs:domain c x p y	x rdf:type c
3	●	●	p rdfs:range c x p y	y rdf:type c
4	●	●	p_1 rdfs:subPropertyOf p_2 x p_1 y	x p_2 y
5	●	○	p rdfs:domain c_1 c_1 rdfs:subClassOf c_2	p rdfs:domain c_2
6	●	●	p_2 rdfs:domain c p_1 rdfs:subPropertyOf $p2$	p_1 rdfs:domain c
7	●	○	p rdfs:range c_1 c_1 rdfs:subClassOf c_2	p rdfs:range c_2
8	●	●	p_2 rdfs:range c p_1 rdfs:subPropertyOf p_2	p_1 rdfs:range c
9	●	●	c_1 rdfs:subClassOf c_2 c_2 rdfs:subClassOf c_3	c_1 rdfs:subClassOf c_3
10	●	●	p_1 rdfs:subPropertyOf p_2 p_2 rdfs:subPropertyOf p_3	p_1 rdfs:subPropertyOf p_3
11	◐	○	x p y y rdf:type rdfs:Ressource	x rdf:type rdfs:Ressource
12	◐	○	x rdf:type rdfs:Class	x rdf:type rdfs:Ressource
13	◐	○	x rdf:type rdfs:ContainerMembershipProperty	x rdfs:subPropertyOf rdfs:member
14	◐	○	x rdf:type rdfs:Datatype	x rdfs:subClassOf rdfs:Literal
15	◐	○	x rdf:type rdf:Property	x rdf:subPropertyOf x
16	◐	○	x rdf:type rdfs:Class	x rdfs:subClassOf x

TABLE 2.3 – Liste des règles d'inférence pour les fragments ρdf et RDFS. Les règles annotées d'un demi-cercle ne sont présentes que dans la version complète des fragments correspondants

Doublons inférés par une même règle

On considère la règle `scm-sco` (définie en sous-section 2.3.1). On souhaite appliquer cette règle sur les triples suivants :

- chat `subClassOf` félin
- félin `subClassOf` animal
- chat `subClassOf` mammifère
- mammifère `subClassOf` animal

Pour appliquer la règle `scm-sco`, on commence par déterminer les couples de triples correspondant aux prémisses de la règle, c'est-à-dire c_1 `subClassOf` c_2 et c_2 `subClassOf` c_3. Les deux couples correspondants sont les suivants :

- chat `subClassOf` félin et félin `subClassOf` animal ;
- chat `subClassOf` mammifère et mammifère `subClassOf` animal.

La conclusion inférée par la règle `scm-sco` est c_1 `subClassOf` c_3, pour chaque couple c_1 `subClassOf` c_2 et c_2 `subClassOf` c_3. Ici on infère donc dans les deux cas le triple chat `subClassOf` animal. Ce triple est donc inféré deux fois par la même règle.

Doublons inférés par deux règles distinctes

On considère maintenant les règles `cax-sco` et `prp-dom`. Cette dernière est définie comme suit :

$$\frac{p \texttt{ domain } c, \quad x\ p\ y}{x \texttt{ type } c} \text{ (prp-rng)}$$

On souhaite appliquer ces règles sur les triples suivants :

chat `subClassOf` animal

Garfield `type` chat

mange `domain` animal

Garfield `mange` souris

On commence par déterminer les couples de triples correspondant aux prémisses des règles, c'est-à-dire c_1 `subClassOf` c_2 et c_2 `subClassOf` c_3 pour `scm-sco` et p `domain` c et $x\ p\ y$.

Application de `cax-sco` :

$$\frac{\text{chat } \texttt{subClassOf} \text{ animal}, \quad \text{Garfield } \texttt{type} \text{ chat}}{\text{Garfield } \texttt{type} \text{ animal}}$$

Application de `prp-rng` :

$$\frac{\text{mange } \texttt{domain} \text{ animal}, \quad \text{Garfield } \texttt{mange} \text{ souris}}{\text{Garfield } \texttt{type} \text{ animal}}$$

Le triple Garfield `type` animal a été explicité deux fois.

2.4 Formalisation

Dans cette section, nous abordons le raisonnement d'un point de vue logique et formel. Après avoir introduit les concepts nécessaires à cette formalisation, nous verrons comment sont définis les graphes RDF et comment ils sont interprétés. Nous serons alors à même de formaliser les règles d'inférence et les systèmes déductifs, équivalents logiques de l'inférence.

2.4.1 Notions préliminaires

Cette sous-section pose quelques bases nécessaires aux définitions des sections suivantes.

Rappelons qu'un triple RDF est composé de trois éléments : un sujet, un prédicat et un objet. Il est souvent souvent représenté comme un triplet (appelé *triple*) (s, p, o). En logique de premier ordre il se présente de cette façon : $p(s, o)$, mettant en avant l'aspect *propriété* du prédicat.

Les éléments du triple sont identifiés par une URI [10], permettant de les identifier de manière unique et permanente. Par exemple, l'URI suivante identifie la propriété `subClassOf` :

`http://www.w3.org/2000/01/rdf-schema#subClassOf`

2.4.2 Graphe RDF et interprétation

Un graphe RDF permet, comme nous l'avons vu, de représenter des connaissances de manière visuelle. Dans cette sous-section, nous nous attachons à formaliser cette représentation, et nous introduisons la notion d'interprétation. Nous utilisons pour cela la formalisation utilisée pour définir le fragment ρdf [60], allégée des définitions particulières au fragment.

Soit trois ensembles disjoints deux à deux : **U** qui représente l'ensemble des références URI, **B** l'ensemble des Blank nodes (représentant des ressources anonymes) et **L** les littéraux (représentant des valeurs constantes). Pour des raisons de lisibilité, on notera leur union par la concaténation de leurs noms (**UB** représentant $\mathbf{U} \cup \mathbf{B}$). Un triple est un vecteur $(s, p, o) \in \mathbf{UBL} \times \mathbf{U} \times \mathbf{UBL}$, avec s le sujet, p le prédicat et o l'objet.

Un graphe RDF G est un ensemble de triples. L'univers d'un graphe, noté $univers(G)$ est l'ensemble des éléments de **UBL** apparaissant dans les triples de G. Le vocabulaire d'un graphe est noté $voc(G)$ et désigne l'ensemble $univers(G) \cup \mathbf{UL}$.

On note μ : **UBL** $\rightarrow$ **UBL** une fonction préservant les URIs et les littéraux, telle que $\mu(u) = u$ pour tout $u \in \mathbf{UL}$.

10. *Uniform Resource Identifier*

On note $\mu(G)$ l'ensemble des triples $(\mu(s), \mu(p), \mu(o))$ tel que $(s, p, o) \in G$. Pour G_1 et G_2 des graphes RDF, on parle d'application μ de G_1 vers G_2, notée $\mu : G_1 \to G_2$, si μ est telle que $\mu(G_1)$ est un sous-graphe de G_2.

2.4.3 Raisonnement

Nous allons maintenant nous focaliser sur le raisonnement, en abordant d'abord les règles d'inférence, puis les systèmes déductifs permettant le raisonnement.

Règles d'inférence

Le processus de raisonnement qui nous intéresse est un système déductif basé sur l'application de règles. Ces règles sont par exemple de la forme suivante :

$$\frac{(\mathcal{A},\texttt{sc},\mathcal{B}),(\mathcal{X},\texttt{type},\mathcal{A})}{(\mathcal{X},\texttt{type},\mathcal{B})}$$

$\mathcal{A}$, $\mathcal{B}$ et $\mathcal{X}$ sont des variables représentant des éléments de **UBL**. Les triples RDF situés au-dessus sont nécessaires à la déduction de ceux situés en dessous. Il s'agit respectivement des prémisses et des conclusions de la règle.

Dans le format de spécification du W3C, la règle précédente, nommée `cax-sco`, est définie comme suit :

	Condition	Déduction
`cax-sco`	$?a$ `rdfs:subClassOf` $?b$ $?x$ `rdf:type` $?a$	$?x$ `rdf:type` $?b$

Une instanciation de règle correspond au remplacement de chaque variable par un élément de **UBL** de telle manière que l'ensemble des triples présents dans la règle soit des triples RDF correctement formés.

Système déductif

Un système déductif est l'application d'un ensemble de règles d'inférence. Cet ensemble est l'équivalent d'un fragment dans le standard défini par le W3C. Le système déductif correspond à l'inférence.

Le système déductif sur le vocabulaire v est noté $\models_v$. Pour deux graphes RDF G et H, $G \models_v H$ si et seulement s'il existe une séquence de graphes $P_1, P_2, \ldots, P_k$, avec $P_1 = G$ et $P_k = H$ et pour j tel que $2 \leq j \leq k$, l'une des propositions suivantes est vraie :

- Il existe une application $\mu : P_j \rightarrow P_{j-1}$;
- $P_j \subseteq P_{j-1}$;
- Il existe une instanciation $\frac{R}{R'}$ d'une règle telle que $R \subseteq P_{j-1}$ et $Pj = P_{j-1} \cup R'$.

L'algorithme 2.1 montre le processus permettant de passer de G à H. L'idée est simple : H est initialisé avec G, les prémisses de chaque règle sont cherchées dans H, les conclusions correspondantes sont ajoutées à H, et ce jusqu'à ce qu'il n'y ait plus de nouvelles conclusions dans H.

ALGORITHME 2.1 – Raisonnement basé sur l'application de règles d'inférence

```
   Données : G, Regles
   Résultat : H
1  H ← G
2  do
3  |  Δ ← ∅
4  |  pour regle ∈ Regles faire
5  |  |  pour regle.premices ∈ H faire
6  |  |  |  Δ ← regle.conclusions\H
7  |  |  |  H ← regle.conclusions
8  |  |  fin
9  |  fin
10 while Δ ≠ ∅;
```

2.5 Conclusion du chapitre

Le Web Sémantique permet, au travers d'un panel d'outils, de représenter et de modéliser des concepts de manière structurée. Les connaissances se présentent sous forme de triples $< s, p, o >$ regroupés dans des bases de connaissances. L'aspect formel de cette représentation lui permet d'être interprétée par une machine, tout en restant compréhensible par un humain. Grâce à cela, il est possible d'appliquer des traitements automatiques sur ces données.

Le raisonnement a pour objectif de rendre explicites les informations implicites cachées dans une base de connaissances. Pour cela, un ensemble de règles d'inférence est appliqué sur les connaissances stockées afin d'en déduire des conclusions faisant apparaître les données implicites. L'ensemble des règles ainsi utilisées pour l'inférence est appelé fragment. Le W3C a défini un certain nombre de règles, regroupées en fragments, notamment ρdf et RDFS.

Les règles d'inférence sont définies par des prémisses à valider afin d'en déduire des conclusions, correspondant aux connaissances inférées. Le paradigme qui nous intéresse est la matérialisation, qui consiste à stocker ces connaissances inférées dans la base de connaissances.

Dans le chapitre suivant, nous étudions les solutions existantes pour le raisonnement, en détaillant les différents types de raisonnement.

3 Solutions pour le raisonnement

Sommaire

3.1 Types de raisonnement et applications

Nous l'avons vu dans la section 2.3, le raisonnement est un processus itératif permettant, grâce à des règles d'inférence, de découvrir des connaissances jusque là implicites. Nous identifions plusieurs manières d'utiliser le raisonnement, illustrées dans la figure 3.1.

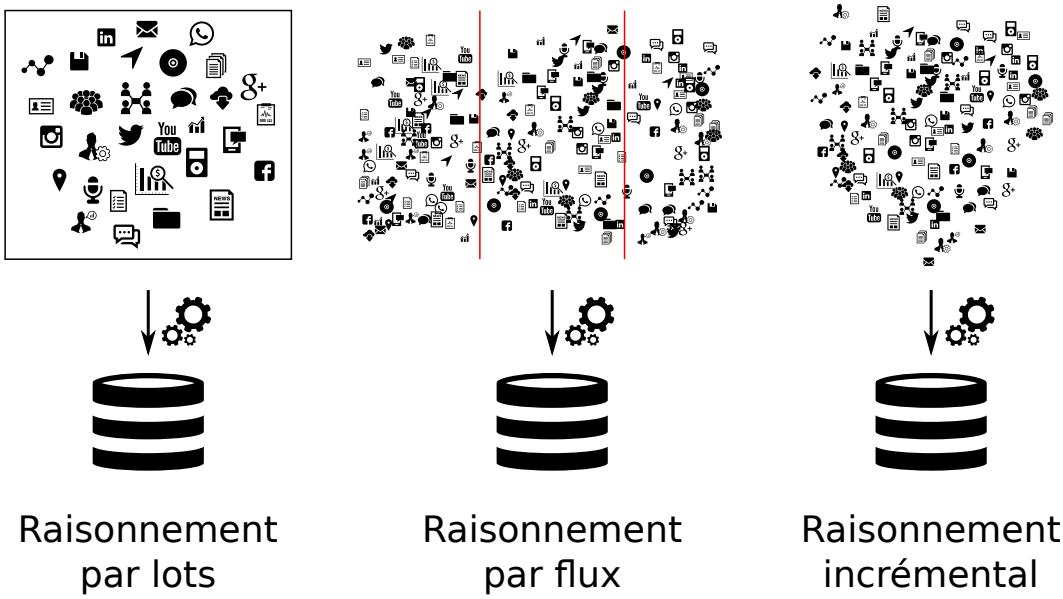

FIGURE 3.1 – Illustration des différents types de raisonnement : par lots, par flux et incrémental

Le raisonnement par lots est l'application du processus de raisonnement sur une ontologie figée qui ne change pas pendant le processus. Les règles d'inférence sont appliquées jusqu'à ce qu'elles ne permettent plus d'inférer de nouvelles connaissances.

Dans le cadre de bases de connaissances non statiques, constamment peuplées de nouvelles informations, de nouvelles formes de raisonnement ont vu le jour, notamment le raisonnement par flux et le raisonnement incrémental. Le raisonnement incrémental permet de maintenir la matérialisation des connaissances implicites au fur et à mesure de l'évolution de la base de connaissances. Le raisonnement par flux, quant à lui, permet de raisonner sur des connaissances présentes dans une fenêtre temporelle.

Plusieurs critères vont diriger le choix vers l'une ou l'autre des manières. Le raisonnement par flux sera privilégié lorsqu'une partie seulement des connaissances est fixe, tandis qu'une seconde partie est composée de données avec une durée de validité réduite. Il sera également favorisé lorsque la vitesse de publication des nouvelles données est trop élevée pour envisager une matérialisation complète.

Le raisonnement incrémental sera choisi lorsque les nouvelles données doivent être ajoutées à la base de connaissances au même niveau que celles déjà présentes et que la matérialisation de toutes les données implicites doit être faite.

Valle introduit en 2009 la notion de *stream reasoning* (ou raisonnement par flux) [30], qui pallie l'incapacité des raisonneurs par lots à gérer d'importants flux de données évoluant rapidement. Ces flux de données sont décrits comme des séquences sans fin d'éléments variant avec le temps.

La combinaison des techniques de raisonnement et de gestion de flux de données donnent naissance au raisonnement par flux. Deux grandes notions sont indissociables du raisonnement par flux [33] : les fenêtres temporelles (*time windows*) et le traitement continu (*continous processing*). Les données venant sous forme de flux sont horodatées et il leur est attribué un temps de validité. Les informations ayant dépassé ce délai ne sont plus considérées par le système. De nombreuses applications génèrent également ce type de données dont la pertinence diminue rapidement avec le temps, comme les réseaux sociaux, les données financières, les systèmes de surveillance du trafic, et bien d'autres.

Les réseaux de capteurs [38, 22, 68] forment un environnement tout désigné pour le raisonnement par flux puisqu'ils génèrent une grande quantité de données éphémères de manière continue, que le système utilise pour orienter des prises de décisions. Ces capteurs peuvent envoyer des informations provenant de sources très hétérogènes : objets connectés, maisons et villes intelligentes, voitures connectées, et bien d'autres. Une application similaire se retrouve dans le *monitoring*, qui remonte une série d'informations rapidement obsolètes comme le font les capteurs. La surveillance de signes vitaux dans le but d'observer l'état de santé de patients a notamment fait l'objet de plusieurs travaux [4, 43]. D'autres applications génèrent de grandes quantités de données n'étant pas valides très longtemps, mais ont également besoin de maintenir la matérialisation de connaissances non éphémères. Dans le cadre de la surveillance du trafic routier [57, 58], les limitations de vitesse et les emplacements de feux de signalisation sont des informations qui ne varient pas ou très peu, mais sont nécessaires pour interpréter d'autres données, comme la quantité de voitures présentes sur une portion de route, l'état des feux de signalisation, ou encore d'éventuels accidents. On peut ici voir un croisement entre la nécessité de maintenir une base de connaissances à jour par rapport à des mises à jour occasionnelles, et celle de traiter à la volée les informations éphémères utiles à une prise de décision.

Dans le cadre de cette thèse, nous ne nous intéressons pas au raisonnement par flux, mais au raisonnement incrémental, dans lequel les concepts de fenêtres temporelles et de péremption des données sont absents.

Il existe des travaux qui proposent d'utiliser le raisonnement afin de gérer les informations événementielles se produisant dans une ville [20]. Grâce aux données sur ces événements reçues en temps réel, l'objectif est de déterminer l'origine des événements, leur impact sur le trafic, ou encore de déterminer si ces événements sont liés.

Le projet WATER-M [1] intègre des outils et des données sémantiques provenant de la surveillance des réseaux de distribution d'eau afin de proposer un modèle de gestion plus intelligent, distribué et apportant de nouveaux services tant pour les consommateurs que les distributeurs. Des projets similaires pour la gestion de l'énergie sont en cours. Dans chaque cas, l'objectif est d'améliorer le processus de distribution de la ressource en question grâce aux outils sémantiques. Par exemple, les maisons équipées seront capables de *discuter* afin de gérer au mieux et de manière autonome la répartition des ressources, afin d'optimiser la consommation tout en allégeant la charge de travail des serveurs du fournisseur.

L'une des applications générales les plus répandues du raisonnement incrémental est la classification continue [3]. La classification est un processus qui permet de lister toutes les classes auxquelles appartiennent toutes les instances d'une base de données. Elle permet de calculer la fermeture transitive des relations hiérarchiques d'une taxonomie. Les relations `subClassOf` et `subPropertyOf` sont privilégiées.

La classification continue permet notamment aux concepteurs d'ontologies de déduire la hiérarchie complète de leur ontologie au fur et à mesure de sa création, sans devoir re-classifier toute l'ontologie. Le temps de cette opération peut être très important. La classification incrémentale permet de gagner un temps considérable et d'avoir un résultat mis à jour quasiment en temps réel. Elle est notamment utilisée dans les logiciels de développement d'ontologies afin de fournir une vue des relations de sous-classes entre les concepts définis.

Après avoir donné un aperçu de leurs domaines d'application, nous nous intéresserons dans ce chapitre à ces types de raisonnement. La section 3.2 détaille le fonctionnement du raisonnement par lots et les travaux qui l'utilisent. La section 3.3 s'intéresse au raisonnement incrémental et aux travaux qui le mettent en œuvre.

1. https://itea3.org/project/water-m.html

3.2 Raisonnement par lots

3.2.1 Solutions basées sur l'algorithme de Rete

L'algorithme de Rete [15] construit un arbre dont chaque nœud, sauf la racine, correspond à un motif apparaissant dans la condition d'une règle d'inférence. Lorsque tous les motifs nécessaires pour déclencher l'application d'une règle sont regroupés, une feuille est atteinte et la règle est exécutée. Deux types de nœuds composent l'arbre : *Alpha* et *Beta*. Les nœuds Alpha sont chargés de filtrer les éléments pouvant être utilisés pour l'exécution d'une règle. Ce sont les premiers nœuds par lesquels transitent les nouvelles données, juste après la racine. Les nœuds Beta ont exactement deux parents, permettent d'agréger les triples sélectionnés par leurs parents et de faire les vérifications supplémentaires (e.g. égalité de concepts). La figure 3.2 montre un exemple de graphe Rete pour l'application de la règles `cax-sco` :

$$\frac{c_1 \texttt{ subClassOf } c_2, \quad x \texttt{ type } c_1}{x \texttt{ type } c_2} \text{ (cax-sco)}$$

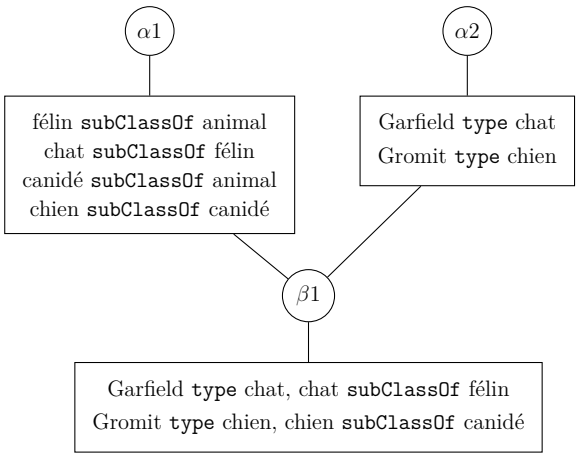

FIGURE 3.2 – Graphe représentant la règle `cax-sco` pour l'algorithme de Rete

Les avantages mis en avant par cet algorithme sont :

- Réduction voire élimination d'une partie de la redondance grâce au partage de nœuds ;
- Stockage de correspondances partielles pour limiter la réévaluation complète des faits en cas de modification des connaissances ;
- Suppression optimisée des éléments lors de l'effacement d'éléments de la mémoire de travail.

Heino et Pan [55] parallélisent le processus de raisonnement sur RDFS en utilisant l'algorithme de Rete et le framework *OpenCL* [23], qui permet de développer des applications parallélisées sur divers dispositifs, notamment CPU [2] et GPU [3]. Tous les concepts sont stockés sous la forme d'entiers 64 bits. Cela permet de garantir une taille constante des éléments, d'accélérer les comparaisons entre les concepts et de réduire la taille totale des données. Un dictionnaire stocké sur le disque permet de conserver la correspondance entre ces entiers et les URI originales. Une phase de synchronisation globale est présente entre l'exécution de chaque règle ou ensemble de règles, comme le montre la figure 3.3.

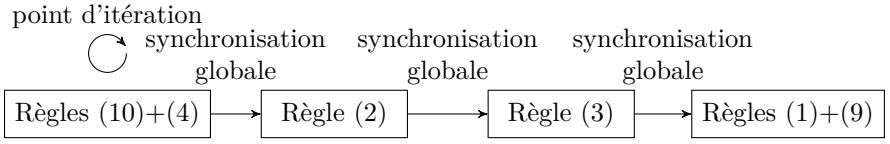

FIGURE 3.3 – Ordonnancement pour l'application des règles de RDFS dans les travaux de Heino, tiré de [55]. Les numéros de règles font référence à la table 2.3

Cette phase de synchronisation, inhérente au framework *OpenCL*, a le désavantage d'obliger le système à attendre la fin de l'exécution de chaque nœud pour pouvoir passer à l'étape suivante. Cela ralentit le temps global d'exécution et diminue l'intérêt de la parallélisation.

Afin de limiter la génération de doublons, deux optimisations sont proposées : une pour les doublons *locaux* et une seconde pour les doublons *distants*. Les locaux sont les triples générés plusieurs fois lors de l'exécution d'une même règle. Ils sont éliminés à la fin de l'exécution de la règle afin de limiter leur impact sur la suite du raisonnement, avant d'être envoyés aux règles suivantes et dans le triple store général. Les doublons distants sont dus à la génération du même résultat par différentes règles. Chaque processus léger a une vision globale des triples stockés, lui permettant de filtrer ceux déjà générés par d'autres règles.

La génération des doublons n'est ici pas empêchée, mais leur dissémination dans le système est stoppée au plus tôt afin d'éviter des calculs inutiles.

Peters propose une solution similaire [44]. Bien que l'évaluation ait été faite sur ρdf, RDFS et OWL Horst, l'algorithme proposé n'est pas dédié à ces fragments et peut être utilisé avec d'autres règles définies suivant les besoins de l'utilisateur. Pour paralléliser le processus, les auteurs utilisent le framework *OpenCL*.

En 2014, les auteurs présentent plusieurs optimisations de leur raisonneur [61]. La répartition de la charge de travail est améliorée pour les nœuds Alpha et Beta. Les doublons sont filtrés grâce à un système de *triple-matches*, afin de limiter la quantité de triples réutilisés pour les prochaines étapes de l'inférence. Ces optimisations ont permis

2. *Central Processing Unit*
3. *Graphics Processing Unit*

au raisonneur de générer jusqu'à 2,7 millions de triples par seconde pour ρdf, et 1,4 million de triples par seconde pour RDFS.

Dans un article de 2015 [62], Peters propose une amélioration des structures de données utilisées afin de réduire la mémoire nécessaire au fonctionnement du raisonneur. Grâce à cette optimisation, le raisonneur est capable de gérer des ontologies plus importantes sans dépasser les capacités d'un ordinateur portable. Le raisonneur est alors capable d'inférer jusqu'à 5 millions de triples par seconde pour ρdf, et 1,6 millions de triples par seconde pour RDFS.

3.2.2 Solutions basées sur *MapReduce*

Map Reduce est un paradigme de programmation mis en avant en 2008 par Dean et Ghemawat [12]. Il permet de développer des applications distribuées grâce à l'implémentation de deux fonctions : *Map* et *Reduce*. *Map* transforme les données en entrées en paires clé/valeur et *Reduce* regroupe et traite les paires ayant la même clé. La figure 3.4 illustre un exemple du fonctionnement de *MapReduce* pour le comptage des mots d'un texte. Il existe plusieurs implémentations de *MapReduce*, dont une proposée par Apache™[78] comme module du projet Hadoop® [69]. Ce framework a grandement contribué à la popularisation du paradigme *MapReduce*.

En 2009, Jacopo Urbani présente une solution pour le raisonnement sur RDFS et OWL Horst [19] : *WebPie* [79]. *MapReduce* est utilisé pour distribuer l'exécution des règles d'inférence dans un cluster. Pour RDFS, les règles d'inférence sont ordonnancées de manière à ce qu'il ne soit pas nécessaire de revenir sur une règle précédente pour compléter l'inférence. Afin de limiter la quantité de doublons générés, les triples sont regroupés durant la phase de *Map*, et l'inférence est faite durant la phase de *Reduce*. De cette manière, les couples de triples permettant d'inférer des connaissances identiques sont regroupés lors de la première phase et le triple correspondant n'est généré qu'une seule fois.

Pour partager des données entre les différents nœuds du cluster, Hadoop® utilise HDFS™ [4], un système de fichiers distribué sur disque. Pour limiter l'impact du temps d'accès aux données, le schéma de l'ontologie – c'est-à-dire la `ABox` – est entièrement chargé dans la mémoire de chaque nœud. L'utilisation d'un système de fichiers partagé sur disque pour les autres données a cependant un impact négatif sur les performances du raisonneur, dû aux temps d'accès disque bien plus importants que ceux d'accès à la mémoire vive.

La table 3.1 contient les résultats de l'inférence menée par les auteurs pour RDFS sur cinq ontologies.

La figure 3.5 montre le passage à l'échelle du raisonneur RDFS en fonction du nombre de nœuds utilisés dans le cluster Hadoop®, pour deux ontologies.

4. *Hadoop Distributed File System*

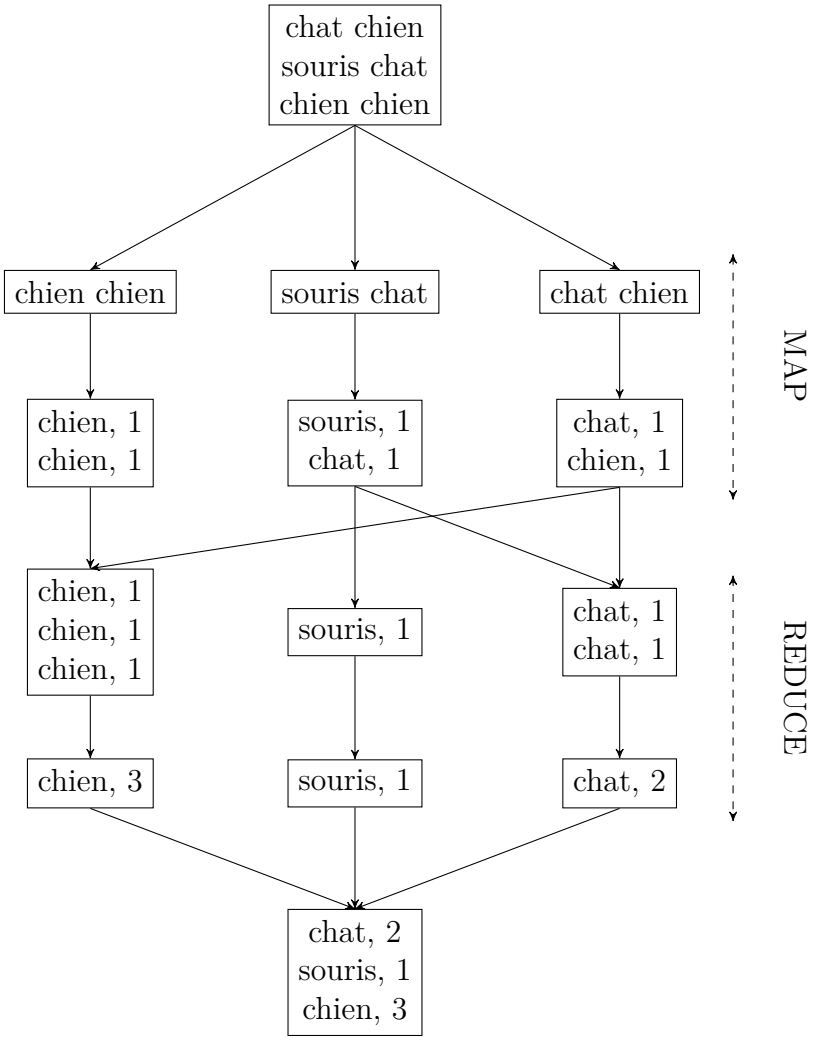

FIGURE 3.4 – Exemple de fonctionnement de *MapReduce* pour le comptage des mots d'un texte

Le gain en performance apporté par un nouveau nœud est conséquent pour les premiers nœuds, mais devient limité après cinq nœuds. Les auteurs expliquent qu'au-delà de 32 nœuds, le coût du framework lui-même est plus important que celui du calcul. Cependant, ce gain est déjà très affaibli après une dizaine de nœuds.

De plus, le temps nécessaire pour l'inférence sur la plus petite ontologie est de 2 minutes 25 secondes, malgré l'utilisation de 85 nœuds pour ce calcul.

Pour OWL Horst, il n'existe pas d'ordonnancement des règles ne nécessitant pas de boucle d'inférence. Différentes optimisations ont donc été introduites pour ce fragment dans [65]. Pour compléter le raisonnement, les règles de RDFS sont d'abord appliquées

Ontologie	Nombre de triples	Triples inférés	Temps d'exec.
Wordnet	1 942 887	3 004 496	02m25s
DBPedia	150 530 169	21 447 573	02m44s
Falcon	32 512 291	832 412 298	04m35s
Swoogle	78 839 999	1 431 765 550	06m33s
1B competition	864 846 753	29 170 100 223	58m43s

TABLE 3.1 – Temps d'inférence et nombre de triples inférés sur RDFS avec WebPie. Les résultats sont tirés de [79]

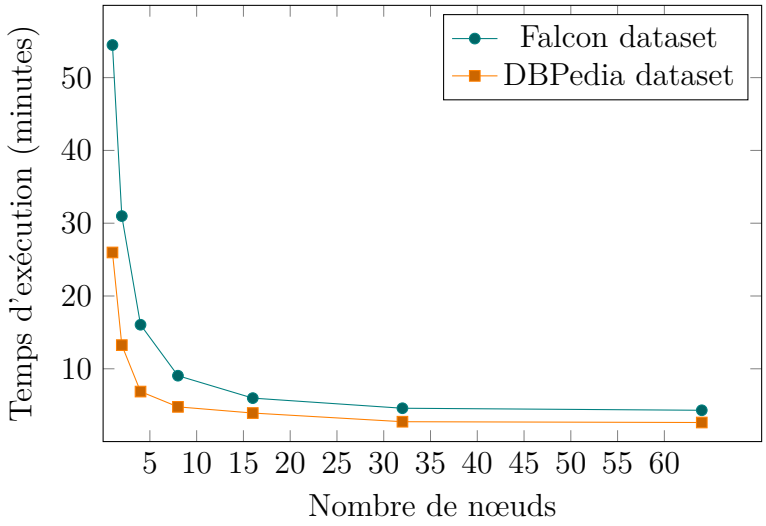

FIGURE 3.5 – Temps d'inférence sur RDFS en fonction du nombre de nœuds utilisés dans WebPie. Les résultats sont tirés de [79]

avec la même méthode que précédemment. Ensuite, les règles de OWL Horst sont appliquées tant que de nouvelles connaissances sont inférées. Le processus est recommencé jusqu'à ce que ni les règles de RDFS ni celles de OWL Horst ne génèrent plus de nouvelles données.

Les expérimentations ont été menées sur le même cluster que précédemment. La table 3.2 regroupe les résultats des expérimentations sur OWL Horst. Le passage à l'échelle donne de bien meilleurs résultats. On remarque cependant qu'il faut près de 45 heures pour l'ontologie la plus imposante.

Malgré la prise en compte de plusieurs contraintes comme l'ordonnancement des règles ou la gestion des doublons, le système souffre de plusieurs faiblesses. Le raisonneur n'est efficace que si la taille du schéma de l'ontologie est suffisamment petite pour la charger en mémoire. L'isolation des données des nœuds rend difficile la limitation de la génération des doublons, puisqu'elle nécessite que les nœuds aient accès à l'ensemble

Données (GTriples)	Inférés (MTriples)	Temps (heures)	Débit (Kt/sec)
1,07	495,5	0,61	455,2
10,71	4971,7	4,06	684,6
102,50	47536,1	45,77	606,8

Nœuds	Temps
8	44,4h
16	22,3h
32	10,6h
64	5,0h

TABLE 3.2 – Temps d'inférence avec WebPie en fonction du nombre de nœuds (à droite) et des données en entrée (à gauche), tiré de [65]

des données disponibles.

MapResolve [64] propose une solution basée sur *MapReduce* pour la classification sur $\mathcal{EL}+$. Les auteurs mettent l'accent sur plusieurs optimisations visant à permettre le passage à l'échelle. Afin de minimiser le nombre de passes nécessaires pour terminer l'inférence, les triples sont classés dans trois ensembles : les triples *utilisables*, *utilisés* et *inférés*. Grâce à cette stratégie, chaque combinaison de prémisses n'est traitée qu'une seule fois, et les doublons sont plus rapidement écartés. À chaque nouvelle itération, les triples *inférés* deviennent des triples *utilisables* afin de compléter l'inférence.

La répartition de la charge de travail est adaptée durant l'inférence en ajustant la quantité n de triples utilisés par chaque fonction *reduce* pour appliquer la règle qui lui correspond. À l'initialisation, tous les nœuds se voient attribuer la même valeur de n. Après chaque exécution, n est réévalué comme suit : $n = n * t_t/t_c$, où t_t est le temps d'exécution théorique et t_c le temps réellement calculé. Cela permet d'ajuster la charge de travail entre les nœuds en fonction de leurs comportements respectifs.

Bien que ce framework permette facilement de développer des applications distribuées, les auteurs mettent en avant plusieurs problèmes inhérents à *MapReduce*. La séparation du processus en séquence de *jobs* ne permet pas aux triples inférés d'être directement envoyés vers le prochain nœud pouvant les utiliser. L'exécution de toutes les instances du *job* courant doivent terminer avant de passer au *job* suivant. Couplé au temps d'accès aux données sur disque, le temps d'exécution est considérablement ralenti. Pour une application visant des performances optimales, les auteurs recommandent un framework permettant la communication entre les nœuds.

3.2.3 Autres méthodes de passage à l'échelle

Dans ses travaux, Soma présente deux méthodes différentes pour partitionner le processus de raisonnement [46, 47]. L'une consiste à partager les données et appliquer l'ensemble des règles sur chaque partie des données. La seconde découpe l'ensemble des règles et chaque nœud d'un système parallèle applique un sous-ensemble de règles sur l'ensemble des données. Afin d'assurer l'efficacité du partitionnement, l'auteur définit des conditions nécessaires, quel que soit le type de partitionnement :

- La quantité de travail soumise à chaque unité de calcul doit être également répartie ;
- Les données transmises entre les nœuds doivent être minimisées ;
- Le nombre de doublons générés doit être minimal ;
- Le partitionnement en lui-même doit être rapide et efficace sur de grands volumes de données.

Afin de partitionner les données, trois méthodes sont présentées : par partition de graphe, par hachage et avec l'aide de connaissances sur le domaine de l'ontologie. Pour le partitionnement des règles, un graphe de dépendance entre règles est utilisé. Les sommets sont les règles et il existe une arête entre deux règles si les triples inférés par la première règle peuvent être utilisés par la seconde. Un algorithme parallèle est présenté, permettant de partitionner l'inférence indépendamment de la méthode utilisée. Une implémentation partitionnant les données suivant le type de règle utilisée est proposée. Elle utilise le raisonneur Jena [34] afin d'inférer de nouveaux triples dans chaque nœud. Les expérimentations sur le passage à l'échelle montrent de bons résultats pour des ontologies dont le graphe est peu dense comme celles générées grâce au générateur d'ontologies LUBM[18]. Le rendement est cependant plus limité sur des ontologies dont le graphe est plus fortement connecté.

En 2014, Motik propose une solution parallèle pour la matérialisation d'ontologies [42]. Le système, s'appuyant sur Datalog [8], supporte les règles d'inférence récursives. Datalog est un langage déclaratif, sous ensemble du Prolog, permettant d'interroger des bases de données déductives. Il permet également de décrire des règles d'inférence sur ces bases. Pour stocker les connaissances, les auteurs utilisent une structure utilisant un minimum de verrous. Elle permet de répondre efficacement à des requêtes et est optimisée pour être mise à jour de manière parallèle. Les concepts sont stockés sous la forme d'entiers. Un dictionnaire permet de faire la correspondance entre un concept et l'entier qui lui est assigné. Un index basé sur le hachage est utilisé pour retrouver rapidement les triples nécessaires à l'application des règles d'inférence. Chaque triple dans la structure utilisée est lié aux triples ayant le même sujet, prédicat ou objet. Cela permet d'accélérer la récupération des triples correspondant aux prémisses d'une règle.

Une variante parallèle de l'algorithme semi-naïf [26] est utilisée pour raisonner. Les auteurs ont travaillé pour réduire les interférences entre les processus légers afin de tirer

au maximum parti de la parallélisation. Les écritures dans la structure sont optimisées pour éviter les défauts de cache, très coûteux.

Le système décrit dans [42] a été implémenté dans RDFox [5]. Pour tester le passage à l'échelle de leur système, les auteurs ont lancé l'inférence en faisant varier le nombre de processus légers utilisés. La figure 3.6 illustre les résultats de cette expérimentation. On peut voir que de manière générale l'utilisation de processus légers permet d'améliorer le temps d'inférence. Les courbes semblent suivre une progression logarithmique. L'amélioration apportée par l'ajout de processus légers est moins importante au-delà de 24 processus. Elle continue cependant à augmenter, montrant que le système réussit à passer à l'échelle.

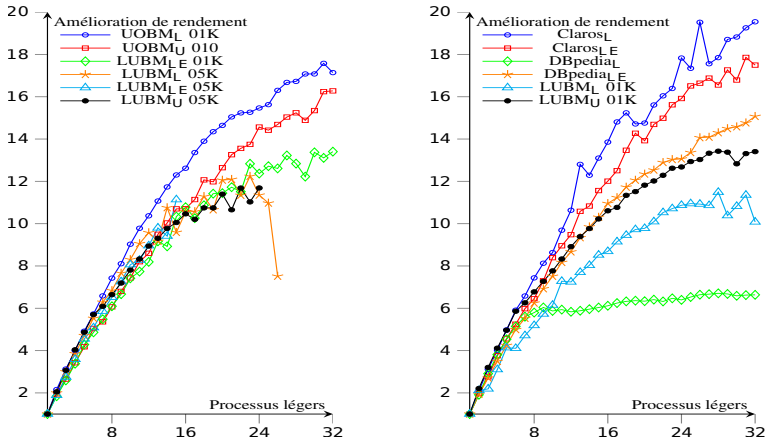

FIGURE 3.6 – Amélioration du temps d'inférence avec RDFox pour différentes ontologies, tiré de [42]

Les auteurs ont ensuite comparé leur système à d'autres méthodes de raisonnement. Seul OWLIM-Lite [6] supporte les règles récursives et stocke les triples en mémoire vive. Pour neuf des ontologies testées, OWLIM-Lite dépasse le temps alloué pour terminer l'inférence. RDFox termine lui pour chaque ontologie. Pour les autres ontologies, RDFox est plus rapide pour compléter la matérialisation de manière significative. Ces résultats sont rapportés dans la table 3.3. RDFox a été configuré pour n'utiliser qu'un seul processus léger.

5. https://www.cs.ox.ac.uk/isg/tools/RDFox/

Ontologie	RDFox	OWLIM-Lite
ClarosL	2062	14293
ClarosLE	4218	-
DBpediaL	143	14855
DBpediaLE	9538	-
LUBML01K	71	316
LUBMLE 01K	765	-
LUBMU 01K	113	-
UOBMU 010	2501	11311
UOBML01K	467	-

TABLE 3.3 – Temps d'inférence pour RDFox et OWLIM-Lite, tiré de [42]. Les temps sont en secondes.

3.3 Raisonnement incrémental

3.3.1 Définition

Alors que le raisonnement par flux est appliqué sur des flux de données éphémères avec une fréquence d'arrivée très élevée, le raisonnement incrémental se place dans le cas de données plus durables, avec un rythme de modification moins soutenu.

Ce paradigme, plus proche du raisonnement par lots, a pour objectif de maintenir la matérialisation d'une base de connaissances au fil de ses modifications. Ce fonctionnement peut être simulé en appliquant le raisonnement par lots sur l'ensemble de la base de connaissances lors de chaque modification. Mais cela oblige à recommencer le calcul d'inférence depuis le début, sans prendre avantage des données déjà inférées. Afin d'éviter la surcharge de calcul qui serait entraînée, le raisonnement incrémental conserve et utilise les données déjà inférées. L'inférence est réduite à son minimum, pour ne traiter que les données nécessaires à la matérialisation des nouvelles données implicites.

Contrairement au raisonnement par flux, toutes les informations sont stockées dans la base de connaissances. Une donnée n'est retirée de la base de connaissances qu'en cas d'inconsistance. Il ne s'agit pas ici de données éphémères, mais de données vouées à rester dans la base de connaissances et permettant de faire grossir la base de connaissances grâce au raisonnement et à la matérialisation.

Quand le raisonnement par flux n'utilise les données du flux que pour diriger une prise de décision sans les conserver, le raisonnement incrémental se comporte plus comme un système de mise à jour de la base de connaissances. En prenant l'exemple d'un réseau social, lors de l'ajout d'un nouvel individu, il est nécessaire de construire son graphe de connaissances, au travers du raisonnement. Il est cependant souhaitable de ne calculer que les nouvelles relations incluant le nouvel individu. Le raisonnement incrémental répond à ce cas d'usage en permettant de conserver les informations déjà inférées et de

s'appuyer sur celles-ci pour n'inférer que les nouvelles relations induites par le nouvel individu.

Les travaux ayant pour sujet le raisonnement incrémental sont nombreux. Rappelons que nous nous intéressons ici au raisonnement par chaînage avant, basé sur l'utilisation de règles d'inférence dans le cas du raisonnement incrémental (c.f. section 2.3.2).

3.3.2 Solutions pour le raisonnement incrémental

En 2005, Volz propose une solution permettant de maintenir à jour la matérialisation d'ontologies de manière incrémentale [67]. En plus des modifications de l'ontologie, elle permet de gérer l'ajout de nouvelles règles d'inférence. Elle n'est pas limitée à un fragment ou à un type d'ontologie et est applicable à tout langage pouvant être traduit en programme Datalog [8], notamment RDFS. L'intégralité de la base de connaissances ainsi que les règles d'inférence sont représentées sous la forme de programmes Datalog. En plus de gérer l'insertion de nouvelles données, le système est capable de propager la suppression de connaissances en révoquant celles qu'elles ont permis de déduire.

Les auteurs utilisent une variante de l'algorithme **DRed** (pour *Delete and Rederive*) [39], composé de trois phases :

1. La *surestimation de la suppression* qui calcule les triples à supprimer ;
2. La *redérivation* qui enlève de cette estimation les triples qui peuvent être ré-inférés à partir des connaissances restantes ;
3. L'*insertion* qui ajoute les nouveaux triples inférés.

Les expérimentations menées montrent que pour un changement de 10% sur l'ontologie, le coût du maintien de la matérialisation reste le même que ce soit pour l'ajout ou le retrait de règles ou de faits. Les coûts des différentes opérations permettant la matérialisation sont très élevés, puisque chaque opération est plus coûteuse que la requête sans matérialisation. Même si la matérialisation est utilisable pour toutes les requêtes suivantes jusqu'à ce que de nouvelles données soient ajoutées, le coût de la mise à jour est bien trop important, jusqu'à plus de 10 minutes par opération, pour une requête sans matérialisation prenant au maximum quelques secondes. L'ensemble de ces résultats est visible dans la figure 3.7.

Dans un article de 2008 [31] Suntisrivaraporn propose une extension de l'algorithme donné dans [1] permettant la classification sur $\mathcal{EL}+$. Pour opérer la classification de manière incrémentale, l'ontologie est séparée en deux parties : $\mathcal{O}_p$ la partie dite permanente de l'ontologie, déjà classifiée, et $\mathcal{O}_t$ la partie temporaire, qui doit encore être traitée. L'idée est de réutiliser les connaissances rendues explicites par la classification de $\mathcal{O}_p$ pour classifier $\mathcal{O}_p \cup \mathcal{O}_t$ sans reprendre le processus du départ. Les auteurs appellent cette configuration *classification incrémentale restreinte*. Cette méthode se rapproche du raisonnement par flux, dans le sens où une partie de l'ontologie est fixe tandis que l'autre est temporaire, rappelant les flux de données éphémères.

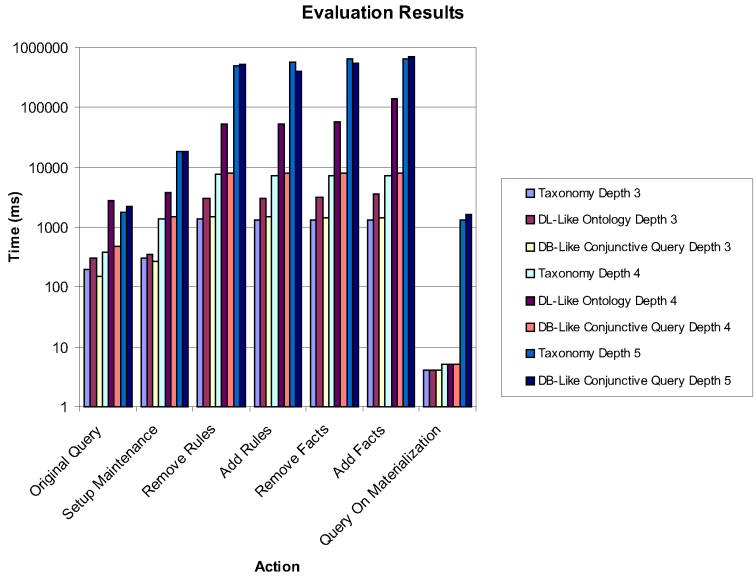

FIGURE 3.7 – Temps d'exécution moyens pour différentes opérations dans les travaux de Volz, pour un changement de 10% de l'ontologie, tirés de [67]

Les auteurs se placent dans le cadre de logiques de description monotones. On a donc $\mathcal{O}_p \models \alpha \rightarrow \mathcal{O}_p \cup \mathcal{O}_t \models \alpha$. Grâce à cette implication, l'appartenance d'un fait à la base de connaissances ne peut pas être remise en question par de nouveaux faits. Cela permet de garantir qu'il ne sera pas nécessaire de supprimer des connaissances suite à une reclassification. La classification sur $\mathcal{O}_p$ est réutilisée afin de ne pas re-classifier l'ensemble $\mathcal{O}_p \cup \mathcal{O}_t$.

Pour mesurer le gain apporté par cette solution, les auteurs ont lancé une série d'expérimentation afin de comparer le temps nécessaire à la classification complète d'ontologies et à la classification incrémentale des mêmes ontologies avec un pourcentage de données supplémentaires, de 0,2% à 1%. La table 3.4 regroupe les résultats de ces expérimentations.

$\mathcal{O}_t$	$\mathcal{O}$NotGalen		$\mathcal{O}$FullGalen		$\mathcal{O}$Nci		$\mathcal{O}$Snomed	
	Class.	Inc.	Class.	Inc.	Class.	Inc.	Class.	Inc.
0,2%	6,53	1,75	486,19	56,94	5,10	2,00	1666,43	55,86
0,4%	6,50	1,88	484,89	59,37	4,81	2,15	1663,51	57,97
0,6%	6,48	2,45	482,13	62,34	4,78	2,37	1661,49	68,58
0,8%	6,43	2,88	466,97	80,52	4,70	2,54	1652,84	83,27
1,0%	6,38	4,46	450,61	109,81	4,59	3,19	1640,11	93,89

TABLE 3.4 – Temps pour la classification complète et incrémentale en secondes sur plusieurs ontologies, pour des ajouts de triples représentant de 0,2% à 1,0%, tiré de [31]

Ces résultats montrent que la classification incrémentale est coûteuse, puisqu'elle nécessite entre 4 et 38% du temps nécessaire à la classification totale pour l'ajout de 0,2 à 1% de nouvelles données. Le gain reste intéressant puisqu'il est dans le pire des cas de 62% du temps total, mais très relatif au vu de la quantité de connaissances ajoutées.

Une autre technique de classification incrémentale basée sur la précédente a été introduite en 2010 [16]. Partant d'une ontologie $\mathcal{O}^1$ déjà classifiée, pour classifier une ontologie $\mathcal{O}^2 = \mathcal{O}^1 \cup \Delta$ (où Δ représente les nouveaux faits), les auteurs ajoutent les triples de Δ à $\mathcal{O}^1$ et appliquent les règles d'inférence impliquant les triples de Δ.

Afin de tester cette solution, le protocole suivant a été mis en place :

1. Supprimer p% des triples de l'ontologie testée ;
2. Classifier l'ontologie en utilisant le raisonneur CEL [50] ;
3. Ajouter les triples supprimés et reclassifier incrémentalement l'ontologie obtenue.

Les résultats obtenus sont présentés dans la table 3.5.

p	GO		NCI		NotGalen$^-$		FullGalen$^-$		SNOMED	
	Fixe	Inc.	Fixe	Inc.	Fixe	Inc.	Fixe	Inc.	Fixe	Inc.
0,5	1,01	0,10	1,86	0,29	1,72	0,06	114,7	7,2	649,7	141,3
1,0	0,98	0,15	1,85	0,44	1,68	0,16	112,5	13,1	638,9	225,9
1,5	0,98	0,24	1,85	0,59	1,60	0,33	107,5	23,1	631,0	348,3
2,0	0,95	0,28	1,81	0,80	1,53	0,52	106,9	23,5	580,4	470,4
2,5	0,94	0,32	1,78	0,99	1,56	0,52	94,5	47,1	580,1	566,7
3,0	0,94	0,38	1,77	1,14	1,58	0,42	101,9	34,5	532,5	683,2
3,5	0,92	0,45	1,73	1,58	1,37	0,91	87,2	64,1	539,1	728,4
4,0	0,86	0,51	1,69	1,75	1,40	0,83	89,5	58,1	468,1	849,9
4,5	0,86	0,51	1,69	1,75	1,24	1,19	80,8	77,2	461,4	916,5
5,0	0,84	0,73	1,64	1,95	1,29	1,19	87,1	63,2	478,8	941,1

TABLE 3.5 – Temps en secondes pour la classification des parties fixes et incrémentales en utilisant CEL, tiré de [16]

Une autre série d'expérimentations a été menée cette fois sur la classification incrémentale des différentes versions d'une ontologie. Il s'agit de SNOMED [48], découpée en seize versions différentes. Les résultats sont présentés dans la table 3.6.

Les auteurs concluent de ces expérimentations que leur solution est efficace en cas de mise à jour importante, mais à condition qu'aucun triple n'ait été enlevé, et que la quantité de nouvelles données n'excède pas 2% de la taille de l'ontologie. En effet, passée cette borne, le temps nécessaire à la classification incrémentale est plus important que celui nécessaire à la classification de la partie permanente.

Version	Concepts	Class.	Transition	Nouv. Concepts	Class. Inc.	Nouv. Relations
v.1	92724	8,0	v.1-v.2	2611(+2,82%)	22,7	34555
v.2	95335	8,9	v.2-v.3	597(+0,63%)	5,7	6380
v.3	95932	9,0	v.3-v.4	668(+0,70%)	10,6	9873
v.4	96600	9,1	v.4-v.5	4107(+4,25%)	32,8	35032
v.5	100707	10,6	v.5-v.6	3910(+3,88%)	138,9	111600
v.6	104617	12,7	v.6-v.7	18229(+17,42%)	306,0	982725
v.7	122846	51,4	v.7-v.8	5(+0,00%)	0,4	10
v.8	122851	52,2	v.8-v.9	4443(+3,62%)	15,8	29428
v.9	127294	52,0	v.9-v.10	178875(+140,0%)	2330,4	4130819
v.10	306169	271,1	v.10-v.11	19687(+6,43%)	406,1	1420405
v.11	325856	420,6	v.11-v.12	7469(+2,29%)	302,3	376095
v.12	333325	461,7	v.12-v.13	11224(+3,37%)	209,1	431405
v.13	344549	511,1	v.13-v.14	8112(+2,35%)	390,8	742571
v.14	352661	602,7	v.14-v.15	4474(+1,27%)	153,1	292081
v.15	357137	615,6	v.15-v.16	4689(+1,31%)	149,3	276475
v.16	361824	637,4	v.16-full	17867(+4,94%)	261,8	398477

TABLE 3.6 – Classification complète et incrémentale de seize versions de l'ontologie SNOMED en utilisant CEL, tiré de [16] (temps en secondes)

Kazakov et Klinov proposent une solution [56] implémentée dans le raisonneur ELK [40]. Elle est présentée comme particulièrement efficace pour des changements incrémentaux mineurs dans une ontologie. Plus précisément, cette solution permet de mettre à jour la classification d'ontologies après l'ajout et/ou la suppression de triples.

Les données sont stockées sous la forme d'*inférences*. Chaque *inférence* comprend un ensemble de prémisses nécessaires à l'application d'une règle et les conclusions sont calculées grâce aux prémisses. Concrètement, pour la règle `scm-sco`, une inférence valide serait :

$$(\{< chat, sc, flin >, < flin, sc, animal >\}, \{< chat, sc, animal >\}) \tag{3.1}$$

Cela permet de garder une trace de la provenance des données inférées. En contrepartie, l'espace nécessaire au stockage de ces inférences est important du fait des répétitions qu'elles entraînent. Chaque triple est répété dans chaque inférence dans laquelle il intervient.

Pour gérer l'ajout de nouveaux triples, le raisonneur utilise la classification existante et la complète en ne calculant que les relations nécessaires pour compléter la classification. La suppression est gérée grâce à une variante de l'algorithme DRed. Les *inférences* sont réparties dans des partitions. Lorsqu'une connaissance est retirée, la partition à laquelle elle appartient est marquée comme "brisée" (*Broken*). Les règles produisant des conclusions appartenant à cette partition sont ensuite ré-appliquées afin de "réparer" la partition endommagée. Ce fonctionnement permet de ne pas ré-appliquer l'ensemble des règles à toute la base de connaissances, mais seulement à un sous-ensemble correspondant aux partitions marquées.

Cette procédure a été implémentée dans la version 0.4.0 du raisonneur ELK. Pour trois ontologies différentes, les auteurs ont ajouté et enlevé un certain nombre de triples, puis reporté le temps nécessaire à la suppression des données, la "réparation" des partitions et l'ajout des nouvelles données. Le nombre de partitions marquées comme "brisées" est également noté. Ces résultats sont regroupés dans la table 3.7.

Ontologie	Class. Init.		Chang.	Suppression			Réparation		Addition		Total	
	#infer.	temps	add.+del.	#infer.	*Broken*	temps	#infer.	temps	#infer.	temps	#infer.	temps
GO	2 224 812	543	84+26	62 384	560	48	17 628	8	58 933	66	138 945	134
GALEN	2 017 601	648	1+1	3 444	36	18	4 321	4	3 055	13	10 820	39
			10+10	68 794	473	66	37 583	17	49 662	52	156 039	147
			100+100	594 420	4 508	214	314 666	96	426 462	168	1 335 548	515
SNOMED	24 257 209	10 133	1+1	4 022	64	120	423	1	2 886	68	7 331	232
			10+10	42 026	251	420	8 343	4	31 966	349	82 335	789
			100+100	564 004	3 577	662	138 633	56	414 255	545	1 116 892	1 376

TABLE 3.7 – Nombre d'inférences et temps en millisecondes pour la classification initiale puis incrémentale sur trois ontologies. Données tirées de [56]

On peut voir dans la table 3.7 que le temps nécessaire à la réparation des partitions marquées *Broken* est négligeable pour les ajouts/suppressions de peu de triples (1 et 10), mais représente la moitié du temps nécessaire à l'ajout pour l'ajout et la suppression de 100 axiomes dans GALEN. Pour l'ontologie GALEN, le temps nécessaire à la re-classification après l'ajout et la suppression de 100 triples (515ms) est très proche du temps nécessaire à la classification initiale (648ms). En revanche, l'évolution du temps nécessaire à la matérialisation par rapport aux quantités de données ajoutées et supprimées semble suivre une courbe logarithmique, ce qui laisse penser que le système devrait être capable de gérer des mises à jour plus importantes.

DynamiTE [66] permet de maintenir à jour la matérialisation de larges bases de connaissances. Plusieurs algorithmes permettent de rapidement recalculer la matérialisation après l'ajout ou la suppression de données, tout en exploitant les connaissances déjà matérialisées. Une version parallèle d'évaluation de Datalog est utilisée pour gérer l'ajout de nouveaux triples. Un autre algorithme est utilisé pour la suppression de connaissances et pour répercuter cette suppression. Ne nécessitant pas le parcours complet de la base de connaissances, il est plus efficace pour cette opération que l'algorithme DRed. Le raisonneur offre donc trois opérations principales : la matérialisation complète de la base de connaissances, la mise à jour de la matérialisation en cas d'ajout de triples et la mise à jour de la matérialisation en cas de retrait de triples.

Afin de minimiser l'espace mémoire nécessaire et les temps d'accès aux données, une compression par dictionnaire et des B-arbres permettent de stocker et d'indexer les triples. Les triples du schéma, c'est-à-dire ceux dont le prédicat est `subClassOf`, `subPropertyOf`, `domain` ou `range`, peuvent être chargés en mémoire afin d'accélérer l'exécution des règles d'inférence. De même, les nouveaux triples envoyés au système sont chargés en mémoire toujours dans le but d'accélérer le traitement de ceux-ci.

Afin de tester les performances de leur solution, les auteurs ont défini six *mises à jour* consistant chacune en un ensemble de triples à ajouter et/ou à retirer de la base

de connaissances. Les données utilisées représentent des universités. La mise à jour 1 ajoute/supprime un triple qui n'entraîne l'application d'aucune règle d'inférence. La mise à jour 2 est similaire à la première avec 16 000 triples. La numéro 3 ajoute/supprime 8 000 triples amenant à l'inférence d'un nombre fixe de triples. La quatrième mise à jour supprime un triple et en ajoute deux, menant à l'inférence d'un nombre de triples proportionnel à l'ontologie en entrée. Les mises à jour 5 et 6 suppriment respectivement une et deux universités complètes de la base de connaissances. Le temps nécessaire au maintien de la matérialisation a été calculé pour chaque mise à jour, la part étant faite entre les ajouts et les suppressions. La matérialisation initiale nécessite quinze minutes sur l'ontologie LUBM(1000) [18]. La table 3.8 résume ces résultats.

Mise à jour	Ajout (sec)	Suppression (sec)
1	0,117	0,117
2	8,2	25,7
3	3,7	25,4
4	31,8	51,0
5	16,8	74,6
6	30,5	135,8

TABLE 3.8 – Temps d'inférence avec DynamiTE pour six mises à jours, après une inférence complète. Les résultats sont tirés de [66]

Les auteurs mettent en avant le goulot d'étranglement dû au temps d'accès au disque, sur lequel est stocké la base de connaissances, qui ralentit le processus général malgré le chargement en mémoire vive d'une partie des connaissances à traiter.

Les temps nécessaires aux mises à jour sont, tant pour l'ajout que pour le retrait de triples, plus que corrects, au maximum 135,8 secondes pour la suppression de deux universités. Bien qu'ils ne soient pas suffisants pour une utilisation temps réel, les résultats obtenus restent plus rentables qu'une re-matérialisation complète.

Afin de tester l'effet de la parallélisation sur l'efficacité du système, les auteurs ont également diminué de moitié le nombre de processus légers alloués. La matérialisation complète est dans cette configuration 12% plus lente, quand l'ajout et le retrait de triples sont respectivement 6% et 30% plus lents. La parallélisation du processus joue donc un rôle essentiel pour les performances obtenues.

Motik propose, en 2015, une solution pour mettre à jour la matérialisation [41] basée sur Datalog. La mise à jour ne concerne pas les triples qui ont été inférés et peut être appliquée à toute règle décrite avec Datalog. Les auteurs se focalisent dans cet article sur la suppression de triples. Ils utilisent pour cela un algorithme basé sur un mélange de chaînage avant et de chaînage arrière, qui fonctionne sur le même principe que l'algorithme DRed. Il est cependant amélioré en réduisant la surestimation des triples à supprimer, en calculant les triples pouvant être dérivés à partir d'autres triples de la base de connaissances en dehors de ceux qui seront supprimés.

Les auteurs ont implémenté cette solution dans le raisonneur RDFox. Ils ont lancé une série d'expérimentations afin de tester les performances du système. La comparaison de l'algorithme proposé avec DRed a montré que l'algorithme proposé surpasse grandement les performances de DRed, de plusieurs facteurs dans certains cas. Les expérimentations menées ne concernent que le retrait de triples et ne donnent pas de résultats concernant les performances pour des mises à jour comprenant l'ajout de nouveaux triples.

3.3.3 Synthèse des solutions pour le raisonnement incrémental

Différentes directions ont été prises dans les travaux que nous avons étudiés pour mettre en place le raisonnement incrémental. On remarque cependant que la généricité du système par rapport aux règles utilisées revient plusieurs fois. Plusieurs optimisations afin d'améliorer le temps d'accès aux données et de réduire l'espace nécessaire à leur stockage sont également introduites. La section 3.4 revient en détail sur ces points, ainsi que ceux apparus dans l'état de l'art des solutions pour le raisonnement par lots de la section 3.2.

L'une des limitations du raisonnement incrémental apparaît lorsque l'ajout de nouvelles connaissances amène à une inconsistance dans la base de connaissances. Il est alors nécessaire de rétracter une partie des connaissances entraînant l'inconsistance. Mais ce n'est pas suffisant. Il faut également rétracter les faits dérivés de ces connaissances.

Il existe, comme nous l'avons vu, plusieurs approches pour gérer l'inconsistance, pouvant être le résultat de l'injection d'information issue de différentes sources de triples. Cependant, cela dépasse le cadre fixé dans cette thèse. Des travaux existent dans notre équipe et ils ont mis en évidence des algorithmes pour rétracter des connaissances dans le cadre de systèmes matérialisant des données évolutives [17].

3.4 Points clés des solutions étudiées

Comme nous venons de le voir, il existe une variété importante dans les solutions de raisonnement au travers de règles d'inférence. On peut cependant dégager plusieurs points clés, que ce soit pour les difficultés rencontrées ou les approches mises en place afin de les contourner. Dans cette section, nous résumons les principales difficultés rencontrées et les optimisations ayant permis le passage à l'échelle dans les solutions présentées.

Nous commençons par voir les méthodes entreprises pour gérer le problème de la génération de doublons (section 3.4.1). Nous présentons ensuite comment l'ordonnancement des règles est calculé dans la section 3.4.2, puis les différentes améliorations mises en place pour assurer la répartition de la charge de travail dans la section 3.4.3. Enfin, nous détaillons les techniques utilisées pour réduire l'empreinte mémoire des structures de données et leur temps d'accès (section 3.4.4). Pour terminer, nous revenons sur l'aspect générique des solutions étudiées face au fragment utilisé pour le raisonnement en section 3.4.5.

3.4.1 Gestion des doublons

La génération de doublons est un obstacle majeur inhérent au processus de raisonnement. Ils peuvent être générés par une même règle à partir de données différentes ou bien par deux règles différentes aboutissant au même résultat (comme nous l'avons vu dans la section 2.3.4). S'ils ne sont pas éliminés, les doublons sont utilisés pour la suite du raisonnement, entraînant l'application de règles sur des prémisses identiques, inférant de fait plusieurs fois les mêmes connaissances. De nouveaux doublons sont alors générés, utilisés pour continuer l'inférence, et ainsi de suite. La surcharge de calcul engendrée par les doublons peut donc être conséquente lorsqu'aucune stratégie n'est mise en œuvre afin de limiter leur dispersion dans le système.

Plusieurs astuces ont été mises en œuvre dans les différents travaux que nous avons étudiés afin de limiter la génération des doublons et de limiter leur propagation dans le système.

Dans [55], les doublons locaux, c'est-à-dire générés par une même règle, sont éliminés dès la fin de l'exécution de cette règle avant d'être envoyés aux autres règles. En ce qui concerne les doublons générés par des règles différentes, chaque module exécutant une règle a une vision globale des données déjà inférées. Cela leur permet de détecter les éventuels doublons en comparant les triples tout juste inférés avec l'ensemble des données inférées jusque là. Seuls les triples n'ayant jamais été inférés avant sont alors envoyés aux autres règles en vue de continuer le processus de raisonnement.

Des opérations sont présentes dans *WebPie* [65, 79] lors de la phase *Map* afin d'éliminer les triples inférés par une même règle, puis de filtrer les triples inférés déjà présents dans les triples en entrée. Une seconde opération permet à *WebPie* d'anticiper la génération de doublons en groupant les ensembles de triples qui, pour une règle, génèrent le même résultat. La règle est appliquée sur ces ensembles de triples de façon à ne pas infé-

rer plusieurs fois les mêmes triples. Cette méthode a l'avantage de prévenir la génération de doublons, mais ne s'applique qu'à certaines règles particulières, comme `prp-spo1`.

MapResolve [64] limite la recherche des doublons dans un ensemble ne contenant que les triples inférés dans le but de réduire le coût de cette opération. Cela ne suffit cependant pas dans le cas où des triples inférés font doublon avec des triples présents dans la base de connaissances.

Peters [62] utilise les connaissances stockées dans la base de connaissances afin de filtrer les doublons. Cela permet d'éliminer à la fois les doublons locaux et distants. La contrepartie est un accès important à la base de connaissances qui doit, pour ne pas ralentir le processus, offrir de bonnes performances pour la vérification d'appartenance des triples.

3.4.2 Ordonnancement des règles

L'ordre dans lequel sont appliquées les règles d'inférence peut avoir un impact primordial sur les performances du raisonnement. Comme cela a été montré par Urbani [79], pour certains fragments comme RDFS, un ordre particulier permet de n'exécuter chaque règle qu'une seule fois. Pour d'autres fragments plus complexes, comme OWL Horst, ce type d'ordonnancement n'existe pas. En revanche, l'ordonnancement peut être choisi de manière à minimiser le nombre de boucles nécessaires pour compléter le raisonnement.

Dans *WebPie* et dans *MapReduce*, l'ordonnancement est optimisé pour les fragments pris en charge par ces solutions. Cela permet d'améliorer les performances en limitant au maximum le nombre de fois qu'il est nécessaire d'appliquer chaque règle afin de compléter l'inférence. Toutefois, ces ordonnancements sont propres aux fragments en question, et, pour appliquer le raisonnement à un autre fragment, un nouvel ordonnancement sera nécessaire.

Dans ses travaux [47], Soma s'affranchit d'une certaine manière de ce problème puisque les règles sont exécutées en parallèle. Il n'y a donc pas à proprement parler d'ordonnancement. Le problème de déterminer l'ordre dans lequel les règles doivent être lancées est remplacé par le problème du découpage des données ou des règles (suivant le mode de découpage utilisé). Afin de séparer au mieux les règles, un graphe de dépendance entre les règles est introduit. Il permet de formaliser l'interdépendance des règles d'inférence. C'est un partitionnement des règles minimisant les échanges nécessaires entre les règles qui remplace l'ordonnancement. Cette méthode s'appuie sur un graphe de dépendance des règles calculable pour tout fragment. Elle tend donc à améliorer la généricité du système face au fragment de règles utilisé.

Motik [42] utilise également une solution parallèle, sans utiliser d'ordonnancement.

3.4.3 Répartition de la charge de travail

La distribution et la parallélisation sont sans équivoque inévitables afin de faire passer à l'échelle le processus de raisonnement. Pour être efficaces, ces opérations nécessitent une attention particulière afin de garantir la meilleure répartition des charges entre les différents dispositifs se partageant le travail (CPU, GPU, cœurs processeur, etc).

Le framework *MapReduce*, utilisé par *WebPie* et *MapResolve*, permet de distribuer facilement le processus de raisonnement. Il est cependant peu adapté au processus de raisonnement, puisque chaque nœud n'a accès qu'à ses propres données, ce qui peut amener à la génération de doublons et ralentir le processus comme nous l'avons vu. De plus, chaque nœud doit avoir terminé sa tâche pour pouvoir passer à la phase suivante, occasionnant une attente des nœuds entre eux et ralentissant le processus. Pour pallier ces problèmes, *MapResolve* est capable d'optimiser à la volée la répartition de charge en adaptant la quantité de triples utilisés pour l'application de chaque règle indépendamment. Cette capacité apporte un avantage non négligeable au système, qui peut adapter son comportement en fonction des données en entrée de chaque règle d'inférence.

Peters a amélioré la répartition de charge lors de l'exécution de l'algorithme de Rete. Alors qu'un nouveau processus léger était créé pour chaque nouveau triple entrant, il a mis en place un partitionnement de ces triples pour assigner des ensembles de triples à chaque processus léger. Le coût d'instanciation et d'exécution d'un processus léger étant non négligeable pour de grands volumes de processus créés, cette optimisation permet de limiter ce coût et d'améliorer la répartition de la charge de travail du raisonneur.

Le partitionnement (soit des données, soit des règles) proposé par Soma [47] a été étudié pour améliorer au maximum la parallélisation et la répartition de charge du raisonneur. Comme nous l'avons vu dans la section précédente, ce travail utilise un graphe de dépendance entre règles. Afin d'améliorer l'efficacité de ce graphe, les auteurs proposent d'ajouter un poids aux arcs du graphe. Ce poids représente la quantité de triples que peut potentiellement inférer la règle d'origine de l'arc en question. Pour utiliser cette méthode, il est nécessaire d'avoir des connaissances sur le comportement des règles d'inférence lors d'exécutions précédentes. Quand *MapResolve* s'adapte pendant son exécution au comportement du raisonneur, les auteurs proposent ici d'utiliser les informations disponibles pour améliorer la répartition de charge en amont. Cela nécessite cependant l'accès à des données historiques sur l'exécution des règles d'inférence utilisées.

Dans Rdfox [42], les requêtes nécessaires pour trouver les triples correspondant aux prémisses d'une règle sont parallélisées. La structure de données permettant de stocker les triples est optimisée pour les accès parallèles. Les triples provenant de l'ontologie sont attribués aux processus légers au fur et à mesure de leur récupération depuis l'ontologie.

3.4.4 Temps d'accès et empreinte mémoire

Que ce soit au travers du système de fichiers distribué de Hadoop®, ou des structures de données partagées entre plusieurs processus légers, le temps d'accès aux données du système a un impact considérable sur les performances du raisonneur. Il est donc primordial de fournir au raisonneur des structures avec une haute accessibilité, tout en proposant les caractéristiques nécessaires au processus (indexation, récupération de triples, etc). Afin de garantir le passage à l'échelle du raisonneur pour de grands volumes de données, il est également essentiel de mettre en place des structures de données compactes, permettant de stocker un maximum de connaissances dans un espace mémoire minimal.

Heino [55] encode les concepts composant les triples sous la forme d'entiers de 64 bits. L'association entre cet entier et le concept original est maintenue grâce à un dictionnaire. Cela permet de minimiser l'espace nécessaire à la représentation de la base de connaissances, les entiers prenant moins de place que les chaînes de caractères. De plus, la taille fixe des entiers permet de prévoir avec précision la quantité de connaissances qu'il sera possible de stocker pour un espace mémoire donné. Le temps nécessaire pour tester l'égalité entre deux chaînes de caractères est bien supérieur à celui nécessaire pour tester l'égalité entre deux entiers, et il en va de même pour un certain nombre d'opérations similaires. Le second avantage de la représentation des concepts sous la forme d'entiers est donc le gain de performances pour le traitement de ces concepts. De nombreuses règles de OWL nécessitent de tester l'égalité entre des concepts pour être appliquées, et le temps nécessaire à ces tests doit être réduit au maximum pour améliorer les performances du raisonneur.

Peters [62] réduit la taille mémoire nécessaire pour l'exécution de l'algorithme de Rete, et utilise une structure compressée pour stocker et indexer les triples. Ici encore, la taille de la représentation des connaissances est réduite au minimum, tout en permettant une récupération rapide des triples grâce à un système d'indexation.

Le raisonneur DynamiTE utilise des B-arbres et un dictionnaire pour indexer les connaissances dans un des fichiers sur disque. On retrouve ici le principe du dictionnaire faisant le lien entre les données encodées dans le système et les concepts sous leur forme originale.

Les temps d'accès à la mémoire vive sont significativement plus rapides que ceux pour un disque dur, même pour des disques SSD [6]. Il est donc du plus grand intérêt de minimiser les accès au disque, pour favoriser le stockage des données utilisées de manière récurrente en mémoire vive. Bien que *WebPie* utilise HDFS™ pour stocker la base de connaissances sur disque, la ABox peut être stockée en mémoire afin d'accélérer l'exécution des règles d'inférence, et donc fluidifier le processus de raisonnement. Dans DynamiTE, le schéma de l'ontologie et les triples nouvellement arrivés sont chargés en mémoire afin d'accélérer le raisonnement.

6. *Solid-State Drive*

En plus de permettre au système de stocker plus de connaissances, et donc de faciliter le passage à l'échelle, l'optimisation de l'empreinte mémoire des connaissances permet également d'en charger un maximum dans la mémoire vive, de manière à accroître significativement les performances du raisonneur.

Pour améliorer le temps d'accès aux données et leur empreinte mémoire RDFox [42] utilise une structure dédiée pour le stockage des triples. Les concepts sont ici aussi encodés sous la forme d'entiers, avec un dictionnaire permettant de faire le lien entre ces entiers et les concepts. Un système d'index basé sur le hachage est utilisé pour accélérer la recherche de triples, et chaque triple est lié aux autres triples avec lesquels il a un sujet, un prédicat ou un objet en commun.

Cette structure a été conçue de manière à utiliser un minimum de verrous. De cette façon, les accès parallèles aux données stockées sont plus rapides. De même, l'ajout de nouveaux triples est plus rapide, et peut être fait de manière parallèle.

3.4.5 Généricité par rapport au fragment

Alors que la plupart des travaux que nous avons étudiés sont dédiés à un ou plusieurs fragments définis [55, 64, 31, 65, 66], d'autres offrent une généricité leur permettant de s'adapter à un éventail plus important de fragments [40, 62, 47, 67].

La généricité de la solution proposée par Soma se fait en deux temps. L'architecture du raisonneur a été définie de manière à ne pas être directement dépendante des règles utilisées. L'utilisation du graphe de dépendances des règles montre une volonté des auteurs de proposer une solution générique. Ce graphe permet en effet de définir comment distribuer le processus de raisonnement, contrairement à l'alternative utilisée. Par exemple, dans *WebPie*, cela consiste à définir manuellement l'ordre optimal d'exécution des règles. Dans un second temps, l'utilisation d'un raisonneur extérieur, en l'occurrence Jena, améliore encore la généricité du raisonneur. Cela permet de ne pas avoir à implémenter chaque règle utilisée pour le raisonnement, tout en utilisant les éventuelles optimisations apportées par le raisonneur extérieur utilisé. Le graphe de dépendance des règles est mis à jour avec cette nouvelle règle, permettant de distribuer le raisonnement pour le nouvel ensemble de règles.

Motik [42] propose une solution utilisant des règles Datalog, permettant au système d'être compatible avec tout fragment dont les règles sont exprimables avec Datalog.

Volz [67] propose une option poussant encore plus loin la capacité d'adaptation de son système. L'ajout ou la suppression de règles d'inférence peut se faire lors de l'exécution du système, pendant l'inférence. Cela peut, entre autre, permettre d'activer ou de désactiver des règles en fonction des besoins de l'utilisateur. Dans le cadre d'un système incrémental fonctionnant en continu, cette fonctionnalité apporte une grande souplesse d'utilisation. Sans cette option, il est nécessaire d'arrêter le système pour ajouter ou supprimer des règles. Si le raisonneur reçoit de nouvelles connaissances venant de flux de données, il est possible que cet arrêt empêche la récupération de ces nouvelles données. Grâce à sa

capacité à ajouter et supprimer des règles pendant son exécution, le raisonneur défini par Volz assure une disponibilité continue même en cas de changement du fragment utilisé.

Les méthodes pour la classification incrémentale décrites dans les travaux de Kazakov [40] et Peters [62] sont présentées pour $\mathcal{EL}^+$, mais ne sont pas spécifiques à ce fragment. Les connaissances y sont représentées sous la forme d'*inférences*, contenant à la fois les prémisses d'une instance de règle et ses conclusions. L'exécution même des règles n'est donc pas formalisée dans cette représentation, permettant à la méthode décrite de s'adapter à d'autres fragments que $\mathcal{EL}^+$.

Alors que les solutions dédiés à un fragment donné doivent revoir leur méthode afin de raisonner sur un fragment différent, les solutions génériques offrent une flexibilité dont l'apport est indiscutable.

Cependant, cette généricité entraîne également des inconvénients. Les solutions génériques ne peuvent comporter les améliorations spécifiques à certaines règles d'inférence ou au fragment en lui-même.

3.5 Bilan sur les solutions de raisonnement

Dans ce chapitre, nous avons étudié les solutions existantes de raisonnement par lots et de manière incrémentale. Malgré la diversité des méthodes utilisées, nous avons pu dégager dans la section précédente des caractéristiques communes à plusieurs de ces solutions, motivées par la résolution de contraintes liées au problème du raisonnement. La table 3.9 récapitule comment ces problèmes ont été adressés par les différentes solutions étudiées. Les points clés mis en évidence dans la section précédente sont les piliers permettant de rendre le raisonnement efficace et évolutif.

La *gestion des doublons* est cruciale pour éviter une congestion du raisonneur et une surcharge de calcul. Il est nécessaire pour limiter l'impact de ces doublons sur le système de filtrer les doublons locaux et distants. La comparaison des triples inférés avec ceux présents dans la base de connaissances est une solution avantageuse. Elle permet d'éliminer tous les types de doublons (locaux ou distants). Il est cependant essentiel de limiter le coût des comparaisons qui en découlent. L'*ordonnancement des règles* est un problème qui, nous l'avons vu, dépend du fragment utilisé. Soma a cependant montré, par ses travaux, qu'il est possible de s'affranchir de ce problème en parallélisant l'exécution des différentes règles. Grâce au graphe de dépendance des règles, il est possible de déterminer comment distribuer des règles entre les différents nœuds. La *répartition de la charge* est primordiale pour assurer la parallélisation du processus de raisonnement. Nous avons vu que les inconvénients de *MapReduce* sont dûs au cloisonnement des données. Il est nécessaire de paralléliser le processus tout en permettant à chaque nœud du système d'avoir une vue d'ensemble des connaissances disponibles. La conception d'un modèle théorique pour un système efficace et capable de passer à l'échelle est indispensable. Mais un certain nombre d'optimisations se rapprochent plus de problèmes pratiques, liés à l'implémentation elle-même du raisonneur. On retrouve notamment la volonté récurrente des auteurs de *minimiser la taille de l'empreinte mémoire* de leurs différentes structures de données et leur *temps d'accès*. Cela nécessite de charger tout ou une partie de ces données dans la mémoire vive. Enfin, la *généricité face au fragment* est une caractéristique apportant avantages et inconvénients. Elle permet principalement au raisonneur de s'adapter à un maximum de situations et de cas d'usages. Cependant, il est difficile d'introduire des optimisations spécifiques à des règles lorsque le système n'est pas dédié à un fragment. L'utilisation d'un raisonneur externe est une alternative à ce problème. Le graphe de dépendance des règles joue ici un rôle en permettant une distribution de n'importe quel fragment de règles d'inférence. Il permet également, avec l'introduction de pondérations, d'influencer cette distribution en fonction des connaissances de l'utilisateur ou des données historiques. La capacité de *MapResolve* à adapter son fonctionnement au comportement du raisonneur pendant son exécution améliore encore l'adaptation du raisonneur. Le tableau 3.9 récapitule pour chaque solution étudiée les optimisations qu'elle propose.

Dans la partie suivante, nous présentons un système pour le raisonnement incrémental, dont la conception a été dirigée par les points clés que nous avons soulignés dans cette section. Cette solution permet l'exécution parallèle du processus de raisonnement

indépendamment du fragment de règles utilisé. Les doublons sont éliminés dès leur génération afin de limiter leur impact sur les performances de l'architecture. Les structures de données utilisées ont été définies dans le but d'optimiser la répartition de charge, les temps d'accès et l'espace nécessaire à la représentation des connaissances.

	Gestion des doublons	Ordonnancement des règles	Répartition de charge	Temps d'accès et empreinte mémoire	Généricité	Incrémental	Par lots
Volz [67]	-	-	-	-	Oui, ajout et suppression de règles pendant l'exécution	✓	
Soma [47]	-	Exécution parallèle	Utilisation du graphe de dépendance des règles et des données historiques	-	Oui, utilisation du raisonneur externe Jena		✓
Urbani [65] WebPie	Filtrés pendant la phase *Map*	Optimisé pour RDFS et OWL Horst	*MapReduce*	-	Dédié à RDFS et OWL Horst		✓
Schlicht [64]	Recherchés dans l'ensemble des triples inférés	Optimisé pour RDFS et OWL Horst	*MapReduce*, adaptation à la volée en fonction du comportement par règle	-	Dédié à $\mathcal{EL}+$, RDFS et Horst		✓
Heino [55]	Doublons locaux éliminés après l'exécution de la règle, doublons distants éliminés par comparaison avec la base de connaissances	Identique à WebPie	-	Entiers 64bits, dictionnaire	Dédié à RDFS		✓
Kazakov [56]	-	-	-	Réduction de l'empreinte des *inférences*	Oui	✓	
Urbani [66] DynamiTE	-	-	-	B-Arbres et dictionnaire	Dédié à ρdf	✓	
Motik [42]	-	Exécution parallèle	Distribution des triples de l'ontologie entre les différents processus légers	Dictionnaire, utilisation mimimale des verrous, indexation par hachage	Oui, règles Datalog	✓	✓
Peters [62]	Filtrés par comparaison avec la base de connaissances	-	Réduction du nombre de processus légers utilisés dans RETE	Réduction de l'empreinte mémoire de RETE	Oui		✓

TABLE 3.9 – Tableau récapitulatif des points clés des solutions étudiées dans l'état de l'art des solutions de raisonnement

III

Contribution

4 Système pour le raisonnement incrémental

Sommaire

Ce chapitre présente l'architecture de notre système de raisonnement incrémental. Nous commençons ce chapitre par situer le problème que nous abordons dans la section 4.1. Après avoir introduit le formalisme du raisonnement incrémental, nous donnons l'ensemble des critères qui nous ont amenés à proposer cette architecture (section 4.3) : exécution parallèle et passage à l'échelle, limitation de la propagation des doublons au plus tôt, gestion de flux de données et agnosticisme au fragment utilisé pour l'inférence. Par la suite, nous détaillons le fonctionnement de notre architecture (section 4.4) en donnant pour commencer une vue d'ensemble, suivie d'une description des différents modules qui la composent, pour terminer par un exemple illustrant comment le système se comporte de l'arrivée de nouveaux triples jusqu'à la fin de l'inférence. Nous revenons ensuite sur un aspect essentiel du raisonneur : sa capacité à s'adapter au fragment, en présentant le graphe de dépendance des règles, indispensable à l'initialisation de l'architecture, puis en détaillant la manière dont les règles sont effectivement appliquées. Nous présenterons après cela les trois paramètres de l'architecture : la taille des buffers, la valeur du timeout et le fragment lui-même. Pour finir, nous présentons deux modes d'inférence permettant d'adapter le fonctionnement de l'architecture à des contraintes temporelles (section 4.7). Le premier mode se place dans le cas de figure où l'utilisateur connaît les connaissances qu'il souhaite utiliser. Il peut, dans ce mode, prioriser ces connaissances afin qu'elles soient inférées en priorité. Le second propose une adaptation sans connaissance a priori sur l'utilisation des données.

4.1 Problématique

L'état de l'art a montré des solutions de raisonnement par lots permettant le passage à l'échelle de ce processus complexe et coûteux. Ces solutions ne permettent pas de maintenir la matérialisation des données inférées après l'ajout de nouvelles données, sans reprendre tout le processus de raisonnement. Le raisonnement incrémental répond à ce besoin en utilisant les matérialisations antérieures et en y appliquant des mises à jour. Les solutions existantes sont cependant peu nombreuses et peinent à passer à l'échelle pour des mises à jour importantes. Par exemple, la solution présentée dans [16] n'est efficace que lorsque les mises à jour ne dépassent pas 2% de l'ontologie de départ.

Dans ce chapitre, nous décrivons notre architecture pour le raisonnement incrémental. Nous avons centré nos efforts pour pallier les inconvénients des solutions que nous avons étudiées tout en intégrant les points clés décrits dans la section 3.4. Pour répondre au problème posé de manière optimale, notre architecture a été définie afin de permettre le passage à l'échelle du processus de raisonnement incrémental, en gérant l'arrivée de nouvelles données sous forme de flux de triples. Les doublons sont gérés dès leur génération afin de limiter leur impact sur les performances. Enfin, le découpage de l'architecture en modules indépendants permet à notre solution d'être agnostique face au fragment utilisé pour le raisonnement. Chaque module est associé à une règle d'inférence afin d'améliorer la répartition de charge. La seule exigence nécessaire pour utiliser cette architecture est de disposer d'une implémentation de chaque règle d'inférence utilisée.

4.2 Formalisation du raisonnement incrémental

4.2.1 Méthode générale

Nous décrivons dans cette section notre méthode permettant de rendre le raisonnement incrémental.

Soit G un graphe RDF, représentant une base de connaissances avant la matérialisation. Soit un système déductif $\models$ et un graphe H tel que $G \models H$. H représente donc la matérialisation de G, c'est-à-dire la base de connaissances après inférence. On souhaite maintenant ajouter des triples dans H, regroupés dans un graphe Δ et calculer la matérialisation sur $H \cup \Delta$. Le système déductif $\models$ permet d'obtenir la matérialisation F telle que $H \cup \Delta \models F$. Puisque H est obtenu en appliquant $\models$ à G, on a également $G \cup \Delta \models F$.

Le problème ici est qu'aucun parti n'est tiré de la matérialisation précédente. On cherche donc le système déductif $\models_\Delta$ tel que $H \models_\Delta F$, qui ne calcule que la matérialisation entraînée par les nouveaux faits de Δ. Plus précisément, pour que les conclusions d'une règle soient ajoutées à la matérialisation, au moins une de ses prémisses doit appartenir à Δ.

Un système incrémental est donc un ensemble de systèmes déductifs $\{\models, \models_{\Delta_1}, \ldots, \models_{\Delta_n}\}$ qui, pour un graphe G, un ensemble de *mises à jour* $\{\Delta_1, \ldots, \Delta_n\}$ et un ensemble de graphes $\{H_1, \ldots, H_n\}$, est défini tel que :

- $G \models H_1$;
- $H_i \models_{\Delta_i} H_{(i+1)}$ $(1 \leq i \leq n)$.

L'algorithme 4.1 présente une adaptation de l'algorithme 2.1 permettant de passer de H_i à H_{i+1}. La première différence est que cet algorithme prend en entrée deux ensembles de triples : H_i contenant la matérialisation résultant du dernier raisonnement et Δ contenant les nouveaux triples ajoutés à la base de connaissances. La seconde est que, comme précisé plus haut, seules les conclusions des règles dont toutes les prémisses sont dans $H_i \cup \Delta$ et au moins une dans Δ, sont ajoutées à H_{i+1}. L'ensemble des prémisses peuvent toutes se trouver dans Δ.

Pour simplifier la notation, on définit $\in^1$ (*au moins 1 dans*) comme suit :

$$G \in^1 H \iff \exists t \in G, t \in H \tag{4.1}$$

ALGORITHME 4.1 – Adaptation de l'algorithme 2.1 pour le raisonnement incrémental

Données : H_i, $Regles$, Δ
Résultat : H_{i+1}

```
1  H_{i+1} ← H_i ∪ Δ
2  do
3  |  Δ' ← ∅
4  |  pour regle ∈ Regles faire
5  |  |  pour regle.premices ∈ H_i ∪ Δ et regle.premices ∈¹ Δ faire
6  |  |  |  Δ' ← regle.conclusions \ H_{i+1}
7  |  |  |  H_{i+1} ← H_{i+1} ∪ regle.conclusions
8  |  |  fin
9  |  fin
10 |  Δ ← Δ'
11 while Δ ≠ ∅;
```

4.2.2 Filtrage des règles

Dans l'algorithme 4.1, les prémisses de chaque règle d'inférence sont recherchées dans $H_i \cup \Delta$. Plutôt que de faire cette recherche sur l'ensemble des règles, l'algorithme 4.2 filtre les règles dont au moins l'une des prémisses peut se trouver dans Δ. Pour cela, une règle n'est utilisée que si au moins un prédicat des triples de Δ contient au moins l'un des prédicats présents dans les prémisses de la règle. Par exemple, la règle `cax-sco` ne sera utilisée que si au moins l'un des triples de Δ a pour prédicat `subClassOf`.

On notera $\mathcal{R}$ l'ensemble des règles d'inférence. Pour des questions de lisibilité, on définit $predicats(G)$ qui retourne la liste $p \in \mathbf{U}$ des prédicats contenus dans les triples de G. On définit une application $Regles : 2^{\mathbf{U}} \rightarrow 2^{\mathcal{R}}$ qui associe à un ensemble d'URI U le plus grand ensemble de règles $R \in \mathcal{R}$ tel que $\forall r \in R, predicats(R) \in U$.

L'algorithme 4.2 contient la modification de la boucle de parcours des règles (c.f. ligne 4) permettant de mettre en place le filtrage des règles.

Avec cette modification, un système $\models_\Delta$, en plus de limiter les conclusions à ajouter à la base de connaissances, n'utilise qu'une partie des règles afin d'alléger le processus.

ALGORITHME 4.2 – Filtrage des règles d'inférence en fonction des triples utilisés pour l'inférence

Données : H_i, $Regles$, Δ
Résultat : H_{i+1}

```
1  H_{i+1} ← H_i ∪ Δ
2  do
3  |  Δ' ← ∅
4  |  pour regle ∈ Regles(predicats(G)) faire
5  |  |  pour regle.premices ∈ H_i ∪ Δ et regle.premices ∈¹ Δ faire
6  |  |  |  Δ ← regle.conclusions \ H_{i+1}
7  |  |  |  H_{i+1} ← H_{i+1} ∪ regle.conclusions
8  |  |  fin
9  |  fin
10 |  Δ ← Δ'
11 while Δ ≠ ∅;
```

4.2.3 Bufferisation et filtrage avancé

La construction des conclusions de règles à partir des prémisses est une opération simple et linéaire. En revanche, la recherche de prémisses dans un ensemble de triples est l'opération la plus coûteuse de l'algorithme 4.2. Il est donc préférable de limiter cette recherche.

Dans cette optique, il est donc déconseillé d'utiliser l'algorithme 4.2 pour chaque nouveau triple. Il est plus intéressant et moins coûteux de ne lancer l'inférence que lorsque Δ contient déjà un certain nombre de triples. Les triples en attente doivent donc être stockés dans une structure dédiée. Les buffers, ou espaces de mémoire tampon, sont les structures utilisées dans ce genre de situation. Il s'agit d'ensembles d'objets, ici de triples, déclenchant une action particulière, ici l'inférence, lorsque le nombre d'éléments présents dépasse un certain seuil.

Comme nous l'avons vu dans la section précédente, il est possible de savoir quels triples sont susceptibles d'être utilisés par une règle. Grâce à cela, le filtrage peut être amélioré en filtrant les triples par règle et non plus de manière globale. Ainsi, chaque règle peut disposer de son propre buffer, ne recevant que les triples qui peuvent être utilisés par cette règle. La quantité de triples dans laquelle les prémisses sont recherchées pour chaque règle est réduite et il en est donc de même pour le coût de l'opération.

Une fois l'inférence pour une règle donnée terminée, les conclusions générées ne seront envoyées qu'aux règles susceptibles de les utiliser.

4.3 Caractéristiques attendues du système

Nous décrivons ici les caractéristiques principales qu'offre l'architecture que nous proposons. Elles sont exposées brièvement dans l'objectif de fournir une vue d'ensemble. La section 4.4 détaille chaque élément du système, tout en décrivant comment ces caractéristiques sont adressées.

4.3.1 Vue d'ensemble

La figure 4.1 présente l'architecture et l'illustre sur la base d'un fragment de trois règles (R_1, R_2 et R_3) et permet de comprendre le circuit qu'emprunte chaque type de triple.

L'architecture de notre système de raisonnement incrémental est modulaire. Des modules nommés **exécuteurs** ont la charge de l'application des règles. Les **exécuteurs** sont indépendants les uns des autres. Chacun est dédié à une règle du fragment logique considéré. Une même règle peut faire l'objet de plusieurs **exécuteurs**. Un **triplestore** central est accessible par les modules afin de stocker et partager l'ensemble des connaissances disponibles. Les triples provenant de flux de données sont envoyés au **distributeur général**, point d'entrée du système.

Les triples sont générés par des objets connectés, des systèmes de connaissances ouverts, ou d'autres systèmes similaires. Ils sont produits continuellement au cours du temps, constituant autant de flux d'entrée. Le **distributeur général** transmet les triples arrivant au **triplestore** pour consignation, ainsi qu'aux **buffers** (ou espaces tampons) d'une sélection de modules : ceux acceptant ce type de triples en entrée. Lorsqu'un **buffer** est saturé ou que le délai de temporisation (timeout) est atteint, la règle associée est appliquée sur les triples contenus dans le **buffer** afin d'extraire les connaissances implicites. Les données présentes dans le **triplestore** sont utilisées pour l'inférence. Un **distributeur** est chargé d'enregistrer les triples inférés dans le **triplestore** et de les redistribuer aux **buffers** de la même manière que le **distributeur général**. Le **distributeur général** est un **distributeur** auquel sont abonnés tous les **buffers**. Le fonctionnement de ces modules est décrit de manière plus précise dans les sections suivantes.

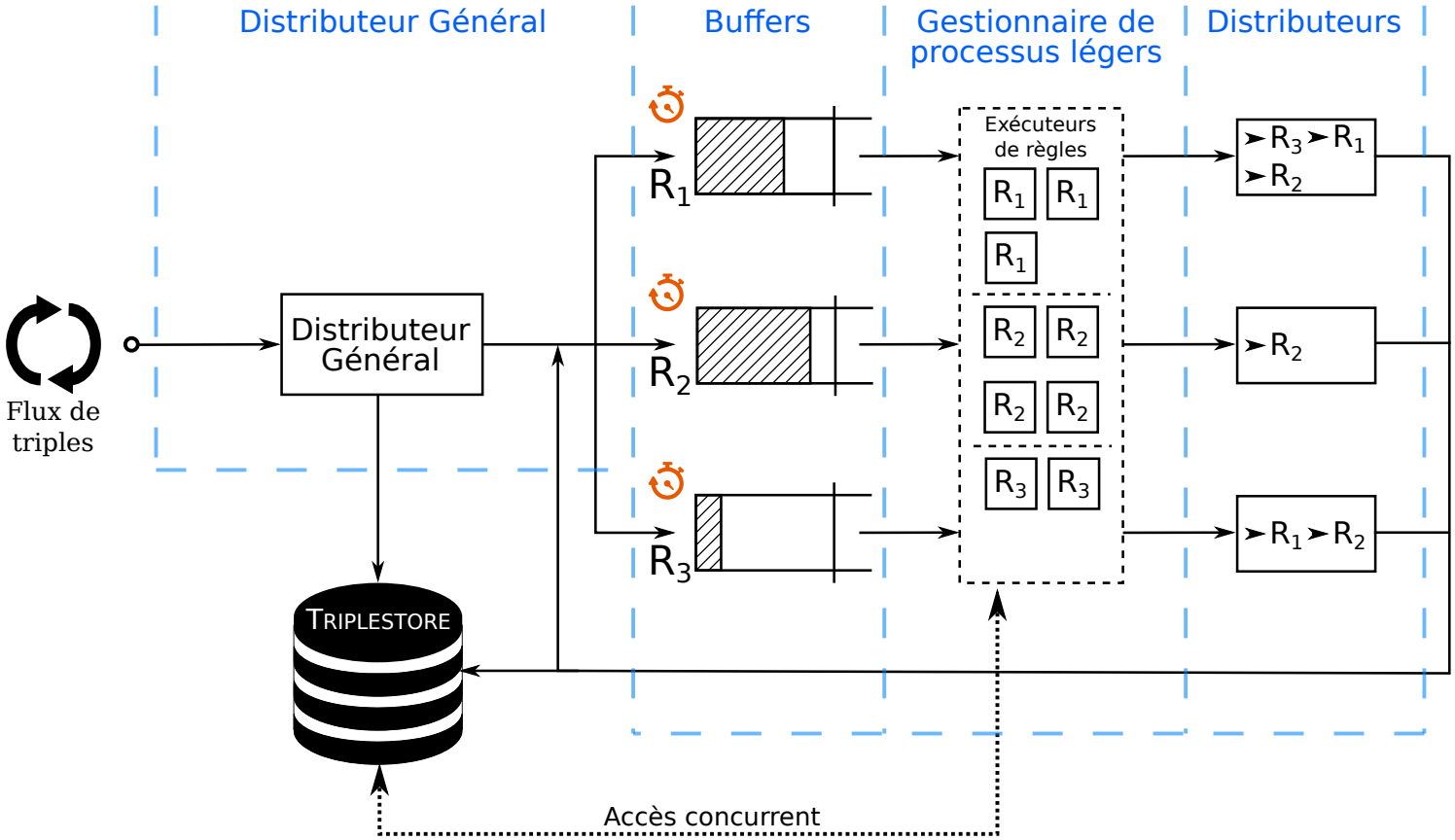

FIGURE 4.1 – Architecture proposée pour le raisonnement incrémental sur des flux de triples

4.3.2 Exécution parallèle et passage à l'échelle

L'un des aspects critiques d'un raisonneur est sa capacité à gérer de grands volumes de données. Afin de permettre à notre solution de passer à l'échelle, son architecture a été conçue sous la forme de modules indépendants. Chaque module dispose de son propre processus léger, permettant à l'ensemble des modules de s'exécuter de manière parallèle. Chaque règle d'inférence est associée à un module indépendant. Ils reçoivent les triples pouvant être utilisés par la règle à laquelle ils sont associés et redistribuent les triples inférés aux autres modules afin de continuer l'inférence.

Plusieurs exécutions d'une même règle d'inférence sur différents ensembles de triples peuvent être lancées par un même module. Cela permet d'éviter une attente du système dans le cas où une règle serait particulièrement sollicitée. Si une règle est déjà en cours d'exécution et que d'autres triples compatibles avec ses prémisses arrivent, une nouvelle instance de cette règle pourra être lancée pour ne pas mettre le système en attente.

De manière à s'assurer que chaque triple envoyé dans le raisonneur est traité, un espace tampon associé à chaque règle récupère les triples qui peuvent être utilisés par la règle. Cela permet de gérer le nombre de triples arrivant ou inférés. Le fonctionnement de ces espaces tampons est détaillé dans la section 4.4.2.

4.3.3 Limitation des doublons

Comme cela a été souligné dans différents travaux de l'état de l'art [55, 64, 46, 65], l'un des problèmes majeurs du raisonnement est la génération inévitable de doublons, qui freine le processus d'inférence. Une solution est de supprimer ces doublons au plus tôt, dès leur génération, afin d'empêcher leur dispersion dans le système. Dans notre architecture, chaque module dispose d'un accès concurrent à l'ensemble des données disponibles, stockées dans un **triplestore**, qu'elles soient inférées ou originaires d'un point d'entrée. Cela permet d'interdire la dispersion d'un triple déjà inféré dans le système. Les doublons sont générés, mais supprimés avant d'être renvoyés dans un autre module du système. De plus, de par son architecture, le **triplestore** est imperméable aux doublons. De manière similaire à la technique présentée dans les travaux de Heino [55], la suppression se fait à deux échelles : au niveau des modules en supprimant les doublons générés à l'intérieur même du module et à un niveau global en vérifiant si les triples inférés dans le module ne sont pas déjà présents dans le **triplestore**. De cette manière, chaque triple n'est considéré qu'une seule fois, réduisant le risque de redondance dans les triples inférés.

4.3.4 Gestion de flux de données

La nature dynamique du Web [25] a mené à la mise à disposition de données sous forme de flux. Le raisonnement est souvent considéré comme un processus ponctuel, utilisant des règles connues par avance. L'évolution constante des bases de connaissances nous impose cependant d'aller vers des systèmes continus et adaptatifs. L'aspect paral-

lèle intrinsèque de notre architecture, gérant des flux de triples circulant à travers les différents modules, lui permet nativement de recevoir des données venant à la fois de sources statiques et de multiples flux de données.

4.3.5 Agnosticisme au fragment

Comme nous l'avons vu dans l'état de l'art, il existe un grand nombre de règles d'inférence, réparties dans différents fragments. Afin de fournir une solution la plus générique possible et apte à répondre à un maximum de besoins, nous proposons une architecture agnostique au fragment. L'accent a été mis sur l'évolutivité du système, plutôt que sur l'optimisation pour un fragment donné. Il est donc possible d'ajouter et/ou de supprimer des règles au fragment utilisé. En plus d'être compatible avec les règles prédéfinies, entre autres par le W3C [77], notre architecture prend en charge toute règle implémentée en suivant une spécification sous la forme d'une interface Java, dont les détails sont donnés dans la section 4.5.2.

4.4 Fonctionnement détaillé

Dans cette section, nous détaillons le fonctionnement interne de notre architecture. Nous nous attarderons sur chaque composant (sections 4.4.1 à 4.4.5), puis nous verrons un exemple simple illustrant le trajet des données au travers de notre architecture en section 4.4.7.

4.4.1 Distributeur général

Ce module reçoit les nouveaux triples envoyés au raisonneur. Il stocke ces triples dans le **triplestore** afin qu'ils soient accessibles par chaque module (cf. section 4.4.5), puis les envoie aux **buffers** associés à chaque règle pouvant utiliser ces triples. Afin de déterminer quels triples peuvent être utilisés par une règle, nous utilisons un graphe de dépendance des règles, présenté en détail dans la section 4.5.1.

Pour des raisons de lisibilité, un seul **distributeur général** apparaît sur la figure 4.1, mais plusieurs instances de ce module peuvent cohabiter pour améliorer la réactivité de l'architecture. Multiplier les instances de ce module permet par exemple de récupérer simultanément des triples de plusieurs sources.

Ce module est un cas particulier de **distributeur** (c.f. section 4.4.4), envoyant des triples à tous les **buffers** du système.

4.4.2 Buffers

Chaque règle se voit assigner une mémoire tampon (**buffer**) en charge de collecter les triples envoyés par les instances du **distributeur général** et des **distributeurs** des différentes règles (voir section 4.4.4). Lorsque le nombre de triples stockés dans un **buffer** dépasse un seuil, appelé taille du **buffer** (c.f. section 4.6.1), une nouvelle instance d'un **exécuteur de règle** est créée. Les triples contenus dans le **buffer** sont envoyés à cette instance pour traitement. Le **buffer** peut alors recevoir de nouveaux triples. De cette manière, les **buffers** ne restent jamais pleins et le système ne se trouve jamais dans une situation de blocage.

L'utilisation de **buffers** répond à deux problèmes. Tout d'abord, ils permettent de collecter les triples en entrée afin de n'appliquer les règles d'inférence que sur un nombre suffisant de triples. Comme nous l'avons vu dans le chapitre précédent, l'exécution d'une règle pour chaque triple est trop coûteuse. L'utilisation de **buffers** permet d'alléger ce coût en retenant les triples arrivant jusqu'à ce que leur nombre justifie l'application d'une règle.

Il existe plusieurs solutions de gestion de **buffers**. La première solution que nous avons envisagée consiste en deux piles de triples de taille fixe. L'une de ces piles reçoit les nouveaux triples envoyés aux **buffers**. Lorsqu'elle est pleine, les piles sont échangées. La deuxième pile, alors vide, prend le relais de la première pour recueillir les nouveaux

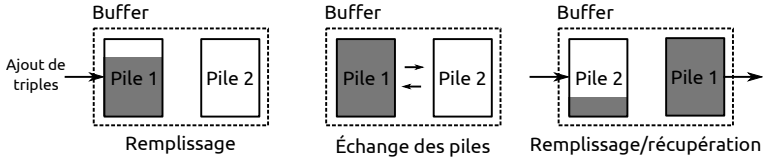

FIGURE 4.2 – Fonctionnement du **buffer** composé de deux piles de triples

triples, pendant que la première, pleine, est vidée. L'avantage de cette méthode, représentée dans la figure 4.2, est qu'elle est non bloquante. Même si le processus de récupération des triples stockés dans le **buffer** n'est pas immédiat, il est toujours possible d'ajouter des triples à la pile non utilisée par le processus de récupération. En revanche, si le processus de récupération n'est pas terminé et que la seconde pile est, elle aussi, pleine, le **buffer** se retrouve dans une situation de blocage. Il ne peut plus recevoir de triples. Une solution serait de ne pas se limiter à deux piles. Mais cette option n'est pas viable car elle ne fait que reporter le problème. Quelque soit le nombre de piles, dans le cas où elles seraient toutes pleines alors que le processus de récupération n'est pas terminé, le même problème se présenterait. Afin de gérer ces blocages, deux attitudes s'opposent. La première consiste à ne pas tenir compte des triples envoyés lors du blocage. Cette solution est inenvisageable dans notre cas puisque la perte de connaissances entraînerait l'incomplétude du raisonnement. La seconde solution consiste à rendre l'ajout de nouveaux triples au **buffer** bloquant. Autrement dit, le processus ajoutant de nouveaux triples doit attendre que le **buffer** soit à nouveau opérationnel pour terminer l'ajout. Mais cette solution n'est pas non plus satisfaisante. Le processus en amont devrait stocker les triples arrivant en attendant de pouvoir les ajouter au **buffer**, ne faisant que reporter le problème vers un autre module du système.

Nous avons donc opté pour une autre forme de **buffer**, illustrée par la figure 4.3. Il est composé d'une file infinie de triples. Un compteur, initialisé à 0, est incrémenté pour chaque triple ajouté à la file. Lorsque ce compteur atteint le seuil de la taille du **buffer**, un **exécuteur de règle** est créé, puis le compteur est remis à 0. L'**exécuteur de règle** instancié récupère un nombre de triples dans la file du **buffer** correspondant à la taille du **buffer**. Grâce à ce procédé, le **buffer** peut constamment recevoir de nouveaux triples, qui seront alors récupérés au fur et à mesure par les **exécuteurs de règle**.

La taille du **buffer** détermine le nombre de triples nécessaires pour déclencher l'instanciation d'un nouvel **exécuteur de règle**. Avec seulement cette contrainte, si le nombre de triples envoyés dans un **buffer** ne dépasse pas sa taille, ces triples ne seront jamais envoyés à un **exécuteur de règle**. Pour éviter cette situation, nous avons introduit une seconde condition permettant de vider un **buffer**. Si aucun triple n'a été envoyé à un **buffer** non vide après un certain laps de temps appelé *timeout*, celui-ci se comporte comme s'il était plein. Une nouvelle instance d'un **exécuteur de règle** est créée et les triples contenus dans le **buffer** sont envoyés à cette instance pour traitement.

Dans le cadre d'un système recevant de nouvelles données en continu, il peut arriver que le système ne reçoive que rarement des triples pouvant être utilisés par des règles.

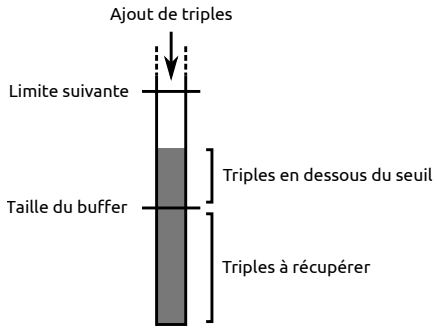

FIGURE 4.3 – Fonctionnement du **buffer** composé d'une file infinie de triples

Dans ce cas, le *timeout* permet aux **buffers** correspondants de déclencher l'instanciation d'un **exécuteur de règle** plus souvent et permet donc d'améliorer la réactivité du système en vidant des **buffers** non pleins.

4.4.3 Exécuteurs de règle

Les **exécuteurs de règle** sont les modules implémentant le raisonnement par chaînage avant (*forward chaining*) sur la base des triples donnés en entrée ainsi que de ceux issus du **triplestore** pouvant concourir à la levée de connaissances pour une règle donnée. Un **exécuteur de règle** correspond à une règle. Lorsqu'un **buffer** est plein ou n'a pas reçu de nouveaux triples depuis un certain temps sans être vide, une nouvelle instance d'**exécuteur de règle** est créée. Cette instance récupère alors les triples contenus dans le **buffer**, dans la limite de la taille du **buffer**. Ces triples sont utilisés conjointement avec les connaissances du **triplestore** pour appliquer la règle d'inférence associée au **buffer**.

Pour paralléliser le processus de raisonnement, deux méthodes sont employées dans l'état de l'art : par découpage des règles ou par découpage des données. Le filtrage opéré sur les triples envoyés aux **buffers** en fonction de la règle à laquelle ils sont associés entraîne un découpage du processus par règle. Les règles d'inférence sont exécutées en parallèle les unes des autres. À ce découpage par règle nous avons ajouté un découpage des données, rendu possible par les exécuteurs de règle. Plusieurs de ces modules exécutant la même règle peuvent fonctionner en parallèle, si le nombre de triples envoyés à un **buffer** est suffisamment conséquent pour déclencher plusieurs instanciations parallèles. Cela permet d'éviter qu'un **buffer** reste plein en attendant la fin de l'exécution d'un **exécuteur de règle**. De plus, les données sont traitées plus rapidement puisque les **exécuteurs de règle** n'ont pas à attendre la fin de l'exécution des autres **excuteurs de règle**. La figure 4.4 illustre l'avantage de l'exécution parallèle d'une même règle.

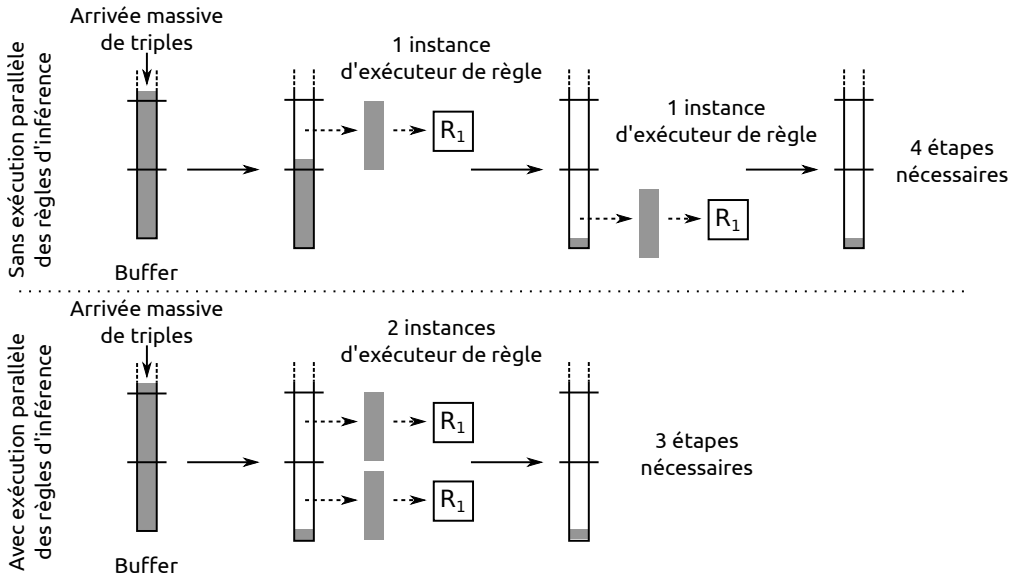

FIGURE 4.4 – Avantage de l'exécution parallèle d'une même règle d'inférence en cas d'arrivée importante de nouveaux triples

Ce mode de fonctionnement amène à une exécution des règles par lots de triples. Cela permet de diminuer la quantité de requêtes faites au **triplestore** (c.f. section 4.5.1). Quelle que soit la quantité de triples récupérés depuis le **buffer**, les requêtes faites au **triplestore** pour trouver les triples complémentaires nécessaires à l'application d'une règle sont les mêmes. Il est donc intéressant d'utiliser les résultats de ces requêtes pour un ensemble de triples, plutôt que d'interroger le **triplestore** pour chaque nouveau triple.

Contrairement à certaines solutions de l'état de l'art [55, 79], l'exécution des différentes règles est régie par l'arrivée des triples dans le système. Il n'y a donc pas d'ordonnancement des règles, qui sont exécutées dès que nécessaire (lorsqu'un **buffer** est plein ou a atteint le timeout). Dans la section 4.7, nous présentons néanmoins deux méthodes permettant de favoriser certaines règles.

4.4.4 Distributeurs

Ces modules fournissent les trois opérations suivantes :

1. Collecter les triples inférés par les **exécuteurs de règle** associés ;
2. Ajouter les triples inférés au **triplestore** ;
3. Redistribuer les triples inférés au sous-ensemble de règles sensibles à cette nouvelle connaissance, i.e. les **buffers** des règles prenant ces triples en entrée.

Cette redistribution est assurée indépendamment du fragment logique considéré, en connectant les **distributeurs** aux **buffers** à l'aide du graphe de dépendance de règles

du fragment considéré, comme présenté plus loin (section 4.5.1).

Les **distributeurs** jouent un rôle primordial dans la répartition des charges du processus de raisonnement. On pourrait penser que le **distributeur général** est suffisant pour la redistribution des triples entrants ou inférés. Il suffit pour cela que chaque **exécuteur de règle** envoie les triples qu'il infère vers le **distributeur général**. Cependant, dans une telle configuration, ce module est central dans le sens où tous les triples transitent par lui. Il représenterait donc un goulot d'étranglement.

Dans la configuration présentée, où chaque règle dispose de son propre distributeur, les nouveaux triples inférés sont directement envoyés aux règles qui peuvent les utiliser, sans devoir passer par un objet central avant d'être redistribués. L'ajout des triples inférés au **triplestore** permet également de vérifier s'ils y sont déjà présents, afin d'éliminer les doublons. L'envoi des triples aux différents **buffers** est faite de manière indépendante, directement par chaque **distributeur**.

4.4.5 Triplestore

Le **triplestore** est le module chargé de récolter, stocker et rendre accessible chaque triple reçu en entrée ou inféré par le système. Chaque nouveau triple, qu'il soit reçu d'une source de données ou inféré par le raisonneur, y est ajouté dès son arrivée dans le système. Cela assure que toute connaissance envoyée au raisonneur ou inférée peut être immédiatement utilisée pour le raisonnement. L'objectif est de rendre tout triple connu disponible au plus tôt par l'ensemble des modules constituant l'architecture. Les **exécuteurs de règle** accèdent à ces nouvelles connaissances pour le raisonnement, sans devoir attendre une prochaine exécution pour les utiliser. Les **distributeurs**, y compris le **distributeur général**, écartent d'éventuels doublons plus efficacement en ayant accès à l'ensemble des données disponibles.

Le **triplestore** doit fournir les fonctionnalités suivantes afin de permettre à notre architecture de fonctionner :

- Ajout d'un ou plusieurs triples dans le **triplestore**, fonctionnalité utilisée par les **distributeurs** ;
- Recherche des triples correspondants aux prémisses d'une règle, fonctionnalité utilisée par les **exécuteurs de règle**.

En plus de ces fonctionnalités, le **triplestore** doit avoir plusieurs caractéristiques permettant au raisonnement d'être le plus rapide possible et donc à l'architecture de passer à l'échelle.

Tout d'abord le stockage des triples doit se faire de manière à optimiser les deux opérations citées précédemment. Une simple file de triples permet l'ajout rapide de nouveaux triples. Mais la recherche d'éléments dans une file est trop coûteuse ($O(n)$). Nous avons opté pour un partitionnement vertical [32]. Les triples y sont indexés par

prédicat, puis par sujet et enfin par objet. À chaque prédicat correspond une liste de sujets et à chaque sujet correspond une liste d'objets. Cette indexation est le meilleur compromis pour la recherche des prémisses des règles de OWL [77]. L'ajout d'un nouvel élément y est rapide, tout en garantissant une recherche efficace.

Le **triplestore** doit également être le plus compact possible afin de stocker un maximum de connaissances, toujours en vue d'un passage à l'échelle plus efficace. Pour cela, nous utilisons un dictionnaire associant un entier à chaque concept. Ces entiers sont utilisés pour représenter les triples dans le partitionnement vertical. Cette méthode présente trois avantages :

- La taille nécessaire à la représentation d'un triple est à la fois petite et fixe ;
- Les tests d'égalité sur des entiers étant plus rapides que ceux entre des structures plus élaborées, les tests d'égalité entre les triples sont de fait plus rapides ;
- Il n'y a pas de répétition des concepts, stockés en un seul exemplaire dans le dictionnaire, réduisant l'empreinte mémoire du **triplestore**.

La figure 4.5 illustre un exemple du stockage des triples.

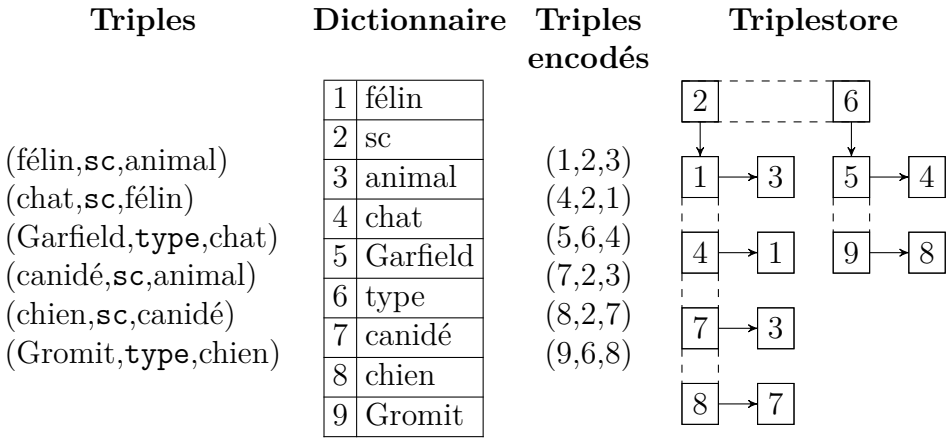

FIGURE 4.5 – Représentation des triples dans le **triplestore** grâce à un dictionnaire de concepts et au partitionnement vertical

Chaque liste de concepts représentant des prédicats, des objets ou des sujets ne permet pas de stocker plusieurs fois la même valeur. Le **triplestore** ne peut donc pas stocker plusieurs fois le même triple. Grâce à cette caractéristique, il n'est pas nécessaire de gérer les doublons à l'intérieur du **triplestore**.

La distribution du processus de raisonnement à la fois par rapport aux règles et aux données entraîne l'exécution parallèle de nombreux **exécuteurs de règle** en simultané. Chacun de ces **exécuteurs de règle** devant accéder au **triplestore**, il est indispensable que celui-ci soit capable de gérer les accès concurrents. Cette gestion de la concurrence

passe par l'utilisation de verrous. Si plusieurs modules peuvent simultanément lire les informations contenues dans le **triplestore**, aucun module ne peut effectuer d'opération de lecture ni d'écriture lorsqu'une opération d'écriture est en cours. De même, si une opération de lecture est en cours, aucune opération d'écriture n'est autorisée. Un ralentissement peut découler de cette situation. Pour pallier cet inconvénient, le **triplestore** doit permettre l'ajout de triples – constituant la principale opération d'écriture – et la recherche de triples – utilisée pour l'inférence et l'élimination des doublons – le plus rapidement possible. Nous détaillerons plus les optimisations proposées dans ce sens dans la section 5.1.2.

4.4.6 Détection de la fin de l'inférence

L'aspect parallèle de notre architecture rend la détection de la fin de l'inférence non triviale. Pour une exécution séquentielle du raisonnement, à la fin de l'exécution de l'ensemble des règles, il suffit de vérifier si des triples ont été inférés ou non. Si c'est le cas, les règles sont à nouveau appliquées sur ces nouveaux triples. Dans le cas contraire, le raisonnement est complet.

Dans notre architecture, les règles sont exécutées en parallèle. Une règle peut être exécutée plusieurs fois alors qu'une autre ne sera pas exécutée si aucun triple ne concordant avec ses prémisses ne se présente. Il n'y a donc pas de phase d'attente entre l'exécution de toutes les règles. Pour déterminer la fin du raisonnement, nous commençons donc par attendre la fin de l'exécution de tous les **exécuteurs de règle**. Cependant ce n'est pas suffisant. En effet, s'il n'y a aucun **exécuteur de règle** en cours d'exécution mais qu'au moins un **buffer** n'est pas vide, cela signifie qu'il reste des triples à traiter.

Le raisonnement est donc terminé lorsque :

- Tous les **buffers** sont vides ;
- Tous les **distributeurs** ont distribué tous leurs triples ;
- Aucun **exécuteur de règle** n'est en cours d'exécution.

Cette vérification, illustrée par l'algorithme 4.3, est faite après chaque exécution de règle. Nous sommes dans le cas d'une inférence *fixed-point*, pour laquelle le processus est répété jusqu'à atteindre un point fixe représenté par plusieurs conditions.

Algorithme 4.3 – Détection de la fin de la procédure d'inférence

```
1 /* Lancement du raisonnement                                    */
2 tant que executeursEnCours()>0 et buffersNonVides()>0 faire
3 |  attendreFinProchainExecuteur()
4 fin
5 /* Fin du raisonnement                                         */
```

4.4.7 Exemple détaillé d'inférence avec notre architecture

Ici nous prenons un exemple simple pour illustrer le fonctionnement de notre architecture. Nous utilisons un fragment composé de deux règles, `scm-sco` (R1) et `cax-sco` (R2), rappelées ci-dessous. Pour des raisons pratiques, `subClassOf` sera noté `sc` dans cet exemple.

$$\frac{c_1 \texttt{ sc } c_2, \quad c_2 \texttt{ sc } c_3}{c_1 \texttt{ sc } c_3} \text{ (scm-sco)} \qquad \frac{c_1 \texttt{ sc } c_2, \quad x \texttt{ type } c_1}{x \texttt{ type } c_2} \text{ (cax-sco)}$$

Dans un premier temps, deux triples sont envoyés en entrée du système : il s'agit de $< Garfield, \texttt{type}, chat >$ et $< chat, \texttt{sc}, félin >$. Le triple $< félin, \texttt{sc}, animal >$ est envoyé dans un second temps. La taille des **buffers** est fixée à 2. Le **triplestore** est vide au début de l'exécution.

Détaillons maintenant les différentes étapes du fonctionnement, en indiquant quelles données seront présentes dans quels modules du système. Les tableaux 4.1 et 4.2 illustrent le déroulement de cet exemple.

Étape 1 Les triples $< Garfield, \texttt{type}, chat >$ et $< chat, \texttt{sc}, félin >$ envoyés dans le système sont récupérés par le **distributeur général**.

Étape 2 Le **distributeur général** envoie les triples qu'il contient au **triplestore** et aux **buffers** dont les règles peuvent utiliser ces triples. $< Garfield, \texttt{type}, chat >$ est donc envoyé à `cax-sco` et $< chat, \texttt{sc}, félin >$ aux deux règles. Le **buffer** de `cax-sco` est plein, puisqu'il contient 2 triples.

Étape 3 Les triples contenus dans le **buffer** de `cax-sco` sont envoyés vers un exécuteur de règle qui applique la règle sur ces triples. La règle infère le triple $< Garfield, \texttt{type}, félin >$.

Étape 4 Les triples inférés par l'exécuteur de règle de `cax-sco` sont envoyés dans le distributeur correspondant.

Étape 5 Le distributeur de `cax-sco` envoie le triple $< Garfield, \texttt{type}, félin >$ au **triplestore** et aux **buffers** qui peuvent l'utiliser, ici `cax-sco`.

Étape 6 Aucun des **buffers** n'étant plein, ils restent dans cet état jusqu'à ce que le délai de temporisation soit atteint. Ce délai dépassé, les deux **buffers** sont vidés et les triples qu'ils contiennent sont respectivement envoyés dans les **exécuteurs de règle** correspondants. L'application des règles ne permet pas d'inférer de nouveaux triples.

Étape 7 L'application des deux règles n'ayant rien inféré, aucun distributeur n'a reçu de triple. Un nouveau triple, $< félin, \texttt{sc}, animal >$, est envoyé au raisonneur et récupéré par le **distributeur général**.

Étape 8 Le **distributeur général** envoie le nouveau triple au **triplestore** ainsi qu'aux deux **buffers**.

Étape 9 Aucun des **buffers** n'étant plein, ils restent dans cet état jusqu'à ce que le délai de temporisation soit atteint. Ce délai dépassé, les deux **buffers** sont vidés et les triples qu'ils contiennent sont respectivement envoyés dans les **exécuteurs de règle** correspondants. Pour exécuter les règles, des triples sont récupérés depuis le **triplestore** afin de valider les prémisses de la règle. Pour `scm-sco`, le triple $< chat, \texttt{sc}, félin >$ est récupéré pour inférer $< chat, \texttt{sc}, animal >$. Pour `cax-sco`, c'est le triple $< Garfield, \texttt{type}, félin >$ du **triplestore** qui permet d'inférer $< Garfield, \texttt{type}, animal >$.

Étape 10 Les triples inférés par l'exécuteur de règle sont envoyés dans le distributeur correspondant.

Étape 11 Les **distributeurs** envoient les triples inférés au **triplestore** ainsi qu'aux **buffers** des règles pouvant les utiliser. Le **buffer** de la règle `scm-sco` est plein.

Étape 12 Les triples du **buffer** de la règle `scm-sco` sont envoyés à un exécuteur de règle. L'application de la règle ne permet pas d'inférer de nouveau triple.

Étape 13 Le **buffer** de la règle `cax-sco` n'est pas plein. Ses triples sont envoyés à son exécuteur de règle après que le délai de temporisation est dépassé.

Étape 14 L'application des règles ne permet pas d'inférer de nouveau triple.

Étape 15 Le système est dans un état *stable*, c'est-à-dire que le **distributeur général**, les **buffers** et les **distributeurs** sont vides. La détection de cet état stable est faite grâce à l'algorithme 4.3 (présenté en section 4.4).

Étapes ↓

#	Distributeur Général	Buffers		Exécuteurs de règles	Distributeurs	Triplestore
1	Garfield type chat chat sc félin	R1				
		R2				
2		R1	chat sc félin			Garfield type chat chat sc félin
		R2	Garfield type chat chat sc félin			
3		R1	chat sc félin			Garfield type chat chat sc félin
		R2		Garfield type chat chat sc félin → Garfield type félin		
4		R1	chat sc félin			Garfield type chat chat sc félin
		R2			Garfield type félin	
5		R1	chat sc félin			Garfield type chat chat sc félin Garfield type félin
		R2	Garfield type félin			
6		R1		chat sc félin → ×		Garfield type chat chat sc félin Garfield type félin
		R2		Garfield type félin → ×		
7	félin sc animal	R1				Garfield type chat chat sc félin Garfield type félin
		R2				
8		R1	félin sc animal			Garfield type chat chat sc félin Garfield type félin félin sc animal
		R2	félin sc animal			

TABLE 4.1 – Exemple de fonctionnement de l'architecture proposée détaillant pour chaque étape le contenu de chaque composant de l'architecture (1/2)

Étapes ↓

#	D.G.	Buffers	Exécuteurs de règles	Distributeurs	Triplestore
9		R1	félin sc animal *chat sc félin* → chat sc animal		Garfield type chat chat sc félin Garfield type félin félin sc animal
		R2	félin sc animal *Garfield type félin* → Garfield type animal		
10		R1		chat sc animal	Garfield type chat chat sc félin Garfield type félin félin sc animal
		R2		Garfield type animal	
11		R1	chat sc animal		Garfield type chat Garfield type animal chat sc félin chat sc animal Garfield type félin félin sc animal
		R2	chat sc animal Garfield type animal		
12		R1	chat sc animal		Garfield type chat Garfield type animal chat sc félin chat sc animal Garfield type félin félin sc animal
		R2	chat sc animal Garfield type animal → ×		
13		R1	chat sc animal		Garfield type chat Garfield type animal chat sc félin chat sc animal Garfield type félin félin sc animal
		R2			
14		R1	chat sc animal → ×		Garfield type chat Garfield type animal chat sc félin chat sc animal Garfield type félin félin sc animal
		R2			
15		R1			Garfield type chat Garfield type animal chat sc félin chat sc animal Garfield type félin félin sc animal
		R2			

TABLE 4.2 – Exemple de fonctionnement de l'architecture proposée détaillant pour chaque étape le contenu de chaque composant de l'architecture (2/2)

4.5 Indépendance au fragment

Nous avons choisi de rendre notre architecture générique face au fragment de règles qu'il est possible d'utiliser. Pour atteindre cet objectif, deux principes ont été mis en place.

Dans la section suivante, nous présentons le graphe de dépendance des règles qui permet de déterminer le routage des triples au travers des règles. Il est également utile au **distributeur général** pour déterminer comment distribuer les triples en entrée du raisonneur. Cela permet un routage adaptatif et automatique des triples au travers des différents modules du raisonneur, ne nécessitant pas de module central dédié au routage des triples.

Ensuite, la spécificité des différents modules de l'architecture a été minimisée. En effet, seuls les **exécuteurs de règle** sont spécifiques à la règle qu'ils appliquent. Le reste du système est générique, de manière à minimiser l'impact du changement de fragment. Nous détaillons dans la section 4.5.2 comment les règles sont définies et exécutées.

4.5.1 Graphe de dépendance des règles

Dans une architecture distribuée, les accès concurrents à des objets partagés sont coûteux. Pour maximiser l'efficacité de notre architecture, il est essentiel de minimiser ces objets et de rendre les différents modules indépendants. Dans cette optique, chaque règle doit pouvoir redistribuer les triples qu'elle a inférés aux autres règles, sans utiliser d'objet central dans l'architecture. C'est le rôle du distributeur qui est associé à chacune d'elles. Ce dernier doit savoir à quelles règles envoyer les triples qu'il reçoit.

Le graphe de dépendance des règles, introduit dans [47] par Soma, permet de répondre à ce problème. Il s'agit d'un graphe orienté dont les sommets représentent les règles du fragment. Il existe un arc entre deux sommets A et B si :

- Au moins un prédicat d'un triple des prémisses de B est une variable, e.g. $(x\ c\ y)$;

  ou

- Au moins un prédicat d'un triple des conclusions de A est également le prédicat d'au moins un triple des prémisses de B. Par exemple si le triple (c_1 `subClassOf` c_3) est présent dans les conclusions de A et si le triple (c_1 `subClassOf` c_2) est présent dans les prémisses de B.

Autrement dit, il existe un arc de A à B si la règle B peut utiliser en entrée les triples inférés par la règle A.

Pour les deux règles `cax-sco` et `scm-sco`, le graphe de dépendance est présenté dans la figure 4.6. Pour des raisons de lisibilités, `subClassOf` est noté `sc`.

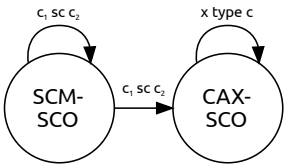

FIGURE 4.6 – Graphe de dépendance des règles pour le fragment composé de `cax-sco` et `scm-sco`

Ce graphe est calculé à l'initialisation du système pour tout ensemble de règles d'inférence, permettant au système d'associer les règles entre elles quel que soit le fragment choisi. À titre d'exemple, le graphe de dépendance du fragment ρdf est illustré dans la figure 4.7. Les règles dites *universelles* sont celles dont au moins un prédicat des prémisses est une variable. Ces règles peuvent de ce fait utiliser n'importe quel triple pour l'inférence. Tout sommet du graphe a donc un arc en direction de chaque règle universelle. Dans le graphe de la figure 4.7, ces règles sont représentées grisées pour améliorer la lisibilité. De même, `subPropertyOf` est noté `sp`. Le graphe de dépendance des règles de RDFS se trouve en Annexe A.1.

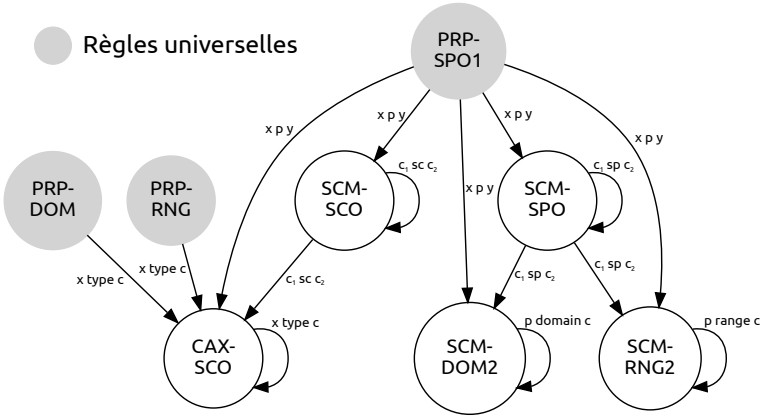

FIGURE 4.7 – Graphe de dépendance des règles pour ρdf

Pour générer ce graphe, nous utilisons l'algorithme 4.4.

Grâce à ce graphe, il est possible non seulement de connecter les règles d'inférence les unes avec les autres, mais également d'avoir un aperçu de la complexité du fragment composé de ces règles. Plus le graphe sera connecté et plus il y aura de boucles, plus l'inférence sera susceptible de nécessiter un nombre important de passes pour terminer la matérialisation.

ALGORITHME 4.4 – Construction du graphe de dépendance des règles

Données : $Regles$
Résultat : GDR le graphe de dépendance des règles

```
pour (regle_1, regle_2) ∈ Regles × Regles faire
    si Conclusions(regle_1) ∈¹ Premices(regle_2) alors
        GDR.ajouterArc(regle_1, regle_2)
    fin
    si Conclusions(regle_2) ∈¹ Premices(regle_1) alors
        GDR.ajouterArc(regle_2, regle_1)
    fin
fin
```

4.5.2 Définition et exécution des règles

Deux solutions pour l'exécution des règles d'inférence s'opposent. La première consiste à utiliser un raisonneur extérieur. Cela permet de réutiliser des implémentations existantes, avec les éventuelles optimisations intégrées. Cependant, l'intégration du raisonneur dans notre solution, notamment pour l'accès à notre **triplestore**, est une tâche complexe, pouvant neutraliser l'avantage apporté par notre solution. Pour cette raison, nous avons opté pour la seconde solution, consistant à définir un **exécuteur de règle** par règle d'inférence utilisée. Cela permet, tout en gardant une solution générique, d'utiliser au mieux les caractéristiques de notre solution, notamment par une utilisation directe des options de recherche de triples du **triplestore**.

Nous avons vu plusieurs exemples de règles prédéfinies dans le cadre de recommandations du W3C. Chaque règle est composée de prémisses et de conclusions, elles-mêmes comprenant un ou plusieurs formats de triples (c.f. section 2.3.1).

Afin d'appliquer une règle, l'**exécuteur de règle** doit donc trouver les triples qui correspondent aux prémisses de la règle. Pour cela, la règle utilise les triples qu'elle reçoit de son **buffer** et recherche dans le **triplestore** les éventuels triples manquants pour compléter les prémisses de la règle. L'algorithme 4.5 illustre l'application de la règle `cax-sco`.

L'algorithme d'exécution d'une règle prend en entrée les triples envoyés par le **buffer** et retourne les triples inférés par la règle. Notre architecture peut intégrer toute règle conforme à la signature de l'algorithme 4.5.

ALGORITHME 4.5 – Application de la règle `cax-sco`

Données : tripleStore, bufferTriples
Résultat : outputTriples

```
1  outputTriples ← ∅
2  pour (s2, type, o2) ∈ bufferTriples faire
3  |  pour (o2, subClassOf, o1) ∈ tripleStore faire
4  |  |  outputTriples ← (s2, type, o1)
5  |  fin
6  fin
7  pour (s1, subClassOf, o1) ∈ bufferTriples faire
8  |  pour (s2, type, s1) ∈ tripleStore faire
9  |  |  outputTriples ← (s2, type, o1)
10 |  fin
11 fin
```

4.6 Paramètres de l'architecture

L'architecture que nous avons présentée dispose de trois paramètres permettant d'ajuster son comportement au besoin de l'utilisateur, en termes de réactivité du système et de consommation des ressources machines. Dans cette section, nous verrons ces paramètres et leur impact sur le comportement du système.

4.6.1 Taille des buffers

À chaque règle d'inférence est associé un *buffer*. Il reçoit les triples susceptibles d'être utilisés par la règle correspondante. Lorsque ce **buffer** est plein, c'est-à-dire lorsque le nombre de triples qu'il contient dépasse une certaine limite, ces triples sont envoyés à un nouvel **exécuteur de règle**. La taille du **buffer** correspond à cette limite.

Le choix de la taille des **buffers** a un impact majeur sur le comportement du système. Plus cette taille est grande, moins fréquentes seront les exécutions de règles et inversement. À l'extrême, si la taille du **buffer** est fixée à 1, une exécution de règle est lancée pour chaque nouveau triple, rendant le système extrêmement réactif, mais également extrêmement coûteux en termes de calcul. À l'inverse, pour une taille de **buffer** importante, la fréquence d'exécution des règles sera plus réduite, mais le processus sera moins coûteux. Comme nous le présentons dans la section 4.2.3, il est en effet moins coûteux de lancer une fois une règle pour n triples plutôt que n fois pour chaque triple.

La taille du **buffer** introduit donc un compromis entre la réactivité du système et la charge de calcul. Ce paramètre est aussi directement lié à la fréquence d'arrivée de nouvelles connaissances. Plus cette fréquence est élevée, plus la limite du **buffer**

sera atteinte rapidement, menant à l'exécution d'une règle. En cas de fréquence faible, l'inférence ne sera mise à jour que rarement, rendant le système peu réactif.

4.6.2 Timeout

Lorsqu'un **buffer** ne reçoit pas de nouveau triple pendant un certain temps sans être vide, les triples qu'il contient sont envoyés à un nouvel exécuteur de règle, comme lorsque le **buffer** est plein. Ce délai de temporisation, ou *timeout*, empêche les triples de stagner dans les **buffers** en attendant qu'ils soient pleins, ce qui ralentirait la réactivité du système.

Le timeout permet également de répartir équitablement l'exécution des règles dans le temps, dans le cas où le délai arrive à expiration avant que le **buffer** ne soit plein. Lorsque le système doit être réactif, le timeout peut être réglé au minimum, de manière à traiter au plus tôt les nouvelles connaissances. Si au contraire, le coût du calcul doit être réduit, le timeout peut être allongé pour réduire ce coût.

4.6.3 Fragment

L'ensemble des règles d'inférence utilisées pour le raisonnement est appelé fragment. Comme nous l'avons présenté, notre architecture prend en charge tout fragment dont chaque règle peut être implémentée par un algorithme ayant la même signature que l'interface de la figure 5.3.

Le choix de ce paramètre introduit le compromis principal que doit faire l'utilisateur. Le cas d'usage dans lequel le raisonneur est utilisé le guide souvent vers un fragment prédéfini. Nous pensons cependant qu'il est essentiel de mesurer l'impact et l'intérêt de chaque règle qui sera utilisée. Le type de connaissance que permet de déduire une règle n'est pas le seul argument qui doit être étudié. Sa complexité influence directement la réactivité du système et le temps nécessaire à l'inférence.

En plus des règles prédéfinies, certains cas d'usage peuvent nécessiter des règles définies par l'utilisateur. L'aspect générique de notre architecture permet d'ajouter de telles règles dans le système en implémentant simplement une fonction.

L'intégration de cette règle se fait grâce aux opérations suivantes :

- Un **buffer** et un **distributeur** sont créés pour cette nouvelle règle ;
- le graphe de dépendance est mis à jour avec la nouvelle règle ;
- le **buffer** est enregistré auprès des autres **buffers** grâce au graphe de dépendance ;
- le **distributeur** enregistre les **buffers** auxquels il doit envoyer les triples inférés, toujours grâce au graphe de dépendance.

Une fois ces opérations effectuées, le **buffer** de la nouvelle règle reçoit les triples qui la concernent et son **distributeur** redistribue les triples qu'elle infère aux autres **buffers** qui peuvent utiliser ces triples. La règle est alors intégrée au système. Cette opération peut être effectuée à chaud, pendant l'exécution du raisonneur.

4.7 Modes d'inférence

4.7.1 Mise en situation

Le raisonnement peut être une procédure qui s'applique à un ensemble fixe de données, permettant de lever des connaissances implicites à partir des données accessibles. Il consiste alors à inférer toutes les connaissances implicites, puis à interroger la base de connaissances contenant la matérialisation complète dans un second temps. L'objectif est de matérialiser ces connaissances le plus rapidement possible.

Dans un système recevant de nouvelles données en continu, il n'est pas possible de séparer ces opérations : les requêtes arrivent pendant l'inférence, qui est un processus continu. L'inférence et l'interrogation de la base de connaissances sont donc effectuées en parallèle, le raisonnement étant appliqué au fur et à mesure de l'arrivée des données, sans empêcher le système de répondre en cas de requête.

Le temps imparti avant l'utilisation des connaissances implicites peut être borné, e.g. dans le cadre de requêtes exécutées à intervalle régulier, voire inconnu lorsque les requêtes sont par exemple déclenchées par des événements extérieurs au système. Il n'est donc pas possible de garantir que la matérialisation soit complète lors d'une réponse à une requête, en particulier puisque de nouvelles données sont susceptibles d'arriver dans le système au moment de la requête. Cette situation nécessite une adaptation du comportement des systèmes de raisonnement.

Dans le cas d'une inférence classique, l'objectif est de minimiser le temps global d'inférence tout en garantissant la complétude du raisonnement. L'ordre dans lequel les triples sont inférés, c'est-à-dire quelles connaissances implicites sont levées en premier, n'a généralement pas d'importance. Si l'on applique cet objectif à notre architecture, cela implique que tous les **buffers** ont la même taille et sont soumis au même timeout. Le paramétrage de l'architecture n'en est que plus simple. Il reste cependant basique et ne permet pas d'adapter le comportement du raisonneur à d'éventuels besoins de l'utilisateur.

Dans le cadre d'un raisonnement continu, une requête a toutes les chances de survenir pendant le processus de raisonnement. Afin d'apporter la meilleure réponse possible avec un raisonnement incomplet, il est assez intuitif de vouloir maximiser la matérialisation des connaissances qui permettent de répondre à cette requête. L'ordonnancement des règles a donc dans ce cas pour vocation de prioriser l'inférence des triples utiles, si nécessaire au détriment du temps nécessaire à la totalité de l'inférence.

Deux situations se font face. Les requêtes à venir peuvent être ou non connues par avance. Dans le cas de requêtes connues, comme par exemple une requête continue, il est simple de définir quelles sont les connaissances, et donc les règles, à privilégier. La seconde situation est plus problématique. Dans le cas de requêtes ponctuelles dont les connaissances requêtées sont a priori inconnues, une solution est de favoriser les règles inférant le plus de triples. De cette façon, la quantité de connaissances accessible au moment de l'accès à la base de connaissances sera maximal, augmentant les chances de

disposer des triples nécessaires pour répondre correctement à cette requête.

Il est important de préciser que, quel que soit le mode de raisonnement, l'inférence est toujours complète. La matérialisation reste identique. Seuls l'ordre dans lequel les triples sont inférés et le temps nécessaire pour terminer l'inférence sont différents. L'objectif est de prioriser certaines connaissances pour qu'elles soient inférées au plus tôt, afin de répondre au mieux aux requêtes arrivant *pendant* le processus de raisonnement.

Dans les sections suivantes, nous verrons comment adapter le comportement du raisonneur en fonction de ces différentes situations. La section 4.7.2 est focalisée sur la situation où les connaissances à prioriser sont connues par avance. La section 4.7.3 présente l'adaptation du raisonneur sans a priori sur les données à prioriser.

4.7.2 Raisonnement dirigé : la qualité d'abord

Dans cette section nous présentons comment choisir les paramètres de notre architecture afin de privilégier l'inférence de certaines connaissances, e.g. pour optimiser la réponse à une requête connue.

Priorisation de connaissances lors de l'inférence

Nous nous plaçons dans une situation où les connaissances qui seront requêtées sont connues par avance. Ce cas peut s'appliquer à un système recevant une requête continue ou à intervalle régulier, ou il peut s'agir d'un choix de l'utilisateur qui, bien qu'il soit intéressé par l'ensemble de l'inférence, souhaite prioriser certaines connaissances. Pour augmenter les chances de fournir un maximum de ces connaissances attendues au moment de leur utilisation, l'objectif est donc de faire en sorte qu'elles soient inférées en premier.

Intuitivement, pour favoriser l'inférence d'un certain type de connaissances, il est nécessaire de lancer en premier les règles générant les triples correspondants. Si, par exemple, les connaissances visées sont la hiérarchie des classes, les règles inférant des triples de la forme (c_1 `subClassOf` c_2) seront exécutées en premier. Ces règles sont triviales à trouver, il suffit de parcourir les conclusions d'une règle pour déterminer si un triple correspond. Mais lancer uniquement ces règles n'est évidemment pas suffisant. Comme nous l'avons vu, les règles d'inférence appartenant à un fragment ne forment que rarement un graphe non cyclique. L'exécution d'autres règles est donc indispensable à l'inférence de toutes les connaissances recherchées. De plus, bien que nous favorisions la découverte d'un type de triples, l'inférence complète est ici toujours d'actualité. Il ne s'agit pas de réduire le fragment utilisé pour le raisonnement, mais seulement d'influencer l'ordre d'inférence des connaissances.

Caractérisation du niveau des règles d'inférence

Afin de déterminer la priorité des règles, nous utilisons le graphe de dépendance des règles. Chaque règle se voit attribuer un niveau d'importance. Les règles de niveau 1, les plus importantes, sont celles inférant les connaissances recherchées. Ensuite, les règles ayant pour filles dans le graphe de dépendance des règles de niveau n sont de niveau $n+1$. Les règles sans niveau à la fin de ce processus ont pour niveau le niveau le plus élevé attribué incrémenté de 1. L'algorithme 4.6 permet de hiérarchiser les règles d'un fragment à partir des triples recherchés et du graphe de dépendance des règles. Les lignes 1 à 8 permettent d'initialiser à 1 le niveau des règles générant les triples à prioriser. Les lignes 9 à 23 vont ensuite attribuer aux règles parentes de celles-ci le niveau $n+1$, et ainsi de suite jusqu'à ce que le graphe de dépendance des règles ait été entièrement parcouru. Pour terminer, les lignes 24 à 28 attribuent aux règles sans niveau à ce stade un niveau supérieur de 1 au plus haut niveau déjà attribué.

Prenons l'exemple suivant. Le fragment utilisé est composé des règles `cax-sco` et `scm-sco`, et les connaissances qui intéressent l'utilisateur sont les relations de sous-classes. La règle `scm-sco` est la seule inférant un triple (x `subClassOf` y), son niveau est donc de 1. Elle possède deux parents : elle-même et `cax-sco`. Comme `scm-sco` a déjà le niveau 1, seule `cax-sco` se voit attribuer le niveau 2.

Adaptation des paramètres de l'architecture

Nous avons défini un algorithme permettant de donner à chaque règle un niveau d'importance par rapport aux données qui seront utilisées en priorité. Nous définissons maintenant les adaptations de notre architecture afin de prendre en compte ces niveaux.

Deux paramètres permettent de modifier l'exécution d'une règle : la taille de son **buffer** et son timeout. Pour qu'une règle soit priorisée, il faut que ces deux paramètres soient plus petits que ceux des autres règles non priorisées. De cette façon, la règle sera exécutée plus tôt et plus fréquemment. Afin de déterminer ces valeurs pour une règle de niveau n, nous utilisons la fonction f définie dans l'équation 4.2. La variable p représente la valeur du paramètre pour une règle de niveau 1 (on a donc $f(1) = p$). Le coefficient α permet de déterminer l'importance de l'écart entre deux niveaux. Le logarithme permet de minimiser cet écart entre les niveaux les plus éloignés des règles priorisées. La différence entre deux règles de niveaux 1 et 2 sera plus grande que la différence entre deux règles de niveaux 8 et 9.

$$f(n) = \lfloor (\alpha p)\, log(n) + p \rfloor\ ,\ \alpha \geq 1,\ p \in \mathbb{N},\ n \in \mathbb{N}* \tag{4.2}$$

L'utilisateur choisit la valeur de taille de **buffer** b et de timeout t pour les règles de niveau 1. Ces valeurs sont ensuite calculées pour les autres règles grâce à $f(b)$ et $f(t)$. L'utilisateur peut également faire varier α pour paramétrer l'écart entre les valeurs pour les différents niveaux.

La figure 4.8 illustre la variation d'une valeur (taille de **buffer** ou timeout) entre les niveaux 1 à 10, pour une valeur initiale de 10 et pour α valant 1, 2, 5 et 10.

ALGORITHME 4.6 – Détermination du niveau des règles d'inférence

Données : D le graphe de dépendance des règles, T l'ensemble des triples à prioriser

```
1  tmp ← ∅
2  // Règles de niveau 1
3  pour règle ∈ D faire
4  |  si règle.conclusions ∩ T ≠ ∅ alors
5  |  |  règle.niveau ← 1
6  |  |  tmp ← tmp ∪ règle
7  |  fin
8  fin
9  // Hiérarchisation des autres règles
10 niveau ← 2
11 tmp2 ← ∅
12 tant que tmp ≠ ∅ faire
13 |  pour règle ∈ tmp faire
14 |  |  // Un niveau n'est attribué que si la règle n'a pas déjà un niveau
15 |  |  si !règle.niveau alors
16 |  |  |  règle.niveau ← niveau
17 |  |  |  tmp2 ← tmp2 ∪ ègle.parents
18 |  |  fin
19 |  fin
20 |  tmp ← tmp2
21 |  tmp2 ← ∅
22 |  niveau + +
23 fin
24 pour règle ∈ D faire
25 |  si !règle.niveau alors
26 |  |  règle.niveau ← niveau
27 |  fin
28 fin
```

Exemple de priorisation des triples de prédicat `domain` dans le fragment ρdf

Nous allons voir un exemple de choix des paramètres dans le but de prioriser la génération des triples du type $< x, \texttt{domain}, y >$. Le fragment utilisé est ρdf (c.f. figure 4.7), la taille de **buffer** de référence est de 500 et le timeout de 10. Nous appliquons maintenant l'algorithme 4.6.

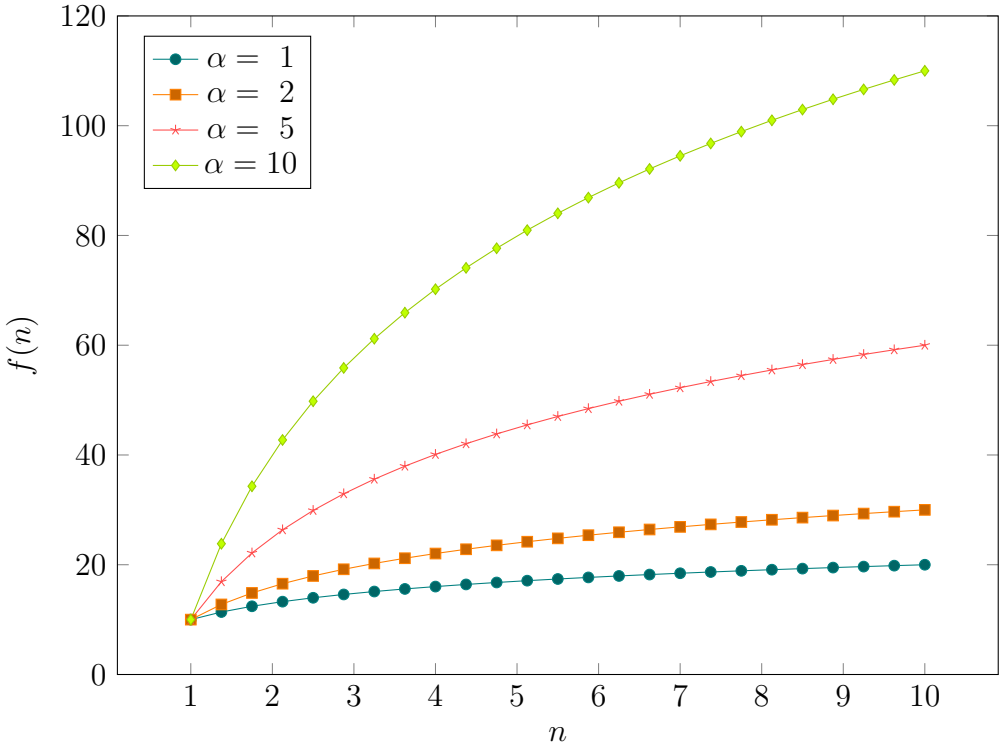

FIGURE 4.8 – Variation de la valeur $f(n)$ d'un paramètre en fonction de son niveau n, avec une valeur initiale $x = 10$, pour différentes valeurs de α

Les règles ayant, dans leurs prémisses, un triple du type $< x, \texttt{domain}, y >$ ont pour niveau 1. Il s'agit pour ρdf des règles `scm-dom2` et `prp-spo1`. Cette dernière est susceptible d'inférer n'importe quel type de triple, dont notamment le type que nous recherchons.

Nous attribuons maintenant les niveaux $n+1$ aux parents des règles de niveau n. Le seul parent des règles `scm-dom2` et `prp-spo1` n'ayant pas encore de niveau attribué est `scm-spo`. Il lui est donc attribué le niveau 2. La règle `scm-spo` est une règle universelle et a donc comme parent toutes les règles du fragment. Celles ne disposant pas d'un niveau déjà affecté se voient donc attribuer le niveau 3. Le tableau 4.3 résume les niveaux de chaque règle.

Niveau	**Règles**
1	{`scm-dom2`, `prp-spo1`}
2	{`scm-spo`}
3	{`cax-sco`, `prp-dom`, `prp-rng`, `scm-sco`, `scm-rng2`}

TABLE 4.3 – Niveau des règles du fragment ρdf pour la priorisation des triples $< x, \texttt{domain}, y >$

Maintenant que nous avons déterminé le niveau de chaque règle, nous allons calculer la valeur des paramètres pour chacune de ces règles. On utilise pour cela l'équation 4.2. Afin d'illustrer l'impact du coefficient α, les tables 4.4 et 4.5 représentent les valeurs respectivement de la taille du **buffer** et du timeout en fonction du niveau et différentes valeurs de α.

		α				
		1	3	5	7	10
Niveau	1	500	500	500	500	500
	2	650	950	1250	1550	2005
	3	735	1215	1690	2165	2885

TABLE 4.4 – Taille du **buffer** calculée en fonction du niveau de la règle et du coefficient α

		α				
		1	3	5	7	10
Niveau	1	10	10	10	10	10
	2	13	19	25	31	40
	3	14	24	33	43	57

TABLE 4.5 – Timeout calculé en fonction du niveau de la règle et du coefficient α

En choisissant la valeur de α à 5, on associe dont les valeurs suivantes aux règles : `scm-dom2` et `prp-spo1` ont une taille de buffer de 500 et un timeout de 10. La règle `scm-spo` a une taille de buffer de 1250 et un timeout de 25. Enfin, `cax-sco`, `prp-dom`, `prp-rng`, `scm-sco` et `scm-rng2` se voient attribué une taille de buffer de 1690 et un timeout de 33.

Grâce à l'algorithme 4.6 et à la fonction $f(n)$, on peut donc associer à chaque règle du fragment utilisé une taille de buffer et un timeout en fonction des connaissances que l'on souhaite prioriser. Le coefficient α permet d'affiner l'importance de l'écart entre les différents niveaux.

4.7.3 Raisonnement à l'aveugle

Dans la section précédente, nous nous étions placés dans un cas d'usage où l'utilisation faite des connaissances matérialisées était connue par avance. Dans de nombreuses applications, cette information n'est pas disponible. Les requêtes sont faites depuis l'extérieur du système, par des utilisateurs ou d'autres systèmes ne suivant pas toujours de

schéma prédéfini. Il faut donc définir de nouvelles méthodes afin d'adapter le système à ce cas de figure.

Lorsque l'on ne dispose pas d'information sur l'utilisation à venir des connaissances, il est assez intuitif de vouloir maximiser le nombre de triples inférés par unité de temps. En effet, plus le nombre de triples matérialisés est important, plus les chances de répondre correctement à une requête sont élevées. Afin de maximiser la quantité de triples inférés par unité de temps, il est nécessaire de favoriser l'exécution des règles générant le plus de triples. Il existe plusieurs méthodes pour déterminer quelles sont ces règles.

Dans le cas d'un système déjà en fonctionnement, il est possible d'obtenir des statistiques sur les quantités de données inférées par chaque règle, et donc de trier les règles grâce à ces informations. Mais ces informations ne sont pas toujours disponibles, en particulier à l'initialisation du système.

Dans cette section, nous verrons comment adapter le raisonnement en fonction des données statistiques disponibles. Enfin, nous proposerons une méthode pour initialiser le raisonneur en l'absence de statistiques.

Adaptation statistique

Les statistiques nécessaires ici sont simples : il s'agit du nombre total de triples inférés par chaque règle du fragment considéré. Ces données peuvent provenir d'une exécution antérieure.

Pour un ensemble de règles d'inférences $\{r_1, \ldots, r_k\}$, on note n_i le nombre de triples inférés par la règle r_i, $i \in [1, k]$. Nous définissons alors la fonction $niveau(x)$ et m la valeur maximale de $\{n_1, \ldots, n_k\}$ (c.f équation 4.3). Le niveau d'une règle r_i est $niveau(n_i)$. De cette façon, la règle inférant le plus grand nombre de triples est de niveau 1 et les autres ont un niveau de plus en plus grand, en fonction de l'éloignement de leur valeur n_i par rapport à la règle de niveau 1.

$$niveau(x) = \left\lfloor \frac{m}{x} \right\rfloor, \ \forall x \in \mathbb{N} \tag{4.3}$$

Une fois le niveau déterminé, la méthode présentée en section 4.7.2 est ensuite utilisée pour déterminer les paramètres de chaque règle.

Afin de rendre le système évolutif, les statistiques, et par conséquent les niveaux et valeurs des paramètres, peuvent être mis à jour tout au long du raisonnement. Par exemple, à chaque exécution d'une règle r_i, la valeur de n_i peut être mise à jour. En cas de changement, la valeur de m est également mise à jour, puis l'ensemble des niveaux des règles est, à son tour, mis à jour. Les paramètres de l'architecture sont alors ajustés en fonction de ces nouvelles valeurs. Le système adapte donc son fonctionnement non seulement en fonction des informations historiques, mais également au fur et à mesure de son fonctionnement.

Initialisation intuitive

Lors de l'initialisation du système, en l'absence de données statistiques, il n'est pas possible d'appliquer la méthode présentée précédemment. Lorsque les statistiques ne sont pas disponibles, nous proposons une méthode d'initialisation basée sur l'intuition suivante : une règle d'inférence universelle, c'est-à-dire dont l'une des prémisses est un triple uniquement composé de variables (e.g. $< x, y, z >$), a plus de chances d'être exécutée qu'une autre règle et donc de lever des connaissances. Partant de cette observation, les règles dites universelles se voient attribuer le niveau 1. Les autres règles sont initialisées au niveau 2. Le tableau 4.6 montre par exemple cette initialisation pour ρdf.

Niveau	**Règles**
1	{`prp-dom`, `prp-rng`, `prp-spo1`}
2	{`scm-spo`, `scm-dom2`, `cax-sco`, `scm-sco`, `scm-rng2`}

TABLE 4.6 – Niveau des règles de ρdf pour la priorisation des triples $< x, y, z >$

Le niveau des règles sera ensuite mis à jour progressivement au fur et à mesure de l'avancée de l'inférence, grâce à la méthode présentée dans la section précédente.

4.8 Bilan de la solution proposée

Dans ce chapitre, nous avons présenté une architecture permettant le raisonnement incrémental sur des flux de triples.

L'architecture que nous avons définie est modulaire et permet une exécution parallèle et le passage à l'échelle. Chaque **exécuteur de règle** s'exécute de manière indépendante et concurrente, évitant les goulots d'étranglement. Un ensemble de **buffers**, composés d'une file de triples infinie, garantit le traitement de toutes les données envoyées au raisonneur. Les **buffers** permettent de grouper les triples afin qu'ils soient traités par lots, réduisant la charge de calcul induite par l'instanciation de processus légers. Les **distributeurs** utilisent le graphe de dépendance entre les règles afin de redistribuer les triples inférés.

Le **triplestore** est partagé et accessible par l'ensemble des modules composant notre architecture. Cela permet de minimiser le nombre d'exécutions de règles nécessaire au calcul de la matérialisation. Les **exécuteurs de règle** accèdent au **triplestore** pour y chercher les triples permettant de compléter les prémisses de la règle qu'ils exécutent. Le **triplestore** est également utilisé par les **distributeurs** pour supprimer les doublons générés. De cette manière, ils ne sont pas disséminés dans le système.

La structure interne de l'architecture manipule des flux de triples, qui transitent à travers les différents modules du système. Cela permet au raisonneur de prendre en entrée un ou plusieurs flux de triples nativement. L'exécution parallèle de l'architecture, et donc du **distributeur général**, permet au raisonneur de recevoir de nouveaux triples en continu pendant son exécution.

Le fonctionnement des différents modules du système est indépendant du fragment considéré, en dehors de l'application de la règle elle-même par les **exécuteurs de règle**. Cette architecture peut raisonner ainsi sur un large panel de fragments. L'ajout d'une nouvelle règle peut être fait à chaud, pendant l'exécution du raisonneur, à condition de disposer de l'implémentation de l'application de la règle. Un **buffer** et un **distributeur** sont créés pour cette règle et sont connectés aux **buffers** et aux **distributeurs** des autres règles. La règle est alors intégrée au système, recevant les triples qu'elle peut utiliser et redistribuant ceux qu'elle infère aux autres règles concernées.

En plus du fonctionnement *classique*, qui consiste à matérialiser l'ensemble des connaissances implicites en un minimum de temps, nous avons introduit deux autres modes d'inférence visant à prioriser l'inférence de certaines connaissances face à deux scénarios. Le premier mode permet d'adapter le processus de raisonnement pour une requête connue par avance, en priorisant l'inférence des connaissances utiles pour répondre à cette requête. Le second mode s'appuie sur l'intuition suivante : si les requêtes à venir ne sont pas connues, il est pertinent de maximiser la quantité de triples inférés par unité de temps. L'idée est simple : plus il y a de triples à disposition, plus il y a de chances que les triples utiles à la requête soient présents. Après une initialisation priorisant les règles

dites *universelles*, le système s'adapte au fur et à mesure de l'inférence en utilisant des statistiques sur la quantité de données inférées par chaque règle.

Afin de mettre en place ces fonctionnements alternatifs, nous avons défini un algorithme pour assigner un niveau à chaque règle, qui sera ensuite utilisé pour adapter la taille du **buffer** et le timeout pour chaque règle.

IV

Validation expérimentale

5 *Slider* : Implémentation du raisonneur incrémental

Sommaire

Dans la partie précédente, nous avons présenté un modèle d'architecture pour le raisonnement incrémental. Une autre contribution majeure de cette thèse est l'implémentation de cette architecture au travers de **Slider**. Raisonneur incrémental par chaînage avant, il prend nativement en charge les fragments ρdf et RDFS et peut être étendu à des fragments plus complexes. Slider permet de raisonner à partir de données statiques et de flux de données.

Entièrement développé en Java, Slider est distribué librement et gratuitement sous licence Apache 2.0 [1]. Il est donc accessible par chacun, pour être utilisé, partagé et amélioré par le plus grand nombre. Afin de favoriser son appropriation, nous maintenons nos efforts pour fournir une documentation approfondie et un maximum d'instructions facilitant son utilisation. Toutes ces informations sont disponibles sur la page Web du projet [2]. Le code source est accessible librement sur Github [3].

Une démonstration du comportement interne du raisonneur, soumise et acceptée [36] à SIGMOD2015 [4], est également disponible en ligne [5].

Ce chapitre est dédié aux détails de l'implémentation de l'architecture que nous avons présentée dans la partie précédente. Nous présentons les structures de données que nous avons utilisées, puis la manière dont la parallélisation est assurée.

5.1 Structures de données utilisées

L'implémentation de notre architecture se découpe suivant les différents modules que nous avons détaillés dans la section 4.4. Le diagramme de classe de la figure 5.1 illustre ces modules. Il regroupe les éléments principaux de cette implémentation et leurs dépendances. On retrouve le distributeur général, chargé d'enregistrer les nouveaux triples dans le dictionnaire et de les répartir entre les différents buffers au travers d'un distributeur. Un nouvel objet est introduit : le module de règle. Il permet de lier un buffer et un distributeur associés à une règle donnée et d'instancier les exécuteurs de cette règle. On retrouve également le triplestore. Les méthodes et variables principales de chaque composant sont également présentes.

Nous présentons dans cette section les structures de données que nous avons choisies pour implémenter ces modules.

1. `http://www.apache.org/licenses/LICENSE-2.0`
2. `http://juleschevalier.github.io/slider/`
3. `https://github.com/juleschevalier/slider`
4. `http://www.sigmod2015.org/`
5. `http://demo-satin.telecom-st-etienne.fr/slider/`

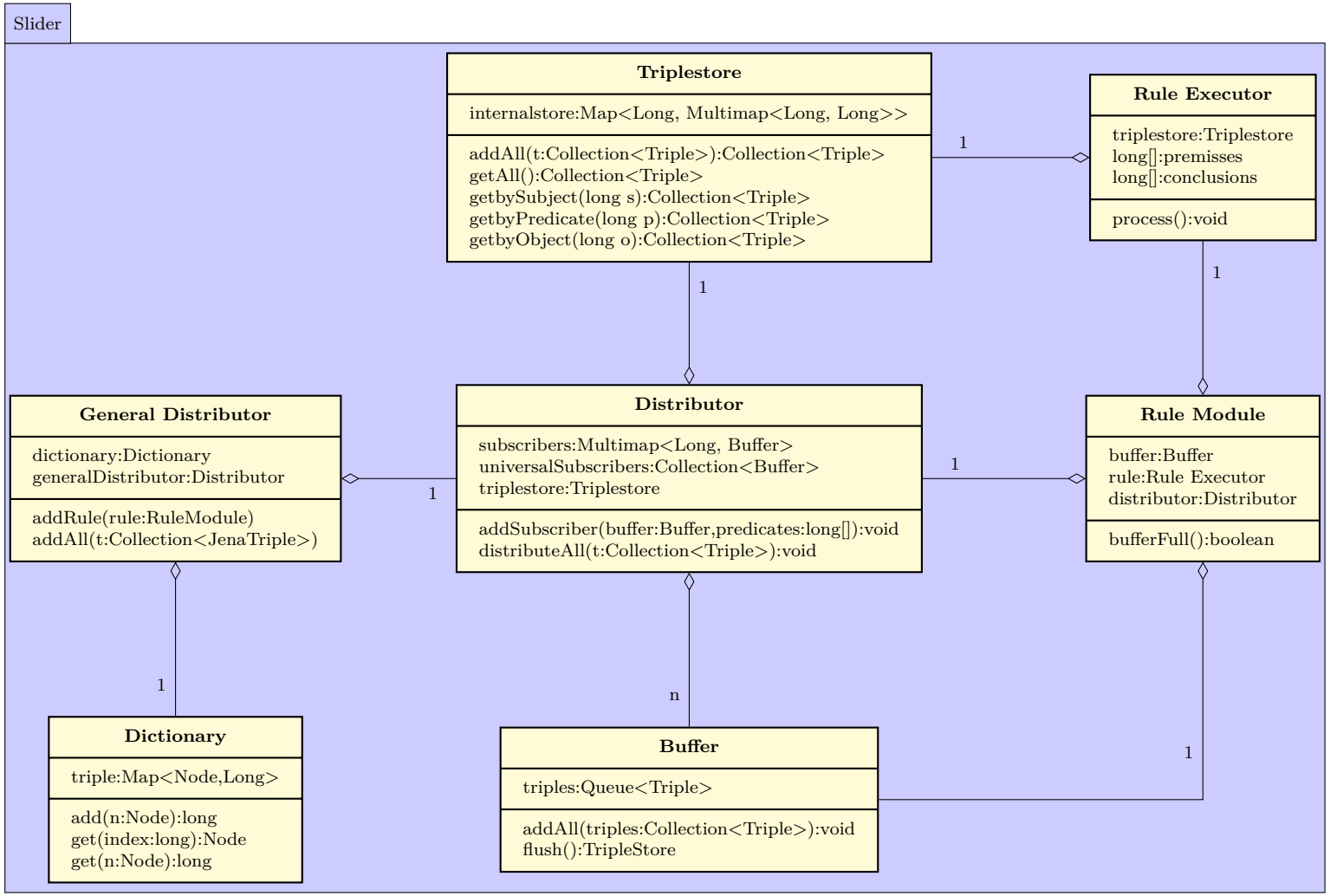

FIGURE 5.1 – Diagramme de classes de Slider

5.1.1 Modules de règle

Comme nous l'avons vu dans le Chaptire 4, chaque règle se voit assigner plusieurs éléments : un **buffer**, un **distributeur** et des **exécuteurs de règles**. Un **module de règle** permet de coordonner ces éléments pour une règle donnée. Plus exactement, lorsqu'un **buffer** notifie le **module de règle**, celui-ci est en charge de récupérer les triples à utiliser et d'instancier l'**exécuteur de règle** qui applique la règle d'inférence sur les triples en question. Il donne également à l'**exécuteur de règle** un accès au **distributeur** afin que celui-ci puisse transférer les nouveaux triples aux différents **buffers**.

La figure 5.2 représente un module de règle ainsi que les objets qui le composent.

5.1.2 Représentation et stockage des triples

Le Triple est l'objet du raisonneur qui est le plus instancié, utilisé, comparé, et échangé. Il est composé de trois éléments : le sujet, le prédicat et l'objet. Plutôt que de stocker trois chaînes de caractères, ou des objets plus complexes représentant chaque élément, nous représentons un triple sous la forme de trois long (entiers 64 bits).

L'utilisation d'un long permet de représenter jusqu'à 2×2^{64} concepts différents. Nous avons créé l'objet Triple pour qu'il soit immuable, afin de s'affranchir des contraintes d'accès concurrents pour cet objet, puisqu'une fois créé un Triple ne peut pas être modifié.

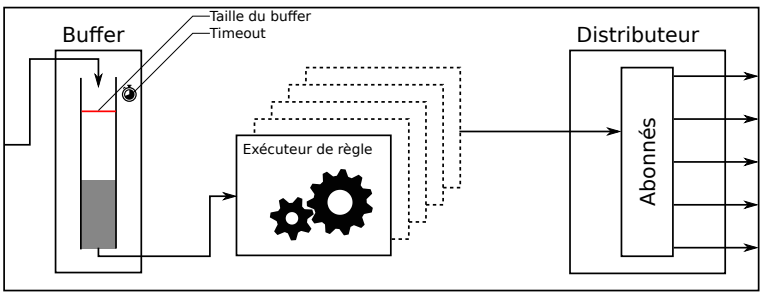

FIGURE 5.2 – Détails d'un module de règle dans Slider

Nous disposons d'un dictionnaire associant un long à chaque concept. Pour stocker ces concepts, nous utilisons l'objet Node fourni dans l'implémentation de Jena [6]. Il permet notamment d'être compatible avec les standards du Web Sémantique. Toutes les informations complémentaires apportées par cette structure (URI, type de concept, valeur, etc) sont conservées dans le dictionnaire, pour être retrouvées lors de l'accès aux données. Une table de hachage (HashMap) est utilisée pour assurer la correspondance entre un concept et l'entier correspondant.

À chaque ajout d'un nouveau concept, le dictionnaire vérifie, dans un premier temps, si l'objet Node correspondant est déjà présent. Si ce n'est pas le cas, un compteur est

6. `https://jena.apache.org/`

incrémenté et sa valeur est associée à l'objet Node. Si c'est le cas l'entier correspondant à l'objet Node est retourné. L'utilisation d'un compteur incrémenté garantit l'unicité des entiers utilisés pour identifier les objets Node.

L'opération consistant à retrouver un objet Node à partir d'un long n'est utile qu'à la fin de l'inférence, pour récupérer les données inférées sous la forme de concepts. Nous avons donc choisi d'utiliser les long comme clés dans la HashMap. Cela permet de retrouver plus rapidement le long correspondant à un objet Node, qui est l'opération la plus souvent utilisée.

Le **triplestore** stocke les triples représentés sous la forme de trois long. Pour optimiser l'accès à ces long, nous utilisons le partitionnement vertical (c.f. figure 4.5), présenté dans [32]. Cette méthode utilise une table de hachage faisant correspondre à chaque prédicat les sujets présents dans les triples contenant ce prédicat. À ces sujets sont associés les objets présents dans les triples contenant à la fois ce prédicat et ce sujet. Pour cela, nous utilisons une HashMap de MultiMap (de la librairie Guava [7]), définie comme ceci : HashMap<long,MultiMap<long,long>>. Une MultiMap permet d'associer plusieurs valeurs à une clé. Le premier long fait référence au prédicat, le second au sujet et le troisième à l'objet, comme présenté dans la figure 4.5.

5.1.3 Buffers

Les **buffers** sont les objets responsables de la récupération des triples pouvant être utilisés par une règle donnée. Ils ont un rôle crucial dans le fonctionnement du raisonneur. Ils permettent de réguler l'arrivée des nouveaux triples, tout en assurant le traitement de tous les triples envoyés dans le système.

Pour représenter la file de triples qui compose le buffer, nous avons utilisé une ConcurrentLinkedQueue<Triple>. Grâce à son fonctionnement *FIFO* (ou PEPS : premier entré premier sorti), les triples sont récupérés par les **exécuteurs de règle** dans l'ordre dans lequel ils sont arrivés. De plus, cette structure intègre des mécanismes de gestion de l'accès concurrent, permettant à plusieurs **distributeurs** d'ajouter des triples dans un **buffer** simultanément. Enfin, c'est une file non bloquante, caractéristique indispensable comme nous l'avons précisé dans la sous-section 4.4.2.

L'algorithme 5.1 illustre l'arrivée d'un nouveau triple dans le **buffer**. Après avoir ajouté le nouveau triple à la file du **buffer**, le compteur est incrémenté. Si ce compteur est plus grand que la limite fixée (la taille du **buffer** dans le chapitre précédent), le compteur est remis à zéro et un **module de règle** est notifié. Un **exécuteur de règle** est instancié par ce dernier.

7. `http://docs.guava-libraries.googlecode.com/git/javadoc/com/google/common/collect/Multimap.html`

ALGORITHME 5.1 – Ajout d'un triple dans un **buffer**

```
   Données : buffer, triple, moduleRegle
1  buffer.file.ajout(triple)
2  buffer.compteur++
3  si buffer.compteur>=buffer.limite alors
4  |  moduleRegle.notifie()
5  |  buffer.compteur← 0
6  fin
```

5.1.4 Distributeurs

Les **distributeurs** reçoivent les triples inférés par un **exécuteur de règle**, les enregistrent dans le **triplestore**, et les envoient aux **buffers** des règles pouvant les utiliser. Chaque **distributeur** dispose d'un ensemble de **buffers** abonnés. Ce fonctionnement est basé sur le patron de conception observateur/observable [8]. Il existe deux types d'abonnés. Les abonnés dit *universels*, correspondant à des règles universelles et à qui sont envoyés tous les triples inférés. Les autres abonnés auxquels n'est envoyée qu'une partie des triples, correspondant à ceux effectivement utilisables par la règle du **buffer**. Le filtrage des triples pour ces derniers est fait grâce aux prédicats des prémisses déclarés par les **exécuteurs de règles**.

L'algorithme 5.2 donne un aperçu de la redistribution des triples au travers des **distributeurs**. Cet algorithme est utilisé pendant l'exécution du raisonneur, lorsqu'un **distributeur** reçoit de nouveaux triples à distribuer. Il vient en complément de l'algorithme 5.5 permettant de lier les **buffers** et les **distributeurs**. L'algorithme 5.5 établit pour chaque **distributeur** la liste des **buffers** auxquels envoyer les triples inférés en fonction de leurs prédicats. L'algorithme 5.2 utilise cette liste pour répartir les triples entre les **buffers** définis par l'algorithme 5.5 en fonction de leur prédicat.

5.2 Exécution parallèle et concurrente

Pour assurer une réactivité maximale, l'exécution des règles est parallélisée. Un certain nombre de mécanismes permet cette exécution parallèle sans conflit d'accès aux données, tout en garantissant des performances optimales.

5.2.1 Exécution parallèle des règles

Chaque règle fait l'objet d'une classe Java conforme à l'interface de la figure 5.3. Une instance de cette classe correspond à un **exécuteur de règle**.

8. `http://design-patterns.fr/observateur`

ALGORITHME 5.2 – Distribution des triples inférés aux différents **buffers** abonnés

Données : abonnes, abonnesUniversels, triplesADistribuer
pour *buffer* ∈ *abonnesUniversels* **faire**
  buffer.envoyer(triplesADistribuer)
fin
pour *triple* ∈ *triplesADistribuer* **faire**
  pour *buffer* ∈ *abonnes.get(triple.predicat)* **faire**
    buffer.envoyer(triple)
  fin
fin

```
public interface RuleExecutor {
    public Collection<Triple> execute(TripleStore ts, TripleStore newTriples);
}
```

FIGURE 5.3 – Interface Java des règles d'inférence utilisables dans l'architecture proposée

Lorsqu'un **buffer** est plein ou a atteint le *timeout*, il notifie un **exécuteur de règle** en charge d'appliquer la règle d'inférence. L'**exécuteur de règle** commence par récupérer les triples contenus dans le **buffer**. Comme nous l'avons vu dans la sous-section précédente, le nombre de triples récupérés dépend de la raison de l'instanciation de l'exécuteur : soit le **buffer** est plein, soit il a atteint le timeout. Si le **buffer** est plein, le nombre de triples récupérés correspond à la limite fixée, ou taille du **buffer**. Si le timeout a été atteint, le nombre de triples récupérés est celui du compteur au moment du timeout. Afin de simplifier la recherche d'éléments dans l'ensemble des triples ainsi récupérés, ils sont regroupés dans une structure identique à celle du **triplestore**, à savoir un partitionnement vertical.

Une fois les triples récupérés depuis le **buffer**, il reste à trouver dans ces triples les ensembles de triples satisfaisant les prémisses de la règle afin d'en déduire les conclusions. Un accès au **triplestore** lui permet de compléter des correspondances. L'algorithme 5.3 illustre l'implémentation de la règle `cax-sco`.

Chaque **exécuteur de règle** est implémenté comme un processus léger (ou *thread*). Toutes les instances d'**exécuteur de règle** sont envoyées à un gestionnaire de processus légers (ou *threadpool*). Ce dernier exécute les processus qui lui sont soumis en fonction des ressources système disponibles. Si celles-ci deviennent insuffisantes, les nouveaux processus sont mis en file d'attente pour être exécutés lorsque les ressources nécessaires

ALGORITHME 5.3 – Implémentation de la règle d'inférence `cax-sco`

```
   Données : triplesbuffer, triplestore
   Résultat : triplesinferes
 1 triplesinferes← ∅
 2 triplesSubclassof←triplestore.predicat(subclassof)
 3 si triplesSubclassof≠ ∅ alors
 4 |   triplesType=triplesbuffer.predicate(type)
 5 |   pour tripleType ∈ triplesType faire
 6 |   |   pour tripleSCO ∈ triplesSubclassof.sujet(tripleType.objet) faire
 7 |   |   |   triplesinferes←(tripleType.sujet,type,tripleSCO.objet)
 8 |   |   fin
 9 |   fin
10 fin
11 triplesSubclassof←triplesbuffer.predicat(subclassof)
12 si triplesSubclassof≠ ∅ alors
13 |   triplesType=triplestore.predicate(type)
14 |   pour tripleType ∈ triplesType faire
15 |   |   pour tripleSCO ∈ triplesSubclassof.sujet(tripleType.objet) faire
16 |   |   |   triplesinferes←(tripleType.sujet,type,tripleSCO.objet)
17 |   |   fin
18 |   fin
19 fin
```

sont libérées. Grâce à ce principe, l'exécution de chaque **exécuteur de règle** est garantie, tout en évitant une surcharge des ressources disponibles.

L'implémentation des **exécuteurs de règle** suit l'interface de la figure 5.3. Toute règle dont l'implémentation est conforme à cette interface peut être utilisée dans Slider, y compris des règles qui ne sont pas définies par le W3C. Pour faciliter la recherche des prémisses dans les triples du **buffer** et maximiser la réutilisation du code existant, ces derniers sont stockés dans une structure identique au **triplestore**.

5.2.2 Gestion de la concurrence

La parallélisation d'un programme nécessite un certain nombre de garanties. En cas d'accès simultané à une ressource par plusieurs processus, nous devons nous assurer que les données restent consistantes, et que les processus ne se retrouvent jamais dans des situations de blocage mutuel. Pour cela, chaque objet accessible par différents processus dispose d'un système de verrous. Deux niveaux de verrous sont utilisés : un pour l'écriture et un pour la lecture. Le verrou en écriture empêche tout processus d'accéder

à la ressource verrouillée, alors que le verrou en lecture autorise les autres processus à lire les données de la ressource, mais interdit toute modification.

Le **triplestore**, le **dictionnaire** et les **buffers**, dont les données sont accessibles par plusieurs processus du raisonneur, disposent donc de verrous en lecture et en écriture. L'objet Java ReentrantReadWriteLock implémente un système de verrous pour la lecture et l'écriture. Les instructions rwlock.readLock().lock() et rwlock.writeLock().lock() permettent respectivement de verrouiller le verrou en lecture et en écriture. Les instructions similaires rwlock.writeLock().unlock() et rwlock.readLock().unlock() permettent de déverrouiller le verrou. Les objets sont verrouillés de manière globale. Nous avons implémenté un **triplestore** gérant différents niveaux de verrous. Le verrou global est utilisé pour l'accès à la liste des prédicats. Un verrou par prédicat est ensuite utilisé pour l'accès aux sujets et aux objets associés. Les performances de cette version du **triplestore** sont cependant moins intéressantes. Cela peut s'expliquer par le surcoût des multiples verrouillages et déverrouillages que son utilisation entraîne. Nous avons donc conservé le **triplestore** utilisant un verrou global.

L'algorithme 5.4 illustre l'utilisation d'un verrou pour l'ajout d'un triple dans un **buffer**. L'accès en écriture sur le **buffer** est verrouillé au début de l'algorithme et n'est relâché qu'après tous les accès aux données du **buffer**, à la fin de l'algorithme. Cela permet de garantir la consistance des données pendant cette opération.

ALGORITHME 5.4 – Ajout d'un triple dans un **buffer** avec l'utilisation d'un verrou

```
   Données : buffer, triple, moduleregle
1  rwlock.writeLock().lock()
2  buffer.file.ajout(triple)
3  s buffer.compteur++
4  si buffer.compteur>=buffer.limite alors
5  |  moduleregle.notifie()
6  |  buffer.compteur← 0
7  fin
8  rwlock.writeLock().unlock()
```

5.2.3 Initialisation et fin de l'inférence

Pour être générique, Slider ne demande aucune intervention supplémentaire lors de son utilisation, hormis le choix du fragment de règles d'inférence. Seules les règles composant ce fragment sont utilisées. Durant la phase d'initialisation, le graphe de dépendance des règles (c.f. section 4.5.1) est calculé et utilisé afin de définir vers quels **buffers** chaque **distributeur** enverra les triples qu'il reçoit. Pour que ce mécanisme fonctionne, il est seulement nécessaire lors de l'implémentation de l'**exécuteur de règle** de définir les

prédicats utilisés par la règle correspondante, et ceux qu'elle est susceptible de générer. Grâce à ces informations, le raisonneur peut déterminer quelle règle peut utiliser les triples inférés par une autre règle. Si la liste des prédicats pouvant être utilisés par la règle est laissée vide, le système considérera cette règle comme universelle.

L'algorithme 5.5 est utilisé à l'initialisation du raisonneur pour connecter les **distributeurs** aux **buffers**. Il donne à chaque distributeur la liste des **buffers** auxquels envoyer les triples inférés en fonction du prédicat du triple. Une MultiMap est une variante de HashMap faisant correspondre à une clé plusieurs valeurs. Nous utilisons cette structure pour faire correspondre à un prédicat un ensemble de **buffers** à qui envoyer les triples avec ce prédicat. Une HashMap contient les **buffers** universels, auxquels tous les triples doivent être envoyés.

ALGORITHME 5.5 – Création des liens entre les règles à l'initialisation du raisonneur

Données : MdR les modules de règles
1 **pour** $(MdR_1, MdR_2) \in MdR \times MdR$ **faire**
2 $premisses_1 \leftarrow Premisses(MdR_1.executeurDeRegle)$
3 $premisses_2 \leftarrow Premisses(MdR_1.executeurDeRegle)$
4 $conclusions_1 \leftarrow Conclusions(MdR_1.executeurDeRegle)$
5 $conclusions_2 \leftarrow Conclusions(MdR_2.executeurDeRegle)$
6 **si** $conclusions_1 == \emptyset$ **alors**
7 $MdR_1.distributeur.abonnerUniversel(MdR_2.buffer)$
8 **fin**
9 **si** $conclusions_1 \in^1 premisses_2$ **alors**
10 $MdR_1.distributeur.abonner(MdR_2.buffer)$
11 **fin**
12 **si** $conclusions_2 == \emptyset$ **alors**
13 $MdR_2.distributeur.abonnerUniversel(MdR_1.buffer)$
14 **fin**
15 **si** $conclusions_2 \in^1 premisses_1$ **alors**
16 $MdR_2.distributeur.abonner(MdR_1.buffer)$
17 **fin**
18 **fin**

Pour ajouter une règle pendant l'exécution du raisonneur, un algorithme similaire est utilisé. Au lieu de boucler sur tous les couples de modules de règles (ligne 1), il ne boucle que sur ceux contenant la nouvelle règle. L'algorithme passe donc d'une complexité $\mathcal{O}(n^2)$ où n est le nombre de règles à $\mathcal{O}(n)$. Le coût à l'exécution est donc réduit, ce qui permet d'ajouter efficacement une nouvelle règle au système pendant son exécution.

Comme nous l'avons décrit dans la sous-section 4.4.6, la détection de la fin de l'inférence n'est pas triviale. Pour cela, nous effectuons une vérification portant sur le nombre d'**exécuteurs de règle** en cours d'exécution et de **buffers** vides. Cette vérification est faite à la fin de l'exécution de chaque **exécuteur de règles**. L'algorithme 5.6

illustre l'implémentation de cette méthode. L'objet *phaser* est un entier immuable (un AtomicInteger) partagé par tous les processus légers du raisonneur. Il est utilisé pour notifier le système de la fin de l'exécution d'une règle, et donc effectuer le test de détection de la fin de l'inférence. Sa seconde utilité est de connaître le nombre d'**exécuteurs de règle** en cours d'exécution. Chaque **exécuteur de règle** incrémente la valeur du *phaser* à son instanciation, et la décrémente à la fin de son exécution. Cela permet de déterminer facilement et efficacement le nombre d'**exécuteurs de règle** en cours d'exécution.

ALGORITHME 5.6 – Détection de la fin du raisonnement

```
1 buffersNonVides = nombreBuffersNonVides() tant que buffersNonVides>0
  faire
2 |   enCours = phaser
3 |   tant que enCours>0 faire
4 |   |   phaser.enAttente()// Notifié par les exécuteurs de règle
5 |   fin
6 |   buffersNonVides = nombreBuffersNonVides()
7 fin
8 /* Fin du raisonnement                                          */
```

5.3 Conclusion sur l'implémentation

Dans ce chapitre, nous avons présenté l'implémentation de notre architecture au travers de Slider.

Le **triplestore** utilisé pour stocker les triples et éliminer les doublons utilise le partitionnement vertical. Il s'appuie sur des HashMap afin d'interdire le stockage de doublons. L'utilisation de verrous en lecture et en écriture permet à tous les modules de Slider d'accéder de manière concurrente au **triplestore**. Les concepts sont stockés sous la forme d'entiers, dont les tests de comparaison sont plus rapides que d'autres objets. Un dictionnaire permet de retrouver les concepts originaux à partir de ces entiers.

Les **buffers** utilisent une file de triples infinie, afin de toujours être en mesure de recevoir des triples. Grâce à cela le système n'est jamais en situation de blocage et garantit le traitement de tous les triples reçus.

À l'initialisation, le graphe de dépendance des règles est utilisé pour connecter les **distributeurs** aux **buffers**. Les **distributeurs** disposent donc d'une liste de **buffers** auxquels envoyer les triples inférés en fonction du prédicat de ceux-ci.

Tous les **exécuteurs de règle** implémentent la même interface RuleExecutor. Toute règle Java implémentée au travers de cette interface peut être ajoutée à Slider. Les **exécuteurs de règle** sont des processus légers, et un gestionnaire de processus léger (ou *threadpool*) est en charge de les exécuter en fonction des ressources système disponibles, et de garantir l'exécution de tous les processus.

Enfin, la détection de la fin de l'inférence permet au système, conçu pour être exécuté en continu dans l'attente de nouveaux triples, de déterminer que plus aucun triple ne peut être inféré, et donc d'arrêter le système uniquement à la fin de l'inférence.

Dans le chapitre suivant, nous présentons les expérimentations menées afin de tester les performances de cette implémentation.

6 Expérimentations

Sommaire

6.1 Présentation des expérimentations

Pour tester au mieux l'implémentation de l'architecture que nous proposons, nous avons lancé une série d'expérimentations sur celle-ci. Nous étudions pour commencer l'impact des différents paramètres sur les performances du raisonneur en section 6.2. Ensuite, nous comparons notre solution à un raisonneur commercial permettant l'inférence sur RDFS et ρdf (section 6.3). Nous évaluons alors les performances incrémentales du raisonneur dans la section 6.4. Pour terminer, nous menons des expérimentations afin de tester le mode alternatif de raisonnement permettant de prioriser l'inférence de certaines connaissances dans la section 6.5.

6.1.1 Environnement de test et ontologies utilisées

Toutes les expérimentations présentées dans ce chapitre ont été exécutées sur une machine sous Ubuntu 14.04 équipée d'un processeur Intel® Xeon® E3-1246 huit cœurs cadencés à 3,5GHz et de 32Go de mémoire vive. Les ontologies utilisées sont stockées sur un disque dur SSD.

Pour ces expérimentations, nous avons utilisé un ensemble de douze ontologies de différents types. Tout d'abord, cinq ontologies ont été générées grâce à l'outil BSBM [71] permettant la génération d'ontologies de tailles variables. Elles sont composées de produits vendus par des vendeurs et notés par différents utilisateurs sur plusieurs sites. Le nombre de chacun de ces éléments dépend de la taille en paramètre du générateur. Ces ontologies permettent de tester le comportement du raisonneur face à un débit important de triples, mais amenant à peu de nouveaux triples inférés.

Ensuite, nous avons utilisé cinq ontologies composées de relations *subClassOf*, dont l'équation 6.1 montre la construction pour une ontologie `subClassOf`n.

$$\begin{array}{l} < 1, type, Class > \\ < i, type, Class > \qquad\qquad i \in \{2, 3, \ldots, n\} \\ < i, subClassOf, (i-1) > \end{array} \tag{6.1}$$

Ces ontologies ont l'avantage d'être rapides et faciles à générer tout en offrant le plus grand intérêt du fait de leur complexité. Après inférence, elles produisent $O(n^2)$ triples uniques, mais $O(n^3)$ triples sont générés en pratique lors de l'inférence (doublons compris) [65]. Ces ontologies permettent donc de tester la capacité des raisonneurs à gérer les doublons.

La dernière catégorie regroupe deux ontologies disponibles en ligne : une basée sur Wikipédia [1], et la seconde sur Wordnet[45]. Elles contiennent de vraies données, contrairement aux ontologies précédentes composées de données générées. Elles sont donc plus représentatives des ontologies qui sont utilisées en pratique en entrée d'un raisonneur.

1. `http://datasets-satin.telecom-st-etienne.fr/cgravier/inferray/wikipediaOntology.zip`

La table 6.1 recense ces ontologies, le nombre de triples qu'elles contiennent et le nombre de triples inférés grâce à ces ontologies pour les fragments ρdf et RDFS.

		Triples inférés			
Ontology	**Taille**	**ρdf**	**ratio**	**RDFS**	**ratio**
BSBM100k	99 914	544	0,54%	33 752	33,78%
BSBM200k	200 007	1 102	0,55%	64 492	32,24%
BSBM500k	500 037	4 347	0,87%	157 831	31,56%
BSBM1M	1000 000	8 664	0,87%	304 065	30,41%
BSBM5M	5000 000	43 212	0,86%	1 449 107	28,98%
wikipedia	458 369	191 574	41,79%	555 653	121,22%
wordnet	473 589	117 659	24,84%	634 692	134,02%
subClassOf50	100	1 176	1 176,00%	1 230	1 230,00%
subClassOf100	200	4 851	2 425,50%	4 955	2 477,50%
subClassOf200	400	19 701	4 925,25%	19 905	4 976,25%
subClassOf500	1 000	124 251	12 425,10%	124 755	12 475,50%
subClassOf1000	2 000	498 501	24 925,05%	499 505	24 975,25%

TABLE 6.1 – Nombre initial de triples et nombre de triples inférés pour ρdf et RDFS, pour chaque ontologie utilisée dans les différentes expérimentations

6.2 Étude des paramètres

Dans cette section, nous présentons les expérimentations que nous avons menées afin de déterminer l'impact des paramètres sur le comportement de notre architecture. Deux paramètres ont été testés : la taille des buffers et le timeout.

Protocole de test L'objectif est de tester l'impact des paramètres sur le comportement du système. Nous faisons varier la taille du buffer entre 1 000 à 100 000 et le timeout entre 5 et 500 millisecondes. Les valeurs intermédiaires ont été choisies pour être équidistantes une fois représentées sur un axe logarithmique.

Pour chaque couple de paramètres dans les bornes définies, nous avons lancé l'inférence douze fois, dont deux exécutions de *warm-up* [2], permettant au programme d'atteindre ses performances optimales. Les temps présentés dans les résultats sont une moyenne des temps d'exécution des dix exécutions suivant le *warm-up*. Chaque test a été effectué pour ρdf et pour RDFS.

Nous avons utilisé quatre ontologies pour cette expérimentation : les ontologies basées sur Wordnet et Wikipédia, ainsi que la plus grande ontologie pour chaque catégorie d'ontologie générée, BSBM et `subClassOf`.

Résultats Les figures 6.1 et 6.2 présentent respectivement les résultats des expérimentations pour ρdf et RDFS [3]. Les temps d'inférence les plus longs sont en foncé, les plus rapides en clair. Les valeurs précises sont fournies en Annexe C.

La première observation que l'on peut faire est que l'impact des paramètres semble être dépendant à la fois du type d'ontologie et du fragment. On peut cependant remarquer des tendances. Pour ρdf, il est globalement plus intéressant d'avoir une taille de buffer plus petite, contrairement à RDFS où le constat est inversé sauf pour l'ontologie `subClassOf`.

Ce comportement peut s'expliquer par la complexité du fragment. En effet, ρdf dispose de moins de règles que RDFS et son graphe de dépendance est moins connexe (c.f. figures 4.7 et A.1). Intuitivement, plus il y a de règles, plus le nombre de triples inférés est important. De surcroît, plus le graphe du fragment est connecté, plus le nombre de règles pouvant utiliser un triple est important, et donc plus les chances qu'une règle soit exécutée après l'ajout d'un triple augmente également. On a donc un nombre d'exécuteurs de règle instanciés plus important avec un graphe de dépendance des règles plus connexe. Il est donc intéressant d'opter pour une taille de buffer plus grande pour limiter ces instanciations, car instancier un exécuteur de règle coûte cher.

2. La machine virtuelle Java est capable d'optimiser le code qu'elle exécute à la volée. Ces optimisations sont optimales après quelques exécutions, appelées *warm-up*. Elles ne sont pas comptabilisées dans les résultats.

3. Pour des raisons de lisibilité après impression, les figures sont en niveaux de gris. Leurs équivalents en couleurs sont en annexes C.1 et C.2

Les expérimentations sur RDFS montrent que, hormis pour `subClassOf`, un faible timeout ralentit le système. Une explication de ce comportement est que lorsque le timeout est faible et la taille du buffer élevée, les buffers sont vidés du fait de l'expiration du timeout et non de la taille du buffer. L'avantage d'une taille de buffer importante pour RDFS est donc neutralisé par un timeout trop faible, entraînant plus d'instanciations d'exécuteurs de règle. Dans la mesure où le timeout a été ajouté pour éviter aux triples de rester indéfiniment dans un buffer, il est logique qu'il ne doive pas interférer avec le fonctionnement du buffer, c'est-à-dire entraîner le vidage d'un buffer recevant encore de nouveaux triples. Le timeout doit donc être réglé en fonction de la taille du buffer.

Les résultats pour l'ontologie `subClassOf` se détachent de ceux des autres ontologies. Les graphiques sont quasiment identiques pour ρdf et RDFS. L'ontologie `subClassOf` est un cas très particulier puisqu'elle ne contient que des relations de subsomption de classes. L'application des règles de ρdf et de RDFS infèrent les mêmes triples, correspondant à la fermeture transitive de la relation `subClassOf`. On voit donc que notre système n'est pas pénalisé par l'ajout de règles qui n'infèrent pas de triples. Cette caractéristique apporte une information supplémentaire : le fonctionnement des composants de l'architecture de Slider (buffers, distributeurs, modules de règles) ne semble pas avoir d'impact sur les performances du raisonneur, puisque ajouter des modules non utiles pour l'inférence ne diminue pas les performances.

Pour l'ontologie `subClassOf`, il est plus intéressant d'avoir une taille de buffer réduite. En effet, à chaque itération, peu de triples peuvent être inférés et ceux-ci sont nécessaires pour continuer l'inférence. Il est donc difficile de paralléliser le raisonnement pour cette ontologie. Avec une taille de buffer importante, la quantité de triples inférés risque de ne pas dépasser la taille du buffer. Le système devra attendre l'expiration du timeout pour instancier un exécuteur de règle, pénalisant les performances du raisonneur du fait de cette attente.

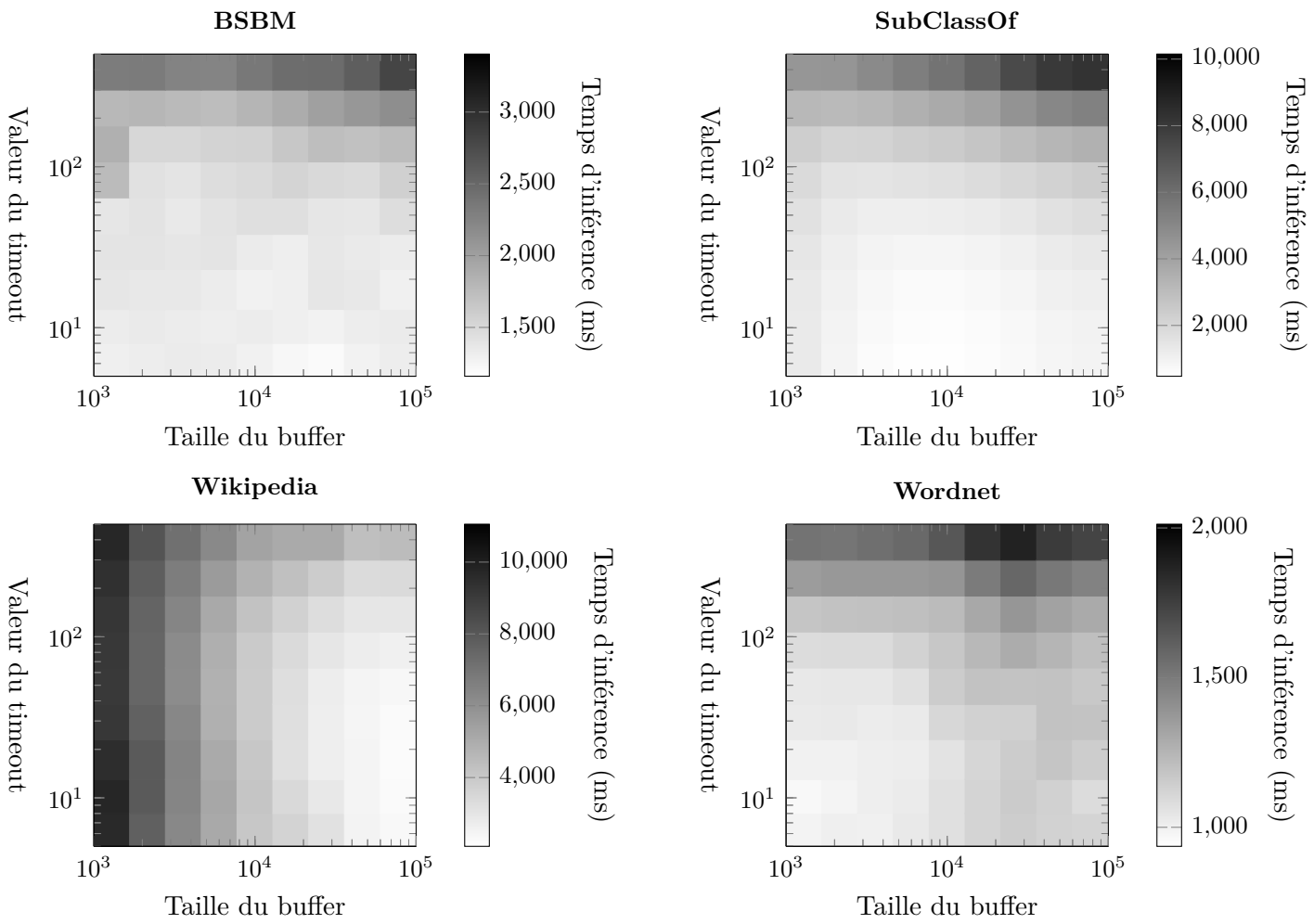

FIGURE 6.1 – Temps d'inférence sur ρdf en fonction de la taille du buffer et du timeout pour les quatre ontologies utilisées

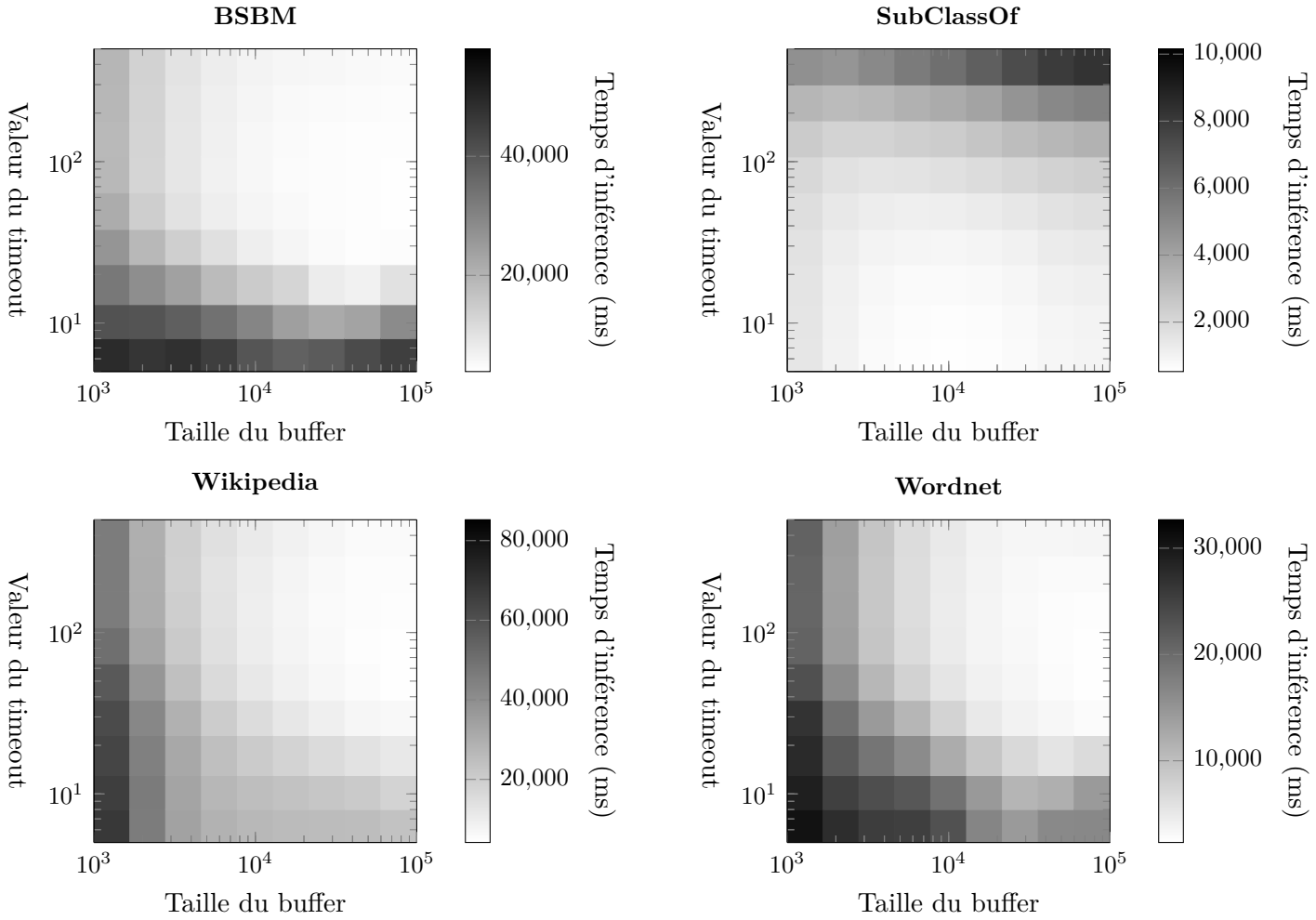

FIGURE 6.2 – Temps d'inférence sur RDFS en fonction de la taille du buffer et du timeout pour les quatre types d'ontologies utilisées

6.3 Comparaison avec les systèmes de référence

Bases de comparaison Dans cette expérimentation, Slider a été comparé au raisonneur OWLIM-SE (*Standard Edition*) [6], un produit commercial fournissant de meilleures performances que Apache Jena comme reporté dans [34] et que Sesame comme reporté dans [53]. Nous avons également comparé Slider aux raisonneurs RDFox [42] et WebPie [65]. Slider a enfin été comparé à Jena et Sesame, mais les performances de ces deux raisonneurs ne sont en rien comparables à celle de OWLIM-SE.

Protocole de test Slider, OWLIM-SE et RDFox ont été configurés pour utiliser les fragments ρdf et RDFS dans son profil `default` [6], le plus utile en pratique. OWLIM-SE supporte nativement le fragment RDFS et fournit également un mécanisme de déclaration de fragment personnalisé, que nous avons employé pour les expériences où OWLIM-SE infère sur le fragment ρdf. WebPie ne permet le raisonnement que sur RDFS. Dans le cas de trop grandes ontologies, le raisonneur peut potentiellement mettre beaucoup de temps suivant le fragment et sa complexité. Par exemple, le problème d'inférence pour RDFS est *NP-complet* (ou en temps polynomial si le graphe après inférence ne contient pas de *blank nodes*). Ainsi, nous avons positionné un timeout de 1 300 secondes comme temps maximal autorisé pour la conduite d'une inférence. Ce timeout d'inférence est à ne pas confondre avec le timeout par buffer de chaque règle, que nous avons positionné à la valeur 10ms. La taille du buffer a été positionnée à 100 000 triples.

Pour chaque ontologie, l'inférence a été lancée dix fois. Les deux premières exécutions ont servi de *warm-up*, permettant à la machine virtuelle Java d'optimiser le code. Les huit exécutions suivantes ont été comptabilisées dans les résultats présentés. Cela permet, pour les deux raisonneurs, d'obtenir des résultats plus parlant.

OWLIM-SE utilise une méthode permettant de réduire le temps d'inférence en procédant en deux phases. Tout comme WebPie [65], les règles *triviales*, ne générant pas de connaissances utilisables par d'autres règles, sont lancées après une première inférence n'utilisant pas ces règles triviales. La seconde étape de l'inférence est alors lancée en utilisant les règles triviales. Afin de rendre l'expérimentation équitable, nous avons implémenté cette procédure en deux phases dans Slider.

Résultats La table 6.2 recense les temps d'inférence de Slider, RDFox et OWLIM-SE pour RDFS et ρdf sur chaque ontologie. Les temps d'inférence de WebPie sont également présents pour RDFS. Les figures 6.3 et 6.4 illustrent les résultats pour Slider et OWLIM respectivement sur ρdf et RDFS. Afin d'assurer une bonne lisibilité, les temps d'inférence pour WebPie ne sont pas représentés dans la figure 6.4.

	ρdf			**RDFS**			
Ontology	**OWLIM**	**Slider**	**RDFox**	**WebPie**	**OWLIM**	**Slider**	**RDFox**
BSBM100k	3 831	544	298	62 293	3 106	847	297
BSBM200k	5 703	848	481	62 408	4 678	1 565	493
BSBM500k	9 728	1 806	1 103	64 365	8 616	3 084	1 095
BSBM1M	17 379	3 619	2 125	61 271	17 137	7 368	2 162
BSBM5M	78 359	24 098	10 689	89 856	73 685	55 989	10 800
wikipedia	8 855	2 984	840	63 325	7 585	6 705	867
wordnet	25 657	7 679	3 073	31 794	21 667	16 608	3 049
subClassOf50	1 212	175	113	62 279	493	235	115
subClassOf100	1 391	236	117	57 748	654	299	112
subClassOf200	1 732	363	133	58 306	892	431	140
subClassOf500	4 401	1 100	379	65 424	3 855	1 229	386
subClassOf1000	23 138	4 201	2 706	65 311	22 310	4 277	2 808

TABLE 6.2 – Temps d'inférence (en millisecondes) avec Slider, RDFox et OWLIM-SE sur ρdf et avec Slider, RDFox, OWLIM-SE et WebPie sur RDFS

Les résultats montrent que Slider apporte en moyenne une amélioration du temps d'inférence par rapport à OWLIM de 78,47% sur ρdf et de 52,22% sur RDFS par rapport à OWLIM-SE. Cette amélioration est au maximum de 85,80% sur ρdf et de 80,83% sur RDFS. On note également que Slider est plus rapide que OWLIM-SE et WebPie pour toutes les ontologies que nous avons considérées.

Les performances de RDFox sont en revanche plus intéressantes que celles offertes par Slider.

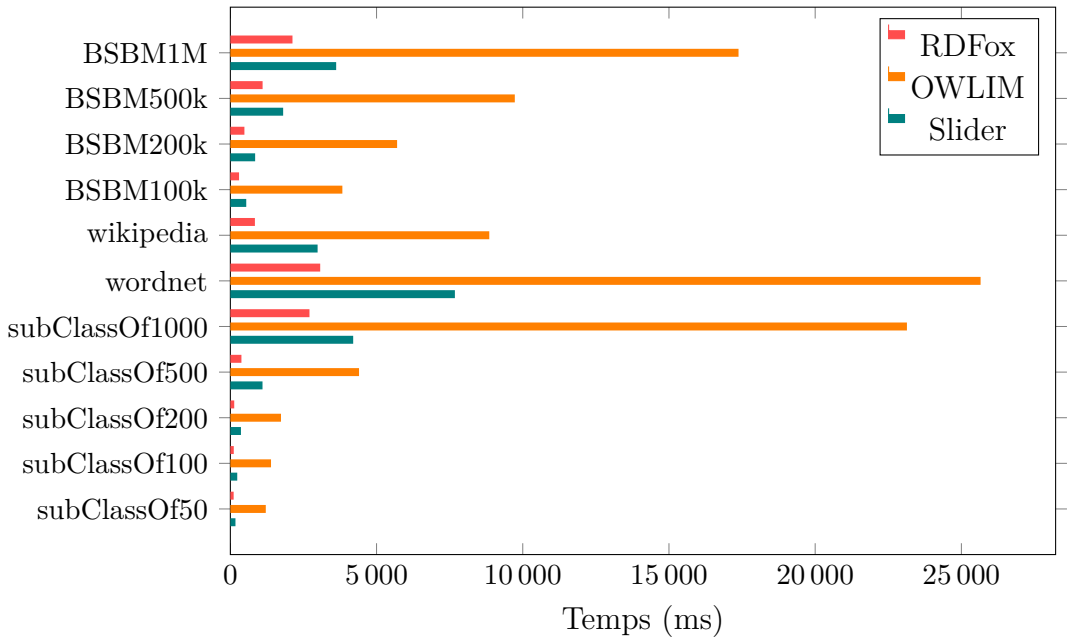

FIGURE 6.3 – Temps d'inférence (en millisecondes) avec Slider, RDFox et OWLIM-SE sur ρdf

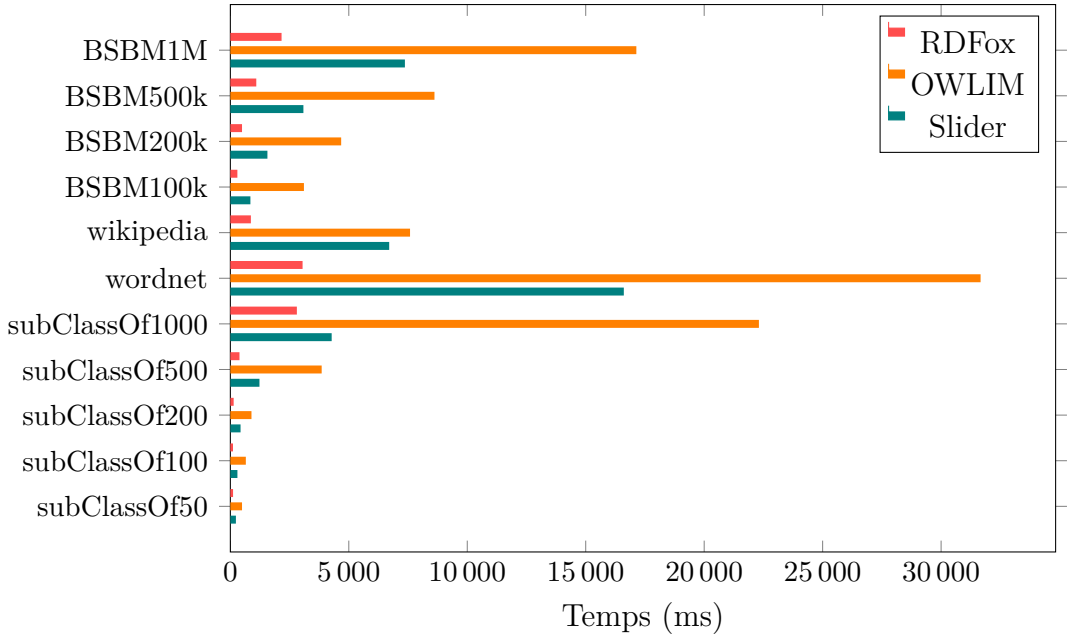

FIGURE 6.4 – Temps d'inférence (en millisecondes) avec Slider, RDFox et OWLIM-SE sur RDFS

6.4 Évaluation des performances incrémentales

L'un des objectifs principaux de cette thèse est de fournir un système incrémental. Tout l'intérêt de l'incrémentalité est de ne pas recommencer le processus d'inférence lors de l'ajout de nouvelles données. À la place, les données déjà inférées sont réutilisées pour ne calculer que le complément nécessaire afin de compléter l'inférence sur les nouvelles données.

Nous avons mené une expérimentation afin de tester les performances incrémentales de Slider. Tous les détails sont présentés dans cette section. Nous commençons par décrire le protocole de test mis en place, avant de donner les résultats de cette expérimentation.

À notre connaissance, le seul raisonneur incrémental dont les fonctionnalités s'approchent de celles de Slider est RDFox, présenté dans l'état de l'art (section 3.2). Il se concentre cependant, dans sa version incrémentale, sur la suppression de triples, contrairement à Slider qui se focalise sur l'ajout de triples.

Protocole de test Nous considérons pour cette expérimentation quatre ontologies. Les deux premières correspondent aux plus volumineuses respectivement pour les ontologies BSBM et subClassOf. Les deux autres sont les ontologies Wikipedia et Wordnet.

Pour cette expérimentation, nous avons séparé chaque ontologie en dix parties égales, notées p_i, la première partie contenant les premiers 10% des triples de l'ontologie, et ainsi de suite. Nous avons choisi de garder l'ordre des triples tel quel. En effet, une autre solution serait de mélanger aléatoirement les triples des ontologies entre les différentes exécutions du raisonneur, pour vérifier si l'ordre des triples a un impact sur les performances. Plusieurs inconvénients nous ont amené à conserver les ontologies telles quelles. Tout d'abord, pour obtenir des résultats réellement concluants, il aurait été nécessaire de tester un maximum de permutations. Or le nombre de permutations pour la plus grande ontologie de BSBM est de $!5\,000\,000$. Même en n'utilisant qu'un faible pourcentage de ces permutations, cela conduit à des expérimentations très longues, et finalement peu représentatives. De plus, dans un cas d'usage réel, on peut intuitivement penser que les triples envoyés ne sont pas sans lien, dans le sens où ils forment certainement une mise à jour avec un sens sémantique global. Nous avons donc décidé d'utiliser les ontologies telles quelles, de manière à conserver les groupes de triples qui ont pu être définis ensemble lors de la création de l'ontologie.

À chaque étape i de l'expérimentation, le raisonnement incrémental est appliqué sur p_i en utilisant les données matérialisées dans l'étape $i-1$. Pour l'étape 1, en l'absence de matérialisation précédente, le raisonnement est fait de la même manière que pour le raisonnement par lots. Les temps mesurés à chaque étape pour la mise à jour de la matérialisation sont consignés. Nous appliquons ensuite le raisonnement par lots sur la concaténation des parties p_1 à p_i pour $1 \leq i \leq 10$. C'est en effet le processus équivalent au précédent pour le raisonnement par lots : il est nécessaire de recalculer toute la matérialisation, sans prise en compte de la matérialisation précédente.

Les temps d'exécution sont consignés pour le raisonnement incrémental et par lots à chaque étape. Grâce à ces mesures, nous comparons le gain de temps apporté en mettant à jour la matérialisation existante plutôt que de la recalculer entièrement.

Résultats Les figures 6.5 et 6.6 illustrent les résultats des expérimentations incrémentales, respectivement pour ρdf et RDFS. Chaque graphique présente les résultats pour une ontologie donnée et compare les temps d'inférence par lots ou incrémentale. Le temps cumulé pour l'inférence incrémentale a été ajouté. Il s'agit de la somme des temps d'inférence de la partie concernée et des précédentes. Par exemple, le temps cumulé pour la partie 30% est la somme des temps d'inférence incrémentale pour les parties 10%, 20% et 30%. Les tables C.5 et C.6 regroupent en annexe les résultats détaillés des expérimentations incrémentales, respectivement pour ρdf et RDFS.

Pour chaque ontologie testée, et tant sur ρdf que sur RDFS, le raisonnement incrémental apporte un gain manifeste. Pour l'ontologie subClassOf et sur ρdf, les deux méthodes, incrémentale et par lots, ont des performances similaires jusqu'à 70%, mais on note qu'après cela le raisonnement incrémental devient bien plus performant. Sur RDFS, toujours pour l'ontologie subClassOf, on remarque un comportement similaire, avec cette fois une amélioration des performances pour le raisonnement incrémental dès 60%. Ce comportement s'explique par la nature particulière de cette ontologie, pour laquelle le raisonneur doit calculer la fermeture transitive des relations de subsomption entre les classes. Malgré la complexité entraînée par ce calcul, le raisonnement incrémental reste aussi performant que le raisonnement par lots avant 60-70%, et plus intéressant après cette limite.

Pour ρdf, la somme des temps pour l'inférence incrémentale est similaire au temps de l'inférence par lots, sauf pour l'ontologie `subClassOf`. Pour RDFS, la somme des temps pour l'inférence incrémentale est plus importante que le temps de l'inférence par lots. Cependant, comme le montrent les figures 6.7 et 6.8, la somme des temps pour l'inférence incrémentale (identique à celle présentée dans les figures C.5 et C.6) reste inférieure pour toutes les ontologies à la somme des temps pour l'inférence par lots. Dans le cas de l'utilisation d'un raisonneur par lots, il serait nécessaire de recalculer entièrement la matérialisation. Comparer les sommes des temps d'inférence pour les deux modes de fonctionnement est donc tout à fait pertinent et permet de comparer les performances de manière équitable.

L'utilisation du raisonnement incrémental par rapport au raisonnement par lots est donc avantageuse pour toutes les ontologies et sur tous les fragments que nous avons testés. Les temps cumulés pour le raisonnement incrémental sont même équivalents aux temps individuels pour le raisonnement par lots. Cela signifie que le coût cumulé de toutes les inférences incrémentales effectuées sur les parties précédentes (permettant donc la mise à jour de la matérialisation et non pas le recalcul complet) est équivalent au temps du raisonnement par lots seulement pour la partie concernée.

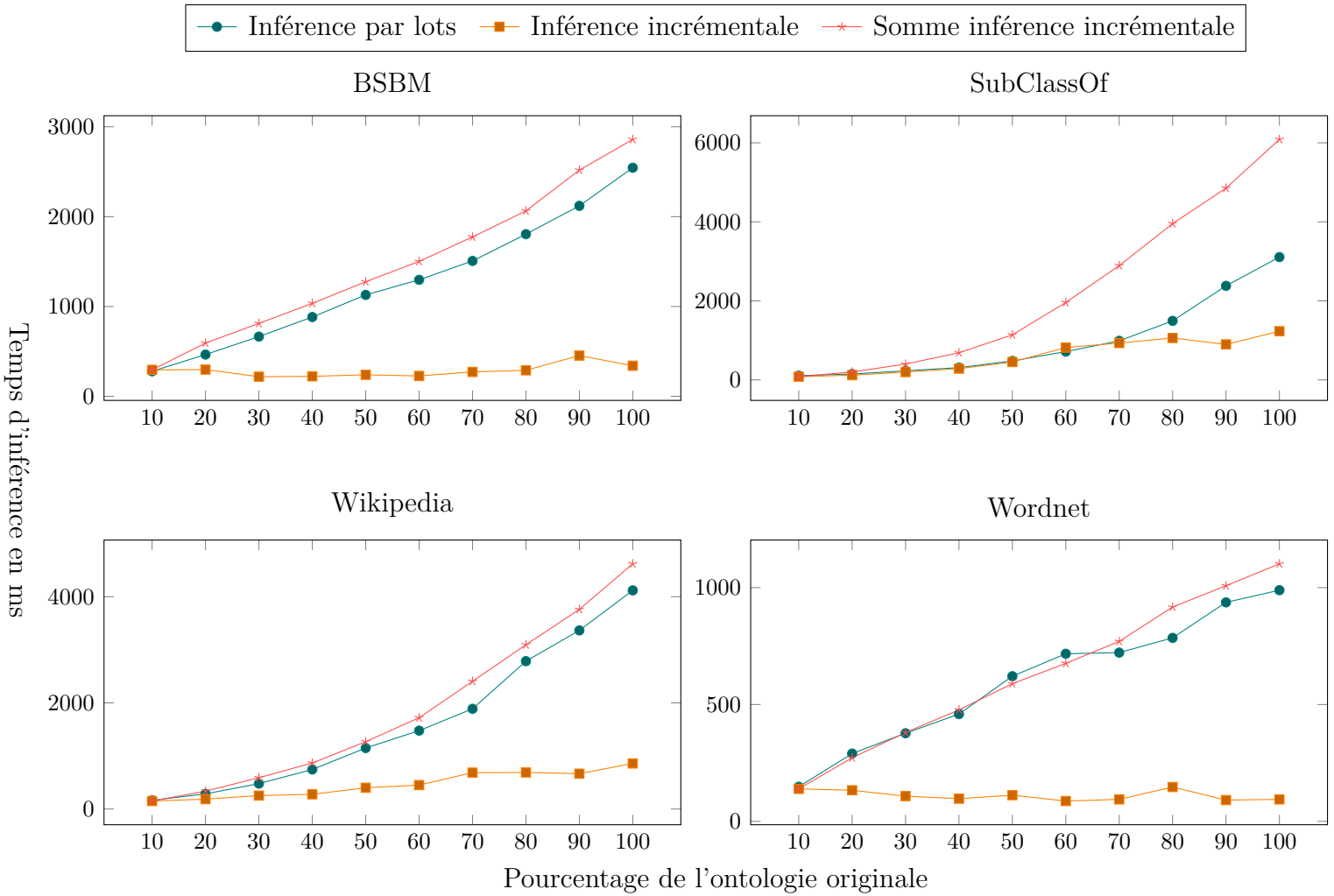

FIGURE 6.5 – Temps d'inférence pour le raisonnement par lots et incrémental avec Slider sur ρdf

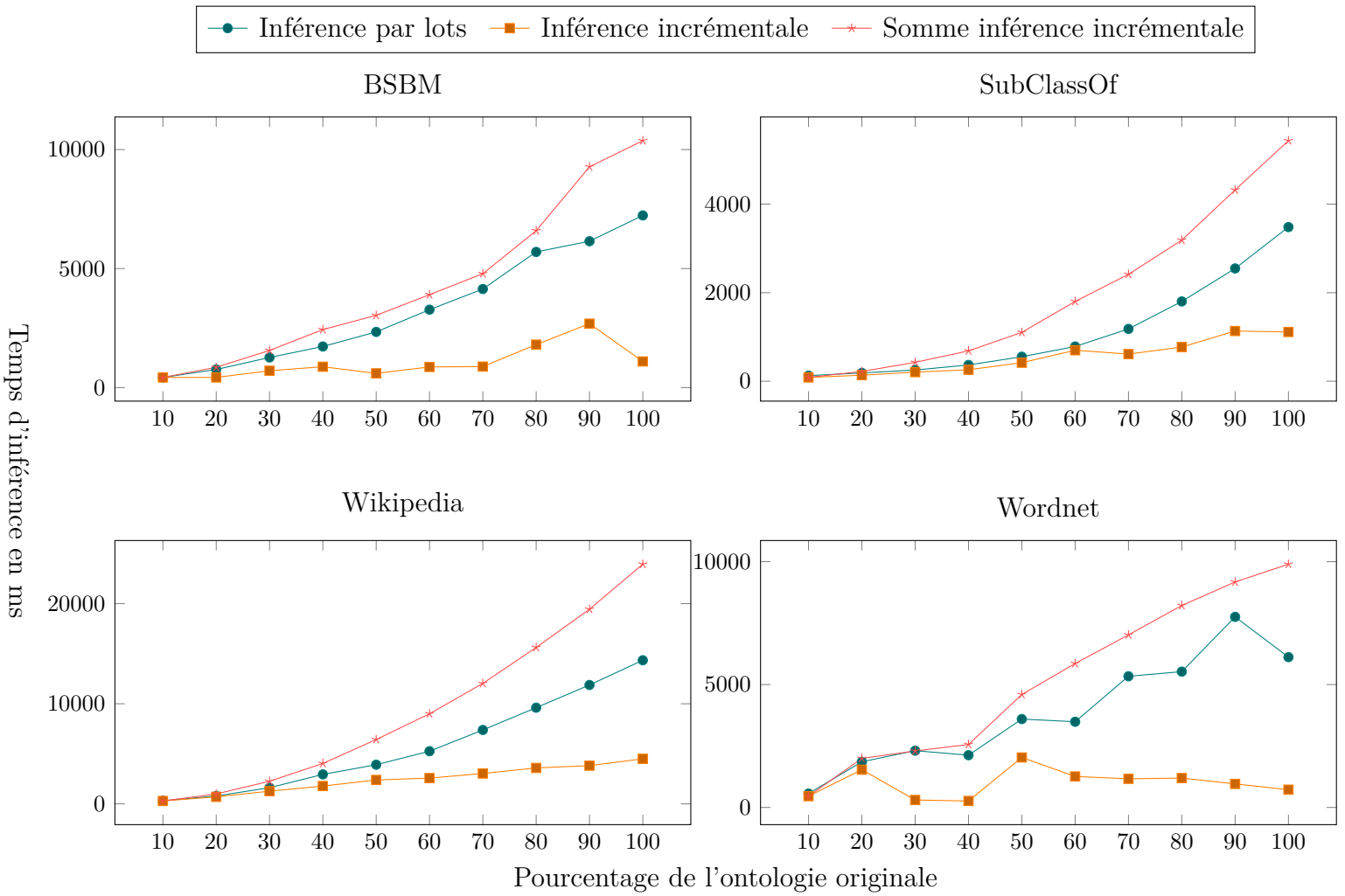

FIGURE 6.6 – Temps d'inférence pour le raisonnement par lots et incrémental avec Slider sur RDFS

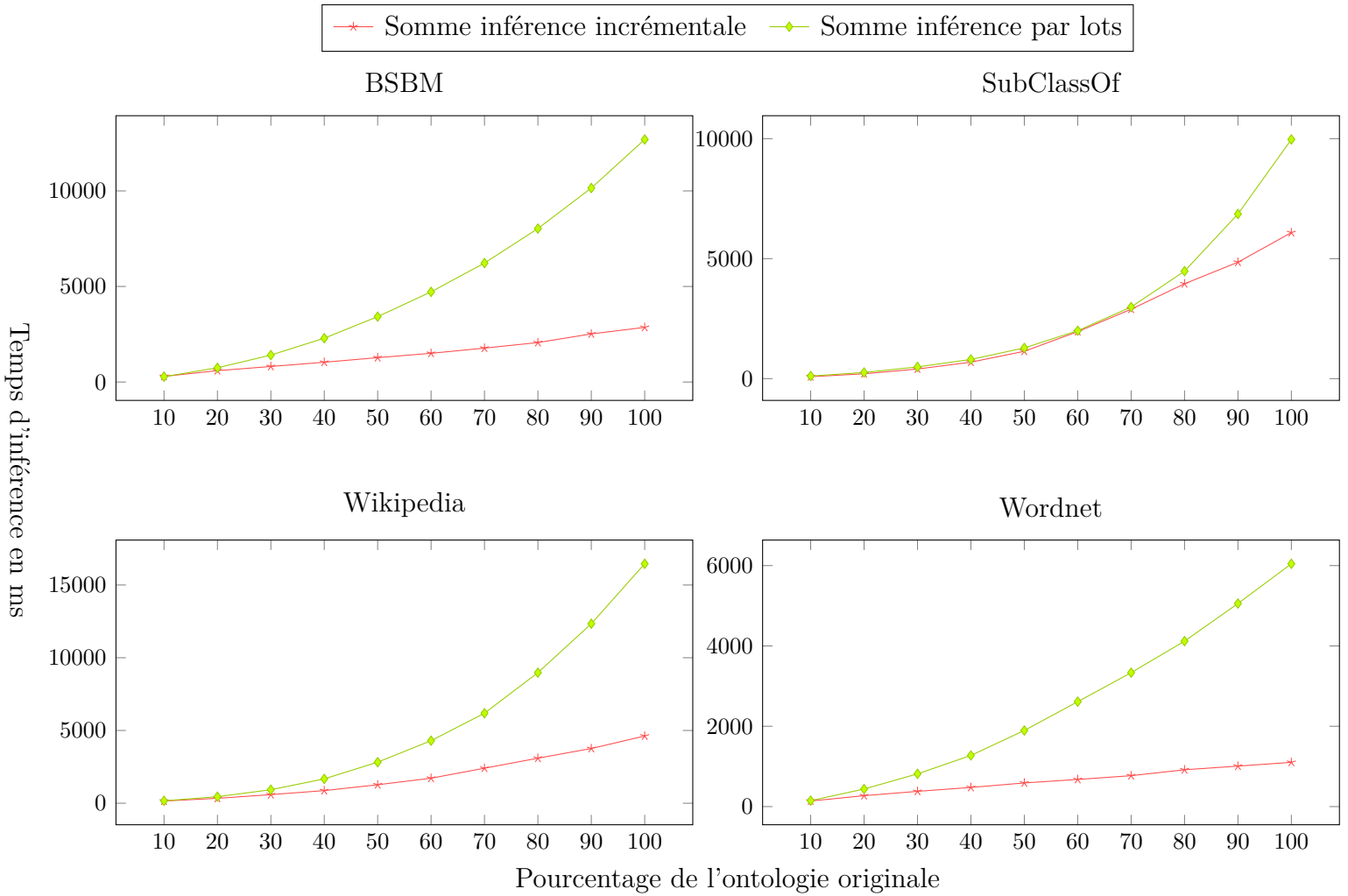

FIGURE 6.7 – Somme des temps d'inférence pour le raisonnement par lots et incrémental avec Slider sur ρdf

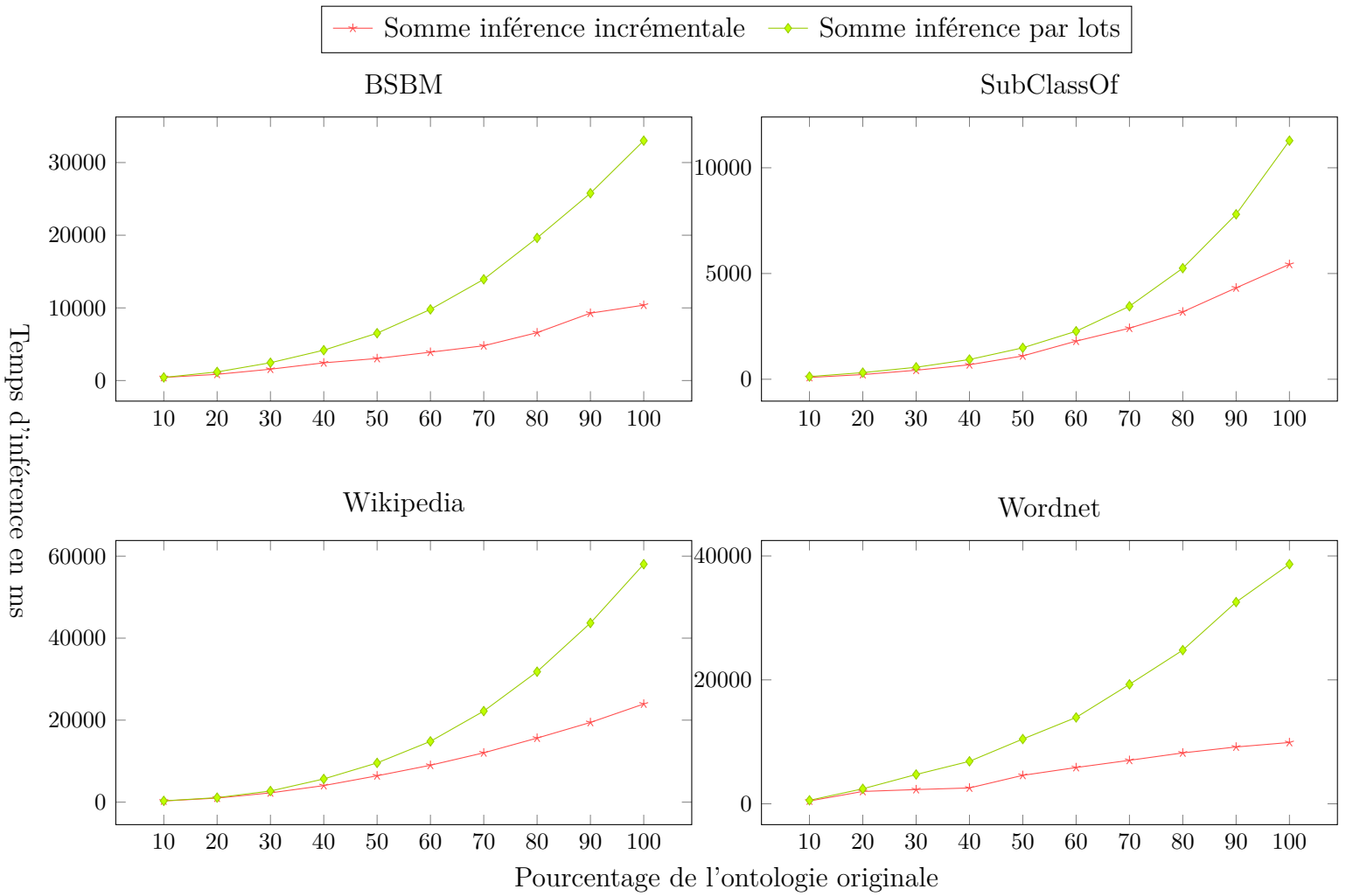

FIGURE 6.8 – Somme des temps d'inférence pour le raisonnement par lots et incrémental avec Slider sur RDFS

6.5 Priorisation des connaissances inférées

Dans cette section nous expérimentons le mode de raisonnement alternatif proposé dans la section 4.7.2. L'objectif est de prioriser l'inférence des triples ayant un prédicat donné. Pour cela, la taille du buffer et le timeout de chaque règle sont calculés dans ce sens, grâce à l'algorithme 4.6.

Protocole de test Dans cette expérimentation nous étudions l'impact de la priorisation des prédicats `subClassOf` et `type` sur l'inférence. Pour cela, nous avons mesuré le nombre de triples inférés par unité de temps, ayant pour prédicat `subClassOf` et `type` [4].

Le raisonnement sur les ontologies BSBM produit peu de triples et ne permet pas d'observer les changements amenés par la priorisation des prédicats. De la même manière, les ontologies subClassOf ainsi que WordNet ne génèrent que des triples avec respectivement les prédicats `subClassOf` et `type`. La priorisation de prédicats n'a donc pas d'intérêt pour ces ontologies.

Nous avons utilisé l'ontologie Wikipedia, qui, au travers de l'inférence, génère un grand nombre de triples tout au long de l'inférence. Les triples générés ont pour prédicat soit `subClassOf` soit `type`. Cette ontologie est donc la plus intéressante des ontologies présentées pour cette expérimentation.

Le raisonnement a été testé dans trois configurations différentes pour chaque fragment. Dans un premier temps, aucun prédicat n'a été priorisé, le raisonneur a été laissé dans sa configuration classique. Ensuite, les prédicats `subClassOf` et `type` sont priorisés à tour de rôle.

La valeur initiale du timeout est de 100 millisecondes et la taille initiale du buffer est de 1000. Les niveaux et les valeurs de ces paramètres lors de la priorisation des prédicats sont présentés pour chaque règle dans les tableaux 6.3 et 6.4. Pour améliorer l'observation de l'impact de la priorisation, nous avons fixé le coefficient α à 50.

Résultats Les figures 6.9 et 6.10 représentent les résultats obtenus pour cette expérimentation respectivement sur ρdf et RDFS. La première observation que l'on peut faire porte sur le fonctionnement classique du raisonneur. Sans priorisation de prédicat, la génération des triples pour chaque prédicat est uniforme tout au long de l'inférence. Des triples avec les prédicats `subClassOf` et `type` sont générés jusqu'à la fin du processus et de manière régulière.

Ces expérimentations valident donc l'hypothèse selon laquelle la priorisation d'un prédicat permet d'inférer les triples avec ce prédicat en priorité. Ceux-ci sont inférés plus rapidement et plus fréquemment. Les triples avec le prédicat qui n'est pas priorisé sont

4. Le dénombrement des triples inférés pour chaque règle entraîne un ralentissement inévitable du système. Les performances et les temps d'inférence ne sont donc pas comparables avec ceux des expérimentations précédentes.

	subClassOf			type		
Règle	Niveau	Taille buffer	Timeout(ms)	Niveau	Taille buffer	Timeout(ms)
cax-sco	3	55930	5593	1	1000	100
prp-dom	3	55930	5593	1	1000	100
prp-rng	3	55930	5593	1	1000	100
prp-spo1	2	35657	3565	2	35657	3565
scm-dom2	3	55930	5593	2	35657	3565
scm-rng2	3	55930	5593	2	35657	3565
scm-sco	1	1000	100	2	35657	3565
scm-spo	3	55930	5593	2	35657	3565

TABLE 6.3 – Niveau, taille du buffer et timeout pour les règles de ρdf pour la priorisation des prédicats `subClassOf` ou `type`

	subClassOf			type		
Règle	Niveau	Taille buffer	Timeout(ms)	Niveau	Taille buffer	Timeout(ms)
cax-sco	3	55930	5593	1	1000	100
prp-dom	3	55930	5593	1	1000	100
prp-rng	3	55930	5593	1	1000	100
prp-spo1	2	35657	3565	2	35657	3565
rdfs12	3	55930	5593	1	1000	100
rdfs13	3	55930	5593	1	1000	100
rdfs4	3	55930	5593	1	1000	100
rdfs8	3	55930	5593	1	1000	100
scm-dom1	3	55930	5593	2	35657	3565
scm-dom2	3	55930	5593	2	35657	3565
scm-rng1	3	55930	5593	2	35657	3565
scm-rng2	3	55930	5593	2	35657	3565
scm-sco	1	1000	100	2	35657	3565
scm-spo	3	55930	5593	2	35657	3565

TABLE 6.4 – Niveau, taille du buffer et timeout pour les règles de RDFS pour la priorisation des prédicats `subClassOf` ou `type`

au contraire inférés à des intervalles moins réguliers, et l'inférence de tous ces triples se termine après l'inférence des triples avec le prédicat priorisé. Ce comportement s'explique par la différence de taille de buffer et de timeout entre les règles d'inférence. De plus, le temps nécessaire pour inférer la totalité des triples dont le prédicat est priorisé est systématiquement inférieur au temps d'inférence sans priorisation. En revanche, comme anticipé dans la section 4.7.2, le temps d'inférence global lors de la priorisation est plus élevé. L'objectif d'inférer certaines connaissances plus rapidement est cependant atteint.

Les résultats obtenus sont donc en adéquation avec nos attentes, puisqu'ils montrent que l'utilisation de l'algorithme 4.6, déterminant la valeur des paramètres pour chaque règle, permet d'influencer l'ordre d'inférence des triples et de prioriser l'inférence pour des prédicats donnés.

Le coefficient α permet de moduler l'écart des paramètres entre les différents niveaux, et donc de régler l'importance de la priorisation. Plus la valeur de α est élevée, plus la priorisation est importante, et vice-versa.

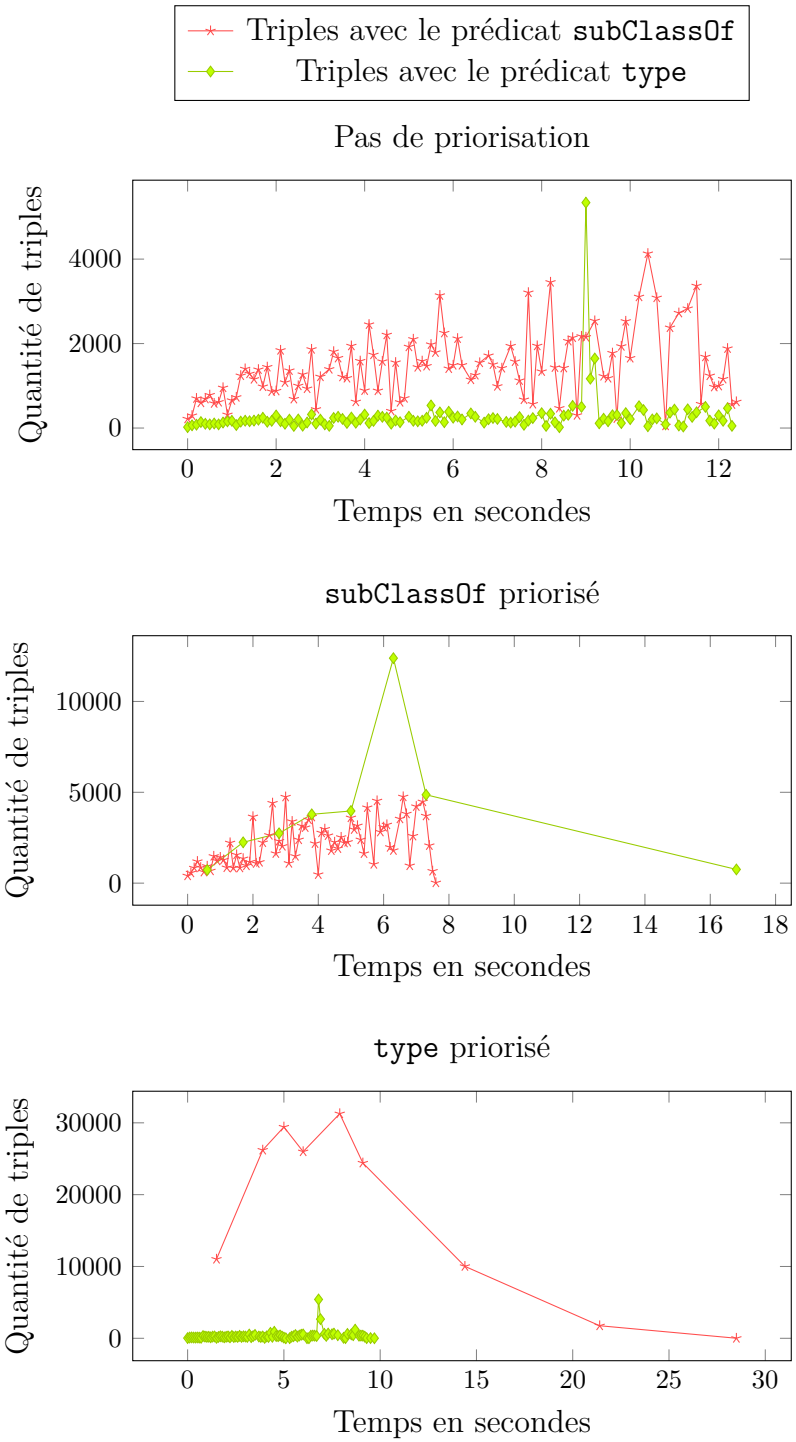

FIGURE 6.9 – Quantité de triples inférés dans le temps avec Slider sur ρdf, suivant les prédicats priorisés

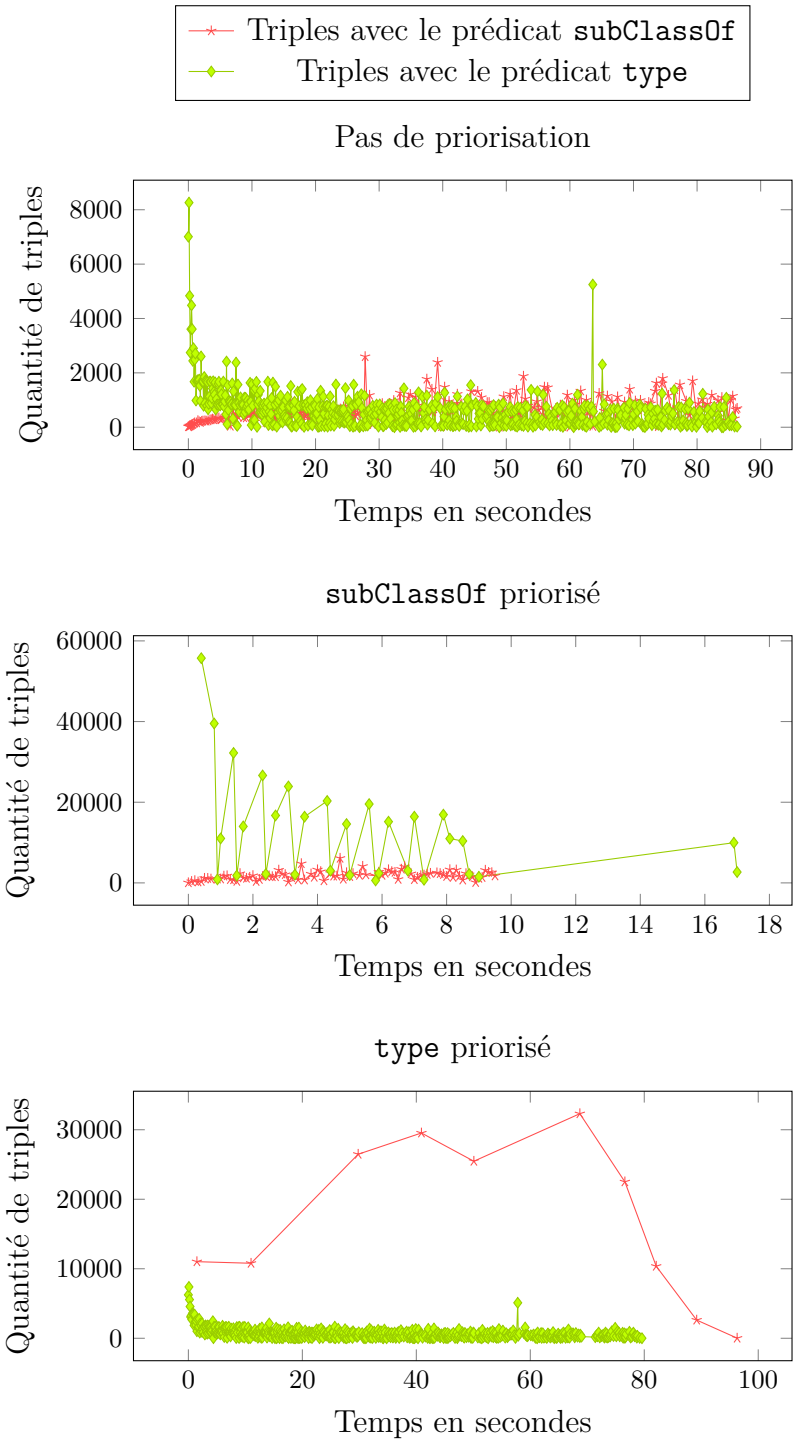

FIGURE 6.10 – Quantité de triples inférés dans le temps avec Slider sur RDFS, suivant les prédicats priorisés

6.6 Reproductibilité

Afin de faciliter la reproduction des expérimentations que nous avons menées dans cette thèse, nous avons détaillé comment les exécuter sur la page d'accueil du projet Slider [5]. Nous avons également mis en ligne les ontologies utilisées ainsi que les scripts ayant permis de les générer le cas échéant. Des exécutables dédiés sont disponibles afin de lancer ces expérimentations.

Toutes ces informations sont détaillées en annexe D. La figure 6.11 présente une copie d'écran de cette page.

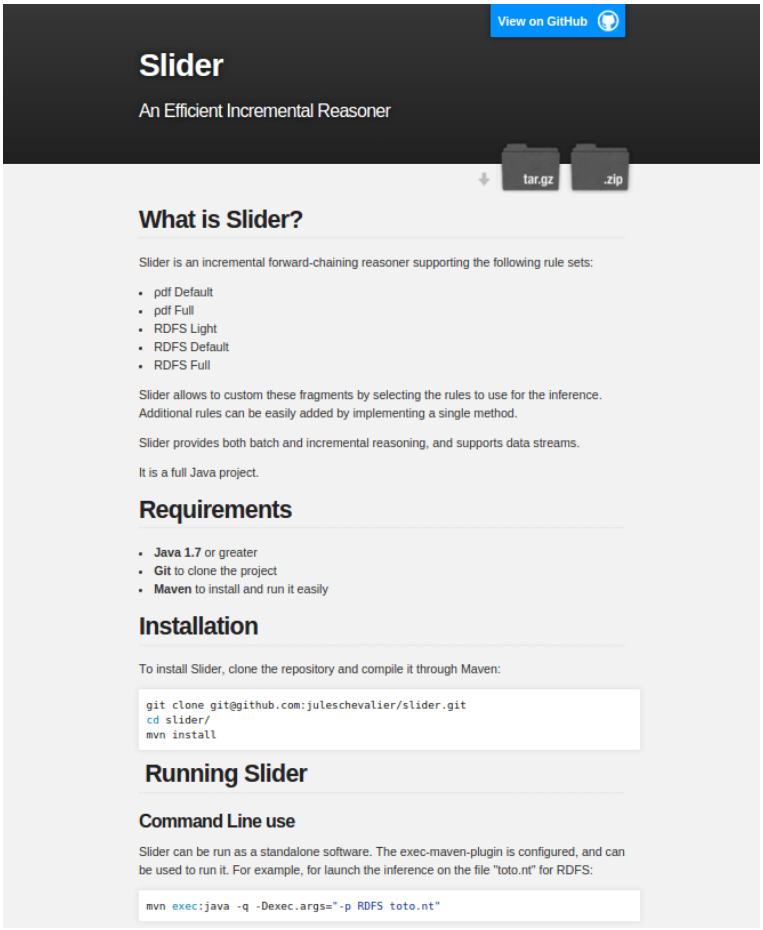

FIGURE 6.11 – Copie d'écran de la page d'accueil du site de Slider

5. http://juleschevalier.github.io/slider/

6.7 Bilan des résultats obtenus

Dans ce chapitre nous avons présenté les différentes expérimentations menées sur l'implémentation de Slider.

Nous avons dans un premier temps utilisé Slider sur quatre ontologies représentatives en faisant varier la taille des buffers et le timeout. Ces expérimentations ont montré que l'influence des paramètres testés sur le comportement du raisonneur est dépendante des ontologies et du fragment. Cependant, nous avons pu observer des tendances générales. Une taille de buffer plus importante est conseillée pour le fragment le plus complexe (ici RDFS), avec un timeout plus important également. Une taille de buffer plus petite est au contraire préférable pour ρdf. Une exception se présente avec l'ontologie subClassOf. En effet, pour celle-ci une taille de buffer réduite permet d'obtenir de meilleurs résultats. Cependant, ce cas particulier nous montre que notre architecture n'est pas pénalisée par l'ajout de règles non utilisées (c'est-à-dire n'inférant pas de triples) dans le système. Le coût de fonctionnement de l'architecture de Slider, en dehors de l'exécution des règles, c'est-à-dire les buffers, les distributeurs, et les modules de règles, semble donc minimal.

Nous avons ensuite comparé Slider à OWLIM-SE et RDFox. Les résultats montrent que Slider est plus rapide que OWLIM-SE pour les douze ontologies testées, et apporte une amélioration du temps d'inférence de 78,47% sur ρdf et de 52,22% sur RDFS, soit 65,35% en moyenne. Cette amélioration est au maximum de 85,80%.

En revanche, RDFox affiche des performances plus intéressantes que Slider sur les ontologies utilisées. Une analyse poussée des structures de données et des algorithmes de ce raisonneur pourrait aider à comprendre les mécanismes qui permettent d'atteindre ces performances. Le développement de RDFox étant récent, cette analyse n'a pas été pu être effectuée dans le cadre de cette thèse.

On peut noter que l'architecture de Slider est conçue pour être à terme distribuée. Les structures de données – buffers, triplestore partagé, exécuteurs de threads, etc – ont donc un coût en terme de performances. Elles permettent entre autre la possibilité de prioriser des connaissances au travers des modes d'inférences présentés. Ces coûts, pénalisant potentiellement Slider dans son implémentation multi processus légers, seront amortis lors de la distribution et le passage à une plus grande échelle.

Nous avons également mis en place des expérimentations sur les capacités incrémentales de Slider. L'utilisation du raisonnement incrémental améliore en moyenne de 230,58% la vitesse d'inférence par rapport à une méthode classique par lots. Cette amélioration est au maximum de 952,13% pour la dernière mise à jour sur RDFS de Wordnet.

Enfin, les expérimentations sur la priorisation des prédicats ont montré l'efficacité du mode de raisonnement alternatif que nous avons proposé dans cette thèse. Les triples contenant les prédicats priorisés sont inférés plus rapidement, et de manière plus régulière.

V

Conclusion

7 Conclusion

7.1 Bilan

Comme nous l'avons vu dans l'état de l'art, les solutions pour le raisonnement incrémental sont rares. Elles montrent des difficultés en ce qui concerne le passage à l'échelle et ne sont efficaces que pour des mises à jour représentant une faible proportion de la matérialisation existante. Notre objectif est de proposer un système de raisonnement incrémental capable de passer à l'échelle.

Dans cette thèse, nous avons défini une nouvelle architecture permettant le raisonnement incrémental sur des flux de triples. Elle est conçue sous la forme de modules indépendants afin de permettre le passage à l'échelle et l'indépendance au fragment de règles utilisé. Notre architecture peut recevoir des triples provenant de plusieurs flux de triples comme de sources de données statiques.

En plus du raisonnement classique consistant à terminer l'inférence le plus rapidement possible, nous proposons également deux autres modes de raisonnement. Le premier permet de modifier le comportement du raisonneur afin d'inférer en priorité certaines connaissances. Le second a pour objectif de maximiser la quantité de triples inférés avant l'arrivée d'une requête pour augmenter les chances d'y répondre correctement.

Nous avons implémenté l'architecture proposée au travers du raisonneur Slider, qui prend en charge nativement les fragments ρdf et RDFS. Il peut être étendu à tout fragment composé de règles implémentables au travers de l'interface correspondante. Les différents modules sont connectés dynamiquement à l'initialisation de Slider, notamment grâce au graphe de dépendance des règles.

Les différentes expérimentations menées nous ont permis d'étudier le comportement de Slider à l'exécution. Nous avons pu déterminer qu'une taille de buffer plus importante est recommandée pour un fragment plus complexe. L'utilisation de l'ontologie `subClassOf` a révélé que l'ajout de règles d'inférence qui n'infèrent pas de triples n'a pas d'impact sur les performances de Slider. La comparaison de Slider avec la référence OWLIM-SE a montré une amélioration des performances de 65,35% en moyenne et au maximum de 85,80% pour certaines ontologies. L'inférence avec Slider a été plus ra-

pide que OWLIM-SE et WebPie pour chaque ontologie testée et sur chaque fragment. Les performances sont par contre moins intéressantes que celles du raisonneur RDFox. Enfin, nos expérimentations ont montré que l'utilisation du raisonnement incrémental apporte un gain de performances de 230,58% en moyenne par rapport à l'utilisation du raisonnement par lots. Cette amélioration est dans le meilleur des cas de 952,13%.

L'implémentation de Slider, les informations pour la reproductibilité des expérimentations et les ontologies utilisées sont disponibles librement à l'adresse `http://juleschevalier.github.io/slider/`. Le code source est libre et distribué sous licence Apache 2.0.

7.2 Perspectives

Notre architecture a été conçue pour être parallèle grâce à sa conception sous forme de modules indépendants. L'unique objet central est le triplestore, auquel tous les modules ont un accès concurrent. Cet accès constitue le principal coût de la parallélisation du raisonnement. Bien que l'accès simultané par plusieurs modules soit possible pour la lecture, il est nécessaire de verrouiller le triplestore pour les opérations d'écriture afin de garantir l'intégrité des données. Ce verrouillage induit des temps d'attente pour les modules, qui même s'ils sont très courts du fait de la simplicité des opérations effectuées, ont un coût sur le processus global.

Pour pallier ce problème, plusieurs solutions s'offrent à nous. Une méthode serait de relâcher des contraintes d'intégrité des données. On ne garantit plus que le raisonnement sera complet, en échange d'une accélération du processus dûe à la suppression des verrous.

Une autre solution est de distribuer le triplestore entre les différents modules pour aller vers une architecture totalement parallèle. Cependant, il est impératif, pour compléter le processus de raisonnement, de permettre à chaque module d'avoir une vue globale des données stockées, notamment pour la gestion des doublons. Dans cette configuration, les modules doivent échanger entre eux les connaissances pour créer une *vue* globale. Ces échanges ont également un coût non négligeable sur les performances du système.

L'ajout de règles dans notre architecture se fait au travers de l'implémentation d'un interface Java. Cela permet une grande généricité des règles prises en charge par notre raisonneur. En contrepartie, la définition d'une nouvelle règle demande des connaissances en programmation. Nous souhaitons donc proposer une méthode plus simple, accessible et standard permettant de définir rapidement de nouvelles règles. Il existe plusieurs méthodes, dont celle utilisée par Jena [1], comme l'illustre l'exemple suivant permettant de définir la règle `cax-sco` :
[cax-sco: (?x rdfs:subClassOf ?y), (?a rdf:type ?x) -> (?a rdf:type ?y)].

1. https://jena.apache.org/

Un tel mécanisme requiert cependant un moteur capable d'interpréter ces règles et de les intégrer ensuite dans notre architecture.

Pour certains modes de raisonnement, les différentes initialisations du raisonneur ne prennent pas en compte les précédentes utilisations du système. Une amélioration possible serait d'utiliser des statistiques collectées lors de ces utilisations précédentes. Ces informations permettraient, par exemple, de déterminer quelles sont les règles générant le plus de triples et ainsi initialiser plus finement le mode *raisonnement à l'aveugle*. Plutôt que de favoriser les règles universelles, la formule 4.3 serait utilisable dès le départ, tirant profit des précédentes utilisations.

Pour le mode de fonctionnement visant à prioriser certaines connaissances spécifiques, ces informations permettraient d'affiner la hiérarchisation des règles en valorisant celles qui, dans les faits, génèrent un maximum de ces triples.

L'utilisation de ces statistiques permettrait donc d'initialiser plus finement les paramètres de notre architecture. Cela repose cependant sur l'hypothèse que les règles auront un comportement similaire d'une utilisation à l'autre. Des expérimentations sont nécessaires afin de valider ou d'invalider cette hypothèse.

Notre architecture a été conçue pour être parallèle. Cela permet à Slider de s'exécuter de manière *multithread*. Une étape suivante pour le passage à l'échelle serait la distribution du raisonneur sur différentes machines. Le *Cloud* est un environnement propice à cette distribution et permettrait au raisonneur de prendre en charge des quantités de données plus importantes et de raisonner sur des fragments plus complexes.

Bien que nous ayons implémenté Slider dans le but qu'il soit parallélisable au maximum, la distribution entre plusieurs machines apporte des contraintes supplémentaires. La première est le partage du triplestore, pour l'instant stocké dans la mémoire vive et donc accessible par tous les processus légers de Slider. De plus, nous utilisons un *thread-pool* pour répartir l'exécution des exécuteurs de règle en fonction des ressources système disponibles. Il est donc nécessaire d'utiliser un mécanisme similaire pour la répartition de la charge sur plusieurs machines.

VI

Annexes

A Fragments

Cette annexe contient la liste des règles d'inférence présentes dans le fragment OWL Horst, ainsi que le graphe de dépendance des règles du fragment RDFS.

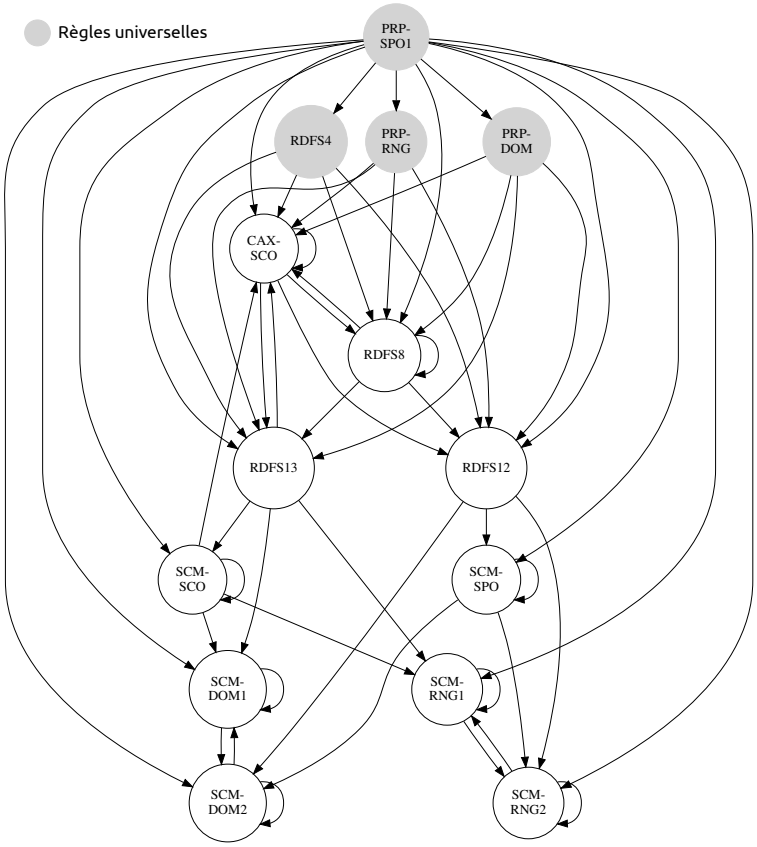

FIGURE A.1 – Graphe de dépendance des règles pour RDFS. Le type des triples échangés a été masqué pour des raisons de visibilité.

#	Condition	Déduction
1	p rdf:type owl:FunctionalProperty $u\ p\ v$ $u\ p\ w$	v owl:sameAs w
2	p rdf:type owl:InverseFunctionalproperty $v\ p\ u$ $w\ p\ u$	v owl:sameAs w
3	p rdf:type owl:SymetricProperty $v\ p\ u$	$u\ p\ v$
4	p rdf:type owl:TransitiveProperty $u\ p\ w$ $w\ p\ v$	$u\ p\ v$
5a	$u\ p\ v$	u owl:sameAs u
6b	$u\ p\ v$	v owl:sameAs v
7	v owl:SameAs w	w owl:sameAs v
8	v owl:sameAs w w owl:sameAs u	v owl:sameAs u
9a	p owl:inverseOf q $v\ p\ w$	$w\ q\ v$
10b	p owl:inverseOf q $v\ q\ w$	$w\ p\ v$
11	v rdf:type owl:Class v owl:sameAs w	v rdfs:subClassOf w
12	p rdf:type owl:Property p owl:sameAs q	p rdfs:subPropertyOf q
13	$u\ p\ v$ u owl:sameAs x v owl:sameAs y	$x\ p\ y$
14a	v owl:equivalentClass w	v rdfs:subClassOf w
15b	v owl:equivalentClass w	w rdfs:subClassOf v
16c	v rdfs:subClassOf w w rdfs:subClassOf v	v rdfs:equivalentClass w
17a	v owl:equivalentProperty w	v rdfs:subPropertyOf w
18b	v owl:equivalentProperty w	w rdfs:subPropertyOf v
19[c]	v rdfs:subPropertyOf w w rdfs:subPropertyOf v	v rdfs:equivalentProperty w
20a	v owl:hasValue w v owl:onProperty p $u\ p\ v$	u rdf:type v
21b	v owl:hasValue w v owl:onProperty p u rdf:type v	$u\ p\ v$
22	v owl:someValuesFrom w v owl:onProperty p $u\ p\ x$ x rdf:type w	u rdf:type v
23	v owl:allValuesFrom w v owl:onProperty p u rdf:type v $u\ p\ x$	x rdf:type w

TABLE A.1 – Règles d'inférence pour le fragment OWL Horst, tirées de [79]

B Sérialisations RDF

Cette annexe expose deux exemples de sérialisations RDF d'une même ontologie. La figure B.1 présente une sérialisation RDF/XML. La figure B.2 présente une sérialisation N-Triples.

```
1  <?xml version="1.0"?>
2  <rdf:RDF xmlns="http://www.semanticweb.org/pet-ontology#"
3       xml:base="http://www.semanticweb.org/pet-ontology"
4       xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
5       xmlns:owl="http://www.w3.org/2002/07/owl#"
6       xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
7       xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#">
8      <owl:Ontology rdf:about="http://www.semanticweb.org/pet-ontology"/>
9
10     <owl:Class rdf:about="http://www.semanticweb.org/pet-ontology#animal"/>
11
12     <owl:Class rdf:about="http://www.semanticweb.org/pet-ontology#canide">
13         <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/pet-ontology#animal"/>
14     </owl:Class>
15
16     <owl:Class rdf:about="http://www.semanticweb.org/pet-ontology#chat">
17         <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/pet-ontology#felin"/>
18     </owl:Class>
19
20     <owl:Class rdf:about="http://www.semanticweb.org/pet-ontology#chien">
21         <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/pet-ontology#canide"/>
22     </owl:Class>
23
24     <owl:Class rdf:about="http://www.semanticweb.org/pet-ontology#felin">
25         <rdfs:subClassOf rdf:resource="http://www.semanticweb.org/pet-ontology#animal"/>
26     </owl:Class>
27
28     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/pet-ontology#Garfield">
29         <rdf:type rdf:resource="http://www.semanticweb.org/pet-ontology#chat"/>
30     </owl:NamedIndividual>
31
32     <owl:NamedIndividual rdf:about="http://www.semanticweb.org/pet-ontology#Gromit">
33         <rdf:type rdf:resource="http://www.semanticweb.org/pet-ontology#chien"/>
34     </owl:NamedIndividual>
35 </rdf:RDF>
```

FIGURE B.1 – Exemple de sérialisation d'une ontologie en RDF/XML

```
<http://www.semanticweb.org/pet-ontology#chat>
    <http://www.w3.org/2000/01/rdf-schema#subClassOf>
    <http://www.semanticweb.org/pet-ontology#felin> .
<http://www.semanticweb.org/pet-ontology#chat>
    <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    <http://www.w3.org/2002/07/owl#Class> .
<http://www.semanticweb.org/pet-ontology#Gromit>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    http://www.semanticweb.org/pet-ontology#chien> .
<http://www.semanticweb.org/pet-ontology#Gromit>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    http://www.w3.org/2002/07/owl#NamedIndividual> .
<http://www.semanticweb.org/pet-ontology#Garfield>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    http://www.semanticweb.org/pet-ontology#chat> .
<http://www.semanticweb.org/pet-ontology#Garfield>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    http://www.w3.org/2002/07/owl#NamedIndividual> .
<http://www.semanticweb.org/pet-ontology#animal>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    http://www.w3.org/2002/07/owl#Class> .
<http://www.semanticweb.org/pet-ontology#felin>
    http://www.w3.org/2000/01/rdf-schema#subClassOf>
    http://www.semanticweb.org/pet-ontology#animal> .
<http://www.semanticweb.org/pet-ontology#felin>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    http://www.w3.org/2002/07/owl#Class> .
<http://www.semanticweb.org/pet-ontology#chien>
    http://www.w3.org/2000/01/rdf-schema#subClassOf>
    http://www.semanticweb.org/pet-ontology#canide> .
<http://www.semanticweb.org/pet-ontology#chien>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    http://www.w3.org/2002/07/owl#Class> .
<http://www.semanticweb.org/pet-ontology#canide>
    http://www.w3.org/2000/01/rdf-schema#subClassOf>
    http://www.semanticweb.org/pet-ontology#animal> .
<http://www.semanticweb.org/pet-ontology#canide>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    http://www.w3.org/2002/07/owl#Class> .
<http://www.semanticweb.org/pet-ontology>
    http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
    http://www.w3.org/2002/07/owl#Ontology> .
```

FIGURE B.2 – Exemple de sérialisation d'une ontologie en N-Triples

C Résultats supplémentaires

Cette annexe contient les résultats complémentaires des expérimentations. Les figures C.1 et C.2 sont les versions couleurs des figures 6.1 et 6.2 présentant les résultats des expérimentations sur l'impact des paramètres. Les tables C.1, C.2, C.3 et C.4 donnent les valeurs exactes des résultats présentés dans les figures 6.1 et 6.2.

Les tables C.5 et C.6 donnent les résultats détaillés des expérimentations sur les performances incrémentales de Slider.

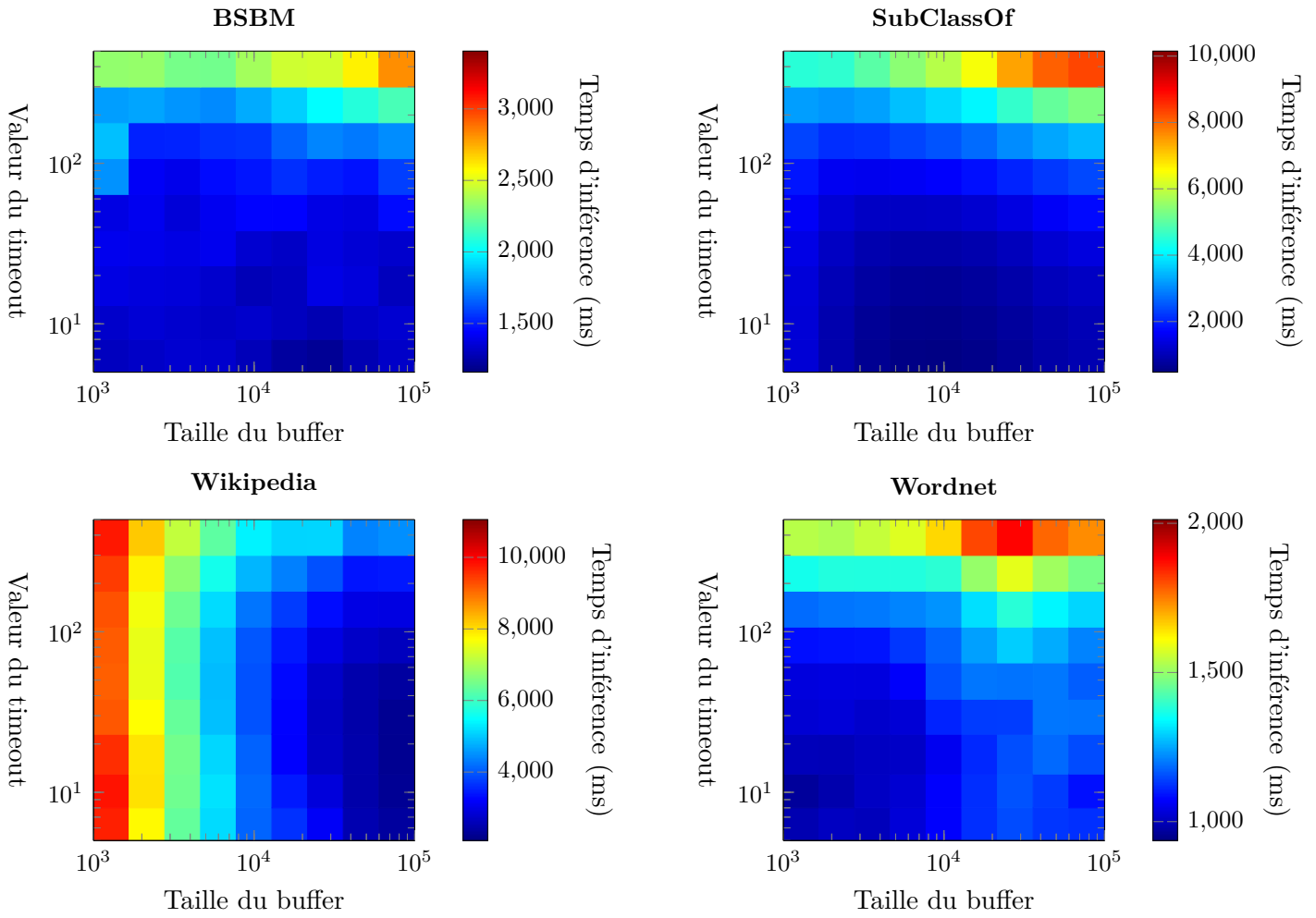

FIGURE C.1 – Temps d'inférence sur ρdf en fonction de la taille du **buffer** et du timeout pour les quatre types d'ontologies utilisées

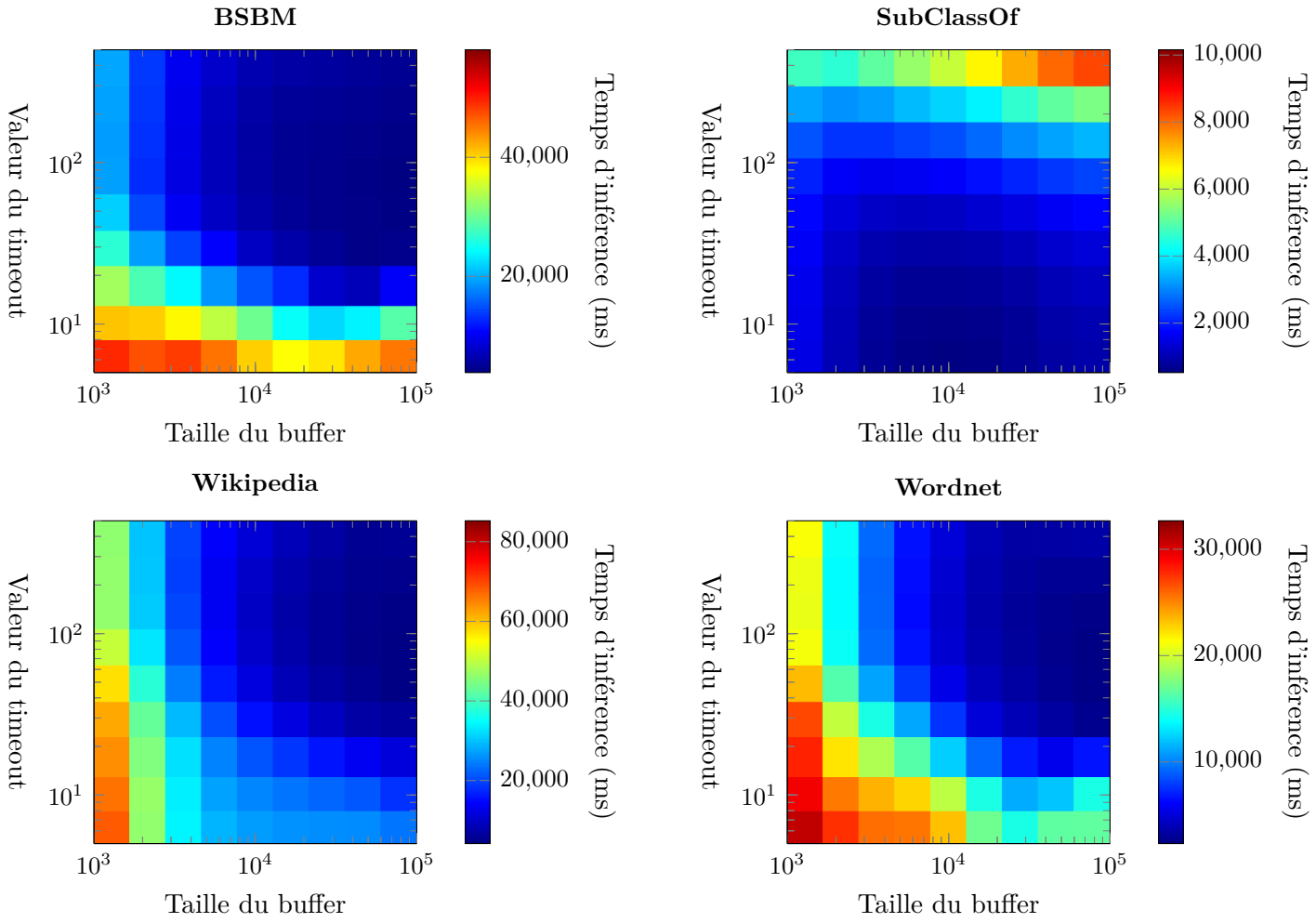

FIGURE C.2 – Temps d'inférence sur RDFS en fonction de la taille du **buffer** et du timeout pour les quatre types d'ontologies utilisées

Taille du buffer	Timeout 5	8	13	23	38	64	107	179	299	500	
1000	2724 +6,61%	2633 +3,06%	2672 +4,58%	3229 +26,36%	3230 +26,39%	2803 +9,69%	2803 +9,70%	2864 +12,07%	3170 +24,04%	3400 +33,06%	ρdf
1668	3294 +28,89%	2764 +8,18%	3193 +24,94%	3160 +23,67%	3260 +27,59%	3238 +26,71%	3338 +30,63%	3508 +37,30%	3050 +19,35%	3800 +48,70%	
2782	3129 +22,46%	3211 +25,68%	2610 +2,14%	3179 +24,41%	2781 +8,85%	2803 +9,71%	2804 +9,72%	2864 +12,09%	3051 +19,40%	3301 +29,20%	
4641	2586 +1,21%	3278 +28,30%	3242 +26,88%	2650 +3,71%	2675 +4,70%	2701 +5,72%	3381 +32,32%	2900 +13,50%	3051 +19,38%	3901 +52,65%	
7742	2579 +0,93%	2618 +2,46%	2713 +6,16%	3216 +25,87%	3193 +24,96%	3227 +26,28%	2804 +9,75%	2901 +13,52%	3052 +19,42%	4003 +56,64%	
12915	3180 +24,44%	3251 +27,21%	2628 +2,85%	**2555 +0%**	3369 +31,84%	3213 +25,75%	2825 +10,56%	3509 +37,32%	3650 +42,82%	3303 +29,25%	
21544	3139 +22,84%	3126 +22,35%	2702 +5,73%	3192 +24,91%	3212 +25,71%	3256 +27,44%	3390 +32,67%	3480 +36,20%	3118 +22,04%	3808 +49,02%	
35938	3095 +21,11%	3237 +26,69%	2581 +1,00%	2662 +4,18%	2669 +4,45%	2712 +6,13%	2857 +11,83%	2981 +16,66%	3778 +47,84%	3820 +49,50%	
59948	3025 +18,37%	3167 +23,93%	2598 +1,66%	2651 +3,73%	3206 +25,47%	2933 +14,77%	3420 +33,85%	3600 +40,87%	3191 +24,86%	4703 +84,03%	
100000	2681 +4,94%	2828 +10,68%	2698 +5,60%	2663 +4,21%	2731 +6,89%	3393 +32,80%	2904 +13,64%	3560 +39,30%	4191 +63,99%	4408 +72,50%	
1000	58060 +1417%	46137 +1105%	36921 +864%	32303 +744%	28448 +643%	23650 +518%	22267 +481%	22518 +488%	22806 +495%	22797 +495%	RDFS
1668	47874 +1151%	44937 +1074%	35544 +828%	26457 +591%	18662 +387%	15788 +312%	14773 +286%	16135 +321%	15439 +303%	16639 +334%	
2782	46738 +1121%	48905 +1177%	32245 +742%	18344 +379%	12803 +234%	10337 +170%	10761 +181%	10914 +185%	10978 +186%	11223 +193%	
4641	55206 +1342%	41756 +991%	29332 +666%	15855 +314%	9224 +141%	7816 +104%	7517 +96,44%	7908 +106%	8081 +111%	9054 +136%	
7742	36920 +864%	46872 +1124%	19133 +399%	9624 +151%	6989 +82,65%	6005 +56,92%	6319 +65,15%	6143 +60,53%	6683 +74,66%	7318 +91,23%	
12915	47484 +1140%	29851 +680%	24211 +532%	7357 +92,25%	5735 +49,89%	5220 +36,42%	4944 +29,22%	5107 +33,46%	5622 +46,92%	6251 +63,36%	
21544	42976 +1023%	30291 +691%	13680 +257%	6224 +62,65%	4840 +26,48%	4402 +15,05%	4599 +20,18%	4623 +20,81%	5245 +37,07%	6029 +57,55%	
35938	45880 +1098%	36857 +863%	7467 +95,14%	4339 +13,38%	4582 +19,74%	4366 +14,10%	4409 +15,22%	4668 +21,99%	5030 +31,44%	5683 +48,51%	
59948	48084 +1156%	38731 +912%	11338 +196%	4021 +5,09%	4093 +6,96%	3922 +2,49%	4039 +5,56%	4327 +13,08%	4620 +20,73%	5257 +37,37%	
100000	48108 +1157%	45274 +1083%	19659 +413%	5919 +54,69%	4119 +7,65%	4168 +8,93%	**3826 +0%**	4590 +19,96%	4425 +15,65%	5040 +31,71%	

TABLE C.1 – Temps d'inférence en fonction de la taille du **buffer** et du timeout pour l'ontologie BSBM. Les pourcentages représentent la différence avec le temps d'inférence le plus bas pour cette ontologie et ce fragment, signalé en gras.

Taille du buffer	Timeout 5	8	13	23	38	64	107	179	299	500	
1000	1473 +211%	1538 +225%	1523 +221%	1540 +225%	1732 +265%	1882 +297%	2354 +397%	2739 +478%	3947 +733%	5505 +1062%	ρdf
1668	1066 +125%	1043 +120%	1068 +125%	1166 +146%	1316 +178%	1494 +215%	1756 +271%	2470 +421%	3588 +657%	4907 +936%	
2782	714 +50,91%	746 +57,58%	797 +68,53%	930 +96,49%	1048 +121%	1264 +167%	1733 +266%	2472 +422%	3887 +721%	5806 +1126%	
4641	548 +15,93%	564 +19,31%	642 +35,78%	768 +62,23%	957 +102%	1254 +165%	1819 +284%	2584 +445%	3902 +724%	6212 +1212%	
7742	**473 +0%**	533 +12,63%	568 +20,03%	735 +55,34%	952 +101%	1331 +181%	1951 +312%	2976 +528%	4555 +862%	7013 +1381%	
12915	502 +6,04%	556 +17,62%	617 +30,42%	766 +61,85%	955 +101%	1337 +182%	2033 +329%	2910 +514%	4531 +857%	7234 +1428%	
21544	590 +24,76%	628 +32,70%	724 +52,98%	862 +82,21%	1079 +127%	1586 +235%	2305 +386%	3556 +651%	5135 +984%	8913 +1782%	
35938	775 +63,71%	782 +65,27%	857 +81,07%	1018 +115%	1209 +155%	1711 +261%	2411 +409%	3795 +701%	5865 +1139%	9688 +1946%	
59948	930 +96,54%	925 +95,40%	1008 +112%	1258 +165%	1516 +220%	2011 +324%	2691 +468%	4122 +770%	6434 +1259%	10151 +2044%	
100000	928 +96,03%	966 +104%	1039 +119%	1198 +153%	1516 +220%	2012 +325%	2765 +484%	4199 +786%	6425 +1257%	10147 +2043%	
1000	1660 +216%	1798 +242%	1775 +238%	1892 +260%	1947 +271%	2086 +297%	2504 +377%	3135 +497%	4007 +663%	6099 +1062%	RDFS
1668	1246 +137%	1213 +131%	1249 +138%	1306 +148%	1441 +174%	1600 +205%	1904 +263%	2506 +377%	3532 +573%	5304 +911%	
2782	802 +52,95%	829 +58,06%	907 +72,89%	964 +83,80%	1109 +111%	1395 +165%	1776 +238%	2648 +404%	3774 +619%	5615 +970%	
4641	615 +17,38%	654 +24,70%	656 +25,12%	831 +58,44%	995 +89,78%	1267 +141%	1840 +250%	2588 +393%	3902 +643%	6802 +1196%	
7742	538 +2,59%	572 +9,04%	632 +20,63%	784 +49,52%	981 +87,00%	1334 +154%	1971 +275%	3009 +473%	4503 +758%	7017 +1237%	
12915	**524 +0%**	560 +6,79%	657 +25,35%	778 +48,46%	997 +90,16%	1414 +169%	2033 +287%	2902 +453%	4526 +762%	8000 +1425%	
21544	607 +15,75%	648 +23,56%	680 +29,78%	873 +66,45%	1117 +113%	1606 +206%	2355 +349%	3581 +582%	5143 +880%	8827 +1582%	
35938	793 +51,32%	808 +54,06%	838 +59,74%	1050 +100%	1263 +140%	1740 +231%	2494 +375%	3770 +618%	5865 +1018%	9480 +1707%	
59948	920 +75,41%	920 +75,49%	1036 +97,56%	1238 +136%	1517 +189%	1986 +278%	2730 +420%	4089 +679%	6438 +1127%	10146 +1834%	
100000	924 +76,29%	962 +83,53%	1054 +101%	1203 +129%	1495 +185%	1986 +278%	2727 +419%	4240 +708%	6432 +1126%	10157 +1836%	

TABLE C.2 – Temps d'inférence en fonction de la taille du **buffer** et du timeout pour l'ontologie subClassOf. Les pourcentages représentent la différence avec le temps d'inférence le plus bas pour cette ontologie et ce fragment, signalé en gras.

Taille du buffer	Timeout 5	8	13	23	38	64	107	179	299	500	
1000	10738 +415%	10740 +415%	10838 +419%	10170 +387%	9813 +370%	10050 +382%	10038 +381%	10464 +402%	10496 +403%	11056 +430%	ρdf
1668	8385 +302%	8790 +321%	8746 +319%	8429 +304%	8284 +297%	8250 +295%	8231 +294%	8184 +292%	8543 +309%	8853 +324%	
2782	6811 +226%	6963 +234%	7297 +250%	7237 +247%	6841 +228%	6638 +218%	6868 +229%	7081 +239%	7414 +255%	7861 +277%	
4641	5727 +174%	5773 +176%	5678 +172%	5636 +170%	5650 +171%	5538 +165%	5803 +178%	5875 +181%	6370 +205%	6992 +235%	
7742	4524 +117%	4512 +116%	4843 +132%	4236 +103%	4069 +95,24%	4172 +100%	4116 +97,49%	4762 +128%	5451 +161%	6273 +200%	
12915	3691 +77,09%	3614 +73,38%	3537 +69,71%	3653 +75,26%	3770 +80,88%	3665 +75,85%	3904 +87,30%	4163 +99,73%	4862 +133%	4792 +129%	
21544	3601 +72,79%	3624 +73,89%	2897 +38,98%	2715 +30,28%	2730 +31,00%	2911 +39,67%	3145 +50,91%	3684 +76,76%	4615 +121%	6068 +191%	
35938	2729 +30,94%	2595 +24,51%	2440 +17,07%	2698 +29,46%	2536 +21,67%	2734 +31,20%	3044 +46,07%	3325 +59,54%	4008 +92,29%	5673 +172%	
59948	2322 +11,44%	2373 +13,85%	2423 +16,24%	2286 +9,69%	2253 +8,09%	2369 +13,69%	2726 +30,79%	2773 +33,05%	3378 +62,08%	4387 +110%	
100000	2526 +21,21%	2167 +3,98%	**2084 +0%**	2160 +3,64%	2279 +9,37%	2581 +23,84%	2840 +36,29%	3477 +66,85%	4022 +92,98%	6080 +191%	
1000	85252 +1948%	77609 +1764%	76170 +1730%	73395 +1663%	72514 +1642%	64839 +1457%	57008 +1269%	54732 +1215%	56784 +1264%	54377 +1206%	RDFS
1668	52352 +1157%	56790 +1264%	53898 +1195%	51549 +1138%	49644 +1092%	42485 +920%	36697 +781%	36718 +782%	36953 +787%	35989 +764%	
2782	36953 +787%	38369 +821%	37218 +794%	36619 +779%	32585 +682%	26812 +544%	24728 +494%	23611 +467%	22947 +451%	23690 +469%	
4641	30602 +635%	29646 +612%	28244 +578%	26292 +531%	20797 +399%	16628 +299%	16221 +289%	14897 +257%	15376 +269%	16172 +288%	
7742	27974 +572%	26578 +538%	24274 +483%	20189 +385%	14837 +256%	12454 +199%	10801 +159%	11306 +171%	11362 +173%	13116 +215%	
12915	26807 +544%	25893 +522%	23398 +462%	16039 +285%	10614 +155%	8864 +112%	8167 +96,24%	8270 +98,72%	9143 +119%	10386 +149%	
21544	27235 +554%	23661 +468%	21512 +416%	12677 +204%	7893 +89,66%	6909 +66,01%	6420 +54,26%	6329 +52,09%	7126 +71,23%	8325 +100%	
35938	26508 +536%	24931 +499%	19607 +371%	9531 +129%	6914 +66,14%	5198 +24,90%	5243 +25,98%	5597 +34,49%	5809 +39,59%	6975 +67,61%	
59948	27251 +554%	23218 +457%	17154 +312%	7030 +68,92%	5095 +22,44%	4582 +10,10%	4519 +8,58%	4935 +18,59%	4774 +14,73%	5316 +27,74%	
100000	23898 +474%	20035 +381%	12319 +196%	9006 +116%	4543 +9,16%	**4161 +0%**	4550 +9,35%	5024 +20,72%	5726 +37,60%	7172 +72,35%	

TABLE C.3 – Temps d'inférence en fonction de la taille du **buffer** et du timeout pour l'ontologie Wikipedia. Les pourcentages représentent la différence avec le temps d'inférence le plus bas pour cette ontologie et ce fragment, signalé en gras.

Taille du buffer	Timeout 5	8	13	23	38	64	107	179	299	500	
1000	970 +3,67%	**936 +0%**	959 +2,48%	1021 +9,03%	1026 +9,59%	1049 +12,09%	1112 +18,82%	1253 +33,85%	1435 +53,27%	1700 +81,59%	ρdf
1668	1049 +12,02%	997 +6,47%	965 +3,08%	1044 +11,58%	1003 +7,16%	1062 +13,48%	1113 +18,92%	1254 +33,94%	1496 +59,78%	1500 +60,21%	
2782	963 +2,90%	1010 +7,86%	964 +2,99%	1007 +7,62%	1064 +13,63%	1024 +9,38%	1155 +23,43%	1253 +33,85%	1495 +59,72%	1600 +70,91%	
4641	1036 +10,72%	987 +5,45%	1070 +14,35%	984 +5,10%	1010 +7,95%	1049 +12,11%	1134 +21,12%	1253 +33,87%	1495 +59,65%	1600 +70,95%	
7742	1018 +8,71%	1077 +15,08%	957 +2,24%	1035 +10,57%	1079 +15,29%	1126 +20,29%	1200 +28,17%	1255 +34,05%	1496 +59,80%	1704 +81,97%	
12915	1055 +12,75%	1124 +20,12%	1114 +18,97%	1115 +19,14%	1195 +27,66%	1219 +30,20%	1158 +23,67%	1292 +38,00%	1499 +60,10%	1904 +103%	
21544	1218 +30,14%	1078 +15,19%	1156 +23,52%	1062 +13,46%	1159 +23,81%	1207 +28,94%	1375 +46,92%	1402 +49,81%	1801 +92,40%	2012 +114%	
35938	1128 +20,48%	1174 +25,40%	1219 +30,26%	1172 +25,25%	1143 +22,08%	1257 +34,30%	1313 +40,26%	1439 +53,70%	1682 +79,65%	2008 +114%	
59948	1115 +19,14%	1088 +16,28%	1048 +11,92%	1287 +37,48%	1180 +26,10%	1204 +28,60%	1233 +31,72%	1342 +43,40%	1573 +68,03%	1821 +94,49%	
100000	1186 +26,66%	1090 +16,45%	1114 +18,97%	1149 +22,79%	1154 +23,30%	1136 +21,36%	1262 +34,79%	1348 +44,02%	1600 +70,87%	1928 +105%	
1000	32170 +1288%	32547 +1304%	31049 +1240%	32677 +1310%	29754 +1184%	24744 +968%	25172 +986%	24046 +937%	25597 +1004%	25385 +995%	RDFS
1668	29572 +1176%	28873 +1146%	24578 +960%	23358 +908%	21296 +819%	17548 +657%	16581 +615%	16743 +622%	16669 +619%	16924 +630%	
2782	25896 +1017%	25622 +1005%	22104 +854%	18217 +686%	15028 +548%	10782 +365%	10765 +364%	10630 +358%	10546 +355%	11238 +385%	
4641	24694 +965%	26242 +1032%	20415 +781%	14810 +539%	9735 +320%	8395 +262%	7260 +213%	7694 +232%	7303 +215%	7836 +238%	
7742	24388 +952%	26348 +1037%	17159 +640%	12610 +444%	7151 +208%	5708 +146%	5157 +122%	5311 +129%	5448 +135%	5706 +146%	
12915	21633 +833%	20384 +779%	13450 +480%	5878 +153%	4902 +111%	3892 +68,01%	4022 +73,63%	3929 +69,61%	4285 +84,95%	4338 +87,24%	
21544	14137 +510%	11881 +412%	12341 +432%	5131 +121%	3820 +64,90%	3614 +56,01%	3197 +38,00%	3216 +38,81%	3363 +45,16%	3545 +53,03%	
35938	17303 +646%	14458 +524%	6322 +172%	3648 +57,48%	3124 +34,87%	2797 +20,74%	2897 +25,04%	2927 +26,35%	3158 +36,31%	3446 +48,74%	
59948	15968 +589%	18337 +691%	8775 +278%	3505 +51,30%	2772 +19,67%	2513 +8,49%	2504 +8,09%	2587 +11,70%	3157 +36,28%	3936 +69,90%	
100000	13216 +470%	18688 +706%	11960 +416%	2374 +2,47%	**2316 +0%**	2347 +1,32%	2467 +6,49%	2757 +19,00%	3102 +33,92%	3768 +62,66%	

TABLE C.4 – Temps d'inférence en fonction de la taille du **buffer** et du timeout pour l'ontologie Wordnet. Les pourcentages représentent la différence avec le temps d'inférence le plus bas pour cette ontologie et ce fragment, signalé en gras.

Ontologie		BSBM	Wordnet	Wikipedia	SubClassOf
	Triples	500000	47358	45836	100
10%	Lot	279	148	164	108
	Inc.	295	139	151	82
	Triples	1000000	94717	91673	200
20%	Lot	465	290	285	149
	Inc.	298	133	186	120
	Triples	1500000	142076	137510	300
30%	Lot	665	377	479	231
	Inc.	219	108	253	199
	Triples	2000000	189435	183347	400
40%	Lot	883	459	745	312
	Inc.	223	97	278	286
	Triples	2500000	236794	229184	500
50%	Lot	1129	621	1149	480
	Inc.	240	112	401	453
	Triples	3000000	284153	275021	600
60%	Lot	1297	717	1477	716
	Inc.	227	87	451	819
	Triples	3500000	331512	320858	700
70%	Lot	1507	722	1888	989
	Inc.	272	94	686	933
	Triples	4000000	378871	366695	800
80%	Lot	1805	785	2786	1494
	Inc.	290	147	689	1062
	Triples	4500000	426230	412532	900
90%	Lot	2120	937	3365	2381
	Inc.	454	91	666	899
	Triples	5000000	473589	458369	1000
100%	Lot	2544	989	4119	3108
	Inc.	341	94	859	1232

TABLE C.5 – Temps d'inférence pour le raisonnement sur ρdf par lots (Lot) et incrémental (Inc.), sur des portions d'ontologies représentant de 10% à 100% de l'ontologie originale. Les temps sont en millisecondes.

Ontologie		BSBM	Wordnet	Wikipedia	SubClassOf
	Triples	500000	47358	45836	100
10%	Lot	426	565	313	126
	Inc.	426	459	289	85
	Triples	1000000	94717	91673	200
20%	Lot	760	1857	786	187
	Inc.	427	1536	701	136
	Triples	1500000	142076	137510	300
30%	Lot	1266	2310	1615	253
	Inc.	708	303	1269	205
	Triples	2000000	189435	183347	400
40%	Lot	1726	2126	2945	367
	Inc.	876	263	1779	259
	Triples	2500000	236794	229184	500
50%	Lot	2339	3596	3910	553
	Inc.	599	2034	2388	418
	Triples	3000000	284153	275021	600
60%	Lot	3273	3486	5259	784
	Inc.	867	1261	2573	699
	Triples	3500000	331512	320858	700
70%	Lot	4141	5337	7380	1181
	Inc.	882	1165	3034	613
	Triples	4000000	378871	366695	800
80%	Lot	5699	5526	9608	1802
	Inc.	1805	1195	3590	772
	Triples	4500000	426230	412532	900
90%	Lot	6149	7754	11873	2547
	Inc.	2685	958	3818	1134
	Triples	5000000	473589	458369	1000
100%	Lot	7232	6116	14347	3482
	Inc.	1098	723	4499	1113

TABLE C.6 – Temps d'inférence pour le raisonnement sur RDFS par lots (Lot) et incrémental (Inc.), sur des portions d'ontologies représentant de 10% à 100% de l'ontologie originale. Les temps sont en millisecondes.

D Reproductibilité des expérimentations

Cette annexe est consacrée à la reproductibilité des expérimentations que nous venons de détailler dans les sections précédentes. La section D.1 explique comment télécharger ou générer les ontologies que nous avons utilisées pour nos expérimentations. Nous donnons dans la section D.2 les détails permettant de lancer les différentes expérimentations.

Afin de maximiser l'utilisation et l'accessibilité de Slider, nous avons choisi de le distribuer sous licence libre. Nous nous sommes tournés vers une Licence Apache, Version 2.0 [1].

Le code source est hébergé sur Github [2] à l'adresse suivante :

- `https://github.com/juleschevalier/slider`

Slider peut être utilisé à la fois comme librairie Java et comme programme indépendant grâce à l'interface en ligne de commande fournie. Pour l'installer, une commande Maven [3] suffit : `mvn install`.

D.1 Ontologies utilisées et exécutables

Les ontologies que nous avons utilisées dans nos expérimentations se divisent en deux catégories. La première comprend les ontologies générées, c'est-à-dire les ontologies BSBM ainsi que `subClassOf`. Nous comptons également dans cette catégorie les ontologies fragmentées utilisées pour les expérimentations sur les performances incrémentales du raisonneur. Les scripts permettant de générer ces ontologies sont disponibles en ligne aux adresses suivantes :

- BSBM : `https://gist.github.com/cgravier/8658389` ;
- subClassOf : `https://gist.github.com/juleschevalier/4bd3410cf14bd51e9811` ;

1. `http://www.apache.org/licenses/LICENSE-2.0`
2. `https://github.com/`
3. `https://maven.apache.org/`

- fragmentées : `https://gist.github.com/juleschevalier/954f69508945b398433f`.

La seconde catégorie d'ontologies contient les deux ontologies provenant du Web : wikipedia et Wornet. Toutes deux téléchargeables ici :

- `http://datasets-satin.telecom-st-etienne.fr/jchevalier/slider/`.

À ce même lien sont téléchargeables l'ensemble des ontologies générées et les deux exécutables Java, `slider.jar` et `slider-incremental.jar`, utilisés dans les expérimentations qui suivent.

D.2 Reproduction des expérimentations

Impact des paramètres

Afin de tester l'impact des paramètres sur le raisonneur, nous avons lancé, pour chaque taille de buffer et valeur de timeout, l'inférence sur les ontologies de la section précédente. Pour cela, nous avons utilisé le script disponible ici :

- `https://gist.github.com/juleschevalier/73de787aca10a3b5dded`

Les tailles de buffer et valeurs de timeout choisies sont équitablement réparties sur un axe logarithmique. Tous les résultats sont stockés dans le fichier CSV `parameters.csv`.

Dans le script fourni, la mémoire vive maximale allouée au processus java est fixée à 16Go.

Comparaison avec les références

Pour mesurer les performances de Slider, le plus simple est d'utiliser l'interface en ligne de commande utilisable grâce au fichier *slider.jar* fourni.

La machine virtuelle Java dispose d'un avantage notable face aux compilateurs traditionnels : sa capacité à optimiser à la volée le code qu'elle exécute. Pour cela, elle utilise les informations obtenues après l'exécution d'un morceau de code afin de l'optimiser pour la prochaine exécution de ce code. Il est donc nécessaire, pour tester les performances optimales d'un code Java, de lancer son exécution plusieurs fois de suite afin de permettre à la machine virtuelle d'optimiser ce code.

Comme nous l'avons indiqué dans la section 6.3, nous proposons de lancer dix fois l'inférence pour chaque ontologie testée, dont deux exécutions de *warm-up*, ou tours de chauffe.

Pour lancer l'inférence cinq fois pour RDFS sur un fichier `ontologie.nt` avec une taille de **buffer** de 1000 et un timeout de 10ms, il suffit de lancer la commande :

```
$ java -jar slider.jar -b 1000 -p rdfs -t 10 -v -i 5 ontologie.nt
```

Expérimentations incrémentales

Pour les tests incrémentaux, nous avons développé un client permettant de lancer tous les tests pour une série d'ontologies. Couplés au script permettant de découper les ontologies, ces deux outils sont suffisants pour obtenir les résultats de la section 6.4. Le script utilisé est disponible à cette adresse :

- `https://gist.github.com/juleschevalier/3f64409244c067830ec5`

La même commande que précédemment est utilisée. Slider lancera alors l'inférence sur toutes les ontologies du dossier désigné les unes après les autres de manière incrémentale, sans réinitialiser le **triplestore** après chaque ontologie.

Dans le script fourni, la mémoire vive maximale allouée au processus Java est fixée à 16Go.

E Publications

Les publications relatives à cette thèse sont énumérées ci-dessous :

Julien Subercaze, Christophe Gravier, Jules Chevalier et Frederique Laforest. « Inferray : fast in-memory RDF inference ». In : *VLDB*. T. 9. PVLDB. New Delhi, India, 2016

Jules Chevalier, Julien Subercaze, Christophe Gravier et Frédérique Laforest. « Slider : un Raisonneur Incrémental Évolutif ». In : *EGC Extraction et Gestion des Connaissances*. 2015

Jules Chevalier, Julien Subercaze, Christophe Gravier et Frédérique Laforest. « Incremental Reasoning on RDFS ». In : *IDA International Symposium on Intelligent Data Analysis*. Poster Session. 2015

Jules Chevalier, Julien Subercaze, Christophe Gravier et Frédérique Laforest. « Slider : An Efficient Incremental Reasoner ». In : *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, p. 1081–1086

Jules Chevalier. « A Linked Data Reasoner in the Cloud ». In : *The Semantic Web : Semantics and Big Data*. Springer, 2013, p. 722–726

Liste des acronymes

CPU	*Central Processing Unit*
GPU	*Graphics Processing Unit*
HDFS™	*Hadoop Distributed File System*
OWL	*Web Ontology Language*
RDF	*Resource Description Framework*
RDFS	*RDF Schema*
SSD	*Solid-State Drive*
URI	*Uniform Resource Identifier*
W3C	*World Wide Web Consortium, communauté internationale fondée en 1994, chargée de développer des standards ouverts afin d'assurer la compatibilité des différentes technologies*
XML	*Extensible Markup Language*

Bibliographie

Articles

[1] Franz Baader, Carsten Lutz et Boontawee Suntisrivaraporn. « Efficient Reasoning in EL+. » In : *Description Logics* 12 (2006) (cf. p. 44).

[2] Franz Baader et Ulrike Sattler. « An overview of tableau algorithms for description logics ». In : *Studia Logica* 69.1 (2001), p. 5–40 (cf. p. 20).

[3] Greg Bak. « Continuous classification : capturing dynamic relationships among information resources ». In : *Archival Science* 12.3 (2012), p. 287–318 (cf. p. 34).

[4] Rachid Benlamri et Luke Docksteader. « MORF : A mobile health-monitoring platform ». In : *IT professional* 3 (2010), p. 18–25 (cf. p. 33).

[5] Tim Berners-Lee, James Hendler, Ora Lassila et al. « The semantic web ». In : *Scientific american* 284.5 (2001), p. 28–37 (cf. p. 13).

[6] Barry Bishop et al. « OWLIM : A family of scalable semantic repositories. » In : *Semantic Web* 2.1 (2011), p. 33–42 (cf. p. 42, 122).

[7] Fernando Bobillo et Umberto Straccia. « Fuzzy ontology representation using OWL 2 ». In : *International Journal of Approximate Reasoning* 52.7 (2011), p. 1073–1094 (cf. p. 14).

[8] Stefano Ceri, Georg Gottlob et Letizia Tanca. « What you always wanted to know about Datalog (and never dared to ask) ». In : *IEEE Transactions on Knowledge and Data Engineering* 1.1 (1989), p. 146–166 (cf. p. 41, 44).

[9] Allan M Collins et Elizabeth F Loftus. « A spreading-activation theory of semantic processing. » In : *Psychological review* 82.6 (1975), p. 407 (cf. p. 12).

[10] Allan M Collins et M Ross Quillian. « Does category size affect categorization time ? » In : *Journal of verbal learning and verbal behavior* 9.4 (1970), p. 432–438 (cf. p. 12).

[11] Allan M Collins et M Ross Quillian. « Retrieval time from semantic memory ». In : *Journal of verbal learning and verbal behavior* 8.2 (1969), p. 240–247 (cf. p. 12).

[12] Jeffrey Dean et Sanjay Ghemawat. « MapReduce : simplified data processing on large clusters ». In : *Communications of the ACM* 51.1 (2008), p. 107–113 (cf. p. 37).

[13] Emanuele Della Valle et al. « It's a streaming world ! Reasoning upon rapidly changing information ». In : *IEEE Intelligent Systems* 6 (2009), p. 83–89 (cf. p. 14).

[14] Cory Doctorow. « Metacrap : Putting the torch to seven straw-men of the meta-utopia ». In : (2001) (cf. p. 15).

[15] Charles L Forgy. « Rete : A fast algorithm for the many pattern/many object pattern match problem ». In : *Artificial intelligence* 19.1 (1982), p. 17–37 (cf. p. 35).

[16] Bernardo Cuenca Grau et al. « Incremental classification of description logics ontologies ». In : *Journal of Automated Reasoning* 44.4 (2010), p. 337–369 (cf. p. 46, 47, 65).

[17] Christophe Gravier, Julien Subercaze et Antoine Zimmermann. « Conflict resolution when axioms are materialized in semantic-based smart environments. » In : *Journal of ambient intelligence and smart environments* to appear (2014) (cf. p. 50).

[18] Yuanbo Guo, Zhengxiang Pan et Jeff Heflin. « Lubm : A benchmark for owl knowledge base systems ». In : *Web Semantics : Science, Services and Agents on the World Wide Web* 3.2-3 (2011) (cf. p. 41, 49).

[19] Herman J ter Horst. « Completeness, decidability and complexity of entailment for RDF Schema and a semantic extension involving the OWL vocabulary ». In : *Web Semantics : Science, Services and Agents on the World Wide Web* 3.2 (2005), p. 79–115 (cf. p. 21, 37).

[20] Freddy Lecue, Spyros Kotoulas et Pól Mac Aonghusa. « Capturing the pulse of cities : A robust stream data reasoning approach ». In : *Position paper, IBM Research, Smarter Cities Technology Centre, Dublin, Ireland* (2011) (cf. p. 33).

[21] Chang-Shing Lee, Zhi-Wei Jian et Lin-Kai Huang. « A fuzzy ontology and its application to news summarization ». In : *IEEE Transactions on Systems, Man, and Cybernetics, Part B : Cybernetics* 35.5 (2005), p. 859–880 (cf. p. 14).

[22] Amit Sheth, Cory Henson et Satya S Sahoo. « Semantic sensor web ». In : *Internet Computing, IEEE* 12.4 (2008), p. 78–83 (cf. p. 33).

[23] John E Stone, David Gohara et Guochun Shi. « OpenCL : A parallel programming standard for heterogeneous computing systems ». In : *Computing in science & engineering* 12.1-3 (2010), p. 66–73 (cf. p. 36).

[24] Quan Thanh Tho et al. « Automatic fuzzy ontology generation for semantic web ». In : *IEEE Transactions on Knowledge and Data Engineering* 18.6 (2006), p. 842–856 (cf. p. 14).

[25] Emanuele Della Valle et al. « It's a streaming world ! Reasoning upon rapidly changing information ». In : *IEEE Intelligent Systems* 24.6 (2009), p. 83–89 (cf. p. 71).

Livres

[26] Serge Abiteboul, Richard Hull et Victor Vianu. *Foundations of databases*. T. 8. Addison-Wesley Reading, 1995 (cf. p. 41).

[27] Sören Auer et al. *Dbpedia : A nucleus for a web of open data*. Springer, 2007 (cf. p. 16).

[28] Franz Baader. *The description logic handbook : theory, implementation, and applications*. Cambridge university press, 2003 (cf. p. 17).

[29] Tim Berners-Lee, Mark Fischetti et Michael L Foreword By-Dertouzos. *Weaving the Web : The original design and ultimate destiny of the World Wide Web by its inventor*. HarperInformation, 2000 (cf. p. 13).

[30] Emanuele Della Valle et al. *A first step towards stream reasoning*. Springer, 2009 (cf. p. 33).

[31] Boontawee Suntisrivaraporn. *Module Extraction and Incremental Classification : A Pragmatic Approach for\ ensuremath {\ mathcal {EL}ˆ+} Ontologies*. Springer, 2008 (cf. p. 44, 45, 55).

In proceedings

[32] Daniel J Abadi et al. « Scalable semantic web data management using vertical partitioning ». In : *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment. 2007, p. 411–422 (cf. p. 77, 107).

[33] Davide Barbieri et al. « Stream reasoning : Where we got so far ». In : *Proceedings of the 4th workshop on new forms of reasoning for the Semantic Web : Scalable & dynamic*. 2010, p. 1–7 (cf. p. 33).

[34] Jeremy J Carroll et al. « Jena : implementing the semantic web recommendations ». In : *Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*. ACM. 2004, p. 74–83 (cf. p. 41, 122).

[35] Jules Chevalier et al. « Incremental Reasoning on RDFS ». In : *IDA International Symposium on Intelligent Data Analysis*. Poster Session. 2015 (cf. p. 165).

[36] Jules Chevalier et al. « Slider : An Efficient Incremental Reasoner ». In : *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM. 2015, p. 1081–1086 (cf. p. 104, 165).

[37] Jules Chevalier et al. « Slider : un Raisonneur Incrémental Évolutif ». In : *EGC Extraction et Gestion des Connaissances*. 2015 (cf. p. 165).

[38] Michael Compton et al. « A survey of the semantic specification of sensors ». In : *CEUR Workshop Proceedings*. 2009, p. 17–32 (cf. p. 33).

[39] Ashish Gupta, Inderpal Singh Mumick et Venkatramanan Siva Subrahmanian. « Maintaining views incrementally ». In : *ACM SIGMOD Record*. T. 22. 2. ACM. 1993, p. 157–166 (cf. p. 44).

[40] Yevgeny Kazakov, Markus Krötzsch et Frantisek Simancik. « ELK Reasoner : Architecture and Evaluation. » In : *ORE*. 2012 (cf. p. 47, 55, 56).

[41] Boris Motik et al. « Incremental Update of Datalog Materialisation : the Backward/Forward Algorithm ». In : *Proceedings of AAAI*. 2015 (cf. p. 49).

[42] Boris Motik et al. « Parallel materialisation of datalog programs in centralised, main-memory RDF systems ». In : *Proceedings of AAAI*. 2014, p. 129–137 (cf. p. 41–43, 52, 53, 55, 59, 122).

[43] Federica Paganelli et Dino Giuli. « An ontology-based context model for home health monitoring and alerting in chronic patient care networks ». In : *21st International Conference on Advanced Information Networking and Applications Workshops*. T. 2. IEEE. 2007, p. 838–845 (cf. p. 33).

[44] Martin Peters et al. « Rule-based Reasoning on Massively Parallel Hardware. » In : *SSWS@ ISWC*. 2013, p. 33–49 (cf. p. 36).

[45] Václav Snášel, Pavel Moravec et Jaroslav Pokornỳ. « WordNet ontology based model for web retrieval ». In : *Proceedings of International Workshop on Challenges in Web Information Retrieval and Integration*. IEEE. 2005, p. 220–225 (cf. p. 116).

[46] Ramakrishna Soma et Viktor K Prasanna. « A Data Partitioning Approach for Parallelizing Rule Based Inferencing for Materialized OWL Knowledge Bases. » In : *ISCA PDCCS*. 2008, p. 19–25 (cf. p. 41, 71).

[47] Ramakrishna Soma et Viktor K Prasanna. « Parallel inferencing for OWL knowledge bases ». In : *37th International Conference on Parallel Processing*. IEEE. 2008, p. 75–82 (cf. p. 41, 52, 53, 55, 59, 84).

[48] Kent A Spackman, Keith E Campbell et Roger A Côté. « SNOMED RT : a reference terminology for health care. » In : *Proceedings of the AMIA annual fall symposium*. American Medical Informatics Association. 1997, p. 640 (cf. p. 46).

[49] Julien Subercaze et al. « Inferray : fast in-memory RDF inference ». In : *VLDB*. T. 9. PVLDB. New Delhi, India, 2016 (cf. p. 165).

In collection

[50] Franz Baader, Carsten Lutz et Boontawee Suntisrivaraporn. « CEL—a polynomial-time reasoner for life science ontologies ». In : *Automated Reasoning*. Springer, 2006, p. 287–291 (cf. p. 46).

[51] Franz Baader et Ulrike Sattler. « Tableau algorithms for description logics ». In : *Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 2000, p. 1–18 (cf. p. 20).

[52] Mustapha Bourahla. « Exact Reasoning over Imprecise Ontologies ». In : *Computer Science and Its Applications*. Springer, 2015, p. 355–366 (cf. p. 14).

[53] Jeen Broekstra, Arjohn Kampman et Frank Van Harmelen. « Sesame : A generic architecture for storing and querying rdf and rdf schema ». In : *The Semantic Web—ISWC 2002*. Springer, 2002, p. 54–68 (cf. p. 122).

[54] Jules Chevalier. « A Linked Data Reasoner in the Cloud ». In : *The Semantic Web : Semantics and Big Data*. Springer, 2013, p. 722–726 (cf. p. 165).

[55] Norman Heino et Jeff Z Pan. « RDFS reasoning on massively parallel hardware ». In : *The Semantic Web–ISWC 2012*. Springer, 2012, p. 133–148 (cf. p. 20, 36, 51, 54, 55, 59, 71, 76).

[56] Yevgeny Kazakov et Pavel Klinov. « Incremental reasoning in OWL EL without bookkeeping ». In : *The Semantic Web–ISWC 2013*. Springer, 2013, p. 232–247 (cf. p. 20, 47, 48, 59).

[57] Freddy Lécué, Anika Schumann et Marco Luca Sbodio. « Applying semantic web technologies for diagnosing road traffic congestions ». In : *The Semantic Web–ISWC 2012*. Springer, 2012, p. 114–130 (cf. p. 33).

[58] Freddy Lécué et al. « Predicting severity of road traffic congestion using semantic web technologies ». In : *The Semantic Web : Trends and Challenges*. Springer, 2014, p. 611–627 (cf. p. 33).

[59] Carsten Lutz, Frank Wolter et Michael Zakharyaschev. « A tableau algorithm for reasoning about concepts and similarity ». In : *Automated Reasoning with Analytic Tableaux and Related Methods*. Springer, 2003, p. 134–149 (cf. p. 20).

[60] Sergio Munoz, Jorge Pérez et Claudio Gutierrez. « Minimal deductive systems for RDF ». In : *The Semantic Web : Research and Applications*. Springer, 2007, p. 53–67 (cf. p. 21, 27).

[61] Martin Peters et al. « Scaling Parallel Rule-Based Reasoning ». In : *The Semantic Web : Trends and Challenges*. Springer, 2014, p. 270–285 (cf. p. 36).

[62] Martin Peters, Sabine Sachweh et Albert Zündorf. « Large Scale Rule-Based Reasoning Using a Laptop ». In : *The Semantic Web. Latest Advances and New Domains*. Springer, 2015, p. 104–118 (cf. p. 20, 37, 52, 54–56, 59).

[63] Matthew Richardson, Rakesh Agrawal et Pedro Domingos. « Trust management for the semantic web ». In : *The Semantic Web-ISWC 2003*. Springer, 2003, p. 351–368 (cf. p. 15).

[64] Anne Schlicht et Heiner Stuckenschmidt. « MapResolve ». In : *Web Reasoning and Rule Systems*. Springer, 2011, p. 294–299 (cf. p. 20, 40, 52, 55, 59, 71).

[65] Jacopo Urbani et al. « OWL reasoning with WebPIE : calculating the closure of 100 billion triples ». In : *The Semantic Web : Research and Applications*. Springer, 2010, p. 213–227 (cf. p. 38, 40, 51, 55, 59, 71, 116, 122).

[66] Jacopo Urbani et al. « Dynamite : Parallel materialization of dynamic RDF data ». In : *The Semantic Web–ISWC 2013*. Springer, 2013, p. 657–672 (cf. p. 20, 48, 49, 55, 59).

[67] Raphael Volz, Steffen Staab et Boris Motik. « Incrementally maintaining materializations of ontologies stored in logic databases ». In : *Journal on Data Semantics II*. Springer, 2005, p. 1–34 (cf. p. 20, 44, 45, 55, 59).

[68] Kamin Whitehouse, Feng Zhao et Jie Liu. « Semantic streams : A framework for composable semantic interpretation of sensor data ». In : *Wireless Sensor Networks*. Springer, 2006, p. 5–20 (cf. p. 33).

Autres sources

[69] *Apache™ Hadoop®*. URL : `https://hadoop.apache.org/` (cf. p. 37).

[70] Tim Berners-Lee. *Linked data, 2006*. 2006 (cf. p. 13).

[71] Christian Bizer et Andreas Schultz. *The berlin sparql benchmark*. 2009 (cf. p. 116).

[72] Dan Brickley et R.V. Guha. *RDF Schema 1.1*. 2014. URL : http://www.w3.org/TR/rdf-schema/ (cf. p. 21).

[73] Dan Brickley et Libby Miller. *FOAF vocabulary specification 0.98*. 2012 (cf. p. 16).

[74] Fabien Gandon et Guus Schreiber. *RDF 1.1 XML Syntax*. 2014. URL : http://www.w3.org/TR/rdf-syntax-grammar/ (cf. p. 19).

[75] RDF Working Group. *Resource Description Framework (RDF)*. 2014. URL : http://www.w3.org/RDF/ (cf. p. 18).

[76] James Manyika et al. *Big data : The next frontier for innovation, competition, and productivity*. 2011 (cf. p. 14).

[77] Deborah L. McGuinness et Frank van Harmelen. *OWL Web Ontology Language Overview*. 2004. URL : http://www.w3.org/TR/owl-features/ (cf. p. 20, 21, 72, 78).

[78] *The Apache Software Foundation*. URL : https://www.apache.org/ (cf. p. 37).

[79] Jacopo Urbani, Eyal Oren et Frank van Harmelen. « RDFS/OWL reasoning using the MapReduce framework ». Mém.de mast. Vrije Universiteit, 2009 (cf. p. 37, 39, 51, 52, 76, 148).

Résumé de thèse

Le web sémantique offre les outils permettant de formaliser des connaissances à partir de données brutes et de lever des connaissances implicites grâce à des algorithmes de raisonnement. Les solutions de raisonnement actuelles souffrent de deux limitations principales : d'une part elles peinent à passer à l'échelle pour les ontologies les plus importantes et, d'autre part, les solutions de raisonnement par lots, offrant les meilleures performances, ne permettent pas de raisonner sur des flux de données.

Nous proposons dans cette thèse une architecture pour le raisonnement incrémental sur des flux de triples. Afin de passer à l'échelle, elle est conçue sous la forme de modules indépendants, permettant l'exécution parallèle du raisonnement. Plusieurs instances d'une même règle peuvent être exécutées simultanément afin d'améliorer les performances. Nous avons également concentré nos efforts pour limiter la dispersion des doublons dans le système, ce qui est un problème récurrent du raisonnement. Pour cela, un triplestore partagé permet à chaque module de filtrer au plus tôt les doublons. Il utilise le partitionnement vertical afin d'optimiser l'ajout et la recherche de triples et interdit, de par sa structure, le stockage de doublons. La structure de notre architecture, organisée en modules indépendants par lesquels transitent les triples, lui permet de recevoir en entrée des flux de triples. Enfin, notre architecture est indépendante du fragment utilisé. Grâce à l'indépendance des modules qui la composent, il est possible d'utiliser tout type de règle d'inférence.

Nous présentons également trois modes d'inférence pour notre architecture. Le premier consiste à inférer l'ensemble des connaissances implicites le plus rapidement possible. Les deux autres modes permettent de prioriser quelles connaissances seront inférées en priorité. Le second mode est utilisable dans le cas où les connaissances à prioriser sont connues par avance. Le troisième propose une priorisation visant à maximiser la quantité de triples inférés par seconde.

Nous avons implémenté l'architecture présentée à travers *Slider*, un raisonneur incrémental prenant nativement en charge les fragments ρdf et RDFS. Il peut être facilement étendu à des fragments plus complexes. Nos expérimentations ont montré une amélioration des performances de plus de 65% par rapport au raisonneur OWLIM-SE. Le raisonneur RDFox, récemment publié, offre cependant de meilleurs performance. Nous avons également mené des tests montrant que l'utilisation du raisonnement incrémental avec Slider apporte un avantage systématique aux performances par rapport au raisonnement par lots, quels que soient l'ontologie utilisée et le fragment appliqué.

Le code source de Slider et les informations permettant de reproduire les expérimentations de cette thèse sont disponibles sur github, à cette adresse : `http://juleschevalier.github.io/slider/`.