Swip : une interface Langue Naturelle à SPARQL programmée en SPARQL

Camille Pradel, Ollivier Haemmerlé, Nathalie Hernandez

Swip : une interface Langue Naturelle à SPARQL programmée en SPARQL

Camille Pradel, Ollivier Haemmerlé, Nathalie Hernandez

IRIT, Université de Toulouse le Mirail, Département de Mathématiques-Informatique, 5 allées Antonio Machado, F-31058 Toulouse Cedex 9
{camille.pradel,ollivier.haemmerle,nathalie.hernandez}@univ-tlse2.fr

Résumé : L'approche *Swip* a pour objectif de traduire en SPARQL des requêtes exprimées en langue naturelle en exploitant des patrons de requêtes préalablement définis. Nous présentons ici le module au cœur du système implémentant cette approche qui repose entièrement sur SPARQL. Les traitements mis en œuvre au sein de ce module sont en effet entièrement réalisés sur une base de triplets RDF par l'intermédiaire de requêtes de mise à jour SPARQL. L'implémentation bénéficie ainsi des capacités du moteur SPARQL employé, ce qui permet d'éviter de mettre en place des fonctions de manipulation et d'appariement de graphes, un moteur SPARQL étant justement conçu et optimisé pour ces tâches.

Mots-clés : SPARQL, appariement, application web.

L'approche *Swip* [1] que nous proposons veut fournir aux utilisateurs finals un moyen d'interroger des bases de connaissances sous forme de graphes à l'aide de requêtes exprimées en langue naturelle, et ce dans le but d'éviter à ces utilisateurs de se confronter à la complexité de la formulation d'une requête graphe dans un langage tel que SPARQL.

D'autres travaux visent à générer automatiquement – ou semi-automatiquement – des requêtes formelles à partir de requêtes exprimées sous forme de mots-clés ou de phrases en langue naturelle. L'utilisateur exprime son besoin en information de façon intuitive, sans avoir à connaître le langage de requêtes ou bien le formalisme de représentation de connaissances utilisé dans le système. Certains travaux ont proposé de traduire des requêtes de haut niveau en requêtes formelles dans différents langages comme SeREQL (Lei *et al.*, 2006) ou SPARQL (Zhou *et al.*, 2007; Cabrio *et al.*, 2012). Dans ces systèmes, la génération de requêtes nécessite les étapes suivantes : (i) appariement des mots de la requête aux entités sémantiques de la base de connaissances ; (ii) construction de graphes requêtes liant les entités détectées à l'étape précédente ; (iii) classement des requêtes construites, (iv) sélection de la bonne requête par l'utilisateur. Les approches se sont pour l'instant focalisées sur des problèmes précis : optimiser l'étape d'appariement en exploitant des ressources externes comme Wordnet ou Wikipedia (Lei *et al.*, 2006; Wang *et al.*, 2008; Cabrio *et al.*, 2012), optimiser l'indexation et l'exploration de la base de connaissances pour la construction de la requête graphe (Zhou *et al.*, 2007), améliorer le classement des requêtes candidates (Wang *et al.*, 2008), optimiser l'identification de relations à l'aide de patrons textuels (Cabrio *et al.*, 2012), ou encore mettre en œuvre un dialogue avec l'utilisateur pour raffiner l'interprétation de la requête utilisateur (Lehmann & Bühmann, 2011; Unger *et al.*, 2012).

L'originalité de notre approche réside aussi bien dans la démarche mise en place pour interpréter la requête que dans son implémentation faisant une utilisation poussée de SPARQL.

1. http://swip.univ-tlse2.fr/SwipWebClient/welcome.html

La section 1 présente les originalités de l'approche et de son déploiment. La section 2 détaille les premières étapes du processus afin de donner une idée précise de son implémentation. La section 3 discute des avantages et inconvénients de cette approche.

1 Cadre

Nous définissons ici le cadre requis pour la mise en œuvre de notre approche. Nous présentons en 1.1 les originalités de l'approche, en 1.2 les ontologies que nous avons conçues afin de fournir un cadre logique à l'implémentation et en 1.3 les quelques moyens logistiques nécessaires.

1.1 Originalités

Une des originalités de *Swip* se caractérise par l'utilisation de *patrons de requêtes* préétablis pour guider le processus d'interprétation. Nos travaux se fondent en effet sur le postulat selon lequel, dans les applications réelles, les requêtes formulées par les utilisateurs sont pour l'essentiel des variations autour de quelques familles typiques de requêtes. Chaque patron de requêtes représente une de ces familles de requêtes. Les patrons de requêtes sont constitués d'un graphe représentant le besoin en informations couvert par le patron et faisant référence à des ressources de la base de connaissances cible, et d'un modèle de phrase descriptive permettant de générer des phrases en langue naturelle présentées à l'utilisateur.

Le processus d'interprétation de la requête utilisateur est décomposé en deux étapes principales, avec un résultat intermédiaire qui est la *requête pivot*. La requête pivot consiste en une première interprétation de la requête utilisateur dans laquelle le besoin en information est exprimé sous une forme proche d'une requête par mots-clés, mais dans laquelle il est possible d'exprimer des relations entre les mots-clés (Pradel *et al.*, 2011). Par exemple, la requête pivot `?"person": "produce"= "In Utero". "In Utero": "album"` est obtenue lors de l'interprétation de la requête en langue naturelle "Who produced the album In Utero ?" (tiré du jeu d'entraînement de la compétition QALD-3 [2]). Cette organisation présente deux avantages principaux : elle permet de représenter les informations importantes issues de l'analyse syntaxique de la requête utilisateur, et elle facilite la mise en œuvre du multilinguisme dans notre approche. En effet, la requête pivot est un format intermédiaire indépendant de la langue, et peut donc être traitée de la même façon quel que soit le langage employé par l'utilisateur. Ainsi, pour adapter notre approche à un nouveau langage, il suffit de modifier la première étape.

Cette première grande étape consiste en l'interprétation de la requête utilisateur et sa traduction en requête pivot. Pour cela, des opérations classiques de traitement automatique des langues (identification des entités nommées, détermination des catégories grammaticales, analyse de dépendances) sont appliquées à la requête en langue naturelle, puis des règles de transformation préétablies sont utilisées pour générer la requête pivot à partir de l'arbre de dépendances de la phrase. Ce premier module est architecturé de façon classique pour une application web : des services web effectuent les sous-tâches basiques du processus d'interprétation et sont exploités par un *workflow*, lui-même accessible sous forme de service web. Chaque service exploite en

2. http://greententacle.techfak.uni-bielefeld.de/~cunger/qald/index.php?x=task1&q=3

entrée le résultat du ou des services situés en amont dans le *workflow*. Ce premier module et l'ensemble des services web qui le composent ont été présentés dans (Pradel *et al.*, 2013b,a).

Dans la seconde étape, les patrons de requêtes sont associés à la requête pivot pour obtenir une liste d'interprétations possibles de cette requête (et donc de la requête utilisateur d'origine) qui sont ensuite ordonnées en fonction de leurs pertinences supposées. Les interprétations peuvent ainsi être soumises à l'utilisateur sous forme de phrases descriptives générées à partir des modèles de phrases descriptives de chaque patron. L'implémentation de ce module est originale et présente selon nous une réelle nouveauté. En effet, dans cette implémentation, les traitements permettant les appariements et associations décrits dans (Pradel *et al.*, 2011) sont entièrement réalisés sur une base de triplets RDF par l'intermédiaire de requêtes de mise à jour SPARQL. Cette approche permet d'éviter l'implémentation de fonctions de manipulation et d'appariement de graphes et d'exploiter les capacités d'un moteur SPARQL qui est justement conçu et optimisé pour apparier des graphes. Ainsi, cette tâche est entièrement réalisée via des requêtes SPARQL. Le résultat de chaque étape est systématiquement enregistré dans la base de connaissances, via les fonctionnalités de mise à jour de SPARQL (*SPARQL updates*), ce qui les rend directement exploitables dans l'étape suivante. A notre connaissance, aucune approche ne repose sur une telle exploitation de SPARQL. Ce second module est présenté dans le reste de l'article.

1.2 Ontologies de patrons et de requêtes

Les triplets manipulés et générés lors du processus d'interprétation exploitent le vocabulaire de deux ontologies construites par nos soins et accessibles depuis la page d'accueil du système *Swip* [3].

L'ontologie *patterns*, présentée dans (Pradel *et al.*, 2012), permet de modéliser des patrons de requête et facilite ainsi la gestion, le partage et l'évolution de ces patrons. Les données RDF représentant les patrons de requêtes au format spécifié par cette ontologie trouvent ici une nouvelle utilité étant donné que, une fois insérées dans la base de triplets, elles sont exploitées telles quelles pour interpréter la requête utilisateur.

L'ontologie *queries* permet quant à elle de modéliser des requêtes pivot et définit les structures communes pour représenter les résultats intermédiaires et finals du processus d'interprétation. Nous ne la détaillons pas ici, étant donné qu'elle a été conçue selon les mêmes principes que l'ontologie *patterns* et se révèle plus simple que cette dernière. Comme nous l'expliquons dans (Pradel *et al.*, 2013a), une requête pivot est un ensemble de sous-requêtes, chaque sous-requête étant un 1/2/3-uplet (ensemble de un, deux ou trois éléments) de requête. Les classes et propriétés de l'ontologie *queries* reflètent logiquement cette structure ; leurs noms sont explicites, et un exemple de requête pivot et une partie des résultats de son interprétation exprimés en RDF et fondés sur cette ontologie est montré plus bas.

Cette ontologie intègre également des axiomes permettant l'inférence de nouveaux triplets. Cependant, ces possibilités d'inférence ne sont ici pas exploitées pour des raisons de performances : contrairement à la description des patrons qui est statique (mis à part dans les cas exceptionnels et ponctuels d'évolution ou d'ajout de patrons), les triplets générés au cours de l'interprétation évoluent en continu, et il n'est donc pas possible d'inférer les connaissances

3. http://swip.univ-tlse2.fr/SwipWebClient/welcome.html

liées à l'ontologie *queries* en pré-traitement. Or, il est très coûteux d'activer un raisonneur sur un jeu de données contenu dans un serveur SPARQL, particulièrement lorsque ce jeu de données fait l'objet de nombreuses mises à jour. C'est pourquoi dans notre implémentation, l'ensemble des triplets exploités dans la suite du processus sont générés explicitement à chaque étape et ce, même s'ils sont redondants et pourraient être obtenus à partir d'autres triplets et des connaissances ontologiques.

1.3 Infrastructure

Le processus d'interprétation est intégralement fondé sur des requêtes SPARQL. La principale infrastructure nécessaire à la mise en œuvre de cette approche est donc un moteur SPARQL qui recevra les requêtes de mises à jour émises par l'interface employée par l'utilisateur. Les triplets générés par chaque requête, exploitant le vocabulaire de l'ontologie *Queries*, seront ajoutés à la base de triplets de ce moteur. Les expérimentations que nous avons menées exploitent *Fuseki* [4], un serveur SPARQL intégrant le moteur de requêtes *ARQ* [5].

Pour mener à bien le processus d'interprétation, il est évidemment nécessaire d'avoir accès au jeu de donnés ciblé par la requête utilisateur et aux patrons de requêtes (préalablement traduits en RDF selon le vocabulaire défini dans l'ontologie *Patterns*). Ces éléments peuvent être directement inclus dans la base de triplets du serveur SPARQL ou simplement, grâce aux fonctionnalités de fédération de SPARQL 1.1, distribués sur le web de données à partir du moment où ils sont accessibles via SPARQL. La figure 1 illustre un scénario possible de déploiement.

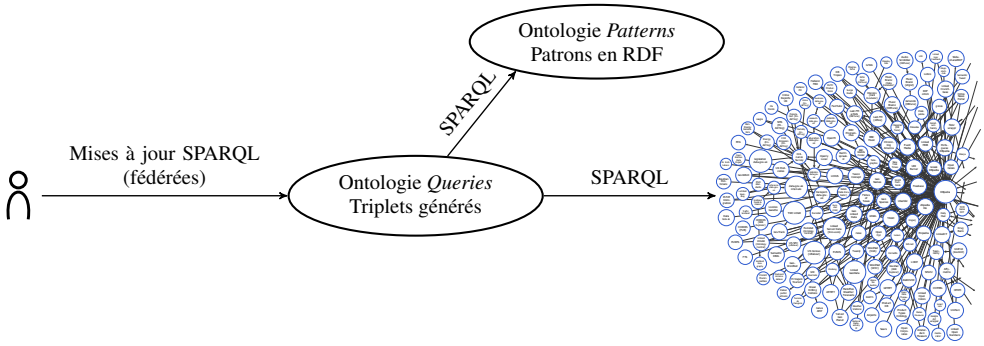

FIGURE 1 – Exemple de déploiement du système *Swip*.

2 Description des premières étapes

Cette section détaille les premières étapes du processus d'interprétation de la requête pivot. Par souci de concision, les étapes suivantes sont présentées plus grossièrement dans la sous-section 2.4. La figure 2 montre toutes les étapes de cette implémentation, chaque étape correspondant à une requête `UPDATE` ou `ASK` (qui permet d'émuler une boucle, comme expliqué plus bas en 2.4) ; ces requêtes sont données sur la page web de présentation de l'approche *Swip* [6].

4. http://jena.apache.org/documentation/serving_data/
5. http://jena.apache.org/documentation/query/
6. http://swip.univ-tlse2.fr/SwipWebClient/welcome.html

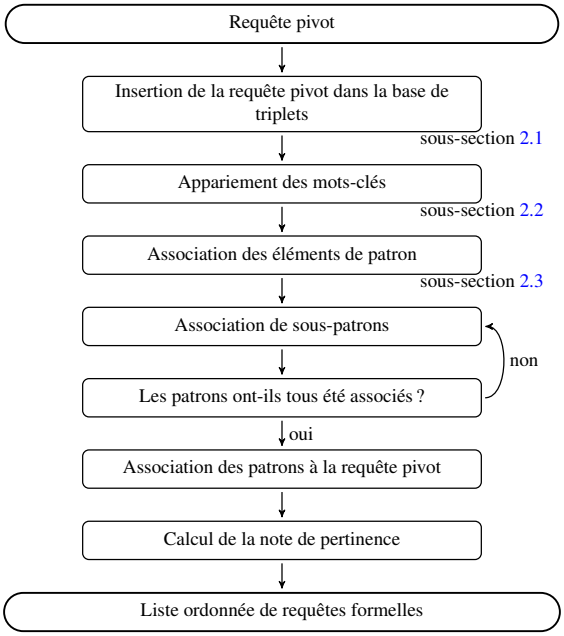

FIGURE 2 – Détail des étapes de formalisation de la requête pivot.

2.1 Insertion de la requête pivot dans la base de connaissances

Dans un premier temps, le système génère un URI qui est unique pour chaque ensemble de requêtes pivot équivalentes. Si cet URI existe déjà dans la base de connaissances, la requête précédente a déjà été traitée et les résultats sauvegardés peuvent être exploités tels quels. Sinon, le système génère un graphe RDF (fondé sur l'ontologie *queries* introduite plus haut) représentant la requête pivot et l'insère au moyen d'une requête de mise à jour SPARQL (`INSERT DATA`) dans la base de connaissances avant de commencer l'interprétation de la requête. Le graphe RDF produit pour la requête `?"person": "produce"= "In Utero". "In Utero": "album"` (qui est la requête pivot issue de la requête "Who produced the album In Utero ?") est montré dans la partie gauche de la figure 3. Cette figure montre les données RDF initiales ainsi que les résultats des mises à jour SPARQL qui sont effectuées au cours de l'étape d'appariement (cf. sous-section 2.2). Les ressources RDF sont représentées dans des nœuds ovales, les littéraux dans des rectangles, et les propriétés sont logiquement matérialisées par des arcs étiquetés. Pour des besoins de lisibilité, les types des littéraux ne sont pas montrés et les classes auxquelles appartiennent les ressources ne sont montrées que lorsque cela aide à la compréhension ; l'URI de la classe d'un nœud est alors indiqué en-dessous de ce nœud.

2.2 Appariement des éléments de la requête à la base de connaissances

La première étape de l'interprétation de la requête pivot consiste à apparier chaque élément de la requête pivot aux entités de la base de connaissances (classes, propriétés et instances) ou aux types de littéraux, en associant à chaque appariement une note de confiance qui représente la qualité supposée de l'appariement et sa vraisemblance.

L'*appariement des mots-clés* est effectué en calculant une mesure de similarité entre les

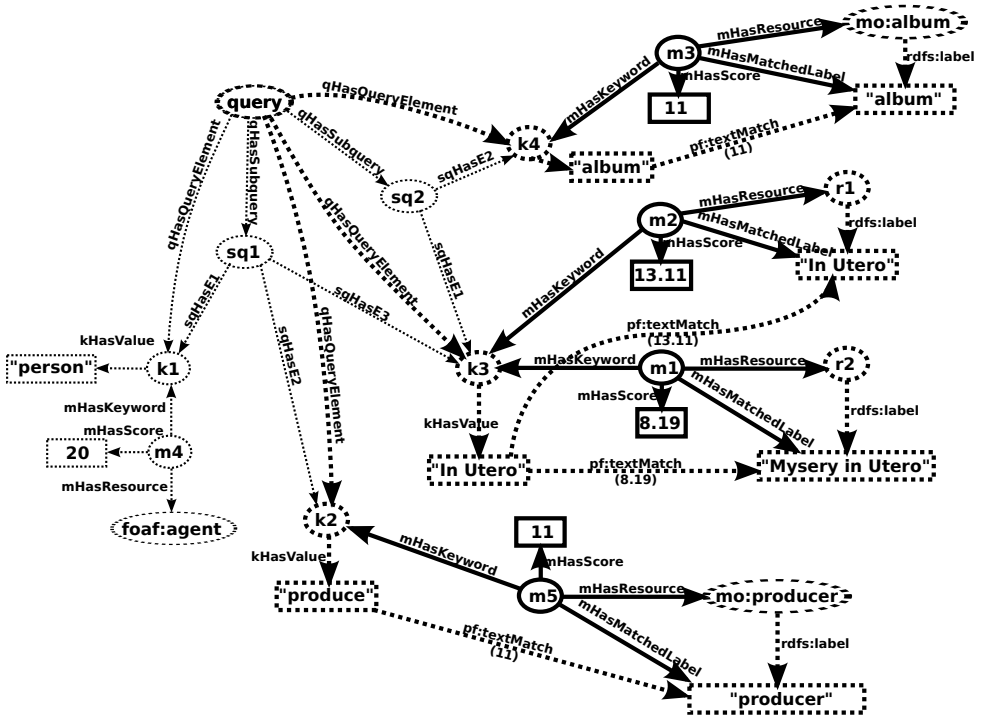

FIGURE 3 – L'étape d'appariement sur notre requête pivot exemple.

mots-clés de la requête pivot et les étiquettes des ressources de la base de connaissances. Pour cela, nous utilisons *LARQ* [7], une extension de SPARQL proposée par le moteur SPARQL *ARQ* qui permet d'exploiter les capacités d'indexation et de recherche dans du texte du moteur de recherche *Apache Lucene* [8]. Cette extension introduit un nouvel élément de syntaxe qui permet de déterminer les littéraux ressemblant à une chaîne de caractères donnée et une valeur représentant le niveau de ressemblance, appelée *score Lucene* : le "triplet" `(?lit ?score) pf:textMatch '+text'` lie à la variable `?lit` tous les littéraux qui sont similaires à la chaîne de caractères `text` et à la variable `?score` le *score Lucene* correspondant. La requête SPARQL intégrale utilisée pour mettre en œuvre cette étape d'appariement est montrée dans la figure 4.

La figure 3 montre un sous-ensemble des appariements obtenus par l'exécution de cette requête et les instances d'appariements (classe `queries:Matching`) générées. Le graphe avant l'exécution de la requête est tracé en pointillés ; la partie de ce graphe qui est appariée au motif de graphe de la clause `WHERE` est en gras, et les ressources et triplets insérés dans la base de connaissances sont en traits pleins. Les figures suivantes utilisent les mêmes conventions graphiques.

Il est important de noter que, bien que cette étape telle que nous l'avons décrite soit mise en œuvre à l'aide d'une extension (non standard) de SPARQL qui la rend peu portable, une alternative peut facilement être implémentée en utilisant des fonctions standard de SPARQL telles que les fonctions de chaînes de caractères `REGEX` et `CONTAINS`, ou encore une simple comparai-

7. http://jena.apache.org/documentation/larq/
8. http://lucene.apache.org/

```
INSERT
{
  ?matchUri a queries:Matching;
            queries:matchingHasKeyword ?keyword;
            queries:matchingHasResource ?r;
            queries:matchingHasScore ?score;
            queries:matchingHasMatchedLabel ?l.
  ?keyword queries:keywordAlreadyMatched "true"^^xsd:boolean.
}
WHERE
{
  <[queryUri]> queries:queryHasQueryElement ?keyword.
  ?keyword a queries:KeywordQueryElement;
           queries:queryElementHasValue ?keywordValue.
  FILTER NOT EXISTS { ?keyword queries:keywordAlreadyMatched "true"^^xsd:boolean. }
  (?l ?score) pf:textMatch (?keywordValue 6.0 5).
  ?r rdfs:label ?l.
  BIND (UUID() AS ?matchUri)
}
```

FIGURE 4 – Mise à jour SPARQL utilisée pour apparier les mots-clés de la requête pivot.

son de chaînes de caractères. Cette version standard présenterait néanmoins des performances dégradées, l'appariement entre chaînes de caractères étant établi de façon exacte.

2.3 Association des éléments de patron aux éléments de requête

Avant de pouvoir associer l'intégralité des patrons à la requête pivot, une première tâche consiste à déterminer pour chaque élément de patron toutes les associations possibles aux éléments de la requête utilisateur et leur note de confiance respective. Ces associations sont appelées *associations d'élément*.

Cette étape consiste à créer une association entre un élément de requête et un élément de patron quand cet élément de requête a été apparié à une ressource qui est liée d'une façon ou d'une autre à l'élément de patron ciblé. Nous définissons plusieurs cas permettant d'établir un lien entre une ressource appariée r et un élément de patron ciblé t :

1. r est une sous-classe de t (ce cas comprend celui où r est la classe t elle-même),
2. r est une sous-propriété de t (ce cas comprend celui où r est la propriété t elle-même),
3. r est une instance de t (ce cas comprend celui des associations d'élément instance-classe, présenté plus bas),
4. r fait référence au même type de littéral que t.

À chaque association d'élément est assignée une note de confiance qui est la même que celle de l'appariement impliqué. La figure 5 montre l'instanciation de quelques appariements via une requête de mise à jour SPARQL. L'élément de patron `cd_info_element5` qui cible la classe `mo:Record` est associé deux fois au mot-clé `k1` (`"In Utero"`) qui a été préalablement apparié à deux instances de cette même classe (cas 3). L'élément de patron `cd_info_element4` qui cible la propriété `mo:producer` est associé au mot-clé `k2` (`"produce"`) qui a été préalablement apparié à cette même propriété (cas 2).

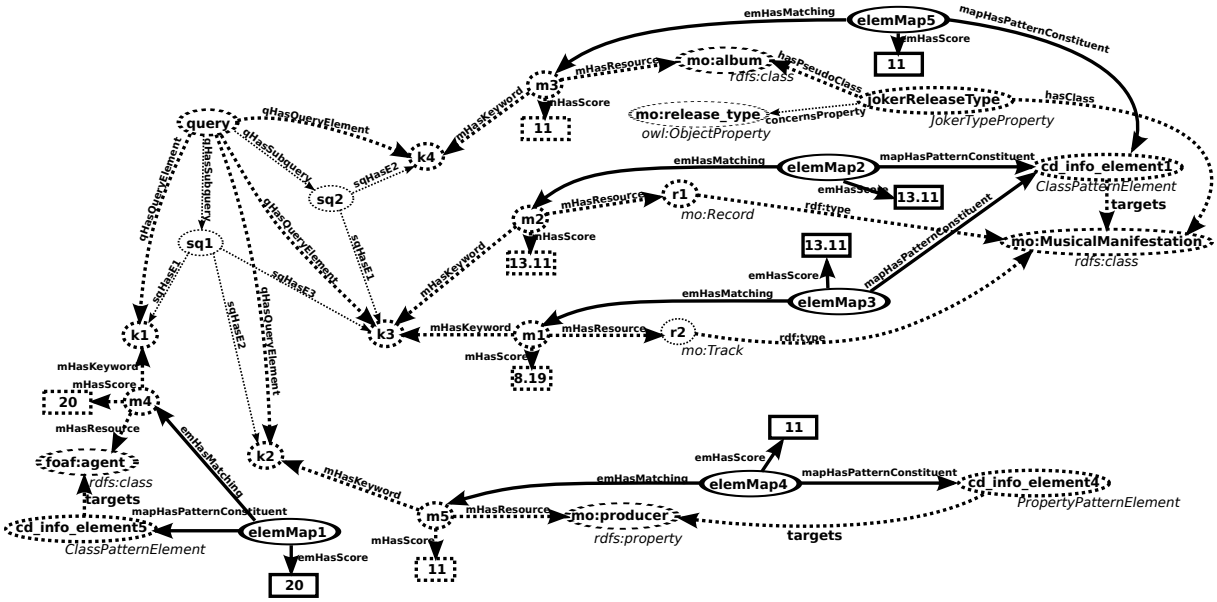

FIGURE 5 – L'étape d'association d'élément sur notre requête pivot exemple.

2.3.1 Le cas de propriétés "*Type"

L'instanciation de l'association d'élément `elemMap5` est issue d'une extension du premier cas établi plus haut. Cette extension est due à l'observation d'un choix de modélisation récurrent fait par certains développeurs d'ontologies, qui consiste à classifier les instances d'une classe c en définissant une propriété d'objet avec pour domaine c, une classe (énumérée) c' qui représente le co-domaine de cette propriété, et des instances de c' qui représentent les différentes façons de classifier les instances de c. Par exemple, dans l'ontologie *music* (Raimond *et al.*, 2007), les instances de la classe `mo:Record` peuvent être impliquées en tant que sujet dans un triplet avec pour prédicat `mo:releaseType` et pour objet une instance de la classe (`mo:Album`, `mo:Single`, `mo:Live`, `mo:Soundtrack`...). Il nous semble que ce choix, bien qu'il ait probablement été guidé par certaines exigences, n'est pas pertinent car il ignore le mécanisme de classification proposé par RDFS et OWL (à savoir le typage d'instances à l'aide de classes) pour exprimer des connaissances qui sont en fait bel et bien une classification. Nous appuyons notre critique en avançant deux éléments que nous considérons comme symptomatiques de ce travers et que l'on retrouve dans notre exemple :

– deux termes qui ont le même statut dans une phrase en langue naturelle (comme par exemple les termes "songs" et "soundtracks" qui sont utilisés de la même façon dans les requêtes "Give me all songs by Aretha Franklin" et "Give me all soundtracks composed by John Williams," toutes deux extraites du jeu de données de la compétition QALD-3) sont modélisés de manière différente dans l'ontologie, le premier par une classe, le second par une instance de la classe `mo:ReleaseType` ;
– la façon dont les developpeurs de l'ontologie eux-mêmes ont nommé la propriété d'objet mise en cause trahit la nature réelle de cette propriété ; en effet, une propriété dont le nom finit par `Type` sera très probablement utilisée pour typer des instances et devrait donc en tant que telle être une sous-propriété de `rdf:type`, or ce n'est pas le cas ; cette seconde

observation est à l'origine du nom que nous avons donné à ce type de propriétés : les propriétés "*Type" (*wildcard type properties* en anglais, d'où l'URI de la classe utilisée plus bas).

Notre approche permet de pallier ce manque de généricité en identifiant explicitement ces cas. Dans la figure 5, l'instance `wildcardReleaseType` de type `WildcardTypeProperty` exprime le fait que la ressource `mo:album` doit être considérée durant le processus d'association comme une sous-classe de `mo:Record`, ce qui permet l'association de l'élément qualifiant `cd_info_element1` au mot-clé `k4` ; elle spécifie aussi que la propriété de typage correspondante (qui devra être utilisée dans la requête SPARQL finale) est dans ce cas `mo:release_type` au lieu du classique `rdf:type`.

2.3.2 Les associations d'élément instance-classe

Un dernier type d'association d'élément peut être produit sur la base des précédents. Ces associations, appelées *associations d'élément instance-classe* sont issues de l'observation de la manière dont les utilisateurs, quand ils s'expriment en langue naturelle, spécifient souvent un terme faisant référence à une instance par un autre terme faisant référence à une classe à laquelle appartient cette instance. On trouve de nombreux exemples dans les requêtes en langue naturelle de la compétition QALD-3 : "the band Dover", "the album In Utero", "the song Hardcore Kids"…

D'après (Pradel *et al.*, 2013a), ce type de formulation se traduit en langage pivot par une sous-requête binaire, composée du mot-clé faisant référence à l'instance qualifié par le mot-clé faisant référence à la classe ; par exemple, "the album In Utero" devient `"In Utero":"album"`. Nous utilisons un type particulier d'association d'élément pour prendre en compte ce cas ; une association de ce type, appelée *association d'élément instance-classe*, associe un élément de patron à deux mots-clés. Sa valeur de confiance est égale à la somme des valeurs de confiance des deux appariements pris en compte.

L'implémentation que nous proposons gère de façon simple ce cas particulier. La figure 6 poursuit le même exemple et illustre l'instanciation d'une association d'élément instance-classe. `elemMap1` et `elemMap4` instanciées préalablement associent `cd_info_element1` respectivement à `k3` (`"In Utero"`) et `k4` (`"album"`), et la ressource appariée par `k3` est une instance (lorsque l'on considère la remarque précédente sur les propriétés "*Type") de la ressource appariée par `k4` ; cela permet d'instancier une nouvelle association d'élément `elemMap6` associant l'élément de patron considéré aux deux éléments de requête. Son score est la somme des notes de confiance des deux associations d'élément originellement impliquées.

2.4 Étapes suivantes

Nous ne détaillons pas autant les étapes suivantes du processus d'interprétation par souci de brièveté. Encore une fois, les requêtes SPARQL utilisées et l'ontologie organisant les triplets générée sont disponibles sur la page d'accueil du système *Swip*.

Comme nous l'avons présenté dans (Pradel *et al.*, 2011), l'étape d'appariement des sous-patrons implique des opérations complexes, comme des combinaisons d'éléments d'un ensemble ou des produits cartésiens entre des ensembles dont le nombre ne peut être déterminé à l'avance. De plus, l'association d'un patron nécessite de commencer par associer les sous-

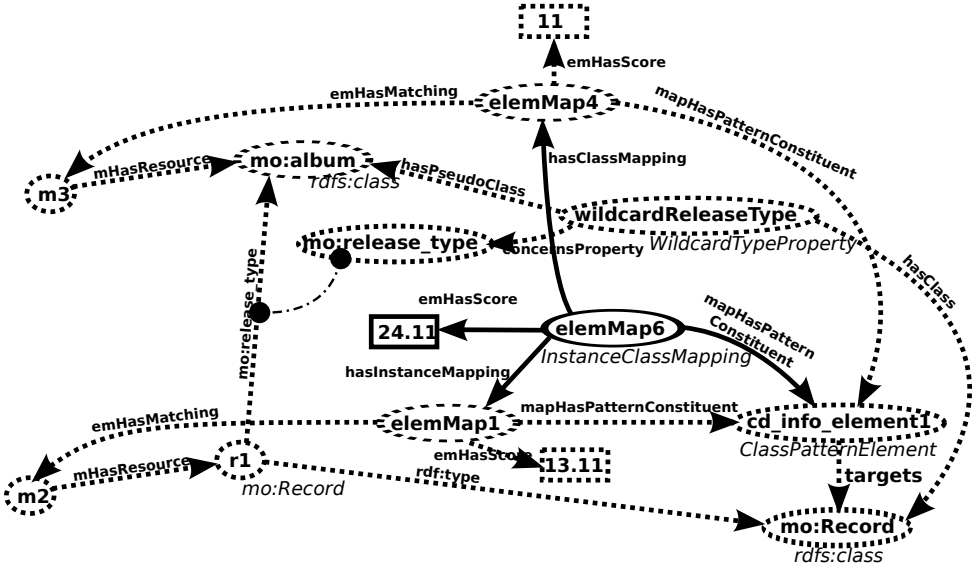

FIGURE 6 – Une association d'élément instance-classe sur notre requête pivot exemple.

patrons les plus simples qui ne contiennent aucun autre sous-patron, puis les sous-patrons qui les contiennent directement, et ainsi de suite jusqu'à avoir associé le patron lui-même. Ces besoins sont traditionnellement résolus en programmation impérative par une structure de contrôle de type *boucle* (*tant que*), elle-même permise par une fonctionnalité de saut conditionnel (*si*).

Une simple succession de mises à jour SPARQL ne peut répondre à ces besoins. Nous avons donc dû ajouter la possibilité d'effectuer un saut conditionnel dans notre implémentation. Ce saut conditionnel est mis en œuvre de manière très simple au travers d'une requête SPARQL `ASK` : deux embranchements sont possibles à l'issue de l'exécution d'une telle requête et celui réellement emprunté dépend du résultat de cette exécution. Comme on peut le voir sur la figure 2, cette méthode est utilisée à la fin de l'étape d'association des sous-patrons, au niveau de la requête (`ASK`) "*Les patrons ont-ils tous été associés ?*" Si la réponse est non (`FALSE`), l'embranchement suivi renvoit à l'exécution de requêtes en amont dans le processus pour revenir au test plus tard. Si la réponse est oui (`TRUE`), alors on passe aux étapes suivantes. Ainsi, une boucle est créée permettant d'assurer que tous les sous-patrons ont été associés avant de passer à la suite.

3 Critique de l'implémentation

L'architecture décrite dans cet article présente pour nous de nombreux avantages. L'un des plus évidents est la facilité d'utilisation d'un système de cache. En effet, étant donné que le résultat de chaque étape de traitement est inséré dans l'entrepôt RDF, il est alors très simple de réexploiter les données générées précédemment. Par exemple, comme expliqué dans la sous-section 2.1, le système Swip se rend compte qu'une requête entrante a déjà été traitée lorsque son URI est déjà présent dans la base de triplets et le résultat de son interprétation peut être directement retourné. De même, l'appariement d'un mot-clé donné n'est réalisé qu'une seule fois pour l'ensemble des requêtes contenant ce même mot-clé.

De plus, cette architecture permet un asynchronisme total et naturel entre le client et le serveur : une fois que l'URI de la requête est construit, il est retourné au client qui peut alors mettre à jour les résultats intermédiaires de l'interprétation en cours progressivement et indépendamment du serveur. Pour cela, il requête directement à l'aide de SPARQL l'entrepôt RDF où sont enregistrés ces résultats.

Cette approche est également homogène et cohérente dans la mesure où les données d'entrée, les données intermédiaires et les données de sortie sont enregistrées et manipulées via des standards. Même la configuration du système et les ajustements sont réalisés de cette manière ; par exemple, les cas identifiés de propriétés "*Type" (présentés plus haut) et certains appariements utiles (c'est-à-dire des appariements de mots-clés au niveau de l'ontologie qui ne pourraient pas être obtenus par mesure de similarité entre chaînes de caractères, comme par exemple entre le mot-clé `"husband"` et la propriété `rel:spouseOf`) sont directement exprimés en RDF, insérés dans la base de triplets et exploités tels quels au cours du processus d'interprétation.

Nous attirons également l'attention du lecteur sur le fait que, bien que la première grande étape du processus d'interprétation (traduction de la langue naturelle vers le langage pivot) ait été implémentée d'une façon plus "traditionnelle," en utilisant des services web, cela pourrait maintenant être fait différemment en exploitant la récente initiative *NLP2RDF* (Hellmann *et al.*, 2013) qui veut fournir un format commun pour les données de sortie des outils de traitement automatique du langage les plus populaires, ce format appelé *NIF (NLP Interchange Format)* étant totalement intégré au cadre du web sémantique.

Enfin, cette architecture permet de déployer le système *Swip* de façon distribuée, ce qui facilite le passage à grande échelle. En effet, pour fonctionner, *Swip* a simplement besoin d'avoir accès via SPARQL aux données à interroger et aux patrons de requêtes. Grâce aux récentes fonctionnalités de fédération de SPARQL (*federated SPARQL* (Prud'hommeaux & Buil-Aranda, 2013)), les données peuvent être regroupées sur un seul serveur SPARQL ou réparties sur plusieurs, et les patrons peuvent eux aussi être regroupés ou répartis, rassemblés avec la base de connaissances qu'ils concernent ou isolés. Cette architecture exploitant des standards est donc très souple et permet de nombreuses variations. De plus, ce sont les serveurs SPARQL qui effectuent les traitements, ce qui rend le programme initial (celui qui émet le requêtes SPARQL) très léger. On peut ainsi imaginer que ce programme, exécuté côté client, met en branle, via la fédération de SPARQL, de nombreux serveurs répartis sur le web et les orchestre pour traiter l'interprétation d'une requête.

Nous retenons également deux inconvénients majeurs qui viennent ternir le tableau. Le premier est le manque de contrôle sur l'exécution des requêtes SPARQL et par conséquent sur les performances générales du système. Le serveur SPARQL est utilisé comme une boîte noire et son efficacité influence directement celle du processus d'interprétation. L'expérience a montré que le serveur ARQ n'est pas performant pour traiter de grosses requêtes (plus de vingt triplets répartis dans plusieurs sous-requêtes) et que la division de ces requêtes en une série équivalente de requêtes successives permet une nette amélioration des performances.

De plus, SPARQL est encore une recommandation relativement jeune qui, malgré les nouvelles fonctionnalités apportées par SPARQL 1.1, propose un panel de fonctions assez limité. Par exemple, il y a très peu de fonctions arithmétiques ; seules les plus basiques sont supportées, ce qui n'est pas le cas de la fonction puissance. En conséquence, une solution dégradée a dû être trouvée pour le calcul de la note de pertinence finale qui impliquait initialement cette fonction.

4 Perspectives

Nous voulons explorer les nouvelles pistes ouvertes par l'implémentation décrite dans ce papier. Nous pensons en effet que l'approche que nous avons utilisée peut être généralisée et utilisée pour d'autres applications : il serait ainsi possible d'implémenter tous types d'algorithmes, et en particulier des algorithmes manipulant des graphes. De plus, pour les raisons exposées en 3, cette approche semble parfaitement adaptée au développement d'applications web et pourrait très bien s'adapter aux cadres proposés visant à intégrer les API web au web de données, comme *RDF-REST* (Champin, 2013).

Nous voudrions donc formaliser cette nouvelle forme de programmation exploitant les mises à jour SPARQL et la comparer à des paradigmes proches, comme la programmation dirigée par les données (*data-driven programming*), ou des approches plus pratiques exploitant des bases de données, comme PL/SQL. Ces travaux nous mèneraient certainement à proposer une extension à SPARQL permettant le saut conditionnel dans le but de faire de ce langage un langage de programmation particulièrement adapté à l'implémentation d'algorithmes impliquant des graphes.

Références

CABRIO E., COJAN J., APROSIO A. P., MAGNINI B., LAVELLI A. & GANDON F. (2012). QAKiS : an open domain QA system based on relational patterns. In *International Semantic Web Conference (Posters & Demos)*, volume 914.

CHAMPIN P.-A. (2013). RDF-REST : a unifying framework for web APIs and linked data.

HELLMANN S., LEHMANN J., AUER S. & BRÜMMER M. (2013). Integrating NLP using linked data.

LEHMANN J. & BÜHMANN L. (2011). AutoSPARQL : let users query your knowledge base. In *The Semantic Web : Research and Applications*, p. 63–79. Springer.

LEI Y., UREN V. & MOTTA E. (2006). Semsearch : A search engine for the semantic web. In *Managing Knowledge in a World of Networks*, p. 238–245. Springer.

PRADEL C., HAEMMERLÉ O. & HERNANDEZ N. (2011). A semantic web interface using patterns : the SWIP system. In *Proceedings of GKR 2011*, p. 172–187, Barcelona, Spain : Croitoru et al.

PRADEL C., HAEMMERLÉ O. & HERNANDEZ N. (2012). Des patrons modulaires de requêtes SPARQL dans le système SWIP. In *23es Journées Francophones d'Ingénierie des Connaissances*, p. 621–636.

PRADEL C., HAEMMERLÉ O. & HERNANDEZ N. (2013a). Natural language query interpretation into SPARQL using patterns. In *COLD@ISWC2013*, Sydney (Australia).

PRADEL C., HAEMMERLÉ O. & HERNANDEZ N. (2013b). SWIP at QALD-3 : results, criticisms and lesson learned. Valencia, Spain.

PRUD'HOMMEAUX E. & BUIL-ARANDA C. (2013). SPARQL 1.1 federated query.

RAIMOND Y., ABDALLAH S. A., SANDLER M. B. & GIASSON F. (2007). The music ontology. In *ISMIR*, p. 417–422.

UNGER C., BÜHMANN L., LEHMANN J., NGONGA NGOMO A.-C., GERBER D. & CIMIANO P. (2012). Template-based question answering over RDF data. In *Proceedings of the 21st international conference on World Wide Web*, p. 639–648.

WANG H., ZHANG K., LIU Q., TRAN T. & YU Y. (2008). Q2semantic : A lightweight keyword interface to semantic search. In *The Semantic Web : Research and Applications*, p. 584–598. Springer.

ZHOU Q., WANG C., XIONG M., WANG H. & YU Y. (2007). SPARK : adapting keyword query to semantic search. In *The Semantic Web*, p. 694–707. Springer.