fr2sql : Interrogation de bases de données en français

Benoît Couderc[1] Jérémy Ferrero[2, 3]
(1) Aix Marseille Université, Marseille, France
(2) Compilatio, 276 rue du Mont Blanc, 74540 Saint-Félix, France
(3) LIG-GETALP, Université Grenoble Alpes, France
benoit.couderc@etu.univ-amu.fr, jeremy.ferrero@imag.fr

Résumé. Les bases de données sont de plus en plus courantes et prennent de plus en plus d'ampleur au sein des applications et sites Web actuels. Elles sont souvent amenées à être utilisées par des personnes n'ayant pas une grande compétence en la matière et ne connaissant pas rigoureusement leur structure. C'est pour cette raison que des traducteurs du langage naturel aux requêtes SQL sont développés. Malheureusement, la plupart de ces traducteurs se cantonnent à une seule base du fait de la spécificité de l'architecture de celle-ci. Dans cet article, nous proposons une méthode visant à pouvoir interroger n'importe quelle base de données à partir de questions en français. Nous évaluons notre application sur deux bases à la structure différente et nous montrons également qu'elle supporte plus d'opérations que la plupart des autres traducteurs.

Abstract.

fr2sql : database query in French

Databases are increasingly common and are becoming increasingly important in actual applications and Web sites. They often used by people who do not have great competence in this domain and who do not know exactly their structure. This is why translators from natural language to SQL queries are developed. Unfortunately, most of these translators is confined to a single database due to the specificity of the base architecture. In this paper, we propose a method to query any database from french. We evaluate our application on two different databases and we also show that it supports more operations than most other translators.

Mots-clés : Base de données (BdD), Requêtes SQL, Traducteur français SQL, Interface Homme-BdD.

Keywords: Databases (DB), Structured Query Language (SQL), French to SQL translator, Natural language interfaces to databases (NLIDB).

1 Introduction

Depuis plusieurs années, les bases de données (BdD) deviennent inévitables pour tous les sites Web ou applications gérant d'importantes masses d'informations, comme des comptes utilisateurs (banques, agences de transport, réseaux sociaux, jeux vidéo, etc.). Internet s'est peu à peu démocratisé et vulgarisé mais les bases de données quant à elles restent encore abstraites pour beaucoup de personnes. Certains postes ne requérant aucune formation en informatique ou en administration de données nécessitent tout de même de travailler étroitement avec des bases de données, comme en comptabilité ou en secrétariat par exemple. C'est dans le but qu'une personne, n'ayant aucune compétence dans le domaine de gestion des BdD, puisse non pas directement en administrer une, mais tout du moins comprendre son fonctionnement, interagir avec elle et effectuer dessus des tâches simples (interrogation, ajout, suppression), que les traducteurs du langage naturel au langage d'interrogation de bases de données ont vu le jour.

Depuis une cinquantaine d'années (Green *et al.*, 1961), le problème d'interrogation d'une base de données en langage naturel est récurrent et fait l'objet de nombreuses recherches. La majorité des outils développés sont très performants mais ne sont malheureusement pour la plupart compatibles qu'avec une seule langue naturelle source et/ou une seule base de données cible. Ils sont développés uniquement dans le but d'être l'interface d'une base de données et sont donc exclusivement compatibles avec celle-ci. En raison de structure, vocabulaire et convention de nommage extrêmement différents d'une base à l'autre, le portage d'un outil non compatible multi-bases sur une base différente de celle prévue

initialement est difficile et le rendrait de toute façon inefficace. C'est en partant de ce constat que cet article a pour objectif de concevoir un traducteur permettant l'interrogation de n'importe quelle base de données à partir du français.

Après avoir défini quelques notions et présenté l'état de l'art, on décrira comment extraire les informations relatives à une base de données cible afin d'en connaître sa structure et son vocabulaire, pour ensuite croiser ces informations avec les mots clefs de la question posée, et ainsi générer en sortie la requête SQL équivalente la plus probable. *La requête sera donc générée en fonction de la présence, du nombre et de l'ordre des mots clefs identifiés dans la phrase entrée par l'utilisateur.* Pour finir, on présentera l'évaluation de notre approche, en comparant les capacités de notre application à celles des applications déjà existantes et en évaluant ses performances sur un jeu de requêtes tests.

2 La traduction du langage naturel aux requêtes SQL

2.1 Les requêtes SQL

Une base de données est un dispositif informatique où est enregistré un ensemble d'informations. Dans une base de données relationnelle, les informations sont stockées sous forme de matrices, appelées tables. Une base de données relationnelle peut comporter une ou plusieurs tables, reliées ou non entre elles. Les entrées (informations) sont réunies dans ce que l'on appelle des colonnes (ou des champs). Un groupe de colonnes ayant trait à une même entité (objet) forme une table.

Un schéma, aussi appelé modèle de données est un diagramme (comme par exemple la figure 4) ou un descriptif textuel décrivant la distribution et l'organisation des données au sein d'une base. Il renseigne les caractéristiques de chaque type de données et les relations entre elles. Un schéma relationnel est le moyen le plus répandu de décrire une base de données relationnelle.

Le SQL (Structured Query Language) est un langage normalisé permettant d'effectuer des opérations (interrogations, modifications, suppression, etc.) sur les bases de données relationnelle.

L'objectif de cet article est de proposer une application permettant d'interroger une base de données relationnelle au moyen d'une question en français. Nous traitons donc seulement l'interrogation d'une base qui se fait à l'aide de la commande SELECT, dont la syntaxe est la suivante :

SELECT column_list
FROM table_list
[JOIN jointure_expression]
[WHERE conditional_expression]
[GROUP BY group_by_column_list]
[HAVING conditional_expression]
[ORDER BY order_by_column_list]

La syntaxe d'une commande SELECT est toujours construite de la même façon. Après le mot clef *SELECT* on énumère la liste des colonnes, qui contiennent les informations que l'on souhaite récupérer. Après le *FROM* on indique dans quelle(s) table(s) se trouvent ces informations. Toutes les lignes suivantes, marquées entre crochets, sont facultatives. Elle permettent respectivement et dans l'ordre, de préciser des tables supplémentaires si des jointures sont nécessaires, d'ajouter des contraintes à l'interrogation et de regrouper ou ordonner les valeurs de retour.

En sachant cela, lorsque l'utilisateur entre une demande du type :

Quel est l'**âge** des **élèves** qui ont pour **prénom Jean** ?

L'application doit exécuter une requête comme celle-ci :

SELECT **age** FROM **eleve** WHERE **prenom** = '**JEAN**'

Le passage de cette première phrase à la seconde représente toute la problématique du projet.

On constate, ci-dessus en gras, des *éléments clefs* communs entre la demande entrée et la requête à produire. C'est sur la correspondance entre ces éléments que la plupart des outils, y compris le nôtre, se positionnent.

2.2 État de l'art

Le lecteur est invité à se reporter à (Androutsopoulos *et al.*, 1995) et (Cimiano & Minock, 2009) pour un état de l'art complet.

L'un des premiers problèmes que soulève l'interrogation d'une base de données par un utilisateur n'ayant aucune connaissance dans ce domaine est qu'il ne connaît ni la structure ni le vocabulaire employé au sein de la base qu'il cherche à interroger. Les solutions les plus triviales consistent soit à limiter le vocabulaire qu'il peut utiliser, soit à employer une grammaire stricte, qui à l'aide de règles, limitera les phrases qu'il pourra construire. Les travaux de (Rao *et al.*, 2010) suivent la première méthode, ils contraignent l'utilisateur à rentrer une question formatée selon un dictionnaire de mots bien précis connus par l'application, se limitant donc à une base en particulier. Les applications françaises MONDE-2000 (Pasero, 1997) et DISQUE (Pasero & Sabatier, 1998) privilégient la seconde méthode. Ces dernières fonctionnent seulement sur une base de données prédéfinie, leur base de données respective, car elles possèdent le vocabulaire et l'ensemble des règles de grammaire adéquates à leur fonctionnement seulement sur ces bases ci. Le problème de cette catégorie de méthodes, d'après l'étude de (Androutsopoulos *et al.*, 1995), est qu'elles ne conviennent pas à l'utilisateur, qui se sent alors « piégé » . C'est pour cela que des méthodes moins contraignantes sont également employées. Les travaux de (Popescu *et al.*, 2003) qui, n'utilisant pas de dictionnaire de synonymes, obligent l'utilisateur à entrer une phrase libre mais contenant un lexique de façon exact ou racinalisé par rapport à celui de la BdD, sont déjà plus permissifs. Néanmoins, cette méthode oblige encore l'utilisateur à avoir une connaissance parfaite de la structure de la base, et plus particulièrement de ses noms de colonnes et tables. Des recherches plus récentes règlent ce problème en croisant les mots clefs de la base et ceux de la phrase entrée par l'utilisateur à un dictionnaire de mots spécifiques à la base (Deshpande & Devale, 2012) ou de façon plus générale à un dictionnaire de synonymes (Chaudhari, 2013).

Chandra (Chandra, 2006) rapporte également des problèmes de linguistique et d'ambiguïté. Il constate que le vocabulaire employé dans la question de l'utilisateur n'est souvent pas le même que dans la base de données et cela entraîne des problèmes de correspondances. Les travaux de (Mohite & Bhojane, 2014) mettent en avant le même phénomène, qu'ils appellent *le problème d'épellation* (spelling mistake). Il s'agit du fait que si l'utilisateur se trompe sur l'orthographe d'un mot clef, mot représentant une table ou une colonne par exemple, dans la demande entrée, il fausse tout le système empêchant une correspondance d'être trouvée. C'est donc pour cette raison que les systèmes utilisant des dictionnaires de synonymes (Chaudhari, 2013) ou des aspects sémantiques (Djahantighi *et al.*, 2008) sont de plus en plus nombreux. *Les ressources lexicales externes jouent un rôle essentiel pour la portabilité des traducteurs.*

Chaudhari (Chaudhari, 2013) développe un traducteur relativement simple mais déjà plus ambivalent. Il se contente d'identifier le type de demande (select ou delete), de transformer les nombres écrits en lettres en chiffres, d'enlever les apostrophes et la ponctuation, d'extraire les mots clefs et de construire la requête en conséquence. Il utilise un dictionnaire de synonymes à compléter à la main pour élargir le vocabulaire accepté par son système. L'inconvénient de cette méthode est qu'à chaque fois que l'on souhaite porter cet outil sur une nouvelle base, il faut effectuer des entrées manuelles dans le dictionnaire.

Pour régler définitivement les problèmes dus à la restriction de vocabulaire d'une base, certaines recherches font intervenir, en plus de la gestion de la synonymie, un aspect sémantique (Djahantighi *et al.*, 2008) afin de trouver des équivalences de sens. Le plus souvent par le biais de méthodes d'apprentissage afin d'établir des correspondances entre une question et une réponse attendue (Giordani & Moschitti, 2012) ou une question et une requête (Giordani & Moschitti, 2009).

Pour la génération des requêtes aussi plusieurs techniques font référence. Certains utilisent des grammaires avec des règles prédéfinies (Alexander *et al.*, 2013), d'autres des grammaires probabilistes (Deshpande & Devale, 2012) ou des automates (Kaur *et al.*, 2013). Certains même utilisent un système d'apprentissage (Giordani & Moschitti, 2009; Minock, 2010; Giordani & Moschitti, 2012). Ils exploitent ensuite la structure de la BdD pour générer des requêtes SQL candidates qui sont ordonnées de la plus plausible à la moins plausible grâce à un SVM-ranker basé sur un système de noyau d'arbres (Giordani & Moschitti, 2012).

Le défaut de la plupart de ces méthodes (Rao *et al.*, 2010; Pasero, 1997; Pasero & Sabatier, 1998) est qu'elles sont efficaces seulement sur une base de données particulière, celle pour laquelle leur grammaire ou leur dictionnaire est fourni, comme le travail de (Safari & Patrick, 2014) qui est uniquement fonctionnel sur sa base de données gérant une clinique, ou les recherches de (Chen, 2014) qui gère une base de données géographique, ou encore l'application de (Alexander *et al.*, 2013) qui dispose uniquement des tableaux des règles d'équivalences et des clefs primaires et liaisons de la base sur laquelle elle travaille.

Côté industriel en revanche, on remarque une plus grande portabilité multi-bases. L'approche commune dans la plupart

des produits opérationnels vise à offrir une interface de création associée à une grammaire sémantique (Minock, 2010) qui interprète les requêtes des utilisateurs sur leur base de données tel que dans l'outil EnglishQuery (Microsoft, 2000) (décrit dans (Popescu *et al.*, 2003)). English2SQL (Hurricane Electric, 2012) quant à lui, analyse la phrase et détermine sa signification à l'aide d'un algorithme non divulgué qui ne tient pas compte d'une grammaire. Cela permet, à la fois de déterminer le sens des phrases qui sont longues et complexes, et de traiter également certaines particularités, comme le problème des *contraintes muettes*. Ce problème récurrent dans les outils procédant par recherche de correspondances entre la phrase entrée par l'utilisateur et les entités de la BdD se rencontre lorsque le nom de la colonne sur laquelle doit s'effectuer la contrainte n'est pas énoncé dans la phrase. La phrase suivante illustre ce problème.

Quel est l'**âge** des **élèves** s'appellant **Jean** ?

Il est alors plus complexe d'opérer une correspondance afin de savoir dans quelle colonne chercher le mot *Jean*, à moins de fouiller toutes les colonnes de la table *eleve* à la recherche d'une valeur *Jean*, ce que fait l'outil English2SQL qui considère que chaque mot de la phrase entrée par l'utilisateur qui n'est pas un nom de table ou de colonne, peut être une valeur de colonne.

Pour finir, on peut citer des travaux issus de la communauté des bases de données (Pound *et al.*, 2010; Patil & Chen, 2012) qui tentent de rapprocher l'interrogation par mots clés de l'interrogation en langue naturelle.

Nous nous positionnons donc de façon originale vis à vis de l'état de l'art pour les raisons suivantes :

— le langage naturel source est le français ;
— la méthode présentée est portable (opérationnelle instantanément sur n'importe quelle base de données SQL) ;
— la méthode présentée possède une grammaire suffisamment permissive pour que l'utilisateur ne ressente aucune restriction lexicale ou syntaxique.

3 Notre approche

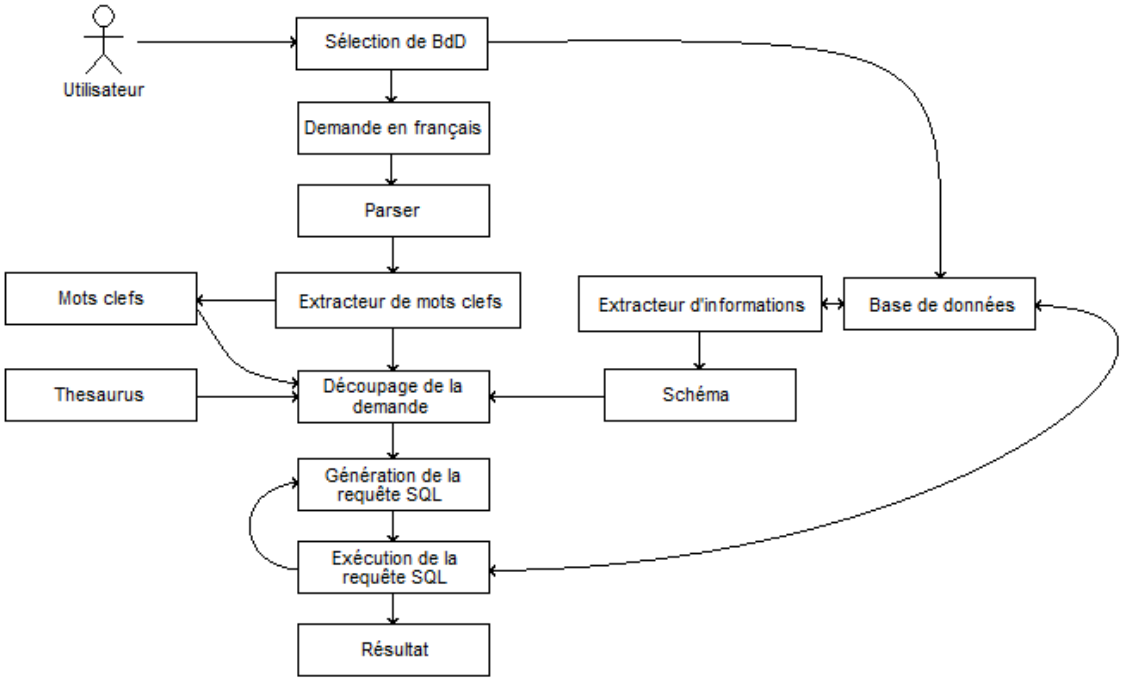

FIGURE 1 – Schéma représentant l'architecture synthétisée du fonctionnement du système fr2sql.

La figure 1 représente le fonctionnement global de l'outil fr2sql. Dans un premier temps, l'utilisateur sélectionne la base de données qu'il souhaite interroger et rentre une demande dans le champ prévu à cet effet. L'application va alors extraire

les mots porteurs de sens de la demande entrée par l'utilisateur (section 3.1). Ensuite, elle va récupérer la structure et les informations nécessaires à son usage dans la base de données qui vient d'être sélectionnée (section 3.2). À l'aide d'un dictionnaire de synonymes (section 3.3), une correspondance est recherchée entre les mots clefs extraits de la demande utilisateur et les entités de la base. Un découpage de la demande entrée est effectué en fonction des correspondances trouvées (section 3.4). Une seconde recherche est opérée afin de trouver des mots pouvant indiquer des opérations supplémentaires lors de la sélection (section 3.5). Une fois que l'application a déterminé quel type de requête était demandée et sur quels éléments, en fonction du découpage opéré et des correspondances trouvées, elle la génère (section 3.6). La requête ainsi générée est ensuite exécutée sur la base de données, si le résultat de l'interrogation ne renvoie aucune erreur (renvoyer une erreur est différent de ne renvoyer aucun résultat), il est affiché à l'utilisateur.

3.1 Extraction de mots porteurs de sens

En premier lieu, l'idée est de récupérer seulement les mots porteurs de sens dans la phrase entrée par l'utilisateur. En effet, il est important de pouvoir par la suite opérer une correspondance (un matching) entre certains concepts entrés par l'utilisateur et des éléments de la base de données. Pour cela on conserve donc plus particulièrement les noms communs, pouvant être le nom d'une table ou d'une colonne, mais aussi les noms propres, chiffres, adjectifs, etc. pouvant s'agir quant à eux d'une valeur de colonne recherchée.

Pour faire ceci, on utilise l'outil TreeTagger (Schmid, 1994) afin de filtrer les mots vides en fonction de leur classe grammaticale (prépositions, pronoms, déterminants, etc.) et d'effectuer une racinisation (stemming) des mots restant. En considérant la phrase :

Quel est l'**âge** des **élèves** qui ont pour **prénom Jean** ?

Le filtre doit retourner les éléments : « *âge, élève, prénom, Jean* ». L'ordre des mots est conservé et a son importance lors des étapes suivantes.

3.2 Récupération de l'architecture de la base de données

La seconde étape du processus consiste à récupérer l'architecture (la structure) de la base de données sur laquelle on va vouloir effectuer les requêtes et ce afin d'en connaître les entités (colonnes, tables, clefs primaires et secondaires, etc.) afin de permettre dans un second temps une correspondance avec les mots extraits de la demande utilisateur lors de la section 3.1.

Deux méthodes ont été implémentées pour parvenir à ce résultat. La première méthode consiste à collecter les informations nécessaires en interrogeant la base de données à l'aide de requêtes SQL du type « *SHOW TABLES, SHOW COLUMNS, DESCRIBE, etc.* ».

La seconde méthode, quant à elle, consiste à analyser le fichier de sauvegarde ou de création de la base de données. Avec cette méthode, une connexion en pré-processing à la base de données n'est plus requis mais un schéma SQL universel est nécessaire (certaines commandes étant syntaxiquement différentes sous MySQL ou Oracle par exemple).

À noter que fr2sql est donc seulement compatible avec une base de données SQL.

3.3 Couplage à un thésaurus

Comme le souligne l'étude de (Mohite & Bhojane, 2014), si l'utilisateur n'entre pas une phrase correctement orthographiée ou dont le vocabulaire employé n'est pas rigoureusement le même que celui de la base de données, aucune correspondance entre sa phrase et des éléments de la base ne sera trouvée et aucun résultat pertinent ne sera retourné. Il est donc important de maximiser le nombre de mots qui donneront lieu à une correspondance pertinente entre un mot clef de la demande entrée et un élément de la BdD. Pour ce faire, parallèlement au processus évoqué dans les sections précédentes, un dictionnaire de synonymes est chargé. Pour chaque mot, on a donc accès à un tableau de concepts contenant tous les mots de la langue par lesquels il peut être substitué. Par exemple, les mots « élèves » et « étudiants » représentent le même concept. Un concept est une idée, un sens représenté par un mot ou un groupe de mots. Dès lors, un concept est représenté par un mot porteur de sens lexical ainsi que tous ses mots de substitution possibles contenus dans son tableau. Le but de ce traducteur est de rendre accessible l'interrogation d'une base de données à une personne n'en connaissant ni la structure ni les mots clefs (noms de tables et de colonnes) et étant donc susceptible d'utiliser un synonyme d'un mot

employé dans la base à la place du mot lui même. Il est dès lors plus judicieux de représenter un mot par un concept, un tableau de tous les mots par lesquels il peut être substitué (un tableau de ses synonymes, lui compris) plutôt que seulement par lui-même. De cette façon pour interroger la table *eleve*, l'utilisateur pourra rentrer le mot *étudiant*.

La table 1 représente une partie des mots de substitution correspondant aux mots porteurs de sens extraits sur la phrase exemple lors de la section 3.1.

Mots porteur de sens	Mots de substitution
âge	ancienneté, ère, période, génération, ...
élève	écolier, étudiant, apprenti, collégien, lycéen, ...
prénom	nom de baptême, surnom, ...
Jean	-

TABLE 1 – Tableau d'une partie des mots de substitution disponibles pour la phrase étudiée.

Lors de cette étude, le dictionnaire utilisé est le thesaurus v.2.3 en date du 20 décembre 2011 de LibreOffice v.3.4. Cette ressource se trouve en accès libre sur internet.

Dans un souci de permettre toutes nomenclatures de nommage des colonnes et tables des bases de données, une interface d'administration du dictionnaire des synonymes a également été développée, permettant ainsi à n'importe quel utilisateur sans connaissance particulière d'ajouter, supprimer ou modifier à sa guise les synonymes de chacun des mots. Ainsi si la table regroupant les informations des étudiants a par exemple pour nom *ETUD_UNIV_01*, et qu'aucun synonyme n'a donc été automatiquement ajouté à ce nom de table, l'utilisateur pourra rentrer manuellement le mot *étudiant* comme synonyme et l'équivalence se mettra automatiquement à jour en ajoutant également tous les synonymes du mot *étudiant* contenu dans le dictionnaire des synonymes.

3.4 Découpage de la demande

À ce stade là du processus, chaque mot clef de la demande entrée par l'utilisateur est donc extrait. L'application a également à sa disposition une liste de synonymes pour chacun de ces mots clefs. L'idée est maintenant de rechercher une correspondance entre les mots clefs de la demande (ou leurs synonymes) et les entités de la base afin d'effectuer une segmentation de la demande en fonction des correspondances trouvées et ainsi de connaître au mieux la structure de la requête à générer. Lors du matching, tous les mots sont mis en minuscule et tous les caractères diacritiques (accents, cédilles, etc.) sont normalisés. Chaque mot clef retrouvé est taggé en fonction de s'il s'agit d'une colonne ou d'une table de la base de données interrogée, ou bien encore d'autre chose toujours inconnue pour le moment.

Tout d'abord, un découpage de la phrase entrée, illustré par la figure 2, est effectué en fonction des mots clefs taggés « table » et « colonne » trouvés dans la phrase, afin de savoir quel segment de la phrase correspond à quelle partie de la requête à construire. La présence d'un segment SELECT et FROM est obligatoire dans la phrase, le premier désigne quel va être le type de sélection et sur quel(s) élément(s) exactement, le second spécifie où chercher, dans quelle(s) table(s), l'élément de la sélection. Les segments TIER et WHERE sont, quant à eux, facultatifs. Le segment TIER sert lors des jointures explicites (section 3.5) et le WHERE à préciser, s'il y en a, les contraintes exercées sur la sélection. *En fonction du nombre et de la position des mots clefs dans la phrase, le découpage n'est pas le même et ne donnera donc pas lieu à la même structure de requête en sortie.* On peut notamment remarquer, que si une demande ne contient aucun mot s'apparentant à une table, elle sera forcément invalide et générera donc une erreur.

3.5 Détermination de la structure de la requête

Ensuite, dans chacun des segments obtenus lors du découpage (section 3.4), on analyse les mots clefs taggés jusqu'à présent *inconnu*. Ces mots peuvent être des facteurs de présence d'une requête dite de comptage, de calculs algébriques, d'une négation, etc., ou bien encore d'une valeur sur laquelle devra s'effectuer une contrainte. De cette façon, si un mot faisant référence au comptage comme par exemple « combien » est trouvé dans le premier segment de la phrase, celui correspondant au SELECT, le système identifie la requête à générer comme étant une requête de comptage, c'est-à-dire un SELECT COUNT(*), première branche du premier segment sur la figure 2. L'application fonctionne de la même façon, avec un système de reconnaissance de mots clefs dans les segments SELECT et/ou WHERE, pour de nombreux autres

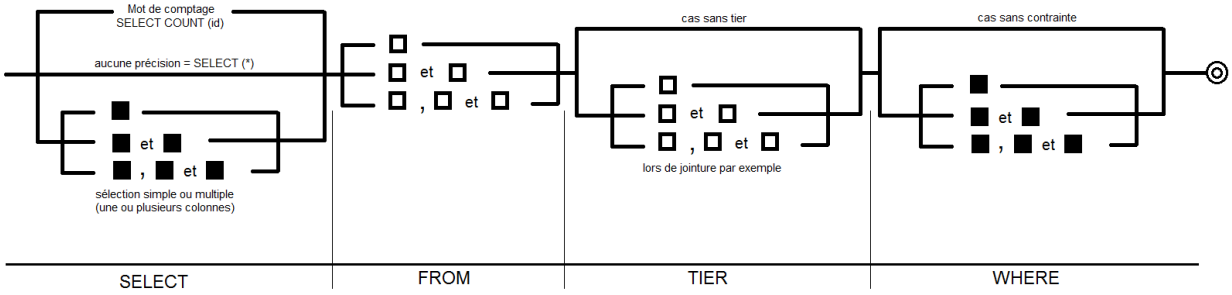

FIGURE 2 – Schéma représentant le découpage opéré sur la phrase entrée par l'utilisateur afin de savoir quel sera le type de requête à générer en sortie, les carrés blancs représentant les tables et les carrés noirs, les colonnes.

types d'opérations. La table 2 énumère de façon volontairement non exhaustive certains mots clefs donnant lieu à ces différentes opérations.

Mots clefs	Opérations
« quel est le nombre », « combien y a-t-il », ...	comptage
« ne [...] pas », « n'[...] pas », ...	négation
« plus grand que », « supérieur à », ...	supériorité
« plus petit que », « inférieur à », ...	infériorité
« quel est la somme », « additionne », ...	agrégat
« quel est la moyenne », ...	moyenne

TABLE 2 – Équivalences entre les mots clefs identifiés et les opérations à générer lors de la sélection.

Nous traitons dans cet article seulement les jointures internes (INNER JOIN). Deux types de jointures internes existent, les implicites et les explicites. Lorsqu'il y a une sélection ou une contrainte sur une colonne qui ne fait pas partie de la table du FROM, autrement dit lorsque la table à laquelle appartient la colonne ciblée n'est pas mentionnée dans la phrase entrée, c'est une jointure implicite. Il faut dans ce cas là, faire une jointure entre la table de la colonne cible et la table du FROM qui est spécifiée dans la phrase. Dans le cas d'une jointure explicite, la table sur laquelle on doit effectuer la jointure est précisée directement dans la phrase, comme dans l'exemple : « Quels sont les élèves ayant un professeur dont le prénom est Jean ? ». C'est dans ce cas là où le segment TIER, apparaissant dans la figure 2, existe et contient une ou plusieurs tables. Ici, il faut faire une jointure entre la table *eleve* et *professeur*, afin de pouvoir sélectionner des élèves tout en faisant une contrainte sur les prénoms des professeurs. Si la colonne de la sélection ou de la contrainte ne se trouve ni dans la table du FROM, ni dans une table accessible par jointure depuis la table du FROM, alors la requête est impossible à construire. La construction des jointures par l'application est rendue possible par la section 3.2, car fr2sql connaissant les clefs primaires et étrangères de chaque table, peut en déduire de façon implicite les liaisons effectives entre les tables et sait donc si une table peut être reliée à une autre et si oui, par quelle(s) table(s) passer. En effet, fr2sql peut créer des jointures passant par plusieurs tables si cela est requis (comme sur la figure 4, pour accéder à la table *professeur* depuis la table *eleve* en passant par la table *enseigner* et *classe*).

3.6 Génération de la requête à l'aide d'une grammaire « laxiste »

La permissivité des applications déjà existantes est due à leur module de matching trop « tolérant », étant donné qu'il recherche un ensemble fini de données dans un espace assez large. C'est ensuite le rôle des règles de leurs grammaires très strictes de réduire à nouveau l'espace des requêtes possibles jusqu'à proposer la plus plausible. Dans fr2sql, c'est en quelque sorte l'inverse qui est opéré. Le matching bidirectionnel réduit en premier lieu l'espace des requêtes possibles, étant donné qu'il effectue une intersection entre un faible ensemble de données et un autre faible ensemble de données. C'est ensuite une grammaire laxiste qui génère la requête en sortie, les règles de cette grammaire ne servant pas à discriminer ou ordonner les requêtes possibles mais seulement à générer la requête déterminée préalablement par le matching. Bien que cette technique rajoute un temps de traitement et des opérations en pré-processing non négligeable, cela amé-

liore sensiblement la discrimination des correspondances (matches) et permet ainsi par la suite de simplement utiliser une grammaire, qui n'a pas pour objectif d'avoir des règles les plus discriminantes possible, étant donné que les étapes précédentes auront déjà filtré la majorité des sources d'erreurs (des correspondances faux positifs), mais qui a uniquement pour objectif de générer une requête en sortie.

En raison du fait qu'un grand nombre de règles, représentant chacune la construction d'une requête possible, existe, nous avons pris la décision de les indexer à l'aide d'une table de hachage pour permettre ainsi leur acquisition plus rapidement (la recherche dans un tableau indexé par des entiers se faisant plus rapidement que la recherche d'une clef constituée de chaîne de caractères). Pour ce faire, un entier est attribué à chaque élément clef de la demande, le 1 pour les éléments de type *table*, le 2 pour les *colonnes*, le 3 pour les *valeurs*, le 4 pour un mot de comptage, etc. La demande entrée donne donc lieu à une série de nombres que l'on concatène, formant ainsi un entier unique. Le 0 correspond au fait que l'élément précédent dans la chaîne peut être présent 1 à N fois au sein de la règle. Ainsi on obtient un tableau de règles indexées par des entiers. Pour retrouver la règle équivalente à une demande entrée, il suffit de rechercher l'entier correspondant à la structure de la demande, qui se trouve également être la clef de la valeur, représentant la structure de la requête à produire en sortie, dans la table.

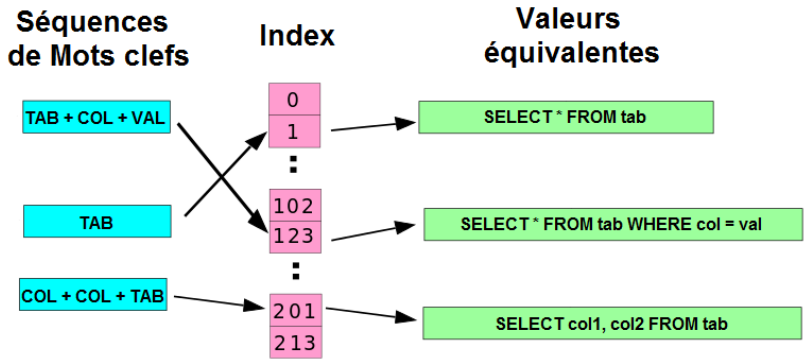

FIGURE 3 – Table de hachage indexant les structures des requêtes à générer en fonction de la structure de la phrase entrée.

Quelques exemples d'équivalences sont présenté dans la table 3. Ces exemples sont les plus triviaux, ils ont pour seul objectif ici d'exposer les équivalences formulées par les suites de mots clefs et les types de requêtes en découlant.

Règles	Opérations notables
NB_KW + TABLE	sélection avec comptage
TABLE	sélection de toutes les colonnes de la table
TABLE + (, TABLE)* + *et* + TABLE	une même demande posée sur plusieurs tables
COLONNE + TABLE	sélection d'une seule colonne
COLONNE + (, COLONNE)* + *et* + COLONNE + TABLE	sélection de plusieurs colonnes
TABLE + COLONNE + VAL	sélection avec contrainte

TABLE 3 – Équivalences entre les mots clefs identifiés et les requêtes à générer.

Maintenant que l'on connaît la structure de la requête à générer en sortie, il suffit de remplacer les tags *variable*, *table* et *colonne* par leur véritable valeur ou nom (les correspondances obtenues dans la base).

La requête est donc générée en fonction de la présence, du nombre et de l'ordre des mots clefs identifiés dans la phrase entrée par l'utilisateur. Nous appelons cette grammaire « laxiste » car elle possède suffisamment de règles pour donner l'impression à l'utilisateur qu'elle accepte toutes formes de demande. De plus, ne vérifiant que la présence et l'ordre des mots clefs, à peu près tous les mots de liaisons ou formes d'écriture sont possibles. Ceci dans le but d'être suffisamment permissif pour que l'utilisateur ne ressente aucune restriction lexicale ou syntaxique.

À noter que le *problème des contraintes muettes* n'est pas supporté par fr2sql. Les demandes telles que « Quels sont les élèves s'appelant Jean ? » ou « Quels sont les élèves de 18 ans ? » ne seront pas traitées correctement par l'application. Ici, la colonne sur laquelle doit s'effectuer la contrainte est implicite, elle n'est pas clairement spécifiée, il est donc impossible pour l'application de la retrouver. Un être humain peut comprendre que dans le premier cas c'est le nom de l'élève que

l'on souhaite et dans le second cas, son âge, mais le système, tel qu'il est conçu, n'a aucun moyen d'y parvenir. On notera également que si une même interrogation est posée sur plusieurs tables dans la même demande, comme par exemple dans la phrase « Quels sont les élèves et les professeurs ayant plus de 25 ans ? », alors le système est capable de produire plusieurs requêtes en sortie, ici deux, pour répondre à la demande ($3^{\text{ème}}$ ligne de la table 3 et $2^{\text{ème}}$ et $3^{\text{ème}}$ branche du segment FROM dans la figure 2).

Une fois une première requête SQL candidate obtenue, on tente de l'exécuter sur la base de données. Si la requête est invalide, c'est qu'elle est mal construite (nom de colonne à la place de table, oubli d'une valeur, oubli d'une colonne, etc.). Le système tente alors de la construire différemment, si en s'exécutant, elle retourne encore une erreur, le système renvoie un message identifiant précisément le type de l'erreur. Ce système permet certes de réduire les erreurs en sorties mais surtout de connaître la catégorie des erreurs renvoyées. D'après les travaux de (Androutsopoulos *et al.*, 1995), l'un des défauts les plus reprochés aux traducteurs du langage naturel à une autre langue, est le manque de clarté. Grâce à ce système, ce problème est en partie résolu.

4 Évaluation et tests

4.1 Base de tests

Aucune base de tests pour un outil en français et supportant plusieurs tables n'ayant été mise à disposition à ce jour, c'est plus de 200 requêtes reprenant tous les éléments standards d'une interrogation classique (sélection simple ou multiple, comptage et calculs algébriques, conjonctions, disjonctions, jointures, conditions, négations, limites et ordonnancement), qui ont été testées sur deux bases différentes, afin de bien illustrer la portabilité multi-bases de notre approche.

Les requêtes ont toutes été écrites manuellement, dont 100 par des personnes ayant des connaissances en BdD et 100 par des personnes n'en ayant aucune dans ce domaine. Toutes ces personnes avaient reçu des informations grossières sur les tables des bases tests mais aucune n'avait pris connaissance de leur structure.

Les deux bases de données tests présentent des structures ainsi que des conventions de nommage différentes. La figure 4 représentent les schémas relationnels de ces bases, elles sont reproductibles et exploitables pour de futurs travaux.

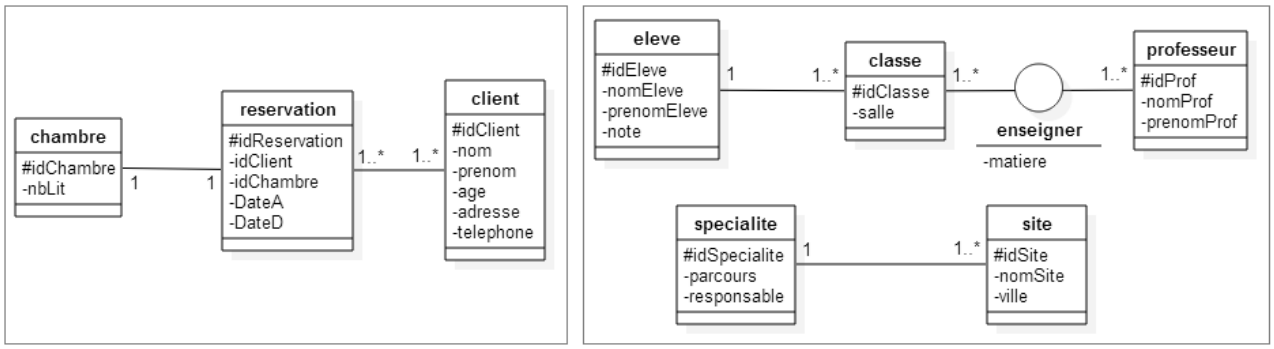

FIGURE 4 – Schémas relationnels des bases de données tests.

4.2 Résultats

Nous adoptons la définition de *précision* et *rappel* décrites dans (Minock *et al.*, 2008) et (Popescu *et al.*, 2003). Une phrase entrée par l'utilisateur peut demander la génération de plusieurs requêtes ou peut ne produire aucune requête.

$$\text{précision} = \frac{card(\text{réponses correctes retournées})}{card(\text{réponses retournées})}$$

$$\text{rappel} = \frac{card(\text{réponses correctes retournées})}{card(\text{phrases entrées})}$$

On constate dans la table 4 que notre méthode supporte autant si ce n'est plus d'opérations que les outils déjà existants, tout en étant compatible instantanément sur n'importe quelle base de données. On constate également que seule fr2sql est compatible sur toutes bases et gère la synonymie de façon à ne pas restreindre les demandes au seul vocabulaire employé dans la base.

	MONDE	SQL-HAL	English2SQL	fr2sql
Sélection sur colonne	OUI	OUI	OUI	OUI
Sélection sur table	OUI	OUI	OUI	OUI
Sélection multiple	NON	OUI	OUI	OUI
Comptage	OUI	OUI	OUI	OUI
Comptage multiple	NON	OUI	OUI	OUI
Contrainte simple	OUI	OUI	OUI	OUI
Contrainte muette	OUI	NON	OUI	NON
Disjonction	OUI	OUI	OUI	OUI
Conjonction	OUI	OUI	OUI	OUI
Contrainte croisée	OUI	OUI	OUI	OUI
Gestion des dates	NON	OUI	OUI	NON
Rangement	NON	NON	NON	OUI
Comparaison	OUI	OUI	OUI	OUI
Calcul algébrique	NON	OUI	OUI	OUI
Négation	OUI	NON	OUI	OUI
Synonymie	NON	NON	NON	OUI
Jointure pour sélection	OUI	NON	OUI	OUI
Jointure pour condition	OUI	NON	OUI	OUI
Requêtes imbriquées	OUI	NON	OUI	NON
Compatibilité sur plusieurs base de données	NON	OUI	OUI	OUI
Restriction de vocabulaire ou de grammaire	OUI	OUI	OUI	NON

TABLE 4 – Opérations supportées par les différentes applications de traductions.

Les résultats dans la table 5 illustrent les performances de notre méthode en fonction du type de demande. Les questions entraînant une sélection sans jointure montrent de bien meilleurs résultats (0.957 de F-score en moyenne) qu'une demande donnant lieu à une jointure (0.761 de F-score). À noter également que les phrases donnant lieu à des requêtes imbriquées ne sont pas correctement traduites par l'application qui ne les gère tout simplement pas encore à l'heure actuelle.

	Précision	Rappel	F-mesure
Tout type de requêtes	0.939	0.850	0.892
Sélections seulement	1	0.969	0.984
Avec jointures	0.832	0.702	0.761
Avec conditions	0.987	0.880	0.930

TABLE 5 – Performances par catégorie de requête de l'application fr2sql.

5 Conclusions

Bien que nous n'avons pas pu effectuer clairement de comparatif, d'après l'état de l'art, notre méthode montre des résultats globalement équivalents à la plupart des applications actuelles (une précision supérieur à 0.90 pour un rappel supérieur à 0.85) avec néanmoins une faiblesse au niveau des jointures (0.761 de F-mesure). On notera également l'impossibilité de celle-ci à gérer les contraintes muettes et à générer des requêtes imbriquées.

Dans de futurs travaux, nous prévoyons de traiter les contraintes muettes, en conservant les verbes comme mots clefs et en ajoutant quelques règles dans la grammaire. Cela permettra par exemple de définir que « l'élève s'appellant Jean » signifie la même chose que « l'élève dont le nom est Jean » ou bien encore que « l'élève de 18 ans » équivaut à « l'élève

ayant pour âge 18 ans ». De plus, il est prévu de détecter la langue de la demande entrée par l'utilisateur afin d'utiliser un dictionnaire de synonymes ayant trait à cette langue et d'ajuster les règles en fonction de la langue pour ainsi rendre le système robuste à d'autres langues que le français.

Pour conclure, bien que perfectible, cette approche permet bien d'interroger n'importe quelle base de données SQL, répondant ainsi aux objectifs de portabilité fixés, tout en gardant des performances dans la moyenne des applications déjà existantes et en couvrant une large palette d'opérations de sélection.

Références

ALEXANDER R., RUKSHAN P. & MAHESAN S. (2013). Natural Language Web Interface for Database (NLWIDB). *In CoRR*.

ANDROUTSOPOULOS I., RITCHIE G. & THANISCH P. (1995). Natural Language Interfaces to Databases - An Introduction. *In Journal of Natural Language Engineering*, **1**, 29–81.

CHANDRA Y. (2006). *Natural Language Interfaces to Databases*. PhD Thesis. University of North Texas, USA.

CHAUDHARI P. P. (2013). Natural Language Statement to SQL Query Translator. *In International Journal of Computer Applications*, **82**(5), 18–22.

CHEN W. (2014). Parameterized Spatial SQL Translation for Geographic Question Answering. *In Semantic Computing (ICSC), 2014 IEEE International*, p. 23–27.

CIMIANO P. & MINOCK M. (2009). Natural Language Interfaces : What is the Problem ? - A data-driven quantitative analysis. *In 14th International Conference on Applications of Natural Language to Information System (NLDB)*, **5723**.

DESHPANDE A. K. & DEVALE P. R. (2012). Natural Language Query Processing Using Probabilistic Context Free Grammar. *In International Journal of Advances in Engineering and Technology*, **3**, 568–573.

DJAHANTIGHI F. S., NOROUZIFARD M., DAVARPANAHAND S. & SHENASSA M. H. (2008). Using Natural Language Processing in Order to Create SQL Queries. *In Proceedings of the International Conference on Computer and Communication Engineering*, p. 600–604.

GIORDANI A. & MOSCHITTI A. (2009). Semantic Mapping Between Natural Language Questions and SQL Queries via Syntactic Pairing. *In Natural Language Processing and Information Systems*, **5723**, 207–221.

GIORDANI A. & MOSCHITTI A. (2012). Translating Questions to SQL Queries with Generative Parsers Discriminatively Reranked. *In COLING 2012 : Posters*, p. 401–410.

GREEN B. F., WOLF A. K., CHOMSKY C. & LAUGHERY K. (1961). Baseball : An automatic question-answerer. In *Papers Presented at the May 9-11, 1961, Western Joint IRE-AIEE-ACM Computer Conference*, IRE-AIEE-ACM '61 (Western), p. 219–224, New York, NY, USA : ACM.

HURRICANE ELECTRIC I. S. (2012). English2SQL powered by he.net. Logiciel.

KAUR J., CHAUHAN B. & KOREPAL J. K. (2013). Implementation of Query Processor Using Automata and Natural Language Processing. *In International Journal of Scientific and Research Publications*, **3**.

MICROSOFT (2000). TechNet : Developing with English Query. Logiciel.

MINOCK M. (2010). C-Phrase : A system for building robust natural language interfaces to databases. *In Data and Knowledge Engineering*, **69**(3), 290–302.

MINOCK M., OLOFSSON P. & NÄSLUND A. (2008). Towards Building Robust Natural Language Interfaces to Databases. *In 13th International Conference on Applications of Natural Language to Information System (NLDB)*, **5039**, 187–198.

MOHITE A. & BHOJANE V. (2014). Challenges and Implementation Steps of Natural Language Interface for Information Extraction from Database. *In International Journal of Recent Technology and Engineering (IJRTE)*, **3**.

PASERO R. (1997). Une interface en français à une base de données discographiques. Logiciel.

PASERO R. & SABATIER P. (1998). Une interface en français à une base de données sur les États du monde. Logiciel.

PATIL R. & CHEN Z. (2012). Struct : incorporating contextual information for english query search on relational databases. In T. W. LING, G. YU, J. LU & W. W. 0011, Eds., *KEYS*, p. 11–22 : ACM.

POPESCU A. M., ETZIONI O. & KAUTZ H. (2003). Towards a Theory of Natural Language Interfaces to Databases. *In Proceedings of the 8th International Conference on Intelligent User Interfaces*, p. 149–157.

POUND J., ILYAS I. F. & WEDDELL G. (2010). Expressive and flexible access to web-extracted data : A keyword-based structured query language. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, p. 423–434, New York, NY, USA : ACM.

RAO G., AGARWAL C., CHAUDHRY S., KULKARNI N. & PATIL D. S. (2010). Natural Language Query Processing using Semantic Grammar. *In International Journal on Computer Science and Engineering*, **2**, 219–223.

SAFARI L. & PATRICK J. D. (2014). Restricted natural language based querying of clinical databases. *In Journal of Biomedical Informatics*.

SCHMID H. (1994). Probabilistic Part-of-Speech Tagging Using Decision Trees. In *Proceedings of the International Conference on New Methods in Language Processing*.