Generating SQL Queries Using Natural Language Syntactic Dependencies and Metadata

Alessandra Giordani and Alessandro Moschitti

Department of Computer Science and Engineering
University of Trento
Via Sommarive 14, 38100 POVO (TN) - Italy
{agiordani,moschitti}@disi.unitn.it

Abstract. This research concerns with translating natural language questions into SQL queries by exploiting the MySQL framework for both hypothesis construction and thesis verification in the task of question answering. We use linguistic dependencies and metadata to build sets of possible SELECT and WHERE clauses. Then we exploit again the metadata to build FROM clauses enriched with meaningful joins. Finally, we combine all the clauses to get the set of all possible SQL queries, producing an answer to the question. Our algorithm can be recursively applied to deal with complex questions, requiring nested SELECT instructions. Additionally, it proposes a weighting scheme to order all the generated queries in terms of probability of correctness.
Our preliminary results are encouraging as they show that our system generates the right SQL query among the first five in the 88% of the cases. This result can be greatly improved by re-ranking the queries with a machine learning methods.

1 Introduction

NLIDB propose a large body of work based manual work for grammar specification and dataset annotation. However the task of question answering, translating natural language (NL) into something understandable by a machine, in an automatic way is rather challenging as it is not possible to hand-crafting all the needed rules.

We address this problem by generating SQL queries whose structure and components match with NL concepts (expressed as words) and grammar dependencies. Our new idea consists on how and where this matching can be found, i.e., exploiting existing knowledge that comes along with each database (metadata). The resulting matching between metadata and words allows for building sets of query components (clauses). These are combined together using a smart algorithm to generate a set of SQL queries, taking into account also the structure of the starting NL.

1.1 Motivations and Problem Definition

A database is not just a collection of data: at design time, domain experts organize entities and relationships giving proper names to tables and columns, defining constraints and specifying the type of the stored data. This additional information is known as metadata and is stored in an underlying database called INFORMATION_SCHEMA (IS, for brevity) that contains, for each database, tables containing columns referring to table names and column names. This self-reference allows for querying metadata with the same technique and technology used to query the database embedding data that answer a given question. In other words, we can execute several SQL queries over IS and a target database and combine their result sets to generate the final SQL query in a straightforward and very intuitive way.

In addition, we can perform question answering against multiple databases, since IS stores metadata of all databases that reside in a single machine. Moreover there is no need to use tailored dictionaries since metadata already embeds a rich knowledge based on the experience of the domain expert that designed the database. Another important feature that should be taken into account is the potential complexity of the NL question (subordinates, conjunction, negation) that can find its own matching in nested SQL query. The general SQL query that our system can deal with has the form:

$$\text{SELECT DISTINCT } COLUMN \text{ FROM } TABLE \text{ [WHERE } CONDITION\text{]} \quad (1)$$

We can *project* a single $COLUMN$ and eventually apply aggregation operators that summarize it by means of SUM, AVG, MIN, MAX and COUNT. Optionally, we can also *select* data for which a $CONDITION$ holds. This is represented as a logical expression where basic conditions, in the form e_L OP e_R, with OP=$\{<,>$,LIKE,IN$\}$, may be combined with AND, OR, NOT operators. While e_L (as well as $COLUMN$) is always in the form `table.column`, e_R could be a numerical value, a string value or a nested query. The meaning of $TABLE$ is more straightforward, since it should contain table name(s) to which the other two clauses refer. This clause could just be a single table or be a join operation, which selectively pairs tuples of two tables.

We found a mapping algorithm that matches dependencies between NL components and SQL structure that allows to build a set of possible queries that answers a given question. To represent textual relationships of the NL sentence we use typed dependency relations. The Stanford Dependencies Collapsed (SDC) representation [1] provide a simple and uniform description of binary grammar relations holding between a governor and a dependent (each dependency is written as *rel*(*gov*, *dep*) where *gov* and *dep* are words in the sentence).

The question answering task of finding an SQL query Q that retrieves an answer for a given NL question q reduces to the following problem. Given question q represented by means of its typed dependencies collapsed SDC_q, generate the three sets of clauses $\mathcal{S}, \mathcal{F}, \mathcal{W}$ such that the set $\mathcal{A}$ =SELECT $\mathcal{S}\times$ FROM $\mathcal{F}\times$ WHERE $\mathcal{W}$ contains all possible answers to q and select the one that maximizes the probability of being the correct answering query Q. In the next section, we show how we can efficiently generate such triples.

2 Building Clauses Sets

The first step before generating all possible queries for a question q is to create their components $\mathcal{S}, \mathcal{F}, \mathcal{W}$, i.e. SELECT, FROM and WHERE clauses, starting

(1)$root$(ROOT, are),	$\Pi = \{\text{capital, state}\}$
(2)$nsubj$(are, capital),	$\Sigma = \{\text{are}\} \Rightarrow \Sigma = \phi$
(3)$prep\_of$(capital, state),	
(4)$nsubj$(border, state),	$\Pi' = \{\text{state, border}\}$
(5)$rcmod$(state, border),	$\Sigma' = \{\text{border, state}\}$
(6)$advmod$(populat, most),	
(7)$amod$(state, populat),	$\Pi'' = \{\text{most, populat, state}\}$
(8)$dobj$(border, state)	$\Sigma'' = \phi$

Fig. 1. Categorizing stems into projection and/or selection oriented sets

from a dependency list SDC_q. This list should be (a) preprocessed using pruning, stemming and adding synonyms, (b) analyzed to create the set of stems used to build $\mathcal{S}$ and $\mathcal{W}$ and (c) modified/cleaned to keep dependency used in a eventual recursive step in order to generate nested queries.

First we prune those relations that are useless for our processing and then reduce *gov*s and *dep*s to stems [2] to obtain the optimized list SDC_q^{opt}. An example showing $SDC_{q_1}^{opt}$ with respect to question q_1: "*What are the capitals of the states that border the most populated state?*" can be found in Figure 1.

Then for each grammatical relation in SDC_q^{opt} we apply an iterative algorithm that adds these stems respectively to Π and/or Σ categories accordingly to a set rules that for lack of space cannot be listed here. However, the key idea is to exploit projection-oriented relations (e.g. $ROOT$ and *nsubj*) and selection-oriented ones (e.g. *prep* and *obj*) to categorize stems recursively.

Next, we use Π to search in metadata all fields that could match with projection-oriented stems. Based on how many matchings are found, a weight w is inferred to each projection, obtaining the SELECT clause set $\mathcal{S}$.

Instead, the selection-oriented set of stems Σ should be divided into two distinct sets of stems Σ_L and Σ_R. The set Σ_L contains stems that find their matching in IS, whereas for remaining stems $\Sigma_R = \Sigma - \Sigma_L$ we should look up in the database to find a matching. In order to build the WHERE clauses set $\mathcal{W}$, $\forall e_L \in \mathcal{W}_L, \forall e_R \in \mathcal{W}_R$ we first generate basic expressions $expr = e_L$ OP $_R$ and combine them by means of conjunction and negation, keeping only those expressions $expr$ such that the execution of $\pi_{count(*)}(\sigma_{expr}(table))$ [1] does not lead to an error for at least a *table* in the database.

It is worth nothing that Σ_R could be the empty set, e.g. when a WHERE condition requires nesting; in this case e_R will be the whole subquery. Moreover, also Σ_L could be empty. This is not surprising since, in a SQL query the WHERE clause is not mandatory. However the absence of selection-oriented stems doesn't necessarily mean that $\mathcal{W}$ should be empty. When this happens all tables and columns of the database are taken into account to find valid conditions: $\mathcal{W}*_R = \{t.c | t \in \pi_{table\_name}(IS.Columns) \text{ and } c \in \pi_{column\_name}(IS.Columns)\}$.

Last, once the two sets $\mathcal{S}$ and $\mathcal{W}$ have been constructed, the generation of the FROM clause $\mathcal{F}$ is straightforward. This set should just contain all tables to which clauses in $\mathcal{S}$ and $\mathcal{W}$ refer, enriched by pairwise joins. Again, this task could

[1] π and σ represents projection and selection operators of relational algebra.

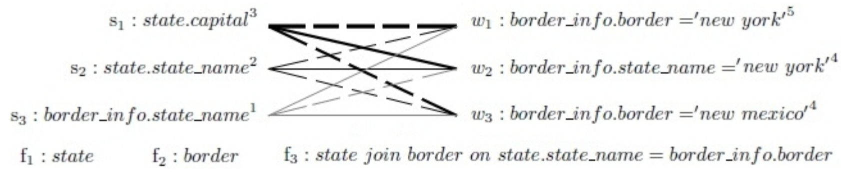
s1 : state.capital³
s2 : state.state_name²
s3 : border_info.state_name¹
w1 : border_info.border ='new york'⁵
w2 : border_info.state_name ='new york'⁴
w3 : border_info.border ='new mexico'⁴
f1 : state
f2 : border
f3 : state join border on state.state_name = border_info.border

Fig. 2. Possible pairings for q_2: "*Capitals of states bordering New York*"

be performed running SQL queries over IS, and especially exploiting metadata stored in table KEY_COLUMN_USAGE. This table identifies all columns in current databases that are restricted by some unique, primary key, or foreign key constraint. That is, for each foreign key column usage in the table, we can determine how many aggregate table columns match that column usage.

3 Generating Queries

The starting point for finding an answering query is to generate the set $\mathcal{A} = \{\mathcal{S}\times\mathcal{F}\times\mathcal{W}\} \cup \{\mathcal{S}\times\mathcal{F}\}$. If such query exists there should be a pairing $\langle s, f, w\rangle \in \mathcal{A}$, such that the execution of `SELECT` s `FROM` f `[WHERE` w`]` retrieve the correct answer. Given that on average each clause set contains up to ten items, their product could result in a very huge set. Actually when generating all pairs some preliminary conditions are verified, e.g. tables appearing in SELECT and WHERE clauses should as well appear in the FROM clause, otherwise the execution of that query will fail. This avoids to generate incorrect queries and to waste time trying to execute them.

At this point the set $\mathcal{A}$ contain all valid pairings, among which there are still someone not useful. One example are meaningless queries: those projecting the same field compared to a value in the selection. For example the pairing $\langle s_3, f_2, w_2\rangle$ in Figure 2 answers the question "*Which state is Texas?*" that is clearly useless.

In particular there are redundant queries that once optimized can lead to a duplicate in the set; hence its cardinality is lower than in theory. For example, the pairing $\langle s_2, f_3, w_1\rangle$ involves that the columns *state.state_name* and *border_info.border* are the same, so w_2 would select same rows of w_1' : *state.state _name* ='*new york*', but this means that table *border_info* is not used and this pairing would be equivalent to $\langle s_2, f_1, w_1'\rangle$, which is a meaningless query.

As we already noticed in previous sections, we add a weight to each clauses in $\mathcal{S}$ and $\mathcal{W}$. This weight is a simple measure consisting in counting how many stems originated the clause. When pairing clauses the combined weight is just computed as the sum of its components, and it's used to order the obtained set $\bar{\mathcal{A}}$ of possible useful queries from the most probable to the less one.

In Figure 2 the higher probability is highlighted by thicker connection lines (dashed lines illustrate pruned queries). The final ordered set answering q_2 is $\bar{\mathcal{A}}=\left\{\langle s_1, f_3, w_2\rangle^7, \langle s_3, f_2, w_1\rangle^6, \langle s_2, f_3, w_2\rangle^6, \langle s_1, f_1\rangle^3, \langle s_2, f_1\rangle^2, \langle s_3, f_2\rangle^1\right\}$. The right answering query can be derived from the pairing with highest weight, that is:
```
SELECT state.capital FROM state join border on state.state_name =
border_info.border WHERE border_info.state_name='new york'.
```

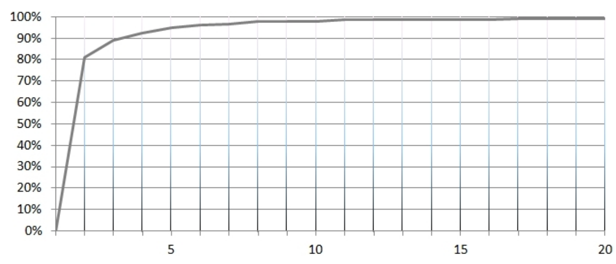

Fig. 3. Precision values when seeking for the answer in a larger subset.

It is worth noting that there could be more queries with the same weight. To cope with that we privilege queries that involve less joins and those that embed the most meaningful (i.e. referenced) table, e.g. `state` in the case of GeoQuery. Note, for example, the order of the second and third pairings in $\bar{\mathcal{A}}$: they have been swapped since f_3 contains a join while f_2 doesn't.

4 Preliminary Experiment and Related Work

In a first preliminary experiment, we have studied the effectiveness of our automatic generation of SQL queries for a given NL question by testing the accuracy of selecting the one that retrieves the correct answer.

We started from the set of 800 NL questions given in the GeoQuery corpus, trying to generate a result set equivalent to the one of associated SQL query in the corpus. Originally these questions were paired with meaning representations [3] that have been translated [4] into SQL queries. However, 23 of 800 queries are wrong or cannot be executed without leading to a MySQL error.

The first implementation of the algorithm as illustrated in this paper, failed to generate the set of possible queries for 71 questions, leading to a recall of 88%. These failures may be due to: (i) empty clauses set $\mathcal{S}$ and/or $\mathcal{W}$, e.g., "*How many square kilometers in the us?*" does not contain any useful stem; (ii) mismatching in nested queries, e.g. "*Count the states which have elevations lower than what alabama has*" contains an implicit reference to a missing piece of question; and (iii) ambiguous questions, e.g. "*Which states does the colorado?*" from which we retrieve an incomplete dependency set.

For all remaining questions from which we succeed in generating an ordered list of possible queries, we find that the query on top of this list retrieves the correct result set 81% of the times. For the other questions it can be found within the first 10 generated queries with a precision of 99%, according to the growing precision curve shown in Figure 3. As pointed out in the plot, the right query is found among the first three in 92% of the cases.

As the previous work suggests [5], similar outcome has been obtained using different approaches: relying on semantic grammar specified by an expert user [6], enriching the information contained in the pairs [4] and implementing adhoc rules in a semantic parser [3, 7]. Our system instead, requires no intervention since the database metadata already contain all the needed information. This result is encouraging since it compares favorably with the state of the art: with respect to the Precise system [4] (100% Precision and 77% Recall) and the Krisp

[3] system (94% Precision and 78% Recall). Our system can always provide an answer, achieving an accuracy of 81%. If we consider valid an answer given in the top three, our accuracy increases to 95%, achieving 99% on the 10-top ranked. Note that the accuracy at top-ranked answer can be improved by learning a reranker, as explained in [8], which can move correct answers in the top position.

5 Conclusions and Future Work

In this paper, we approach the question answering task of implementing a NL interface to databases by automatically generating SQL queries based on grammatical relations and matching metadata. The complexity of generated queries is fairly high indeed, since we can deal with questions that require nesting, aggregation and negation in addition to basic projection, selection and joining (e.g. "*How many states have major non-capital cities excluding Texas*"). To our knowledge, the underlying idea that we propose for building and combining clauses sets is novel. Preliminary experiments on automatic question generation system show a satisfactory accuracy, i.e. 81%, although large improvement is still possible.

In the future we plan to extend this research by introducing learning approaches to classify and rank generated queries.

Acknowledgement

The research described in this paper has been partially supported by the European Community's Seventh Framework Programme (FP7/2007-2013) under the grants #247758: EternalS – Trustworthy Eternal Systems via Evolving Software, Data and Knowledge, and #288024: LiMoSINe – Linguistically Motivated Semantic aggregation engiNes.

References

1. Marie-Catherine de Marneffe, B.M., Manning, C.D.: Generating typed dependency parses from phrase structure parses. In: Proceedings LREC 2006. (2006)
2. Porter, M.: Porter stemmer. `http://tartarus.org/~martin/PorterStemmer/`
3. Kate, R.J., Mooney, R.J.: Using string-kernels for learning semantic parsers. In: Proceedings of the 21st ICCL and 44th Annual Meeting of the ACL, Sydney, Australia, Association for Computational Linguistics (July 2006) 913–920
4. Popescu, A.M., A Etzioni, O., A Kautz, H.: Towards a theory of natural language interfaces to databases. In: Proceedings of the 2003 International Conference on Intelligent User Interfaces, Miami, Association for Computational Linguistics (2003)
5. Giordani, A., Moschitti, A.: Corpora for automatically learning to map natural language questions into sql queries. In: Proceedings of LREC'10), Valletta, Malta, European Language Resources Association (ELRA) (may 2010)
6. Minock, M., Olofsson, P., Näslund, A.: Towards building robust natural language interfaces to databases. In: NLDB '08: Proceedings of Natural Language and Information Systems, Berlin, Heidelberg (2008)
7. Ruwanpura, S.: Sq-hal: Natural language to sql translator. `http://www.csse.monash.edu.au/hons/projects/2000/Supun.Ruwanpura`
8. Giordani, A., Moschitti, A.: Syntactic structural kernels for natural language interfaces to databases. In: Proceedings of Machine Learning and Knowledge Discovery in Databases: Part I. ECML PKDD '09, Springer-Verlag (2009)