QUEST: A Natural Language Interface to Relational Databases

Vadim Sheinin, Elahe Khorashani, Hangu Yeo, Kun Xu, Ngoc Phuoc An Vo, Octavian Popescu
IBM Research, Yorktown Heights, US
firstname.lastname@us.ibm.com

Abstract

Natural language interfaces to databases systems allow the user to use natural language to interrogate a database. Current systems mainly focus on simple queries but neglect nested queries, which are predominant in real cases. We present a NLIDB system, QUEST, which is able to cope with nested logic queries, without imposing any restrictions on the input query. QUEST outperforms a strong baseline system by 11% accuracy.

Keywords: natural language interface to relational database, machine learning.

1. Introduction

Natural language interfaces to database systems (NLIDB) allow the user to use natural language to interrogate a database. This is an old desideratum, but there is no easy way to attain it. To begin with, even for simple queries that can be solved into a single SQL select query, namely **S**imple **L**ogic **Q**ueries (SLQs), may be challenging to get the SQL query because of different joins needed to be executed. Moreover, the queries that need a succession of select in SQL, namely **N**ested **L**ogic **Q**ueries (NLQs), are very difficult to be handled automatically. NLQs usually display a wide range of linguistic phenomena like ellipsis and anaphora, garden path parsing, etc. Due to this, many systems tend to impose strict limitations on the input query language, in a way that makes the extraction of logical relationships controllable. But this is not actually a very realistic scenario.

There is a large literature on this research field describing systems that are purely rule based, semi-supervised, and unsupervised. We point to excellent overviews of such systems in (Copestake and Jones, 1990; Androutsopoulos et al., 1995; Li and Jagadish, 2014).

The recent developments in deep learning algorithms seem to bring back into the foreground the NLIDB. Working on related topics, such as learning inferences over constrained outputs, global inference in natural languages, open domain relation extraction (Schwartz et al., 2014) , or more recently, resolving algebraic problems expressed in natural language (Koncel-Kedziorski et al., 2015), provide a strong ground for approaching the NLIDB by taking advantage of the power of semi-supervised and unsupervised techniques. However, most systems did not deal with nested logic queries.

In this paper we present a semi-supervised methodology to deal with NLQs in the most general setting: (i) we do not presuppose any restriction on the input query, and (ii) we do not presuppose any restriction on the structure of the database. From a practical point of view, we do not rely on large training data, but rather we build on different technologies that carry out linguistic processing of a query up to a point where a specialized engine is able to produce a series of SQL queries that return the answer in a recursive bottom-up process of computation. We developed a system, QUEST, on top of IBM Watson UIMA pipeline (McCord et al., 2012) and Cognos[1], which aims to provide effective SQL execution performance. In a nutshell, firstly a database connection is created in Cognos. In a Watson UIMA pipeline, a NLQ is decomposed into a set of sentences that individually can be resolved by Cognos. The modules involved in processing the query and handling Cognos are transparent to the user, who interacts only with the web interface. The user is unaware of the complexity involved either in the language analysis or in the nested SQL.

As shown in Figure 1, our system is based on a rule-based kernel whose role is to ensure a correct translation from simple queries to SQL. Generally, we have two main steps: (1) rule based semantic parsing (*Quest Engine* Box); and (2) further SQL generation and execution on databases (*Cognos Engine* Box). In the first step, we generated the lexicalized rules that would be used in the semantic parsing (offline part). Specifically, we relied on the schema independent rule templates to automatically extract the lexicalized rules from the schema annotation file. In the online part, we built a semantic parser using several Watson NLP components in a pipeline fashion, namely English Slot Grammar parser (ESG), Predicate Argument Structure (PAS), and Subtree Pattern Matching Framework (SPMF). Our semantic parser will produce a list of SQL sub-queries that are fed to the second step. In the step 2, we employed the Cognos server to combine these sub-queries into a final SQL query which is then executed on the DB2 server. Experimental results suggest that this approach helps to improve 11% accuracy over a very strong baseline.

2. Nested Question Decomposition

One of the main challenges in the NLIDB task is to handle nested questions which require additional aggregation operations over the intermediate results. For example, "*which company manufactured more products than Samsung*" is a nested question. To answer this question, we need to solve the following two sub-questions orderly:

1. how many products did Samsung manufacture ?
2. which company manufactured products more than <the answer of question #1> ?

[1] `http://www.ibm.com/analytics/us/en/technology/cognos-software/`

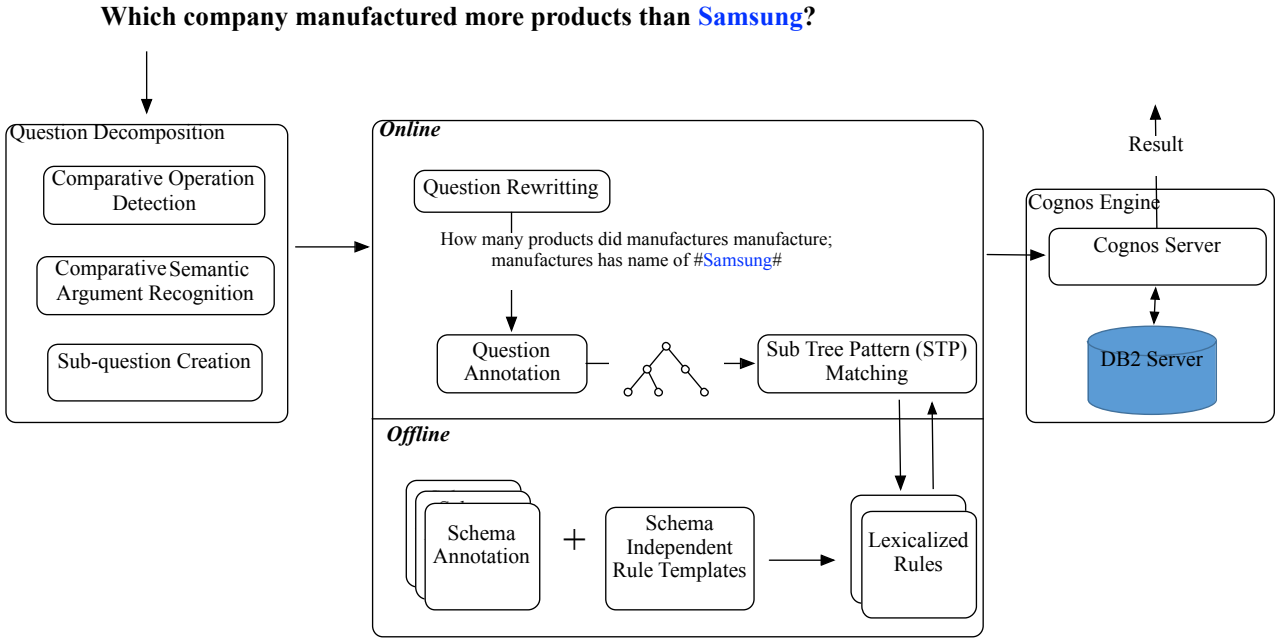

Figure 1: The Quest Architecture.

Intuitively, one way to solve the nested questions is to first decompose the question into several sub-questions, which may be then resolved in a specific order.

By analyzing real query logs in our system, we observed that in most cases, nested questions include comparative expressions, such as "*more than*" and "*higher than*". Being motivated by this observation, we proposed a decomposition method based on analyzing the comparative expressions and their arguments. Specifically, we first developed a syntactic based method to detect these comparative expressions. Then we used a syntactic scope delimiter to determine the arguments of the expressions, such as "*products*" and "*Samsung*" in Figure 2. Note that, sometimes the arguments may not be compatible due to the ellipsis in natural language, also indicating that this question needs the decomposition. For example, "*products*" which is the left argument of the comparative phrase is countable, while "*Samsung*" is not. To resolve this ellipsis, we identified the semantic arguments of the comparative phrase, which in our case, is "*the products manufactured by Samsung*". To represent the decomposition, we created the sub-questions for these arguments and replaced these arguments with symbols in the original question.

3. Quest Engine

After the nested question decomposition, each decomposed question is fed to the Quest engine, which consists of two main components, rule generation (offline part) and rule-based semantic parsing (online part). The main idea of the Quest engine is to automatically create a set of schema-dependent rules, such as sub tree patterns, and finding matches among these rules for different parts of the parsed input question. From those matched rules, we can create a set of SQL sub-queries which will be later combined and executed in the Cognos server.

3.1. Rule-based Semantic Parsing

With recent progress in the structured databases, semantic parsing against these databases has attracted more and more research interests. Almost all of the existing semantic parsers need to be trained either with the direct supervision such as logical forms (Kwiatkowski et al., 2011) or with the weak supervision such as question-answer pairs (Berant et al., 2013). However, in real application scenarios, collecting sufficient question-answer pairs to train the model remains a challenge. To address this, in this paper we proposed a rule-based semantic parser which consists of the following three components.

Question Rewriting The input question is first processed through the entity annotator, which detects entities in the question and rewrites the question by replacing these entities with their semantic types as described in schema annotation. We explicitly specified the values of these semantic types with additional description sentences. The key motivation behind this is that entities that appear in a database column can be meaningless or unrecognizable to the parser. Also, replacing these entities with relative less and finite semantic types can further ease the pattern matching problem.

Question Annotation After detecting entities and rewriting the question, the question is processed through a sequence of annotators, such as numerical annotator, date annotator, comparator annotator and predicate argument structure (PAS) annotator. Specifically, comparator annotators detect particular types of values and operators for those values that appear in the question. PAS annotator detects the predicate argument structure of the question. For instance, for the question in Figure 1, comparative annotator may create an annotation with operator "*more*". Date annotator may create an annotation if there is any temporal expression in the question.

Nested SQL

which company manufactured more products than SAMSUNG

```
"query1" AS
  (
  SELECT
    COUNT_BIG("PRODUCTS"."PRODUCT_ID") AS "PRODUCTS_PRODUCT_ID",
    "MANUFACTURERS"."NAME" AS "MANUFACTURERS_NAME"
  FROM
    "NLIDBSCH"."MANUFACTURERS" "MANUFACTURERS"
      INNER JOIN "NLIDBSCH"."PRODUCTS" "PRODUCTS"
      ON "MANUFACTURERS"."MANUFACTURER_ID" = "PRODUCTS"."MANUFACTURER_ID"
  WHERE
    LOWER("MANUFACTURERS"."NAME") LIKE LOWER('SAMSUNG')
  GROUP BY
    "MANUFACTURERS"."NAME"
  ),
"query2" AS
  (
  SELECT
    COUNT_BIG("PRODUCTS"."PRODUCT_ID") AS "PRODUCTS_PRODUCT_ID",
    "MANUFACTURERS"."NAME" AS "MANUFACTURERS_NAME"
  FROM
    "NLIDBSCH"."MANUFACTURERS" "MANUFACTURERS"
      INNER JOIN "NLIDBSCH"."PRODUCTS" "PRODUCTS"
      ON "MANUFACTURERS"."MANUFACTURER_ID" = "PRODUCTS"."MANUFACTURER_ID"
  GROUP BY
    "MANUFACTURERS"."NAME"
  ),
"auxQuery1" AS
  (
  SELECT
    SUM("query1"."PRODUCTS_PRODUCT_ID") AS "PRODUCTS_PRODUCT_ID"
  FROM
    "query1"
  ),
"auxQuery2" AS
  (
  SELECT
    SUM("query2"."PRODUCTS_PRODUCT_ID") AS "PRODUCTS_PRODUCT_ID",
    "query2"."MANUFACTURERS_NAME" AS "MANUFACTURERS_NAME"
  FROM
    "query2"
  GROUP BY
    "query2"."MANUFACTURERS_NAME"
  )
SELECT DISTINCT
  "auxQuery2"."MANUFACTURERS_NAME" AS "MANUFACTURERS_NAME"
FROM
  "auxQuery1"
    INNER JOIN "auxQuery2"
    ON "auxQuery2"."PRODUCTS_PRODUCT_ID" > "auxQuery1"."PRODUCTS_PRODUCT_ID"
```

Figure 2: SQL formulas generated by QUEST.

Sub Tree Pattern Matching The last step of our semantic parser is to search previous annotation results to find all the patterns in the lexicalized rules that match the input question. These patterns determine the types of the SQL sub-queries that should be produced and the values of the filter items if there are any.

3.2. Rules Generation

We automatically generate the lexicalized rules that are used in the sub tree pattern matching from the schema annotation and rule templates. The rule templates include a set of rule patterns for generic sentence structures. There is no schema related lexical information in these patterns, but only grammatical information that is applicable to any schema. The schema annotation is manually created for each specific database schema. It describes the different elements (tables and columns) of a database and their relationships with each other. Since the schema annotation is primarily created manually it limits the coverage of expressions against the relations in the database. Although the schema annotation is primarily created manually, it is enriched using automatically acquired paraphrases.

3.3. SQL Generation and Execution

For each question, the generated SQL sub-queries from the Quest engine are sent to the Cognos server, where the final SQL queries are produced and submitted to database.

4. Experiments

In this section we introduce the experimental setup, the main results and detailed analysis of our system. We looked separately at SLQs vs. NLQs, as indeed a large gap in performance is noticed between these two cases. Quest has been evaluated against a complex corpus, which includes both SLQs and NLQs. The baseline we used is a system that achieves state of the art results on language to logical form conversion.

4.1. Baseline and Evaluation Corpora

We implemented a deep neural network combining the insights from (Wang et al., 2015) and (Dong and Lapata, 2016). The network is a sequence to tree long short term memory (seq2tree LSTM). The input query is decoded into its logical form by a hierarchical tree decoder which identifies the arguments of a predicate.

First, we used the GeoQuery dataset (Zelle and Mooney, 1996) which contains 880 geography questions annotated with logical forms. This corpus is used only to evaluate the effectiveness of the baseline system as there are no SQL queries associated with these questions, but only their logical forms. Moreover, all these queries are simple logic queries that makes it possible for the baseline to attain an accuracy above 90%. GeoQuery dataset is standardly split into 600 queries for training, and 280 queries for testing.

We manually created a dataset from real queries on golf tournaments, named GolfQuery, made up of 2,350 queries. This corpus is made out of two parts: GF_NLQ contains 1,200 queries that are nested SQL queries, and GF_SLQ is made of the remaining 1,150 queries that are all simple logic queries. From GF_NLQ we randomly chose 249 queries for testing, the remaining 951 queries were used for training. From GF_SLQ a random set of 350 queries were chosen for testing, the rest 800 were used for training.

4.2. Results and Discussion

In the first experiment we examined the baseline system using the GeoQuery dataset. The accuracy computed as ratio between the number of correct resolved queries vs. the total number of queries, was 92% which differs insignificantly from the current state of the art performances on this corpus, and proves that this is indeed a strong baseline.

In Table 1 we presented the accuracy of the baseline system Seq2Tree and of our system Quest on GF_NLQ and GF_SLQ. Overall, Quest attained near 73% accuracy, leading to 11% increase over the baseline. While for SLQs, both system attain very good performances, above 92%, for NLQs, the Quest improves the baseline by 17%.

Looking at the results of GF_SLQ vs. GF_NLQ we can see a large difference between the accuracy of SLQs vs. NLQs. We wanted to understand how this gap is related to the number of training examples, and if there is a scalability problem for the baseline system.

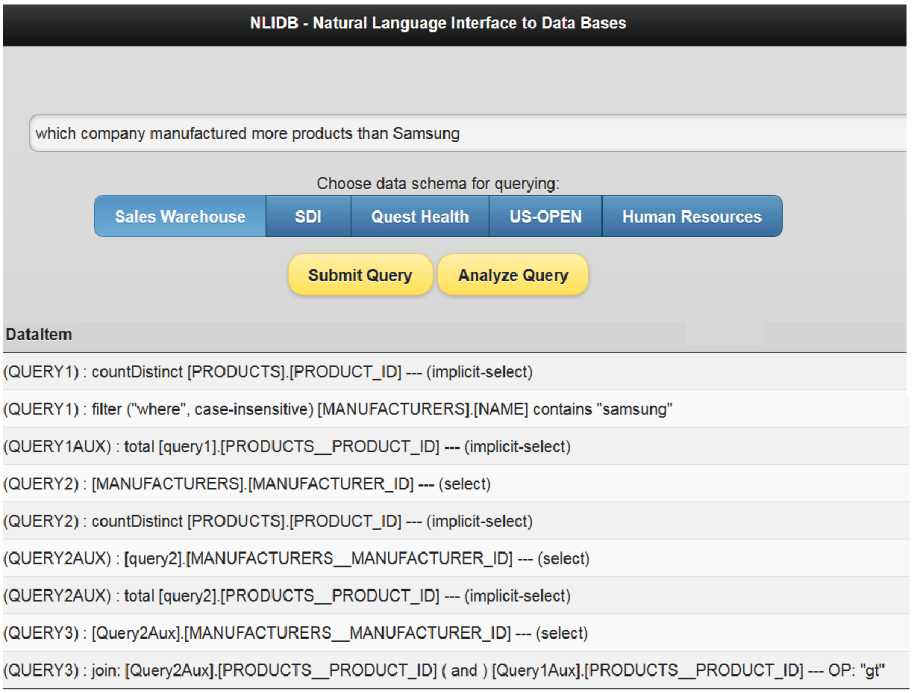

Figure 3: The natural language interface of Quest to the relational databases.

Figure 4: Two running examples of Quest system.

Scalability To evaluate the scalability of the baseline approach, we investigated the relation between the number of training examples vs. SLQ and NLQ accuracy. As shown in Figure 5, the accuracy of the baseline system grows quickly as the number of training examples increases for SLQs, both for GeoQuery and GF_SLQ datasets. However, for GF_NLQ, the learning curve looks different both for the direct test on the chosen 249 testing queries (GF_TST) of

	GF_SLQ	GF_NLQ	overall
Quest	98%	38%	73%
Seq2Tree	92%	21%	62%

Table 1: Quest and seq2Tree on Golfquery

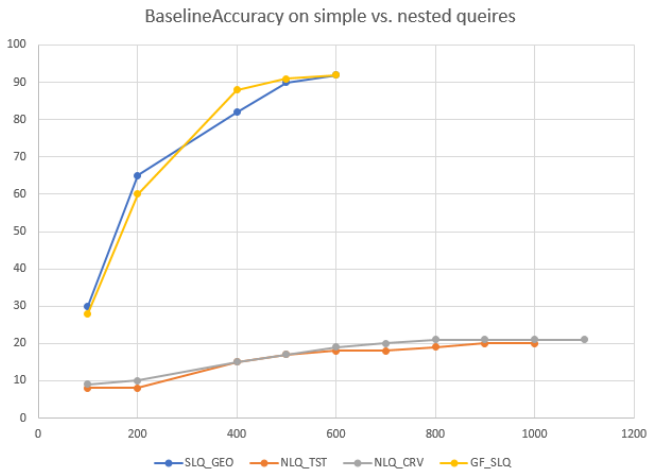

Figure 5: Seq2Tree learning curve

GF_NLQ, and for 10-fold cross validation (GF_CRV) for the whole GF_NLQ dataset. This result suggests that the number of training examples needed for a higher accuracy may be very high. This is a serious bottleneck, as in all real scenarios we are aware of, training data is unavailable.

Nested question decomposition We selected a set of 64 sentences from the GF_NLQ test set that are particularly complex. Then we run Quest with 10-fold cross validation on GF_NLQ dataset, and we imposed that a third of the test queries of each fold to be chosen randomly from the 64 complex sentences. The performance on each fold varied considerably from 30% to 63%. This is mainly due to ambiguous attachment which is an issue that we plan to address in the future.

4.3. Error Analysis

We analyzed 155 questions that are not answered correctly by our system. These errors are mainly due to three reasons: (1) QUEST now does not support certain logical operations, such as "NOT" or "OR", (2) limited sub tree pattern rules can not cover all queries, and (3) schema annotation does not define some operations. For example, although the QUEST was able to answer the question "*Who was the runner up at the 1950 US Open?*", it could not answer the question "*Who was the second place in the 2015 US Open?*" because the semantic of "*second*" was not defined in schema annotation file.

4.4. Demo

In Figure 3, we showed the system output for the query whose automatic SQL was presented in Figure 2. In Figure 4, we displayed an example of complex queries, in which the right order of operators "*per date*" and "*highest*" must be inferred. As being showed, the system solved them correctly.

5. Conclusion

In this paper we presented the main characteristics of an NLIDB system which outperforms a strong baseline (seq2Tree LSTM). It showed that the system obtained much better accuracy on nested logic queries which are the major source of difficulty in this task. The most efficient further directions for improvement are of two types: (i) negation and logical operators handling in natural language, and (ii) a better linguistic processing of the sentences in order to correctly identify the arguments of SQL queries. According to the experiments that were carried out, this could add a significant increase in accuracy.

6. Bibliographical References

Androutsopoulos, I., Ritchie, G. D., and Thanisch, P. (1995). Natural language interfaces to databases - an introduction. *Natural Language Engineering*, 1(1):29–81.

Berant, J., Chou, A., Frostig, R., and Liang, P. (2013). Semantic parsing on freebase from question-answer pairs. In *EMNLP*.

Copestake, A. A. and Jones, K. S. (1990). Natural language interfaces to databases. *Knowledge Eng. Review*, 5(4):225–249.

Dong, L. and Lapata, M. (2016). Language to logical form with neural attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers*.

Koncel-Kedziorski, R., Hajishirzi, H., Sabharwal, A., Etzioni, O., and Ang, S. D. (2015). Parsing algebraic word problems into equations. *TACL*, 3:585–597.

Kwiatkowski, T., Zettlemoyer, L. S., Goldwater, S., and Steedman, M. (2011). Lexical generalization in CCG grammar induction for semantic parsing. In *ACL*, pages 1512–1523.

Li, F. and Jagadish, H. V. (2014). Constructing an interactive natural language interface for relational databases. *PVLDB*, 8(1):73–84.

McCord, M. C., Murdock, J. W., and Boguraev, B. (2012). Deep parsing in watson. *IBM Journal of Research and Development*, 56(3):3.

Schwartz, E. M., Bradlow, E. T., and Fader, P. S. (2014). Model selection using database characteristics: Developing a classification tree for longitudinal incidence data. *Marketing Science*, 33(2):188–205.

Wang, Y., Berant, J., and Liang, P. (2015). Building a semantic parser overnight. In *ACL*.

Zelle, J. M. and Mooney, R. J. (1996). Learning to parse database queries using inductive logic programming. In *AAAI*, pages 1050–1055.