λόγος: A System for Translating Queries into Narratives

Andreas Kokkalis[1], Panagiotis Vagenas[1], Alexandros Zervakis[1]
Alkis Simitsis[2], Georgia Koutrika[3], Yannis Ioannidis[1]

[1] University of Athens
Athens, Hellas
{a.kokkalis,p.vagenas,a.zervakis,yannis}@di.uoa.gr

[2] HP Labs
Palo Alto, CA, USA
alkis@hp.com

[3] IBM Almaden
San Jose, CA, USA
gkoutri@us.ibm.com

ABSTRACT

This paper presents Logos, a system that provides natural language translations for relational queries expressed in SQL. Our translation mechanism is based on a graph-based approach to the query translation problem. We represent various forms of structured queries as directed graphs and we annotate the graph edges with template labels using an extensible template mechanism. Logos uses different graph traversal strategies for efficiently exploring these graphs and composing textual query descriptions. The audience may interactively explore Logos using various database schemata and issuing either sample or ad hoc queries.

Categories and Subject Descriptors

H.2.m [**Database Management**]: Miscellaneous

Keywords

Query translation, SQL queries, Natural Language

1. INTRODUCTION

"Logos, originally ... (meant) ... 'opinion', 'word', 'speech', 'reason', became a technical term in philosophy, beginning with Heraclitus (535–475 BC), who used the term for a principle of order and knowledge. … For Heraclitus logos provided the link between rational discourse and the world's rational structure. … Aristotle (384–322 BC) applied the term to refer to 'reasoned discourse'." [Wikipedia].

Logos (λόγος in Greek) is the system we have developed for generating natural language translations for SQL queries.

Explaining queries in text may be useful in a handful of cases. For example, many applications (e.g., museum portals, digital libraries, e-commerce sites, and so forth) offer a form-based environment for formulating queries to search (web-based) databases. In addition, emerging Do-It-Yourself (DIY), database-driven web application platforms empower non-programmers to easily create and evolve applications customized to their needs by manipulating visual elements. In all these scenarios, involving searching and programming over a database, user interactions with the interface are translated to structured queries. Explaining these implicitly built queries without exposing the details of the underlying query language becomes vital especially when executing a query may have a different outcome from what the user anticipated. Translation of a user's choices on a certain form in a narrative can assist her in forming queries correctly, even without being expert in use of a specific interface or a query language.

Query translation can be also helpful when users use a structured query language. Before sending the query for execution, seeing it expressed in a more familiar way can help check whether it captures correctly the intended meaning. A user trying to understand an error message concerning her mistaken query might prefer to have an explanation of that query in a familiar language, instead of getting back an error code and a generic error description. As another example, when a query returns an empty answer, an explanation of the query may help identify parts of the query that are responsible for the failure. Similarly, when a query returns a very large number of answers, a query explanation may help the user understand the reasons and possibly rewrite the query into one that returns fewer results. As yet another example, translating queries into natural language descriptions can be handy in self-guided exercises as part of a database class for students to get better familiarized with database query languages.

Traditionally, the application of natural-language techniques to the front-end of an information system environment focused mainly on the opposite direction of the one studied here: from NL requests to queries production (e.g., [3, 6]). Other research efforts have tried to provide visual explanations to queries (e.g., [1]).

Finding a correct and meaningful translation for a query is not trivial. Some queries do not have obvious semantics. Consider for example the following query (Query 1, Q1):

```
select a.id, a.name from MOVIE m, CAST c, ACTOR a
where m.id = c.mid and c.aid = a.id
group by a.id, a.name having count(distinct m.year) = 1
```

One may see a typical aggregate query, but in reality, it is the count aggregate that implies *all* and dominates the query. Hence, it is not obvious how to produce a correct narrative, such as:

"*Find the actors whose movies are all in the same year*"

Other queries contain different type of difficulty. A query with many joins might be straightforward to be translated by simply following primary key (PK) – foreign key (FK) relationships, but producing a meaningful translation might not be that trivial. Consider for example the following query, which involves a large number of relations interconnected via PK-FK join relationships:

```
select a.name, m.title from MOVIE m, CAST c, ACTOR a,
       DIRECTED r, DIRECTOR d, GENRE g
where m.id=c.mid and c.aid=a.id and m.id=r.mid and r.did=d.id
      and m.id = g.mid and d.name = `Coppola'
      and g.genre = `action'
```

Due to the number of join relationships, a straightforward translation would be lengthy and convoluted. However, with appropriate templates, this query (Query 2, Q2) can be translated as follows:

"Find the titles of action movies directed by Coppola and the names of actors that play in these movies" or

"Find the actors and titles of action movies directed by Coppola"

Our system, Logos, deals with such difficulties through two key mechanisms: (*a*) a template mechanism that allows the translation of peculiar syntactic patterns and the production of more natural text, and (*b*) a set of query graph traversal strategies that generate text from a query avoiding repetition of certain phrases or nouns and lengthy narratives.

2. BACKGROUND

In this section, we provide an overview of our approach.

DB Schema and Queries as Graphs. We take a graph-theoretic approach for representing a database schema and various forms of structured queries as directed graphs. The database schema graph is a directed graph that captures the basic roles of relations and attributes in queries over the database. Its nodes are relations and attributes, and its edges are either membership (connecting attributes to relations), selection (from relations to attributes), or predicate edges (from attribute to attribute, i.e., joins). A simple query graph captures the possible semantics of an SPJ query and it is an extension of the database schema graph. Its nodes are relation (one for each tuple variable in the query), attribute (one for each attribute occurrence), and value nodes (one for each value or a set of values specified in the query). Its edges are membership (capturing projections), predicate (capturing joins), and selection edges (capturing selection conditions). For more complex query types, a query graph may contain other edge and node types that capture functions as well as order-by, group-by and having clauses. For details, we refer the interested reader to [2].

Graph Annotations and Templates. We give semantics to the various parts of a query by annotating the query graph edges with template labels using an extensible template mechanism.

Each node that can be part of a query graph may be annotated by a label that signifies the meaning of the node in natural language. For example, for the relation MOVIE, the label may be '*movies*'. Similarly, each edge (or path) connecting two nodes can be annotated by a label that signifies the meaning, in natural language, of the relationship between the source and destination node. For instance, the membership edge connecting MOVIE to its attribute Title may have the label '*of*'. Labels are stored on the database schema graph for both nodes and edges. A query graph inherits these edges from the database graph.

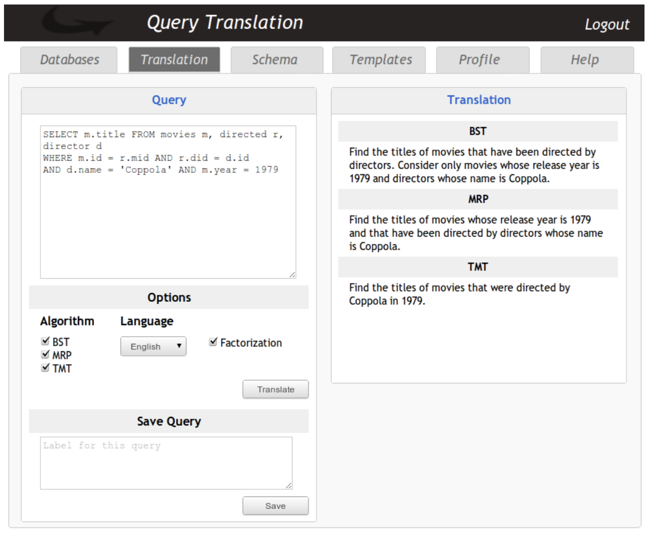
Our translation methods traverse the query graph and create phrases by composing labels found on the way. For producing more natural results, we use template labels at different granularity levels and an extensible template mechanism to fuse these labels. A *template label*, $l((v, u))$, is assigned to an edge (v, u) or to a path connecting v to u. This template is used for the interpretation of the relationship between v and u in a narrative. A template label may have the form:

$$l((v, u)) = expr_1 + l(v) + expr_2 + l(u) + expr_3$$

Template labels are created manually by a human, and can produce high-quality, concise text. As a short example, a template for the selection edge e_σ(MOVIE, year) may be the following:

$l(e_\sigma(\text{MOVIE, year})) = l(\text{MOVIE}) +$ " released in " + MOVIE.year.<val>

Query Translation as Graph Traversal. We use three domain-independent graph traversal strategies for efficiently exploring query graphs and composing query descriptions as narratives.



```
SELECT m.title FROM movies m, directed r,
director d
WHERE m.id = r.mid AND r.did = d.id
AND d.name = 'Coppola' AND m.year = 1979
```

Algorithm: ✓ BST, ✓ MRP, ✓ TMT
Language: English
✓ Factorization

BST

Find the titles of movies that have been directed by directors. Consider only movies whose release year is 1979 and directors whose name is Coppola.

MRP

Find the titles of movies whose release year is 1979 and that have been directed by directors whose name is Coppola.

TMT

Find the titles of movies that were directed by Coppola in 1979.

Figure 1. Translation form

The first strategy (BST algorithm) composes separate clauses for each part of the query. First, it translates the membership edges, then it connects all query relations to the query subject through the joins in the query, and finally, it reads the paths that connect relations to value nodes that are specified for attributes on these relations. The translation is performed in a depth-first way on the query graph starting from the query subject to all relations through the joins on the graph. The query subject represents what the query refers to. Identifying the query subject is important because it determines how we traverse the query graph, i.e., the query translation direction, and what kind of clauses we generate. Naturally, it is a relation with attributes projected in the select-clause that holds a central position in the query graph (see [2]).

In the second strategy (MRP algorithm), the translation is realized in a holistic manner, where information from all parts of the query graph is blended in the translation as we traverse the graph. The key idea here is that, while in BST all query components refer always to the query subject, in MRP we use additional reference points to avoid long, possibly unnatural, sentences. In this way, we semantically split the translation at multiple points.

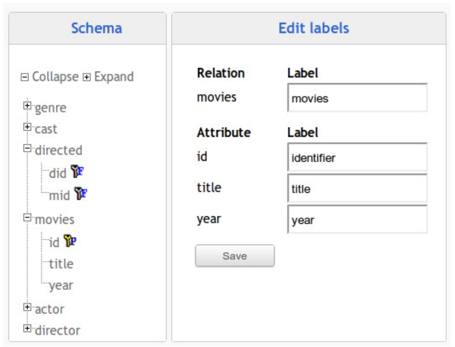
Finally, the third strategy (TMT algorithm) enables the use of predefined, richer, templates for query parts in an effort to produce more concise translations.

Example translations for each strategy are shown in Figure 1. Next, we present the features of Logos.

3. INTERACTING WITH LOGOS

User Perspective. Typically, a regular user is interested in the query translation and user profile forms.

Query Translation. Query translation is achieved through a form, where the user provides a query and sees its translation into natural language (see Figure 1). The translation can be customized by setting parameters such as preferred language, translation algorithm, and factorization. This latter option controls whether special words such as articles or relative pronouns are repeated or not, thus tuning translation verbosity. It is possible to get more than one translation in the same form by selecting multiple translation algorithms. Then, the user may issue a new query or refine the existing translation by modifying the parameters. It is also possible to label and store a query for later use.



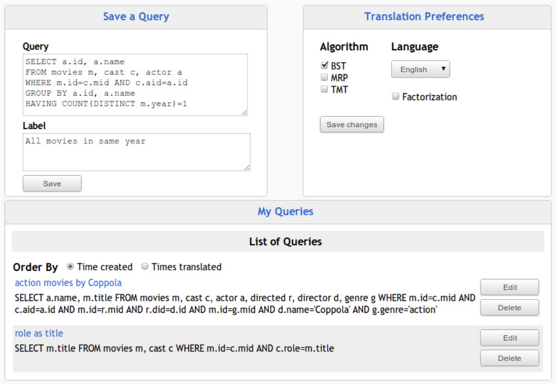
Relation	Label
movies	movies

Attribute	Label
id	identifier
title	title
year	year

Figure 2. Schema browsing and labeling

For issuing a query, one needs to be aware of the database schema. Logos features a tree-like schema explorer (see Figure 2) that enables a quick review of the database schema. First, the user is presented with the top-level nodes of the structure tree; i.e., the relations in the database. Expanding relation nodes displays their child attribute nodes, whereas selecting a specific node –either relation or attribute– displays its label (see Section 2; e.g., 'movies' for the relation MOVIE). In the latter case, label editing –which typically is an administrator task– is also available.

User Profiles. Logos features an interactive web interface that allows for customized query translation. In order to provide a seamless user experience, profile data are stored. This data includes translation preferences, such as language, translation algorithm, and factorization, as well as a list of saved queries. Through a user profile page (see Figure 3), translation preferences may be modified and then persisted, while queries in the history log may be selected for translation. Users may also assign labels to queries in the history log.



Query

```
SELECT a.id, a.name
FROM movies m, cast c, actor a
WHERE m.id=c.mid AND c.aid=a.id
GROUP BY a.id, a.name
HAVING COUNT(DISTINCT m.year)=1
```

Label: All movies in same year

Algorithm: [✓ BST] MRP TMT

Language: English

Factorization ☐

Order By [✓ Time created] Times translated

action movies by Coppola

SELECT a.name, m.title FROM movies m, cast c, actor a, directed r, director d, genre g WHERE m.id=c.mid AND c.aid=a.id AND m.id=r.mid AND r.did=d.id AND m.id=g.mid AND d.name='Coppola' AND g.genre='action'

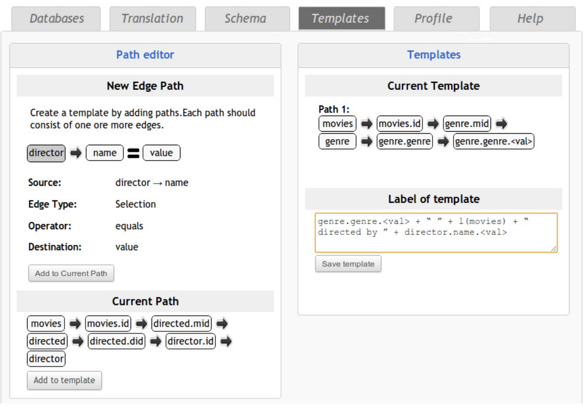
role as title

SELECT m.title FROM movies m, cast c WHERE m.id=c.mid AND c.role=m.title

Figure 3. User settings

Administrator Perspective. Logos can be used for tuning a database schema in a way that enables meaningful and useful translations. Logos connects to a DBMS and extracts the metadata of a database schema. Having connected to a specific database, template definition is carried out using Logos' integrated administration console. Our implementation does support default labels (e.g., "of" for membership edges), but as the designer provides the system with more fine-tuned labels, the translation results are even more descriptive.

For fine-tuning, template graph synthesis is performed in a step-by-step, incremental manner assisted by Logos. Figure 4 shows the template building process for the part of Q2 that results into the "*action movies directed by Coppola*". The template graph involved consists of two paths. The first one starts from the MOVIE relation node, gets to DIRECTOR through DIRECTED, using PK-FK



Create a template by adding paths.Each path should consist of one ore more edges.

Source:	director → name
Edge Type:	Selection
Operator:	equals
Destination:	value

```
genre.genre.<val> + " " + l(movies) + "
directed by " + director.name.<val>
```

Figure 4. Template construction

attribute nodes, then moves to the name attribute of the DIRECTOR relation and finally to its value node. The second path starts from the MOVIE relation node, gets to GENRE using PK-FK attribute nodes, then moves to the genre attribute of the GENRE relation and finally to its value node.

For creating the template, we work as follows. First, we use the path editor to determine a path by designating successive edges.

Let us examine the first path. Using the *source* tree-like selector, the source relation node (MOVIE) and its respective attribute (id) are determined. Then the type of the current path operation is specified as a *join*. After selecting, say, the equality operator as the predicate operator, we use the *destination* tree-like selector to determine how the join ends.

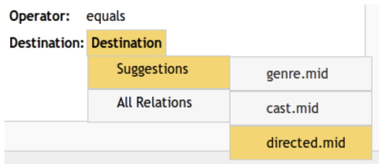


Figure 5. Suggestions

Logos exploits PK-FK relationships to offer destination node suggestions (see Figure 5) and therefore, to facilitate and accelerate the entire task. Using this feature, we can easily select the DIRECTED relation and the respective mid attribute and thus, conclude this join. Then, this join is added to the current path. We then repeat this process twice to build the entire path: first, we add the join from DIRECTED to DIRECTOR and then, the selection from DIRECTOR to their name value. After the path has been completed, it can be added to the current template.

Next, Logos aggregates multiple joint paths. To add the second path of the template, we first specify the node that it shares with the first path and then repeat the same steps.

Once the template graph has been completed, an appropriate template language expression [2,5] has to be assigned to it before it gets stored. The template language expression in this case is:

GENRE.genre.<val> + " " + l(MOVIE) + " directed by " + DIRECTOR.name.<val>

Finally, the designer may assign this template label to the produced graph and store the template.

Multilingual Query Translations. Logos also offers translations into languages other than English. In many plain localization settings, supporting a language only consists in string assignment. Natural language generation however requires full expressiveness and flexibility. Since every language has its own grammatical and

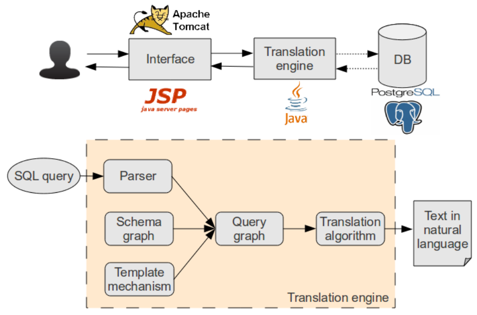


Figure 6. System architecture

syntactical peculiarities, simple string assignment are not enough in this case. Logos tackles this challenge by considering language extensibility at a more abstract level, so that grammatical and syntactical aspects are embraced. In Logos API, each language is defined through a programmatic module, which determines its specific attributes and behavior in terms of the translation. In addition, module inheritance allows for flexibility through code reuse. With this approach, a new language can be supported by creating a new module and integrating it into the system.

As a proof of concept, apart from English, Logos may also generate translations in Spanish and Greek. For example, Q1 would be translated in Spanish as: *"Buscar el nombre de los actores cuyas películas son todas del mismo año"* and in Greek: *"Βρες το πεδίο 'όνομα' των αντικειμένων 'ηθοποιοί' των οποίων οι ταινίες είναι όλες στον ίδιο χρόνο"*.

4. DEMONSTRATION

We offer an interactive demonstration, where the participants will be able to connect to a handful of databases and experiment with queries provided by our system or ad hoc queries, and view and customize query translations. We provide example database schemata with varying number of tables, number of attributes per table, and number of PK-FK relationships, so that participants can experiment over schemata of different complexity.

The demonstration will serve two purposes: (*a*) to present how Logos can be used by a regular user and (*b*) to highlight its extensibility and design features presenting how an administrator may use it. Hence, participants can interchangeably assume the role of a regular user or administrator while interacting with Logos.

From a regular user's perspective, we will demonstrate the query translation and user profile forms as well as the multi-lingual functionality. Participants may choose from a predefined set of queries and also provide ad-hoc queries. The predefined set comprises queries over different database schemas with different level of difficulty and particularities. Our goal here is two-fold: (*a*) to show how the query translation engine can deal with all such cases, and (*b*) to show the benefits of query translation as we go from fairly simple queries to quite complex ones that do not lend themselves to a straightforward translation.

Participants will also experiment with the different query translation strategies. Furthermore, they will be able to use the multi-lingual translation and see how a query can be explained in text in Spanish or Greek. They will be also able to store their favorite query translation method, selected queries, and other parameters as part of their own profile in the system. This functionality would for instance help a SQL instructor to prepare queries and their translations as part of self-guided exercises for her students to use.

The second part of our demonstration is to show how Logos can be used for enriching a database schema in a way that enables meaningful and useful translations. Participants will be able to see how template definition is carried out using Logos' integrated administration console. Our goal is to show how the designer can provide the system with fine-tuned labels without much effort and make the translation results over a particular database more descriptive. The administrator may have her own set of queries that she uses for testing how query translation looks like using different templates. An administrator may also specify and store user profiles in the system that contain specific sets of queries and query translation options for the users to use.

5. ARCHITECTURE

Logos comprises two core components, namely the *Translation Engine* and the *GUI* (see Figure 6). The Translation engine comprises five modules: the Schema and Query graphs, the Parser, the Template mechanism, and the Translation algorithm module. The engine is implemented in Java, using the Apache collections generic library. Its input is a database schema, an SQL query, and the translation preferences, and the output is the translation of the query in natural language. We use PostgreSQL as our database.

The *Schema* and *Query graphs* represent database schemas and SQL queries, respectively. Nodes and edges are decorated with labels that signify their meaning in natural language. Both graphs are implemented using the Java Universal Network/Graph Framework (JUNG) library.

The *Parser* is a top down parsing module that performs lexical and syntactic analysis on the query and generates parts of the Query graph, using information from the Schema graph and the Template mechanism. For its implementation, we have used the JavaCC parser generator.

The *Template mechanism* is a powerful feature for defining template labels and assists in producing high quality, concise text. Our template strategy tries to cover the query graph with the optimum combination of template labels. To facilitate this procedure we use the powerSet methods found in the Google Core Libraries for Java (Guava). The template language [2,5] used for registering new templates is also implemented in JavaCC.

The *Translation algorithm* module embraces the various graph traversal strategies, modeled separately as different classes. This approach fosters extensibility, since it allows new traversal algorithms to be seamlessly introduced into the system.

The GUI is a web application that runs on Tomcat and is implemented in Java EE using servlets and JSP, along with jQuery. We also use the JSTL library to avoid scriptlets in the JSP, the Joda Time, and the Apache Commons Validator library.

6. REFERENCES

1. J. Danaparamita, W. Gatterbauer. QueryViz: Helping Users Understand SQL Queries and their Patterns. In EDBT, 2011.
2. G. Koutrika, A. Simitsis, Y.E. Ioannidis. Explaining structured queries in natural language. In ICDE, 2010.
3. M. Minock. A phrasal approach to natural language interfaces over databases. In NLDB, 2005.
4. A. Simitsis, Y.E. Ioannidis. DBMSs Should Talk Back Too. In CIDR, 2009.
5. A. Simitsis, G. Koutrika, Y. Alexandrakis, Y.E. Ioannidis. Synthesizing structured text from logical database subsets. In EDBT, 2008.
6. V. C. Storey, R. C. Goldstein, H. Ullrich. Naive semantics to support automated database design. In IEEE TKDE, vol. 14, no. 1, 2002.