Explaining Structured Queries in Natural Language

Georgia Koutrika [1], Alkis Simitsis [2], Yannis E. Ioannidis [3]

[1]*Computer Science Dept., Stanford University, California, USA.*
koutrika@stanford.edu

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

[3]*Dept. of Informatics and Telecom., University of Athens, Greece.*
yannis@di.uoa.gr

Abstract— **Many applications offer a form-based environment for naïve users for accessing databases without being familiar with the database schema or a structured query language. User interactions are translated to structured queries and executed. However, as a user is unlikely to know the underlying semantic connections among the fields presented in a form, it is often useful to provide her with a textual explanation of the query. In this paper, we take 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. We present different graph traversal strategies for efficiently exploring these graphs and composing textual query descriptions. Finally, we present experimental results for the efficiency and effectiveness of the proposed methods.**

I. Introduction

Structured query languages are powerful tools at the hands of advanced searchers and experienced developers but the vast majority of users are not familiar with them. For this reason, 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 rapidly and cheaply create and evolve applications customized to their needs by manipulating visual elements [1], [2]. In all these scenarios (i.e., 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 or effect from what the user has anticipated. Translation of a user's choices on a certain form in a narrative would assist her in forming queries correctly, even without being familiar with a specific interface or a query language. Especially in large forms, a user is likely to not know the underlying semantic connections among the fields presented in the form, and a textual explanation may come in handy.

Explaining queries in text may be useful in some cases for users that use a structured query language for writing queries. Before the query is sent for execution, it may be useful to see the query expressed in a more familiar way in order to check that it captures correctly the intended meaning. A user trying to understand an error message concerning her mistaken query would 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 highlight the reasons, in case a rewrite would reduce this number significantly and serve the user better.

In general, in any situation where explanation of queries is warranted, such textual interpretation may be very useful and effective. Insertions, deletions, and updates, especially those with complicated qualifications or nested constructs, will benefit from a translation into natural language. Likewise for view definitions and integrity constraints, which borrow most of their syntax from queries. Although here we focus on SQL, similar arguments can be made about RDF queries in SPARQL or RQL, even Datalog programs, and others.

The requirement for translating structured queries to text is further dictated by current trends. Automating computer-to-human speech translation is recognized as one of the seven most important IT challenges for the next 25 years by Gartner analysts who examine technologies that will have a broad impact on all aspects of people's lives [3].

Translating queries into narratives has been largely ignored so far. Traditionally, the application of natural-language techniques to the front-end of an information system environment has been one-directional: from NL requests for information to queries production (e.g., [4]). Unfortunately, the fact that NLP tools are trying to match SQL query patterns with NL queries significantly bounds the idea of reversing their functionality for getting the NL translation of an SQL query.

The problem we are studying can be informally stated as follows. Given a query q over a database D, we would like to generate a narrative that captures the intended meaning or objective of q. Translating a structured query to text is challenging due to a number of reasons, including insufficient SQL semantics and the complexity of the queries, which may have nested queries, complex query conditions and different query constructs (group-by, order-by, etc.). In addition, there are several alternative expressions of a query in a formal language that are equivalent, based on associativity, commutativity, and other algebraic properties of the query constructs. Capturing the query elements in the right order so that the corresponding textual expression is natural and meaningful independent of the way the user has expressed the query is not straightforward.

We take a graph-based approach for representing various forms of structured queries as directed graphs. We annotate the graph elements with labels using an extensible template mechanism. We present three translation strategies. In the first

978-1-4244-5446-4/10/$26.00 © 2010 IEEE

Departments(DepID, DepCode, Name) Courses(CourseID, DepID, Title)
Instructors(InstrID, Name) Students(SuID, Name, Class, GPA)
CourseSched(CourseID, Year, Term, InstrID, TimeSlot)
StudentHistory(SuID, CourseID, Year, Term, Grade)
Comments(SuID, CourseID, Year, Term, Text, Rating, Date)

Fig. 1. An example course database

one (BST algorithm), the translation consists of a composition of clauses each one focusing on specific query semantics. 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 last strategy (TMT algorithm) enables the use of predefined, richer, templates for query parts in an effort to produce more concise translations. Our approach mainly targets queries like those found in [2]. However, a query language has different semantics than a spoken language. In our previous work, we have presented a taxonomy of queries based on their complexity and expressivity [5]. There are queries that are very difficult or even impossible to be translated in a meaningful way. We can still handle some of these using pre-defined patterns.

Contributions. In summary, our contributions are:

- We introduce a novel query graph model for capturing the possible semantics of a query.
- We give semantics to the various parts of a query by annotating the query graph edges with template labels using an extensible template mechanism.
- We present different, domain-independent graph traversal strategies for efficiently exploring query graphs and composing query descriptions as phrases in natural language.
- We present an algorithm for selecting the best templates for a query given (possibly overlapping) templates for different query parts.
- We compare the translation algorithms and show their applicability and effectiveness through experimental results.

II. Query Representation

We focus on relational databases and SQL queries. In this section, we introduce our query graph representation that captures query elements and their semantic associations.

A. Database Graph

A database D comprises a set of relations. A relation R_i has a set of attributes. We use A^i_j to refer to an attribute of R_i. We represent the database D by its *database graph* $\mathbf{G}(\mathbf{V}, \mathbf{E})$, a directed graph corresponding to the schema of D extended to capture the basic roles of attributes in queries over the relations of the database. Nodes in $\mathbf{V}$ are: (a) *relation nodes*, $\mathbf{R}$ - one for each relation in the schema; (b) *attribute nodes*, $\mathbf{A}$ - one for each attribute in the schema. Edges in $\mathbf{E}$ are:

- *membership edges*, $\mathbf{E}^\mu$ - connecting an attribute node to its container relation node. A membership edge μ between an attribute A^i_j and a relation R_i formally is: $e_\mu : R_i \xleftarrow{\mu} A^i_j$.
- *selection edges*, $\mathbf{E}^\sigma$ - connecting each relation to each of its attributes. A selection edge σ between a relation R_i and its attribute A^i_j represents a possible selection of tuples from R_i based on a condition that involves A^i_j. Formally: $e_\sigma : R_i \xrightarrow{\sigma} A^i_j$ or $e_\sigma : R_i \xleftarrow{\sigma} A^i_j$.
- *predicate edges*, $\mathbf{E}^\theta$ - emanating from an attribute node and ending at another attribute node. A predicate edge θ between two attributes A^i_j and A^m_k represents a potential join between two relations R_i and R_m using the attributes. Formally: $e_\theta : A^i_j \xrightarrow{\theta} A^m_k$.

Therefore, the database schema graph is a directed graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$, where $\mathbf{V} = \mathbf{R} \cup \mathbf{A}$ and $\mathbf{E} = \mathbf{E}^\mu \cup \mathbf{E}^\sigma \cup \mathbf{E}^\theta$.

For our examples, we consider an example course database depicted in Figure 1. Figure 2 shows how a join between two relations, Students and StudentHistory, is captured on the database graph: we can start from Students and join to StudentHistory using the path Students $\xrightarrow{\sigma}$ SuID $\xrightarrow{\theta}$ SuID $\xrightarrow{\sigma}$ StudentHistory or vice versa using the path StudentHistory $\xrightarrow{\sigma}$ SuID $\xrightarrow{\theta}$ SuID $\xrightarrow{\sigma}$ Students. This example shows how our query graph representation captures the query semantics (in contrast to other query representations [6], [7]): operationally the two paths may be equivalent, e.g., in the case of equi-joins. Semantically they may have different translations. As we will see in Section IV, for the same join between two relations, we may choose one path over the other (e.g. choose between "courses taken by the students" or "students have passed courses") depending on the query and the translation.

B. Query Graphs

We first consider SPJ queries and then we extend our graphs to handle queries that contain query elements, such as functions and groupings, as well as subqueries.

A SPJ query q is represented by its *query graph* $G_q(V_q, E_q)$, a directed graph that is an extension of the database graph. Nodes in V_q are: (a) *relation* nodes - one for each relation and tuple variable in the query; (b) *attribute* nodes - one for each attribute in the query, possibly repeated if the attribute is found in different parts of the query; and (c) *value* nodes - one for each value or a set of values specified in the query qualification. Edges in E_q are defined as follows:

- *membership edges*: for each attribute A^i_j projected from a relation R_i, there is a membership edge: $e_\mu : R_i \xleftarrow{\mu} A^i_j$.
- *predicate edges*: for each predicate of the form $A^i_j \ \theta \ \Omega$, where Ω can be a single value or a set of values or an attribute, and θ denotes a comparison operator (e.g., $=$, $<$, $>$, $<>$ and $LIKE$), there is a predicate edge. We distinguish two cases: If Ω is a single value or a set of values, then it is a selection predicate edge: $e_\theta : A^i_j \xrightarrow{\theta} \Omega$. If Ω is an attribute A^m_k, then it is a join predicate edge: $e_\theta : A^i_j \xrightarrow{\theta} A^m_k$. In this case, we also capture the inverse direction: $e_{\theta'} : A^i_j \xleftarrow{\theta} A^m_k$, where θ' is the inverse of θ (e.g, if θ is $>$ then θ' is $\leq$).
- *selection edges*: for each predicate of the form $A^i_j \ \theta \ \Omega$, where Ω is a value (or set of values), there is a selection edge from its container relation R_i to A^i_j: $e_\sigma : R_i \xrightarrow{\sigma} A^i_j$. If Ω is an attribute, then there is also $e_\sigma : R_i \xleftarrow{\sigma} A^i_j$.

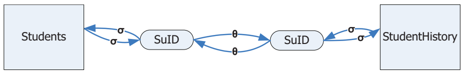


Fig. 2. A join on database graph

Example 1 Let us consider the following query.

```
select   s.name, s.GPA, c.title, i.name, co.text
from     students s, comments co
         studenthistory h, courses c, departments d,
         coursesched cs, instructors i,
where    s.suid = co.suid and
         s.suid = h.suid and h.courseid = c.courseid and
         c.depid = d.depid and
         c.courseid = cs.courseid and cs.instrid = i.instrid and
         s.class = 2011 and co.rating > 3 and
         cs.term = 'spring' and d.name =' CS'
```

Its graph representation is shown in Fig. 3. We observe that each join in the query is mapped to two paths with inverse directions between the relations joined. We also observe how a condition involving an attribute and a value, e.g., s.class $= 2011$ is captured as a path composed of selection and predicate edges, like Students$\xrightarrow{\sigma}$ class $\xrightarrow{=}$ 2011.

To capture functions, expressions, and renaming operations as well as order-by, group-by and having clauses, we extend the query graph with the following edge and node types:

- *function nodes*: A function node f is used for representing a function, an expression or a renaming operation that is applied on an attribute A^i_j or a set of attributes.
- *transformation edges*: A transformation edge r is used for connecting an attribute A^i_j with a function f that is applied to A^i_j. If A^i_j is in the select clause of the query, then the edge is defined: $e_r : A^i_j \xleftarrow{r} f$. If A^i_j is in the where clause of the query, then the edge is defined: $e_r : A^i_j \xrightarrow{r} f$.
- *order edges*: An order edge o is used for representing an ordering. If the query results are ordered based on the attributes A^i_j, A^k_l, ... (in that order), then we consider a set of order edges, the first one starting from the container relation R_i to A^i_j ($e_o : R_i \xrightarrow{o} A^i_j$), and each of the remaining ones starting from each attribute and ending at the subsequent in the order attribute ($A^i_j \xrightarrow{o} A^k_l$, ...). o shows if it is an ascending or descending order.
- *grouping edges*: A grouping edge γ is used to represent a grouping. If the grouping attributes are A^i_j, A^k_l, ... (in that order), then we consider a set of grouping edges, the first one starting from the container relation R_i to A^i_j ($e_\gamma : R_i \xrightarrow{\gamma} A^i_j$), and each of the remaining ones starting from each attribute and ending at the subsequent in the grouping attribute ($e_\gamma : A^i_j \xrightarrow{\gamma} A^k_l$, ...).
- *having edges*: a having edge h is used to show attributes in having clauses. For each participating attribute A^i_j of a relation R_i, there is an edge: $e_h : R_i \xrightarrow{h} A^i_j$.

Example 2 Let us consider the following query.

```
select     year, term, max(grade)
from       studenthistory
group by   year, term     having    avg(grade) > 3
```

Its graph representation is shown in Fig. 4(a). The grouping attributes are year and term, hence there are two grouping

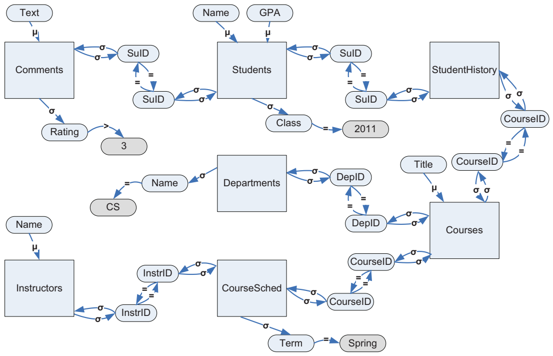


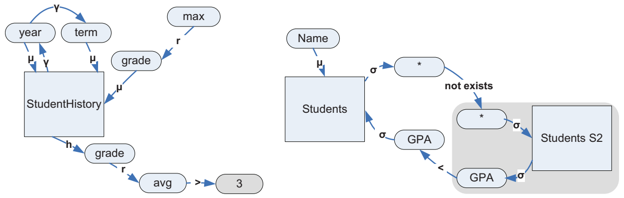
Fig. 3. A SPJ query

edges γ, one from the relation where year belongs and the other from year to the next attribute in the grouping order, i.e., term. The projecting attributes are year, term and grade; but the latter is an aggregated attribute, which is connected with a transformation edge to an aggregate function node max. Moreover, the attribute grade aggregated (with a different function here) is in the having clause. This is captured as a path involving a having edge, connecting the attribute with its relation, a transformation edge, connecting the attribute with the function node avg, and a predicate edge connecting the function node with the value. If a function involved more than one attribute, then more than one attribute node will be connected to the same function node in the query graph through transformation edges. Finally, we use two copies of the attribute grade depending on its role for making the example clear. We could have one instance of this attribute.

We now consider queries with nesting. A parenthesized select-from-where statement (subquery) can be used in a number of places: in a from clause, where it is treated as a table that is joined to other tables in the query, in a select, where it is treated as a set of attributes to be projected or in a where or having clause, where it can be treated as a list of values or a single value that participates in a predicate in this clause. We consider that in a predicate of the form $A^i_j\ \theta\ \Omega$, θ denotes a comparison operator (e.g., $=$, $<$, ...), or *a set comparison operator*, such as $(NOT)EXISTS$, $(NOT)IN$, $\theta' ANY$ and $\theta' ALL$, where θ' is a comparison operator.

Given a query q (the "parent" query), *each subquery block* q_m in q is represented as a separate query subgraph. This subgraph is treated as a "virtual" relation and it is connected to the parent graph depending on its position as follows:

- Each predicate in q of the form $A^i_j\ \theta\ q_m$, where q_m *returns* an attribute A^m_k, is represented as a path connecting the attribute A^i_j with its relation R_i through a selection edge and with the respective A^m_k through a predicate edge that starts from A^i_j and ends at A^m_k, i.e.: $R_i \xrightarrow{\sigma} A^i_j \xrightarrow{\theta} A^m_k$.
- Each predicate of the form $A^i_j\ \theta\ A^m_k$ in q where A^m_k is an attribute *returned* by the subquery q_m and A^i_j belongs to a relation in the parent query is represented as usual.
- Each predicate of the form $A^i_j\ \theta\ A^m_k$ *defined in the subquery* q_m, where A^m_k is an attribute defined in the scope of q_m and A^i_j is an attribute defined outside q_m (i.e., in the



(a) A group-by query (b) A nested query

Fig. 4. Example queries

query q) is represented as a path connecting the attribute A_k^m with its relation R_m through a selection edge and with the respective A_j^i through a predicate edge that starts from A_k^m and ends at A_j^i and A_j^i with its relation R_i in q with a selection edge, i.e.: $R_m \xrightarrow{\sigma} A_k^m \xrightarrow{\theta} A_j^i \xrightarrow{\sigma} R_i$.

A query in the where-clause is an example of the first case above, unless queries are correlated; then it is the last case. A query in the from-clause is an instance of the second case.

Example 3 Let us consider the following query.

```
select   s.name    from    students s
where    NOT EXISTS  (select *  from students s2
                                  where s2.GPA > s.GPA)
```

Figure 4(b) shows the query graph for this query. This example covers the first and third cases: the subquery is in the where-clause and it references an attribute of the parent query (i.e., correlated queries). That is why we observe the two paths between the two relation nodes: one going towards the subquery (for the first case) and one coming out of it (for the third case). Here we see that if θ is $(NOT)EXISTS$, the subquery q_m does not return any attributes. In that case, we use dummy attribute nodes, one connected to the query subgraph and one connected to the parent query graph. This example also shows how multiple instances of a relation each one corresponding to a different tuple variable over the relation are mapped in the query graph.

III. Capturing Query Semantics

In this section, we describe a template mechanism that allows us to represent semantics of query graph elements.

Labels. Each node v that can be part of a query graph over a database D has a *conceptual* meaning. For example, the conceptual meaning of a relation node represents its entity type; e.g., for Students the conceptual meaning is ‘students’. The conceptual meaning of a function captures its outcome (e.g., the function max represents “the greatest” of its input.) For expressions or unknown functions, we consider default labels, such as “an expression on” or “a function of”.

We define as the *label* l of a node v the conceptual meaning of the node, and we denote it as $l(v)$. For example, the label l(Name) of the attribute node Name may be “name”. Values are treated as literals, so for a value node val: $l(val) = val$.

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 nodes. For example, each membership edge from an attribute A_j^i to its container relation R_i is annotated by a label that signifies the meaning of the relationship of A_j^i with R_i’s conceptual meaning. Referring to Fig. 3, the membership edge connecting Students to its attribute Name may have the label “of”, and the predicate edge participating in the join of Students and Courses (in this direction) may have the label “have taken”. Fig. 5 shows example labels for the graph of Fig. 3.

Labels are stored on the database graph for both nodes and edges. A query graph inherits these edges from the database graph. Node labels can be automatically extracted from the names of database constructs using schema matching and entity resolution techniques. As a second step (or even first when such names are not meaningful), the system designer should correct or complement these findings. 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.

Templates. Our translation methods (Section IV) traverse the query graph and create phrases by composing labels found on the way. For producing more natural results, we define template labels at different granularity levels and we provide an extensible template mechanism to fuse these labels.

A *template label*, $l((v, u))$, is assigned to an edge (v, u) or, if it is more generic, 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 generic template label may have the form:

$$l((v, u)) = expr_1 + l(v) + expr_2 + l(u) + expr_3 \qquad (1)$$

where $expr1$, $expr2$, $expr3$ are alphanumeric expressions and the operator “$+$” acts as a concatenation operator. For using or registering template labels, we use a template language (based on [8]) that supports variables, loops, functions, and macros. Example macros implemented in our system, are:

(a) $l_M(v)$, which creates a phrase containing information of all template labels involving the membership edges of v (if any); i.e., $l((x, v)), \forall edge\ (x, v) \in E^\mu$.

(b) $l_V(v)$, which creates a phrase containing information of all template labels involving the paths starting from v and ending to its values (if any); i.e., $l((v, y))$, $\forall (v, y) \in E^\sigma$, $(y, z) \in E^\theta, z$ is a value node.

(c) $l_{MV}(v)$ = $l_M(v)$ + $expr1$ + $l(v)$ + $expr2$ + $l_V(v)$; this macro provides a *full translation* of v, in the sense that it translates anything related to v.

We consider templates of two types: *generic* and *specific*. The former are defined on edges and are constructed automatically following the form (1). Example generic templates are depicted in Table I. A generic template is essentially database-agnostic. Applying $l(e_\sigma)$ template to Students $\xrightarrow{\sigma}$ Name gives the label “students whose name”. Templates for predicates, $l(e_\theta)$, contain labels for operators, $l(\theta)$ (Table I). Also, for ensuring the extensibility of templates, we encourage the use of template variables. For example, for combining a sentence with a noun phrase, we use the variable $CONJ\_NOUN$ that coordinates this conjunction. Table I shows the default values used in our implementation. The designer can change these values globally or change a value corresponding to a subset of constructs (e.g., only the label of a specific e_μ).

TABLE I

EXAMPLE LABELS FOR TEMPLATE CONSTRUCTS

Operators	Translation	Translation variables	Translation	Template labels	Description
$l(=)$	'is'	VAL_SEL	"whose"	$l(e_\sigma)$	$l(R_i)$+VAL_SEL+$l(A_j^i)$
$l(\leq)$	"does not exceed"	$COORD\_CONJ$	"and"	$l(e_\theta)$	$l(A_j^i) + l(\theta) + l(\Omega)$
$l(>)$	"greater than"	$CONJ\_NOUN$	"that"	$l(e_\mu)$	" the " + $l(A_j^i)$ + " of " + $l(R_i)$
$l(LIKE)$	"looks like'	$CONJ\_PROJ, CONJ\_SEL$	"and"	$l(R_i \xrightarrow{\sigma} A_j^i \xrightarrow{\theta} A_j^i \xrightarrow{\sigma} R_i)$	$l(R_i)$ + " with the same " + $l(A_j^i)$

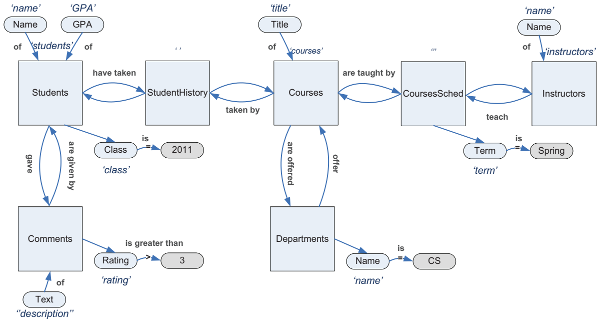


Fig. 5. Our running example with labels

Specific templates can be defined not only on edges, but on paths as well. These are created manually by a human, and can produce high-quality, concise text. Hence, when they exist, they should be preferred for query translation. As a short example, a specific template for the selection edge $e_\sigma(\mathsf{Students}, \mathsf{Name})$ may be the following: $l(e_\sigma(\mathsf{Students}, \mathsf{Name})) = l(\mathsf{Students}) +$ " named ".

Note, template labels follow the direction of edges; thus, if between two nodes there exist two edges with inverse directions, then template labels may be assigned to both.

IV. QUERY TRANSLATION

In this section, we present algorithms for the translation of SPJ queries. We first discuss the selection of the query subject, i.e., the primary entity of interest in a query, and then we present three strategies for translating the information we wish to know about the subject and the subject qualification, i.e., which particular entities are of interest. The query described in the *Example 1* and its query graph (Fig. 3) will serve as a reference example. Fig. 5 depicts a simplified version of the graph annotated with labels, where each join path has been replaced by a single "virtual" edge. In Section IV-D, we extend the translation algorithms to grouping and ordering semantics.

A. Query Subject

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. Unfortunately, when more than one relation projects attributes, the query subject cannot be determined solely based on the select-clause, due to the limited semantics of SQL. For example, a request for the names of students and the titles of the courses they took and a request for the titles of the courses and the names of the students that took these courses are both expressed with the same SQL query.

Definition 1: Primary relation R_P. A relation storing information for a set of entities of the same type is called *primary*.

Definition 2: Secondary relation R_S. A relation that stores information for a relationship of entities that are stored in different relations is called *secondary*.

For example, referring to Fig. 5, $\mathsf{Students}$ is a primary relation, whereas $\mathsf{StudentHistory}$ shows how courses and students connect and is secondary. Primary relations can be identified either by the designer or inferred during the construction of the database schema from an E/R diagram (entities in a E/R diagram make primary relations). The rest are secondary. Primary relations whose attributes are projected in the query result, are candidates for query subject. Intuitively, since the query subject is a reference point around which the query explanation is built, it is reasonable to select one that is "central" in the query graph, so that all references to it can be as short and concise as possible. A formal definition follows.

Consider a query q and its query graph $G_q(V_q, E_q)$. $\mathbf{R}$ is the set of nodes corresponding to the query relations. The distance $\delta(R_i, R_x)$ between two relations R_i and R_x on the graph G_q is the length of the shortest path between the two relations. Since in our context query graphs are typically connected it holds that $\delta(R_i, R_x) > 0, \forall R_i, R_x \in \mathbf{R}, R_i \neq R_x$.

Definition 3: Query subject, R_q. The query subject is a primary relation $R_q \in \mathbf{R}$ with attributes projected in q s.t.:
$\underset{R_x \in \mathbf{R}}{max}(\delta(R_q, R_x)) \leq \{\underset{R_x \in \mathbf{R}}{max}(\delta(R_i, R_x)) : \forall R_i \in \mathbf{R}\}$.

In Fig. 5, primary relations with projected attributes in the query are $\mathsf{Students}$, $\mathsf{Courses}$, $\mathsf{Comments}$, and $\mathsf{Instructors}$. The longest path of each one of them has length: 12, 9, 15, 15, respectively (recall that each "virtual" edge between two relations contains 3 edges). $\mathsf{Courses}$ has the minimum longest path to a relation on the graph and becomes the query subject.

1) QSUB Algorithm: The algorithm for selecting the query subject computes for each primary relation with attributes projected in the query, the shortest paths on the query graph to all reachable relation nodes and the resulting distances performing a breadth-first traversal of the graph. The length of the shortest path between each pair of nodes R_i and R_x in the graph is stored in a distance matrix D in $D[R_i][R_x]$. Then, for each primary relation with projected attributes, the longest path distance is found and the query subject is the relation with the shortest longest path. We resolve ties by preferring a relation with more attributes projected. If more than one candidate meets the criteria, we pick one or show alternative translations using different query subjects. If there is no primary relation in the query (i.e., the query involves only one secondary relation), then we use as query subject a primary relation of the database graph, which is the closest to the query relation (details are provided in [9].)

$$v \xrightarrow[\sigma]{e} t \xrightarrow{eo} to \qquad\qquad v \xleftarrow[ei]{\mu} ti$$

Fig. 6. Cases examined in BST

B. Query graph traversal

1) BST *Algorithm:* Our first strategy composes separate clauses for each part of the query. First, it translates the membership edges (clause $pStr$), then it connects all query relations to the subject through the joins in the query (clause $fStr$), and finally it reads the paths that connect relations to value nodes that are specified for attributes on these relations (clause $wStr$). The translation is performed in a depth-first way on the query graph $G_q(V_q, E_q)$ starting from the query subject R_q to all relations through the joins on the graph. The generated clauses are enriched with a few descriptive expressions (e.g., 'Find', 'for', etc.) and combined to the final text, as follows:

'Find ' + $pStr$ + ' for ' + $fStr$ + '.' +
' Return results only for ' +$wStr$ +'.'

BST is a recursive algorithm with inputs a graph $g(V_g, E_g)$ and a root node v. It also takes as input the strings $pStr$, $fStr$, and $wStr$ that comprise its output as well. The first time it is called, its initial inputs are $G_q(V_q, E_q)$ and the query subject R_q (details can be found in [9]).

For creating the latter two, we need to translate paths that start from the v node (ln:3). Hence, we examine the outgoing edges $e = (v, t)$ of v (see Fig. 6(left)). For looking for paths that end at value nodes, we examine the outgoing edges $eo = (t, to)$ of the v's neighbor t. If to is a value, then the translation of this path $v \dashrightarrow to$ contributes to $wStr$ (ln:3.1). Referring to Fig. 5, if v=Students and t=CLASS, then (ln:3.1.1) str = 'students'+VAL_SEL+'class'+' '+'is'+' '+'2011' = 'students whose class is 2011'. VAL_SEL is used for automating the conjunction of a noun ('student') to a noun clause. Since there may exist more than one such path from v, we create a single clause for each one and then, we combine the clauses with a coordinating conjunction $CONJ\_SEL$ (e.g., 'and').

If to is not a value, then we are interested only in the edge e. If we haven't visited t so far and if e is a selection edge, i.e., $e \in E_g^\sigma$, the path $v \dashrightarrow t$ represents part of a join and thus, it contributes to $fStr$ (ln:3.1.2). The translation of this part is stored in $fStr$ and then, (ln:4) we enrich the labels of v's children with appropriate expressions, so when these children are about to be translated their produced phrases will be combined appropriately with the existing one. In particular, for translating the contents of query's from-clause, we enrich the produced clauses with suitable words for smoothing the conjunction of a main phrase to a noun clause (i.e., $CONJ\_NOUN$ – for example 'that') or for combining equivalent phrases (i.e., $COORD\_CONJ$ – for example 'and') (ln:4). Referring again to Fig. 5, if v=Courses, then t can be Courses's ids used for joining eventually Courses with Instructors and Departments (and Students too). Thus, the $fStr$ is created as follows:

v=Courses, $fStr$ = 'courses'
v=Instructors, $fStr$ += 'that are taught by instructors'
v=Departments, $fStr$ += 'and are offered by departments'

and finally: 'courses that are taught by instructors and are offered by departments'.

```
Algorithm BST
Input:   node $v$, graph $g(V_g, E_g)$, list $open$, list $close$,
         and clauses $pStr$, $fStr$, and $wStr$
Output:  clauses $pStr$, $fStr$, and $wStr$
Begin
0.      $sel\_edges = 0$; $children = \oslash$;
1.      $close \leftarrow v$;
2.      If ($v \notin R_S$) $fStr$ += $l(v)$;
3.      Foreach edge $e\ (v,t) \in E_g$, $t \in V_g$ {
3.1       Foreach edge $eo\ (t,to) \in E_g$, $to \in V_g$ {
3.1.1       If ($to\ is\ a\ value$) {
              $str$=$l(v)$+$VAL\_SEL$+$l(t)$+' '+$l(eo)$+' '+$l(to)$+' ';
              $wStr$ += $make\_lbl(wStr, str, CONJ\_SEL)$;
            }
3.1.2       If (($t \notin close$) && ($to\ is\ not\ a\ value$)) {
              If ($e \in E_g^\sigma$) $sel\_edges$++;
              $children \leftarrow t$;
              $fStr = fStr + l(e)$+" ";
3.2.    } } }
4.      While ($children \neq \oslash$) {
4.1       $tv \leftarrow children.pop()$;
4.2       If  (--$sel\_edges > 0$)  $l(tv)$ += $COORD\_CONJ$;
4.3       Else If ($sel\_edges = 0$)  $l(tv)$ += $CONJ\_NOUN$;
4.4       $open \leftarrow tv$;
4.5     }
5.      Foreach edge $ei\ (si,v) \in E_g$, $si \in V_g$
5.1       If ($ei \in E_g^\mu$)    $children \leftarrow (l(si), l(ei))$;
6.      $str$ = '';
7.      While ($children \neq \oslash$) {
7.1       $(x,y) \leftarrow children.pop()$;
7.2       $str$ += ' the ' + $x$ + ' ' + $y$ + ' ' + $l(v)$ ;
7.3       If ($sizeof(children) \neq 1$)   $str$ += ', ';
7.4     }
8.      If ($str \neq$'')   $make\_lbl(pStr, str, CONJ\_PROJ)$;
9.      If ($open \neq \oslash$) {
9.1       $v \leftarrow open.pop()$;
9.2       BST($v$,$g$,$open$,$close$,$fStr$,$pStr$,$wStr$);
9.3     }
End
    $make\_lbl(clause, label, def)$ {
      If ($clause$ ='') return $clause = label$;
      Else return $clause$ += $def$ + $label$;
    }
```

Fig. 7. BST: dfs-like query translation in three steps.

Before leaving v, we examine its incoming edges $ei = (ti, v)$ (ln:7) (see Fig. 6(right)). If ei is a membership edge, i.e., $e \in E_g^\mu$, then it contributes to $pStr$. All incoming edges ei are stored in $children$, in order to find their actual number. Then, $pStr$ is created using conjunctive expressions (e.g., 'and') in appropriate places (ln:7-8). Regarding Fig. 5, if v=Students, then ti can be both NAME and GPA. Then, $pStr$ can be 'the gpa of students, the name of students'. For smoothing results, we use a simple find-and-replace mechanism, termed $resolve\_common\_expressions$ (RCE) [8], that removes repeating information (not shown in Fig. 7 due to space limits). The final result will be: 'the gpa and name of students'.

This process is repeated until we have visited all nodes of the query graph. Regarding the example of Fig. 5, the final result (having used RCE too) will be:

> *"Find the title of courses, the name of instructors, the gpa and name of students, and the description of comments for courses that are taught by instructors, are taken by students that gave comments, and are offered by departments. Return results only for courses whose term is spring, students whose class is 2011, comments whose rating is greater than 3, and departments whose name is CS."*

```
Algorithm MRP
Input:   nodes $v$, $rp$, $u$, graph $G_q(E_q, V_q)$, lists $open$,
         $close$, $path$, and clause $cStr$
Output:  clause $cStr$
Begin
0.      $close \leftarrow v$;
1.      If $((u, v) \in E_q)$ $path.push\_back() \leftarrow (u, v)$;
2.      If ($v$ is RP) {
3.        $pr = rp$; $rp = v$;
4.        If $(\exists(a, v) \in E_q^{\mu}, a \in V_q)$ {
4.1.        $cStr$ += $l_{MV}(rp)$;
4.2.        While $(path \neq \oslash)$ {
4.2.1         $(x, y) \leftarrow path.pop\_back()$;
4.2.2         If $(x \neq pr)$ $cStr$ += $l(y, x)$ + $l_V(x)$;
4.2.3         If $(x = pr)$ $cStr$ += $l(y, x)$ + $l(x)$;
4.2.4     }}
5.        If $(\nexists(a, v) \in E_q^{\mu}, a \in V_q)$ {
5.1.        $cStr$ += $l(pr)$;
5.2.        While $(path \neq \oslash)$ {
5.2.1         $(x, y) \leftarrow path.pop\_front()$;
5.2.2         $cStr$ += $l(x, y)$ + $l_V(y)$;
5.2.3     }}
6.        $path = \oslash$;
7.      }
8.      Foreach $(v, t) \in E_q$
8.1       If $(t \notin visited)$ $open.push\_back() \leftarrow \{v, rp, u\}$;
9.      If $(open \neq \oslash)$ {
9.1       $\{v, rp, u\} \leftarrow open.pop\_back()$;
9.2       MRP($v$,$rp$,$u$,$G_q$,$open$,$close$,$path$,$cStr$);
9.3     }
End
```

Fig. 8. MRP: dfs-like query translation using reference points.

2) MRP *Algorithm:* Our second strategy blends all three types of information that BST considers individually. The challenge is to avoid creating extremely complex and lengthy phrases. For instance, observe the part of Fig. 5 that relates to the primary relations Students, Courses, and Instructors and assume Students as a starting point. Then, an attempt to translate this part at once produces the following long phrase:

> *"Find the names of students and the titles of the courses taken by these students and the names of the instructors that taught courses taken by these students"*

In order to avoid long, possibly unnatural, sentences, we semantically split the translation at multiple points, called *reference points*, RP, as in the example below:

> *"Find the names of students and the titles of the courses taken by these students and the names of the instructors that taught these courses"*

Starting the traversal of query graph from the query subject, we identify a subset of relations as reference points, as follows.

Definition 4: Reference point, RP. It is a relation on the query graph that satisfies at least one of the following properties: (a) RP is a primary relation with μ edges, or (b) RP is a branching point, i.e., a relation that connects to more than one relation through paths directed from this relation to the other relations, or (c) RP is a leaf relation, i.e., a relation with no outgoing paths to other relations on the query graph, or (d) the minimum distance of RP from the closest reference point is greater than a pre-defined threshold ψ.

The last property allows us to tune the semantic length of the resulting phrases by regulating the distance among the reference points, or equivalently, by regulating the number of reference points used in the translation.

The Multi Reference Points algorithm (MRP) translates a query graph $G_q(V_q, E_q)$ based on the notion of reference points (RP). MRP's inputs are the query subject R_q and the query graph G_q. However, due to its recursive nature, it uses the following parameters as well: v is the node being processed in each turn; rp the reference point for v; u the parent node of v; the lists $open$ and $close$ for storing the nodes to be visited and the already visited ones, respectively; the list $path$ for storing the edges between rp and v; and finally, the clause $cStr$ that stores the translation of G_q. $cStr$ is MRP's output.

MRP "collects" and combines projections, selection and join predicates as it traverses G_q. The following text shows MRP's effect on the query of Fig. 5.

> *"Find the title of courses for courses that are offered by departments whose name is CS, and also, the gpa and name of students for students whose class is 2011 and that have taken these courses, and also, the description of comments for comments whose rating is greater than 3 and that are given by these students, and also, the name of instructors that teach courses whose term is spring."*

MRP traverses the query graph in a dfs-like manner. Interestingly, although it traverses the graph following a certain direction, the actual translation is happening by flipping directions depending on where it stands. MRP just traverses the graph until it reaches a reference point. Then, it creates a phrase containing the translation of the subgraph connecting the previously met RP, *pr*, and the current one, *rp*. This subgraph may contain joining relations and paths that end at value nodes. We distinguish two cases: (a) If *rp* does not have μ edges, then it can be either a branching point, a leaf relation or its distance from *pr* is greater than the allowed threshold ψ. Then, the translation always follows a direction from *pr* to *rp*. (b) If *rp* has μ edges, the translation follows the direction from *rp* to *pr*. The algorithm's behavior is explained by the need to connect reference points correctly. In case (a), the new reference point rp is a "weak" point in the sense that it provides no information of interest to the query and hence it cannot "stand" by itself. Hence, we make the previous reference point *pr* to textually connect to rp. In case (b), rp has information of interest (i.e., projected attributes) and we can ask for this information and then link back to pr.

Going back to Fig. 5, for the sake of the example assume that Students is the starting point. Then, Students, Courses, and Instructors are RPs. If only Students and Instructors contained μ edges, then the translation directions would be: Students → Courses and Courses ← Instructors.

At the end, $cStr$ contains the MRP translation, which produces results as the one of the previous example. Observe that this example is enriched with extra words –placed in italics– that serve as coordinating conjunctions (e.g., "and"), conjunctions to noun clauses (e.g., "that"), and so on. For presentation simplicity, we have not overloaded Fig. 8 with such information as we did with BST. For example, one could add the expression " there " before $l(pr)$ in (ln:5.1) and before $l(x)$ in (ln:4.2.3).

C. Template Selection

The previous query translation algorithms compose generic templates on the edges of the query graph as they traverse the graph. In this section, we present a flexible algorithm that can

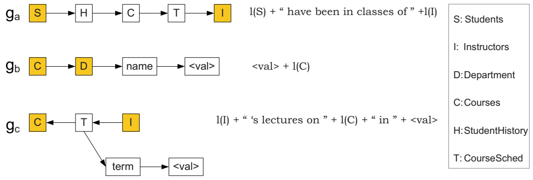


Fig. 9. Example graphs and their templates

build on-the-fly the best combination of generic and specific templates for a query graph based on how templates can be "glued" together. To illustrate the basic idea of the algorithm, imagine the query graph as a puzzle that we have to fill (cover) and we have various pieces (i.e., templates) of various sizes. The objective is to use as few pieces that can be glued together as possible (few big pieces will make a clearer picture, while many small pieces would pixelize the result).

A generic template is automatically defined on an edge whereas a specific one, provided by a designer, may be defined over a path or subgraph. Therefore, we consider that a template is assigned to a *template graph* g, which is a DAG, and has a set $R(g)$ of reference points, which are nodes on g. The template provides a translation of the relationship involving the reference points on g.

Definition 5: Composeable templates. Two templates g, with reference points $R(g)$, and g', with reference points $R(g')$, are *composeable* if $R(g) \bigcap R(g') \neq \emptyset$.

In words, the reference points are the points where two templates can be glued together. Two templates are composeable if they share at least one reference point.

Assume that the templates shown in Fig. 9 are defined for the example of Fig. 5 (colored nodes represent reference points). Observe that the graphs g_a, g_b are not composeable, since they refer to different things: students and instructors in the case of g_a and courses and departments for g_b. g_c is composeable with both g_a and g_b. Thus, the challenge is: given the fact that templates cannot be freely combined but they may still be combined if taken in the right order, then which templates to pick and in what order to "solve the puzzle".

Consequently, given a query graph $G_q(V_q, E_q)$ and a set of (both generic and specific) templates, we need to: (a) find the minimum set of composeable template graphs that cover the query graph and (b) compose them based on the query graph and the template "composeability". A set $\{g_1, g_2, ...g_N\}$ of template graphs are composeable, if for each $g_i, i = 1...N$, there is a $g_j, j = 1...N, i \neq j$, s.t. g_i and g_j are composeable. We first describe the algorithm TS for template selection.

1) TS algorithm: Formally, the template selection problem is defined as follows: Given a query q and its query graph $G_q(V_q, E_q)$, we consider the set $\mathbf{g}$ of template graphs that are contained in $G_q(V_q, E_q)$, i.e.,:

$\mathbf{g} = \{g_i(V_i, E_i) | G_q$ is a supergraph of $g_i, \; i = 1..N\}$.

A set $\mathbf{g_k} \subseteq \mathbf{g}$ *covers* $G_q(V_q, E_q)$ iff $\bigcup_{g_x \in \mathbf{g_k}} E_x = E_q$ (query graphs are connected, hence this condition also implies $\bigcup_{g_x \in \mathbf{g_k}} V_x = V_q$.) All sets that cover the query graph can be ordered by their size, i.e., $\mathbf{g_k} > \mathbf{g_m} \Leftrightarrow |\mathbf{g_k}| > |\mathbf{g_m}|$. We are interested in the set $\mathbf{g_t} \subseteq \mathbf{g}$ of composeable template graphs s.t. $\nexists \mathbf{g_m} \subseteq \mathbf{g}$ of composeable graphs with $\mathbf{g_m} > \mathbf{g_t}$.

```
Algorithm TS
Input:    query graph $G_q(V_q, E_q)$, index $I$
Output:   a minimum set of composeable graphs $\mathbf{g_t}$
Begin
0.    initialize $M[][]$
1.    Foreach $e$ in $E_q$ {
2.        use $I$ to retrieve $\mathbf{g_e}$;
3.        Foreach $g \in \mathbf{g_e}$ {
3.1           $M[e][g] \leftarrow 1;$
3.2   }       }
4.    Foreach column $g$ in $M$ {
5.        If $sum(M[][g]) = |E_q|$ {
5.1           $\mathbf{g_{con}}.push() \leftarrow g;$
5.2           $QP.push() \leftarrow (\{g\}, E_g, R(g));$
5.3   }
6.    While $(QP \neq \oslash)$ {
7.        $(\mathbf{g_{com}}, E_{sat}, R_{\mathbf{g}}) \leftarrow QP.pop\_front();$
8.        Foreach $(g(V_g, E_g) \in \mathbf{g_{con}}, g \notin \mathbf{g_{com}}, R(g) \cap R_{\mathbf{g}} \neq \oslash)$ {
8.1           $\mathbf{g}'_{\mathbf{com}} = \mathbf{g_{com}} \cup \{g\};$
8.2           $E'_{sat} = E_{sat} \cup E_g;$
8.3           $R'_{\mathbf{g}} = R_{\mathbf{g}} \cup R(g);$
8.4            If $(E'_{sat} = E_q)$ return $\mathbf{g}'_{\mathbf{com}}$;
8.5            Else $QP.push() \leftarrow (\mathbf{g}'_{\mathbf{com}}, E'_{sat}, R'_{\mathbf{g}});$
8.6   } }
9.    return $\oslash$;
End
```

Fig. 10. TS: algorithm for template selection.

Fig. 10 provides the algorithm for template selection. First, we find the template graphs that are contained in the query graph. We keep an inverted index I over template graphs. Given a query graph $G_q(V_q, E_q)$, we probe the index with the edges of E_q. For each edge $e \in E_q$, the index returns the list $\mathbf{g_e}$ of graphs that contain e. A graph g is contained in G_q, if g is found in n lists returned for the edges in E_q and $n = |E_q|$. To identify the qualifying graphs, we keep a matrix $M[][]$ with rows mapping to edges and columns mapping to the template graphs returned by I for the query. We set $M[e][g]$ to 1 if the index returns g for e (ln:3). We keep only graphs that correspond to columns in the matrix with sum of 1's equal to $|E_q|$ (ln:5). These graphs are inserted into a list $\mathbf{g_{con}}$ in decreasing order of the graph size (number of edges).

The next step is to find the minimum set of composeable graphs from $\mathbf{g_{con}}$ that cover G_q. This is a set covering problem but not all combinations of template graphs are valid, since we are interested in composeable sets. For this reason, the algorithm's strategy is to build solutions by combining the largest composeable template graphs. Solutions that cannot extend to the whole query graph are pruned.

A candidate solution is represented as a tuple $(\mathbf{g_{com}}, E_{sat}, R_{\mathbf{g}})$, where $\mathbf{g_{com}}$ is a set of composeable graphs, E_{sat} is the set of edges covered by these graphs, and $R_{\mathbf{g}}$ is the union of their reference point sets. The size of a solution is the size of $\mathbf{g_{com}}$, i.e., the number of graphs in $\mathbf{g_{com}}$. TS keeps a list QP of candidate solutions in increasing order of size. Same size solutions are ordered in decreasing $|E_{sat}|$. At each round, it picks the head of QP (ln:7) and generates solutions (if any) that extend this one with a graph from $\mathbf{g_{con}}$ (ln:8). All solutions are inserted into QP unless one covers all edges of the query graph. Then it is a minimum solution, and the algorithm terminates.

2) TMT algorithm: The algorithm TMT (Fig. 11) uses the set $\mathbf{g_t}$ of composeable templates returned by TS to generates a query translation ($cStr$) for a query graph G_q. The challenge

```
Algorithm TMT
Input:    query graph $G_q(V_q, E_q)$, index $I$
Output:   a clause $cStr$
Begin
0.    $\mathbf{g_t} \leftarrow \mathsf{TS}(G_q, I)$;
1.    $E_{\mathbf{g_t}} = \cup_i E_{g_i}, g_i \in \mathbf{g_t}$;
2.    Foreach relation $r$ s.t. $\nexists\ edge\ (x, r) \in \{E_{g_t} - E_q^{\mu}\}, x \in V_q$;
3.        $QP.push() \leftarrow r$;
4.    While $(QP \neq \oslash)$ {
5.        $r_i \leftarrow QP.pop\_front()$;
6.        $V_{r_i} = \{\}$; $E_{r_i} = \{\}$; $leaves.push() \leftarrow r_i$;
7.        Foreach $g \in \mathbf{g_t}$ with $root(g) \in leaves$ {
7.1           $V_{r_i}.push() \leftarrow R(g)$;
7.2           $E_{r_i}.push() \leftarrow g$;
7.3           $leaves.push() \leftarrow R(g) - \{root(g)\}$;
7.4           }
8.        $L \leftarrow biased\_topological(G_{r_i})$;
9.        While $(L \neq \oslash)$ {
9.1           $g \leftarrow L.pop\_front()$
9.2           $cStr_i$ += $l(g)$;
          } }
10.       $cStr$ += $cStr_i$ ;
11.   return $cStr$;
End
```

Fig. 11. TMT: algorithm for specific template composition.

is to find how to compose the templates on the query graph. To illustrate, for the query of Fig. 5 and the specific templates shown in Fig. 9, TS would have found that these templates combined with generic templates for the parts of the query not covered can be used for translating the query. The result (using appropriate auxiliary phrases) would be:

> *"Find the gpa and name of students whose class is 2011 and have been in classes of instructors and find the name of these instructors, whose lectures on courses are in spring and find the title of these CS courses and the description of comments whose rating is greater than 3 given by these students."*

We observe that using specific templates may generate smaller and more natural text. On the other hand, since specific templates can be arbitrary, none of the previous query translation algorithms that read and translate edges on the query graph can be used. TMT uses a dynamic strategy to find in what order the templates should be read and how they must be combined. For the example of Fig. 9, the right order to read the specific templates is g_a, g_c, g_b.

TMT proceeds as follows. It finds a root r_i on the query graph that is a relation node and has no incoming edges (except for possible membership edges) (ln:2). For example, Students is the only root for the templates above. With r_i as root, it builds a DAG $G_{r_i}(V_{r_i}, E_{r_i})$ by connecting template graphs found in $\mathbf{g_t}$ that can be composed with r_i (ln:5-7). Recall that each graph g is a DAG and can be seen as a "super edge" connecting the nodes in its set $R(g)$ of reference points (following the direction of the edges in g). Hence G_{r_i} is composed of such "super edges" (which in the case of generic templates are edges on the query graph but in the case of specific templates may map to subgraphs). In order to create this DAG, the algorithm gradually expands the graph (which initially contains only r_i) with graphs whose root is one of the current leaves of G_{r_i}.

When G_{r_i} is built, TMT performs a topological sort and stores G_{r_i}'s graphs (in the order produced) in a list L. The topological sort is biased to give from a relation priority to membership edges, then selection edges to values, and finally all other cases (ln:8). Then, TMT pops g_is out of Q and builds a phrase $cStr_i$; if a g_i has more than one child, it uses appropriate coordinating conjunctions (e.g., 'and') (not shown in the figure for simplicity) (ln:9).

Since more than one root r_i might exist (ln:2), possibly covering different parts of the graph, TMT constructs and translates a DAG G_{r_i} for each root and at the end concatenates the partial translation results, $cStr_i$'s, to $cStr$ (ln:10).

D. Discussion

The algorithms presented so far, cover SPJ queries (paths, nested, etc.) as discussed in Section II. For simplicity, we left out of the discussion the grouping and ordering query parts. These two are handled separately, since they apply to the whole translation result. For both parts we work as in BST and create the phrases $gStr$ and $oStr$ by simply following the paths of γ or o edges, respectively. (A slightly different case involves nested queries, where these phrases may blend into the rest of the translated text; however, due to space considerations we have omitted this part from the descriptions of our algorithms.) Similarly, we work for r edges and functions f (also omitted from our discussion).

In most examples we used queries having conjunctive predicates. Nothing changes for disjunctive predicates. Extra care should be taken for the choice of the words responsible for coordinating phrases conjunction (e.g., apart from "and" we need "or" too). Operators' priority should be considered too.

An interesting issue concerns the "corporate" queries, which contain large chains of joins, and most importantly many projected attributes. Although, there is no inherent problem with the function of the proposed algorithms, there is a question regarding the usefulness of such translations. Clearly, the produced text is not pretty, but neither the queries themselves are. Since such queries usually are used by people with advanced technical skills, a translation might not be that useful. However, we can leverage the power of templates for producing "query summaries", which can be used as starting points; e.g., for facilitating query documentation. Using TMT and appropriate templates (with macros) we can put limits on the number of projected attributes in the result. [8] defines the notion of the *heading* attribute, which represents the most characteristic attribute of a relation (e.g., the attribute $name$ for Students). Such attributes may be defined in the database graph. When a query involving many projected attributes comes, we could provide a first translation using the predefined attributes, so that the user would get insight into the query mechanism. Then, we can expand it according to user needs.

As a final remark, TMT has been proven especially useful in the case of queries that are difficult to translate [5]. For example, queries containing predicates like "having count(distinct year) = 1" and "where year < all (<subquery>)", require extra knowledge for capturing their semantics. The first case implies "all" (e.g., as in "find students whose classes are *all* in the same year") and the second implies "earliest" (e.g., as in "find students who have taken courses in *the earliest* years that have been taught").

V. Experiments

We use General SQL Parser, an off-the-shelf tool that reads SQL queries and generates an XML-representation of the parse tree (sqlparser.com). This is the input to our query translation module that converts it to a query graph. The query translator is implemented in C++ and makes use of the Boost library for graphs (boost.org). Our experimental study aims at shedding light on the *performance* as well as the *effectiveness* of the proposed methods changing various features of the queries:

- the *depth*: the ♯ of relations on the longest path in the query graph,
- the *out-degree*: the ♯ of edges emanating from a relation node pointing to other relations if query graph is converted to a DAG,
- the *μ-degree*: the total ♯ of membership edges,
- the *σ-degree*: the total ♯ of value nodes,
- the *compactness*: (♯ of relations)/(♯ of relations on the longest path in the query graph).

In addition, we studied the effect of: the query subject (on BST), the ♯ of reference points (on MRP) and the ♯ of templates (on TMT). We used the schema of the course database used in CourseRank (courserank.stanford.edu) comprising 20 relations with average number of fields around 5 [10].

A. Effectiveness

For these experiments, we recruited a few experts in SQL. We intentionally chose experts because they can judge whether an SQL query is translated well. We used different sets of queries, with the same features, for which we automatically generated query explanations with all algorithms. For MRP, we also set the default minimum distance between reference points to be 3 in order to ensure the use of reference points even when the query structure does not provide such points. For TMT, we have provided sufficient templates to make sure that query translation can exploit them in order to leverage its expressivity. In addition, we asked two experts to write an explanation for each query (USER). Then, we involved the other experts in the following experiments. The queries used in each one of them were different in order to allow users judge the results without any recollection of queries seen before.

1) SQL → NL: The first set of experiments (Fig. 12) investigates the direction from SQL to text and it has two parts. The first part measures the effectiveness of some particular choices of the algorithms, namely the selection of a query subject, the use of reference points, and the use of templates. The benefits of reference points and specific templates are also discussed in other experiments, so we will focus on the query subject and summarize our findings for the other two due to space considerations. The second part of the experiments compares the various translation strategies.

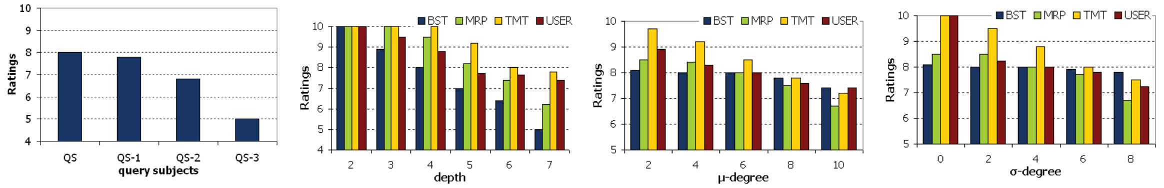
Fig. 12(a) shows the effect of query subject (QS). We considered a set of 10 path queries of *depth* equal to 10 with 6 of them being primary with one attribute projected. We translated them using BST with QS calculated by the algorithm and with manually set, alternative, query subjects that were 1, 2, ..., hops away (denoted $QS-1$, $QS-2$,...). For each query, the experts were presented with the results of all translations using the different QS's and rated them from 1 to 10. The figure confirms the intuition behind the selection of QS: a central QS collects much more information around it and generates more balanced clauses referring to it.

Fig. 12(b)-12(d) show results from the second part of the SQL → NL experiments. Each point in the figures represents the average of the user ratings for a set of 7 queries with the same characteristics. Each user was presented with a random subset of the queries used in each experiment and rated all possible query explanations (BST, MRP, TMT, and USER) from 1 to 10 depending on their comprehensibility and naturality (10 is the best).

Fig. 12(b) shows ratings for queries as a function of the ♯ of relations (*depth*: 2 to 7). First, let us observe that the ratings given to all query translations (even those user-generated) decrease as the *depth* increases. We observe similar trends when other aspects of the query (e.g., the *μ-degree* in Fig. 12(c)) change resulting in bigger, and more complex queries. Thus, the quality of translation is inevitably affected by the query size and complexity, which essentially means that even for humans it becomes very difficult to explain a query in a concise and elegant way. On the other hand, for simple queries, all translation methods provide very satisfactory results.

Focusing now on the effect of the *depth* in Fig. 12(b), we observe that BST is the most sensitive, because the longer the paths on the query graph it translates, the more unnatural the translation result is. On the other hand, the quality of TMT's translation is smoother because it can make use of specific templates that "explain" bigger parts of a graph and as a result while the query graph may grow, the algorithm can still find a translation combining fewer templates than the other algorithms. MRP's results provide a good compromise between TMT (which relies on human input) and BST (which is fully automated): it combats the "long-path" effect of BST by inserting intermediate reference points and breaking long paths to shorter segments. In a way, it tries to "mimic" the behavior of TMT by constructing templates (using of course generic ones) for bigger but digestible parts of the graph. As a final note, it is worth noting that the USER translation results for small queries most of the times tended to follow a BST-like approach, while for bigger queries they approach MRP and finally, they adopt a TMT-like approach trying to simplify parts of the query in order to make its description shorter. We observed similar trends for other query parameters. We think that this provides an indication that such a hybrid translation approach may be useful, and we intend to explore it as a continuation of this work.

Fig. 12(c) shows user ratings for queries as a function of the ♯ of projections (*μ-degree*:2 to 10). We consider two projections per relation (which explains why in the figure *μ-degree* increases by increments of 2). We observe that TMT (although it starts with better translations), it is affected by *μ-degree* because, although we have specific templates for the main body of queries, i.e., for covering joins, for projections, we cannot do much improvement. On the other hand, BST seems more immune to *μ-degree* and we observe



(a) d:10, *o-deg.*:1,μ*-deg.*:6, σ*-deg.*:0 (b) *out-deg.*:1, μ*-deg.*:1, σ*-deg.*:1 (c) d:5, *out-deg.*:1, σ*-deg.*:1 (d) d:5 *out-deg.*:1, μ*-deg.*:1,

Fig. 12. From SQL to NL

that for high values it has better results than the others which also coincide with USER. We observed in the user-generated texts that as queries had more projections they tend to adopt a BST strategy (i.e., first the describe the projections and at last the value selections).

Fig. 12(d) shows user ratings as a function of the σ*-degree*. We would like to highlight few points here. Overall, BST is the least affected simply because all value selections form a separate clause appended at the end of the query translation. For a certain number of value selections, MRP is not affected and actually it seems to blend them nicely in the query text. However, for many selections, the clean-cut solution of BST seems more preferable. We observe that USER ratings lie somewhere between BST and MRP ratings. We looked at the user-generated text and we saw that as σ*-degree* increases, the users were either trying to group selections at the end or they were trying a mixed strategy like MRP.

Observe that TMT translations often got higher ratings than USER translations. This is because we chose to evaluate the users' first translation attempts, which were based mostly on a SQL-driven translation. (In general, USER results get improved at users' second or third attempt.) On the other hand, by using appropriate templates in TMT, one may get more declarative, smoother translations. The effort to come up with appropriate, fine-tuned templates is a one-time effort worth to place as it is compensated by the effectiveness of TMT.

2) NL → SQL: This set of experiments looked at the inverse direction. Each person (except for those who provided query explanations) was presented in random order different explanations for the same set of SQL queries and was asked to write the SQL query. We measured the ♯ *of hits* (i.e., how many times they got the right SQL query). We also asked users to rate a text from 1 to 10 based on how easily they could build the corresponding SQL query (10 is the best rating). Fig. 13 and 14 show user ratings as a function of *depth* and μ*-degree*, respectively. (For space considerations, we do not show results for the σ*-degree* since we observed very similar trends.) These figures reveal that the text generated by BST suits better the purpose of finding which SQL query is described. BST groups projections, joins, and selections in a query in separate clauses and dictates how to write the SQL query. MRP also allows to easily write the query but in the presence of several projections it becomes harder to reconstruct the SQL query. TMT provides a higher level description of the query and makes it easier to catch the meaning but then one needs to work harder to map the semantic associations to real joins based on the underlying schema. It is also worth noting that (although not shown in the figures), we run experiments where we observed that reverse-engineering TMT text to SQL queries, may lead to equivalent but not exactly the same queries (e.g., in the case of nesting).

Fig. 15 shows the effect of *compactness* on the quality of query translations as expressed through user ratings. If it is equal to 1, then it shows a path query. The higher the compactness is the more branches the graph has. We considered queries of 12 relations of different compactness. Each subset of queries used for each compactness has 4 projections and four selections that are always distributed to the leave relations of the query graph. The purpose of this setting is to make it hard to pick a query subject on the query graph that is a central relation. (We have seen the effect of the query subject above.) We observe that the more compact the query graph becomes, BST translations improve thanks to the ability to select a more central query subject, and MRP translations are better due to the use of reference points and improve as they can also pick a better query subject. Finally, both generate equally good translations since for a very compact graph, MRP does not need to consider reference points. Interestingly, as the graph becomes compact, for our configuration, there were not too many specific templates to combine or they had large overlap.

Concluding: If one is willing to invest some effort on designing some specific templates, TMT can generate better translations. It can capture the important semantic associations between entities in the database providing an abstraction level that can hide the particularities of the schema, (such as normalization, re-structuring, and so on). If an automated method is preferred, we could apply a different technique (BST or MRP), depending on the purpose the text will serve (explaining a query or helping a user write the query himself) and depending on the query characteristics.

B. Performance

Independent of the type of the nodes and edges in the query graph, our experiments have shown that execution times depend mainly on the graph size, which is ♯ *of relation nodes* + ♯ *of attribute nodes* + ♯ *of value nodes*. (Other parameters that affect performance include the number of templates TMT has to process, but for the template database used we did not witness a significant overhead.) Fig. 16 shows times for the three algorithms in seconds. BST is the most efficient since it only makes one pass of the graph. MRP traverses parts of the graph that correspond to joins to both directions as it goes forth to find a reference point and back to connect it to the previous reference point. TMT reads the query graph edges more than once in order to find candidate templates and to find how to compose them.

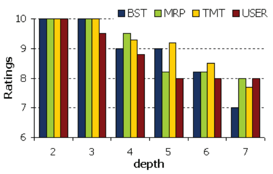


Fig. 13. NL to SQL (*out-deg.*:1, *μ-deg.*:1, *σ-deg.*:1)

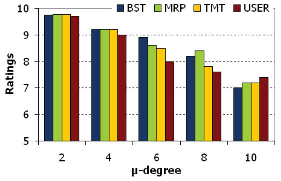


Fig. 14. NL to SQL (*depth*:5, *out-deg.*:1, *σ-deg.*:1)

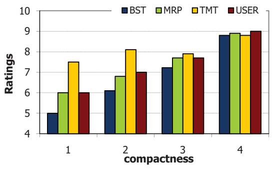


Fig. 15. NL to SQL (*depth*:12, *μ-deg.*:4, *σ-deg.*:4)

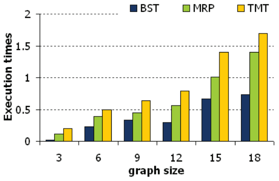


Fig. 16. Performance

VI. Related Work

NL and DB. Earlier interaction of DB and NL processing has focused mainly on the opposite direction of the one studied here, such as NL Querying [4], NL and Schema Design [11], NL and Database interfaces [12], and Question Answering [13]. In the past, we have worked on translating small databases with content or query answers under certain constraints [8]. We have also discussed the usefulness of translating SQL queries into narratives and examined the space of the problem [5]. The problem of query translation that we study in this paper is more difficult than content translation because the size and complexity of a query are essentially arbitrary with no upper bounds (whereas database contents necessarily follow the schema structure, which is bounded).

Query graph representations. Query graphs have been proposed for query optimization purposes [6], [14], [7]. For instance, the query graph model (QGM) defines a conceptually more manageable representation of an SQL query [7]. These models form the key structure for representing information relevant to query optimization and processing, such as operations and data flows. We are not interested in the operational aspects of a query (i.e., "how" an answer will be generated) but more in its semantics (i.e., "what" the query describes). Our query graph model captures query semantics by identifying the elements of a query and capturing their semantic associations.

Graph and set problems. Our template selection problem is divided into a graph containment and a graph cover subproblem. Graph containment problems have recently gained attention and indexes and pruning methods have been proposed for very large graph databases [15]. We follow an exact-match approach that works well for the size of (template) graph databases we consider. Our graph cover problem is: Given a (query) graph g and a set of subgraphs $\{g_1, g_2, ...\}$, find the minimum subset of composeable subgraphs that cover g. Two subgraphs are composeable if they share specific nodes. This problem differs from graph decomposition [16], [17], where a graph g is decomposed into a set of subgraphs, that have disjoint sets of edges and keep the structural properties of g. Viewing graphs as sets (of edges), our problem can be seen as a set cover problem [18], [19]. However, in our case, sets cannot be freely combined, and a set cover for g may not be an acceptable solution. Our algorithm is designed to take these constraints into account and finds a graph cover.

VII. Conclusions and Future Work

In this paper, we explored a new problem, translating SQL queries to text. We have described a model for representing various forms of structured queries as directed graphs and we have captured query semantics by annotating the graph edges with template labels using an extensible mechanism. We have mapped the query translation problem to a graph problem and presented different graph traversal strategies for efficiently exploring these graphs and composing textual query descriptions. Among them is an algorithm that can capture important semantic associations between entities in the database providing an abstraction level over the db schema with very promising results as our experiments have indicated.

This is the first effort towards structured query translation and it is an open field for research. We are interested in an adaptive method that can follow different translation strategies depending on the query characteristics. A related problem is to apply techniques for finding interesting or frequent associations in queries (for example, by mining query logs) in order to recommend to a designer for template assignment.

References

[1] C. Botev, N. Gupta, J. Shanmugasundaram, and F. Yang, "A WYSIWYG development platform for data driven web applications," in *VLDB*, 2008.

[2] K. Kowalczykowski, K. W. Ong, K. K. Zhao, A. Deutsch, Y. Papakonstantinou, and M. Petropoulos, "Do-it-yourself custom forms-driven workflow applications," in *CIDR*, 2009.

[3] Gartner, "Gartner identifies seven grand challenges facing IT," in *Gartner Emerging Trends Symposium - ITxpo*, 2008.

[4] E. Métais, J. Meunier, and G. Levreau, "DB schema design:A perspective from NL techniques to validation & view integration," in *ER*, 1993.

[5] A. Simitsis and Y. E. Ioannidis, "DBMSs Should Talk Back Too," in *CIDR*, 2009.

[6] U. Dayal, "Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers," in *VLDB*.

[7] H. Pirahesh, J. Hellerstein, and W. Hasan, "Extensible/rule based query rewrite optimization in Starburst." *SIGMOD Rec.*, vol. 21, no. 2, 1992.

[8] A. Simitsis, G. Koutrika, Y. Alexandrakis, and Y. E. Ioannidis, "Synthesizing structured text from logical database subsets," in *EDBT*, 2008.

[9] G. Koutrika, A. Simitsis, and Y. E. Ioannidis, "Explaining structured queries to users," Available at: Infolab Pub Server, Tech. Rep., 2009.

[10] B. Bercovitz, F. Kaliszan, G. Koutrika, H. Liou, Z. M. Zadeh, and H. Garcia-Molina, "Courserank: A social system for course planning," in *SIGMOD*, 2009.

[11] V. C. Storey, R. C. Goldstein, and H. Ullrich, "Naïve semantics to support automated database design," *IEEE TKDE*, vol. 14, no. 1, 2002.

[12] M. Minock, "A phrasal approach to natural language interfaces over databases," in *NLDB*, 2005.

[13] E. Sneiders, "Automated question answering using question templates that cover the conceptual model of the database," in *NLDB*, 2002.

[14] O. Hartig and R. Heese, "The SPARQL query graph model for query optimization," in *ESWC*, 2007.

[15] C. Chen, X. Yan, P. S. Yu, J. Han, D.-Q. Zhang, and X. Gu, "Towards graph containment search and indexing," in *VLDB*, 2007.

[16] D. Eppstein, "Subgraph isomorphism in planar graphs and related problems," *J. of Graph Algorithms and Applications*, vol. 3, no. 3, 1999.

[17] D. W. Williams, J. Huan, and W. Wang, "Graph database indexing using structured graph decomposition," in *ICDE*, 2007.

[18] B. Gao, M. Ester, J. Cai, O. Schulte, and H. Xiong, "The min. consistent subset cover problem and its applications in data mining," in *KDD*, 2007.

[19] D. Johnson, "Approximation algorithms for combinatorial problems," *JCSS*, vol. 9, 1974.