Provenance for Natural Language Queries

Daniel Deutch
Tel Aviv University
danielde@post.tau.ac.il

Nave Frost
Tel Aviv University
navefrost@mail.tau.ac.il

Amir Gilad
Tel Aviv University
amirgilad@mail.tau.ac.il

ABSTRACT

Multiple lines of research have developed Natural Language (NL) interfaces for formulating database queries. We build upon this work, but focus on presenting a highly detailed form of the *answers* in NL. The answers that we present are importantly based on the *provenance* of tuples in the query result, detailing not only the results but also their *explanations*. We develop a novel method for transforming provenance information to NL, by leveraging the original NL query structure. Furthermore, since provenance information is typically large and complex, we present two solutions for its effective presentation as NL text: one that is based on provenance factorization, with novel desiderata relevant to the NL case, and one that is based on summarization. We have implemented our solution in an end-to-end system supporting questions, answers and provenance, all expressed in NL. Our experiments, including a user study, indicate the quality of our solution and its scalability.

1. INTRODUCTION

Developing Natural Language (NL) interfaces to database systems has been the focus of multiple lines of research (see *e.g.* [35, 4, 34, 48]). In this work we complement these efforts by providing *NL explanations to query answers*. The explanations that we provide elaborate upon answers with additional important information, and are helpful for understanding *why* does each answer qualify to the query criteria.

As an example, consider the Microsoft Academic Search database (`http://academic.research.microsoft.com`) and consider the NL query in Figure 1a. A state-of-the-art NL query engine, `NaLIR` [35], is able to transform this NL query into the SQL query also shown (as a Conjunctive Query, which is the fragment that we focus on in this paper) in Figure 1b. When evaluated using a standard database engine, the query returns the expected list of organizations. However, the answers (organizations) in the query result lack *justification*, which in this case would include the authors affiliated with each organization and details of the papers they have published (their titles, their publication venues and publication years). Such additional information, corresponding to the notion of *provenance* [28, 7, 9, 25, 26] can lead to a richer answer than simply providing the names of organizations: it allows users to also see relevant details of the qualifying organizations. Provenance information is also valuable for validation of answers: a user who sees an organization name as an answer is likely to have a harder time validating that this organization qualifies as an answer, than if she was presented with the full details of publications.

```
return the organization of authors who published papers
in database conferences after 2005
```

(a) NL Query

```
query(oname) :- org(oid, oname), conf(cid, cname),
pub(wid, cid, ptitle, pyear), author(aid, aname, oid),
domainConf(cid, did), domain(did, dname),
writes(aid, wid), dname = 'Databases', pyear > 2005
```

(b) CQ Q

Figure 1: NL Query and CQ Q

```
TAU is the organization of Tova M. who published
'OASSIS...' in SIGMOD in 2014
```

Figure 2: Answer For a Single Assignment

We propose a novel approach of presenting *provenance information for answers of NL queries, again as sentences in Natural Language*. Continuing our running example, Figure 2 shows one of the answers outputted by our system in response to the NL query in Figure 1a.

Our solution consists of the following key contributions.

Provenance Tracking Based on the NL Query Structure. A first key idea in our solution is to leverage the *NL query structure* in constructing NL provenance. In particular, we modify `NaLIR`[1] so that we store exactly which parts of the NL query translate to which parts of the formal query. Then, we evaluate the formal query using a provenance-aware engine (we use `SelP` [16]), further modified so that it stores which parts of the query "contribute" to which parts of the provenance. By composing these two "mappings" (text-to-query-parts and query-parts-to-provenance) we infer which parts of the NL query text are related to which provenance

[1]We are extremely grateful to Fei Li and H.V. Jagadish for generously sharing with us the source code of `NaLIR`, and providing invaluable support.

Proceedings of the VLDB Endowment, Vol. 10, No. 5

parts. Finally, we use the latter information in an "inverse" manner, to translate the provenance to NL text.

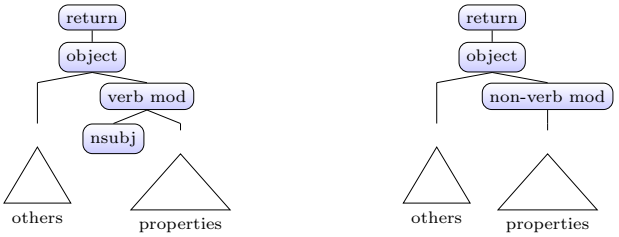
Factorization. A second key idea is related to the provenance size. In typical scenarios, a single answer may have multiple explanations (multiple authors, papers, venues and years in our example). A naïve solution is to formulate and present a separate sentence corresponding to each explanation. The result will however be, in many cases, very long and repetitive. As observed already in previous work [8, 42], different assignments (explanations) may have significant parts in common, and this can be leveraged in a *factorization* that groups together multiple occurrences. In our example, we can *e.g.* factorize explanations based on author, paper name, conference name or year. Importantly, we impose a novel constraint on the factorizations that we look for (which we call *compatibility*), intuitively capturing that their structure is consistent with a partial order defined by the parse tree of the question. This constraint is needed so that we can translate the factorization back to an NL answer whose structure is similar to that of the question. Even with this constraint, there may still be exponentially many (in the size of the provenance expression) compatible factorizations, and we look for the factorization with minimal size out of the compatible ones; for comparison, previous work looks for the minimal factorization with no such "compatibility constraint". The corresponding decision problem remains coNP-hard (again in the provenance size), but we devise an effective and simple greedy solution. We further translate factorized representations to concise NL sentences, again leveraging the structure of the NL query.

Summarization. We propose *summarized* explanations by replacing details of different parts of the explanation by their synopsis, *e.g.* presenting only the number of papers published by each author, the number of authors, or the overall number of papers published by authors of each organization. Such summarizations incur by nature a loss of information but are typically much more concise and easier for users to follow. Here again, while provenance summarization has been studied before (*e.g.* [3, 44]), the desiderata of a summarization needed for NL sentence generation are different, rendering previous solutions inapplicable here. We observe a tight correspondence between factorization and summarization: every factorization gives rise to multiple possible summarizations, each obtained by counting the number of sub-explanations that are "factorized together". We provide a robust solution, allowing to compute NL summarizations of the provenance, of varying levels of granularity.

Implementation and Experiments. We have implemented our solution in a system prototype called `NLProv` [15], forming an end-to-end NL interface to database querying where the NL queries, answers and provenance information are all expressed in Natural Language. We have further conducted extensive experiments whose results indicate the scalability of the solution as well as the quality of the results, the latter through a user study.

2. PRELIMINARIES

We provide here the necessary preliminaries on Natural Language Processing, conjunctive queries and provenance.



(a) Verb Mod. **(b)** Non-Verb Mod.

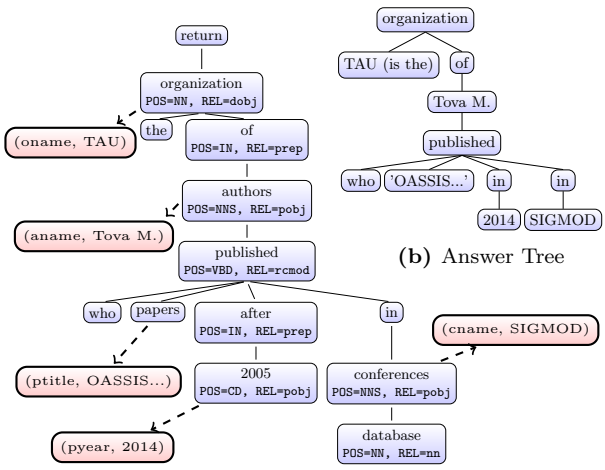
Figure 3: Abstract Dependency Trees

2.1 From NL to Formal Queries

We start by recalling some basic notions from NLP, as they pertain to the translation process of NL queries to a formal query language. A key notion that we will use is that of the *syntactic dependency tree* of NL queries:

DEFINITION 2.1. *A dependency tree $T = (V, E, L)$ is a node-labeled tree where labels consist of two components, as follows: (1) Part of Speech (POS): the syntactic role of the word [32, 36] ; (2) Relationship (REL): the grammatical relationship between the word and its parent in the dependency tree [37].*

We focus on a sub-class of queries handled by `NaLIR`, namely that of Conjunctive Queries, possibly with comparison operators $(=, >, <)$ and logical combinations thereof (`NaLIR` further supports nested queries and aggregation). The corresponding NL queries in `NaLIR` follow one of the two (very general) abstract forms described in Figure 3: an object (noun) is sought for, that satisfies some properties, possibly described through a complex sub-sentence rooted by a *modifier* (which may or may not be a verb, a distinction whose importance will be made clear later).



(a) Query Tree **(b)** Answer Tree

Figure 4: Question and Answer Trees

EXAMPLE 2.2. *Reconsider the NL query in Figure 1a; its dependency tree is depicted in Figure 4a (ignore for now the arrows). The part-of-speech (POS) tag of each node reflects its syntactic role in the sentence – for instance, "organization" is a noun (denoted "NN"), and "published" is a verb in past tense (denoted "VBD"). The relation (REL) tag of each*

node reflects the semantic relation of its sub-tree with its parent. For instance, the REL of "of" is prep ("prepositional modifier") meaning that the sub-tree rooted at "of" describes a property of "organization" while forming a complex sub-sentence. The tree in Figure 4a matches the abstract tree in Figure 3b since "organization" is the object and "of" is a non-verb modifier (its POS tag is IN, meaning "preposition or subordinating conjunction") rooting a sub-sentence describing "organization".

The dependency tree is transformed by NaLIR, based also on schema knowledge, to SQL. We focus in this work on NL queries that are compiled into Conjunctive Queries (CQs).

EXAMPLE 2.3. *Reconsider our running example NL query in Figure 1a; a counterpart Conjunctive Query is shown in Figure 1b. Some words of the NL query have been mapped by* NaLIR *to variables in the query, e.g., the word "organization" corresponds to the head variable (oname). Additionally, some parts of the sentence have been complied to boolean conditions based on the MAS schema, e.g., the part "in database conferences" was translated to dname =′ Databases′ in the CQ depicted in Figure 1b. Figure 4a shows the mapping of some of the nodes in the NL query dependency tree to variables of Q (ignore for now the values next to these variables).*

The translation performed by NaLIR from an NL query to a formal one can be captured by a *mapping* from (some) parts of the sentence to parts of the formal query.

DEFINITION 2.4. *Given a dependency tree $T = (V, E, L)$ and a CQ Q, a dependency-to-query-mapping $\tau : V \to Vars(Q)$ is a partial function mapping a subset of the dependency tree nodes to the variables of Q.*

2.2 Provenance

After compiling a formal query corresponding to the user's NL query, we evaluate it and keep track of *provenance*, to be used in explanations. To define provenance, we first recall the standard notion of *assignments* for CQs.

DEFINITION 2.5. *An assignment α for a query $Q \in CQ$ with respect to a database instance D is a mapping of the relational atoms of Q to tuples in D that respects relation names and induces a mapping over variables/constants, i.e. if a relational atom $R(x_1, ..., x_n)$ is mapped to a tuple $R(a_1, ..., a_n)$ then we say that x_i is mapped to a_i (denoted $\alpha(x_i) = a_i$, overloading notations) and we further say that the tuple was* used *in α. We require that any variable will be mapped to a single value, and that any constant will be mapped to itself. We further define $\alpha(head(Q))$ as the tuple obtained from $head(Q)$ by replacing each occurrence of a head variable x_i by $\alpha(x_i)$.*

Assignments allow for defining the semantics of CQs: a tuple t is said to appear in the query output if there exists an assignment α s.t. $t = \alpha(head(Q))$. They will also be useful in defining provenance below.

EXAMPLE 2.6. *Consider again the query Q in Figure 1b and the database in Figure 6. The tuple (TAU) is an output of Q when assigning the highlighted tuples to the atoms of Q. As part of this assignment, the tuple (2, TAU) (the second tuple in the org table) and (4, Tova M., 2) (the second tuple of the author table) are assigned to the first and second atom of Q, respectively. In addition to this assignment, there are 4 more assignments that produce the tuple (TAU) and one assignment that produces the tuple (UPENN).*

```
(oname,TAU)·(aname,Tova M.)·(ptitle,OASSIS...)·
(cname,SIGMOD)·(pyear,14')+
(oname,TAU)·(aname,Tova M.)·(ptitle,Querying...)·
(cname,VLDB)·(pyear,06')+
(oname,TAU)·(aname,Tova M.)· (ptitle,Monitoring..)·
(cname,VLDB)·(pyear,07')+
(oname,TAU)·(aname,Slava N.)·(ptitle,OASSIS...)·
(cname, SIGMOD)·(pyear,14')+
(oname,TAU)·(aname,Tova M.)·(ptitle,A sample...)·
(cname,SIGMOD)·(pyear,14')+
(oname,UPENN)·(aname,Susan D.)·(ptitle,OASSIS...)·
(cname,SIGMOD)·(pyear,14')
```

Figure 5: Value-level Provenance

We next leverage assignments in defining provenance, introducing a simple value-level model. The idea is that assignments capture the *reasons* for a tuple to appear in the query result, with each assignment serving as an *alternative* such reason (indeed, the existence of a single assignment yielding the tuple suffices, according to the semantics, for its inclusion in the query result). Within each assignment, we keep record of the value assigned to each variable, and note that the *conjunction* of these value assignments is required for the assignment to hold. Capturing alternatives through the symbolic " + " and conjunction through the symbolic " · ", we arrive at the following definition of provenance as sum of products.

DEFINITION 2.7. *Let $A(Q, D)$ be the set of assignments for a CQ Q and a database instance D. We define the* value-level provenance *of Q w.r.t. D as*

$$\sum_{\alpha \in A(Q,D)} \Pi_{\{x_i, a_i | \alpha(x_i) = a_i\}}(x_i, a_i)$$

.

Rel. *org*

oid	oname
1	UPENN
2	TAU

Rel. *author*

aid	aname	oid
3	Susan D.	1
4	Tova M.	2
5	Slava N.	2

Rel. *pub*

wid	cid	ptitle	pyear
6	10	"OASSIS..."	2014
7	10	"A sample..."	2014
8	11	"Monitoring..."	2007
9	11	"Querying..."	2006

Rel. *writes*

aid	wid
4	6
3	6
5	6
4	7
4	8
4	9

Rel. *conf*

cid	cname
10	SIGMOD
11	VLDB

Rel. *domainConf*

cid	did
10	18
11	18

Rel. *domain*

did	name
18	Databases

Figure 6: DB Instance

EXAMPLE 2.8. *Re-consider our running example query and consider the database in Figure 6. The value-level provenance is shown in Figure 5. Each of the 6 summands stands for a different assignment (i.e. an alternative reason for the tuple to appear in the result). Assignments are represented*

as multiplication of pairs of the form (var, val) so that var is assigned val in the particular assignment. We only show here variables to which a query word was mapped; these will be the relevant variables for formulating the answer.

By composing the dependency-to-query-mapping (recall Definition 2.4) from the NL query's dependency tree to query variables, and the assignments of query variables to values from the database, we associate different parts of the NL query with values. We will use this composition of mappings throughout the paper as a means of assembling the NL answer to the NL query.

EXAMPLE 2.9. *Continuing our running example, consider the assignment represented by the first monomial of Figure 5. Further reconsider Figure 4a, and now note that each node is associated with a pair (var, val) of the variable to which the node was mapped, and the value that this variable was assigned in this particular assignment. For instance, the node "organization" was mapped to the variable oname which was assigned the value "TAU".*

3. FIRST STEP: A SINGLE ASSIGNMENT

We now start describing our transformation of provenance to NL. We show it first for the case where the query has a single assignment with respect to the input database. The solution will serve as a building block for the general case of multiple assignments.

3.1 Basic Solution

We follow the structure of the NL query dependency tree and generate an answer tree with the same structure by replacing/modifying the words in the question with the values from the result and provenance that were mapped using the dependency-to-query-mapping and the assignment. Yet, note that simply replacing the values does not always result in a coherent sentence, as shown in the following example.

EXAMPLE 3.1. *Re-consider the dependency tree depicted in Figure 4a. If we were to replace the value in the organization node to the value "TAU" mapped to it, the word "organization" will not appear in the answer although it is needed to produce the coherent answer depicted in Figure 2. Without this word, it is unclear how to deduce the information about the connection between "Tova M." and "TAU".*

We next account for these difficulties and present an algorithm that outputs the dependency tree of a detailed answer; we will further translate this tree to an NL sentence.

Recall that the dependency tree of the NL query follows one of the abstract forms in Figure 3. We distinguish between two cases based on nodes whose *REL* (relationship with parent node) is *modifier*; in the first case, the clause begins with a verb modifier (*e.g.*, the node "published" in Fig. 4a is a verb modifier) and in the second, the clause begins with a non-verb modifier (*e.g.*, the node "of" in Fig. 4a is a non-verb modifier). Algorithm 1 considers these two forms of dependency tree and provides a tailored solution for each one in the form of a dependency tree that fits the correct answer structure. It does so by augmenting the query dependency tree into an answer tree.

The algorithm operates as follows. We start with the dependency tree of the NL query, an empty answer tree T_A, a dependency-to-query-mapping an assignment and a node *object* from the query tree. We denote the set of all modifiers by MOD and the set of all verbs by $VERB$. The algorithm is recursive and handles several cases, depending on *object* and its children in the dependency tree. If the node *object* is a leaf (Line 2), we replace it with the value mapped to it by dependency-to-query-mapping and the assignment, if such a mapping exists. Otherwise (it is a leaf without a mapping), it remains in the tree as it is. Second, if $L(object).REL$ is a modifier (Line 5), we call the procedure *Replace* in order to replace its entire subtree with the value mapped to it and add the suitable word for equality, depending on the type of its child (*e.g.*, location, year, etc. taken from the pre-prepared table), as its parent (using procedure *AddParent*). The third case handles a situation where *object* has a non-verb modifier child (Line 9). We use the procedure *Adjust* with a $false$ flag to copy T_Q into T_A, remove the *return* node and add the value mapped to *object* as its child in T_A. The difference in the fourth case (Line 12) is the value of $flag$ is now $true$. This means that instead of adding the value mapped to *object* as its child, the *Adjust* procedure replaces the node with its value. Finally, if *object* had a modifier child *child* (verb or non-verb), the algorithm makes a recursive call for all of the children of *child* (Line 16). This recursive call is needed here since a modifier node can be the root of a complex sub-tree (recall Example 2.2).

Algorithm 1: ComputeAnswerTree

input : A dependency tree T_Q, an answer tree T_A (empty in the first call), a dependency-to-query-mapping τ, an assignment α, a node $object \in T_Q$
output: Answer tree with explanations T_A

```
1  child := null;
2  if object is a leaf then
3  |  value = α(τ(object));
4  |  Replace(object, value, T_A);
5  else if L(object).REL is mod then
6  |  value = α(τ(child_{T_Q}(object)));
7  |  Replace(tree(object), value, T_A);
8  |  AddParent(T_A, value);
9  else if object has a child v s.t. L(v).REL ∈ MOD and
   L(v).POS ∉ VERB then
10 |  Adjust(T_Q, T_A, τ, α, object, false);
11 |  child := v;
12 else if object has a child v s.t. L(v).REL ∈ MOD and
   L(v).POS ∈ VERB then
13 |  Adjust(T_Q, T_A, τ, α, object, true);
14 |  child := v;
15 if child ≠ null then
16 |  foreach u ∈ children_{T_Q}(child) do
17 |  |  ComputeAnswerTree(T_Q, T_A, τ, α, u);
18 return T_A;
```

EXAMPLE 3.2. *Re-consider Figure 4a, and note the mappings from the nodes to the variables and values as reflected in the boxes next to the nodes. To generate an answer, we follow the NL query structure, "plugging-in" mapped database values. We start with "organization", which is the first object node. Observe that "organization" has the child "of" which is a non-verb modifier, so we add "TAU" as its child and assign true to the hasMod variable. We then reach Line 15 where the condition holds and we make a recursive*

call to the children of "of", i.e., the node object is now "authors". Again we consider all cases until reaching the fourth (Line 12). The condition holds since the node "published" is a verb modifier, thus we replace "authors" with "Tova M.", mapped to it. Then, we make a recursive call for all children of "published" since the condition in Line 15 holds. The nodes "who" and "papers" are leaves so they satisfy the condition in Line 2. Only "papers" has a value mapped to it, so it is replaced by this value ("OASSIS..."). However, the nodes "after" and "in" are modifiers so when the algorithm is invoked with object = "after" ("in"), the second condition holds (Line 5) and we replace the subtree of these nodes with the node mapped to their child (in the case of "after" it is "2014" and in the case of "in" it is "SIGMOD") and we attach the node "in" as the parent of the node, in both cases as it is the suitable word for equality for years and locations. We obtain a tree representation of the answer (Fig. 4b).

So far we have augmented the NL query dependency tree to obtain the dependency tree of the answer. The last step is to translate this tree to a sentence. To this end, we recall that the original query, in the form of a sentence, was translated by `NaLIR` to the NL query dependency tree. To translate the dependency tree to a sentence, we essentially "revert" this process, further using the mapping of NL query dependency tree nodes to (sets of) nodes of the answer.

EXAMPLE 3.3. *Converting the answer tree in Figure 4b to a sentence is done by replacing the words of the NL query with the values mapped to them, e.g., the word "authors" in the NL query (Figure 1a) is replaced by "Tova M." and the word "papers" is replaced by "OASSIS...". The word "organization" is not replaced (as it remains in the answer tree) but rather the words "TAU is the" are added prior to it, since its POS is not a verb and its REL is a modifier. Completing this process, we obtain the answer shown in Figure 2.*

3.2 Logical Operators

Logical operators (and, or) and the part of the NL query they relate to will be converted by `NaLIR` to a logical predicate which will be mapped by the assignment to *one value* that satisfies the logical statement. To handle these parts of the query, we augment Algorithm 1 as follows: immediately following the first case (before the current Line 5), we add a condition checking whether the node *object* has a logical operator ("and" or "or") as a child. If so, we call Procedure HandleLogicalOps with the trees T_Q and T_A, the logical operator node as u, the dependency-to-query-mapping τ and the assignment α. The procedure initializes a set S to store the nodes whose subtree needs to be replaced by the value given to the logical predicate (Line 2). Procedure HandleLogicalOps first locates all nodes in T_Q that were mapped by the dependency-to-query-mapping to the same query variable as the sibling of the logical operator (denoted by u). Then, it removes the subtrees rooted at each of their parents (Line 8), adds the value (denoted by *val*) from the database mapped to all of them in the same level as their parents (Line 9), and finally, the suitable word for equality is added as the parent of *val* in the tree by the procedure *AddParent* (Line 10).

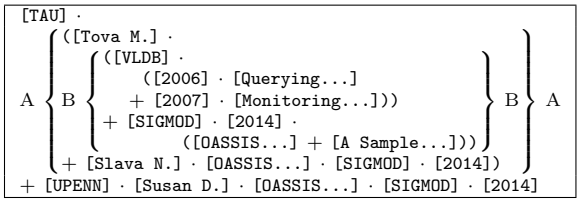
4. THE GENERAL CASE

In the previous section we have considered the case where the provenance consists of a single assignment. In general, as illustrated in Section 2, it may include multiple assignments. We next generalize the construction to account for this. Note that a naïve solution in this respect is to generate a sentence for each individual assignment and concatenate the resulting sentences. However, already for the small-scale example presented here, this would result in a long and unreadable answer (recall Figure 5 consisting of six assignments). Instead, we propose two solutions: the first based on the idea of provenance factorization [42, 8], and the second leveraging factorization to provide a summarized form.

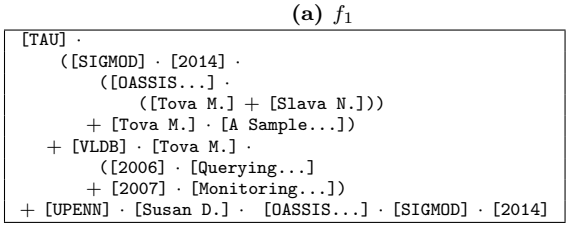
Procedure HandleLogicalOps

input : A dependency tree T_Q, T_A, $u \in V_{T_A}$, dependency-to-query-mapping τ and an assignment α

1 $w \leftarrow parent_{T_Q}(u)$;
2 $S \leftarrow \{w\}$;
3 $var \leftarrow \tau(children_{T_A}(w) \setminus u)$;
4 $val \leftarrow \alpha(\tau(children_{T_A}(w) \setminus u))$;
5 **for** $z \in siblings_{T_A}(w)$ **do**
6 **if** *z has child mapped by* τ *to var* **then**
7 $S.Insert(z)$;
8 $parent_{T_A}(w).children_{T_A}().Remove(S)$;
9 $parent_{T_A}(w).children_{T_A}().Insert(val)$;
10 $AddParent(T_A, val)$;



(a) f_1



(b) f_2

Figure 7: Provenance Factorizations

4.1 NL-Oriented Factorization

We start by defining the notion of factorization in a standard way (see *e.g.* [42, 17]).

DEFINITION 4.1. *Let P be a provenance expression. We say that an expression f is a factorization of P if f may be obtained from P through (repeated) use of some of the following axioms: distributivity of summation over multiplication, associativity and commutativity of both summation and multiplication.*

EXAMPLE 4.2. *Re-consider the provenance expression in Figure 5. Two possible factorizations are shown in Figure 7, keeping only the values and omitting the variable names for brevity (ignore the A,B brackets for now). In both cases, the idea is to avoid repetitions in the provenance expression, by*

taking out a common factor that appears in multiple summands. Different choices of which common factor to take out lead to different factorizations.

How do we measure whether a possible factorization is suitable/preferable to others? Standard desiderata [42, 17] are that it should be short or that the maximal number of appearances of an atom is minimal. On the other hand, we factorize here as a step towards generating an NL answer; to this end, it will be highly useful if the *(partial) order of nesting of value annotations in the factorization is consistent the (partial) order of corresponding words in the NL query.* We will next formalize this intuition as a constraint over factorizations. We start by defining a partial order on nodes in a dependency tree:

DEFINITION 4.3. *Given an dependency tree T, we define $\leq_T$ as the descendant partial order of nodes in T: for each two nodes, $x, y \in V(T)$, we say that $x \leq_T y$ if x is a descendant of y in T.*

EXAMPLE 4.4. *In our running example (Figure 4a) it holds in particular that $authors \leq organization$, $2005 \leq authors$, $conferences \leq authors$ and $papers \leq authors$, but $papers$, 2005 and $conferences$ are incomparable.*

Next we define a partial order over elements of a factorization, intuitively based on their nesting depth. To this end, we first consider the *circuit form* [6] of a given factorization:

EXAMPLE 4.5. *Consider the partial circuit of f_1 in Figure 8. The root, $\cdot$, has two children; the left child is the leaf "TAU" and the right is a $+$ child whose subtree includes the part that is "deeper" than "TAU".*

Given a factorization f and an element n in it, we denote by $level_f(n)$ the distance of the node n from the root of the circuit induced by f multiplied by (-1). Intuitively, $level_f(n)$ is bigger for a node n closer to the circuit root.

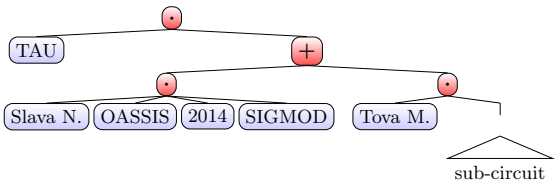


Figure 8: Sub-Circuit of f_1

Our goal here is to define the correspondence between the level of each node in the circuit and the level of its "source" node in the NL query's dependency tree (note that each node in the query corresponds to possibly many nodes in the circuit: all values assigned to the variable in the different assignments). In the following definition we will omit the database instance for brevity and denote the provenance obtained for a query with dependency tree T by $prov_T$. Recall that dependency-to-query-mapping maps the nodes of the dependency tree to the query variables and the assignment maps these variables to values from the database (Definitions 2.4, 2.5, respectively).

DEFINITION 4.6. *Let T be a query dependency tree, let $prov_T$ be a provenance expression, let f be a factorization of $prov_T$, let τ be a dependency-to-query-mapping and let $\{\alpha_1, ...\alpha_n\}$ be the set of assignments to the query. For each two nodes x, y in T we say that $x \leq_f y$ if $\forall i \in [n] : level_f(\alpha_i(\tau(x))) \leq level_f(\alpha_i(\tau(y)))$.*

We say that f is T-compatible if each pair of nodes $x \neq y \in V(T)$ that satisfy $x \leq_T y$ also satisfy that $x \leq_f y$.

Essentially, T-compatibility means that the partial order of nesting between values, for each individual assignment, must be consistent the partial order defined by the structure of the question. Note that the compatibility requirement imposes constraints on the factorization, but it is in general far from dictating the factorization, since the order $x \leq_T y$ is only partial – and there is no constraint on the order of each two provenance nodes whose "origins" in the query are unordered. Among the T-compatible factorizations, we will prefer shorter ones.

DEFINITION 4.7. *Let T be an NL query dependency tree and let $prov_T$ be a provenance expression for the answer. We say that a factorization f of $prov_T$ is* optimal *if f is T-compatible and there is no T-compatible factorization f' of $prov_T$ such that $| f' | < | f |$ ($| f |$ is the length of f).*

The following example shows that the T-compatibility constraint still allows much freedom in constructing the factorization. In particular, different choices can (and sometimes should, to achieve minimal size) be made for different sub-expressions, including ones leading to different answers and ones leading to the same answer through different assignments.

EXAMPLE 4.8. *Recall the partial order $\leq_T$ imposed by our running example query, shown in part in Example 4.4. It implies that in every compatible factorization, the organization name must reside at the highest level, and indeed TAU was "pulled out" first in Figure 8; similarly the author name must be pulled out next. In contrast, since the query nodes corresponding to title, year and conference name are unordered, we may,* within a single factorization*, factor out e.g. the year in one part of the factorization and the conference name in another one. As an example, Tova M. has two papers published in VLDB ("Querying..." and "Monitoring") so factorizing based on VLDB would be the best choice for that part. On the other hand, suppose that Slava N. had two paper published in* 2014*; then we could factorize them based on* 2014*. The factorization could, in that case, look like the following (where the parts taken out for Tova and Slava are shown in bold):*

```
[TAU] ·
([Tova M.] ·
(***[VLDB]*** ·
    ([2006] · [Querying...]
  + [2007] · [Monitoring...]))
+ ***[SIGMOD]*** · [2014] ·
        ([OASSIS...] + [A Sample...]))
+ ([Slava N.] ·
  (***[2014]*** ·
    ([SIGMOD] · [OASSIS...]
   + [VLDB] · [Ontology...])))
```

The following example shows that in some cases, requiring compatibility can come at the cost of compactness.

EXAMPLE 4.9. *Consider the query tree T depicted in Figure 4a and the factorizations $prov_T$ (the identity factorization) depicted in Figure 5, f_1, f_2 presented in Figure 7.*

$prov_T$ is of length 30 and is 5-readable, i.e., the maximal number of appearances of a single variable is 5 (see [17]). f_1 is of length 20, while the length of f_2 is only 19. In addition, both f_1 and f_2 are 3-readable. Based on those measurements f_2 seems to be the best factorization, yet f_1 is T-compatible with the question and f_2 is not. For example, $conferences \leq_T authors$ but "SIGMOD" appears higher than "Tova M." in f_2. Choosing a T-compatible factorization in f_1 will lead (as shown below) to an answer whose structure resembles that of the question, and thus translates to a more coherent and fitting NL answer.

Note that the identity factorization is always T-compatible, so we are guaranteed at least one optimal factorization (but it is not necessarily unique). We next study the problem of computing such a factorization.

4.2 Computing Factorizations

Recall that our notion of compatibility restricts the factorizations so that their structure resembles that of the question. Without this constraint, finding shortest factorizations is coNP-hard in the size of the provenance (*i*.e. a boolean expression) [27]. The compatibility constraint does not reduce the complexity since it only restricts choices relevant to part of the expression, while allowing freedom for arbitrarily many other elements of the provenance. Also recall (Example 4.8) that the choice of which element to "pull-out" needs in general to be done separately for each part of the provenance so as to optimize its size (which is the reason for the hardness in [27] as well). In general:

PROPOSITION 4.10. *Given a dependency tree T, a provenance expression $prov_T$ and an integer k, deciding whether there exists a T-compatible factorization of $prov_T$ of size $\leq k$ is coNP-hard.*

Greedy Algorithm. Despite the above result, the constraint of compatibility does help in practice, in that we can avoid examining choices that violate it. For other choices, we devise a simple algorithm that chooses greedily among them. More concretely, the input to Algorithm 2 is the query tree T_Q (with its partial order $\leq_{T_Q}$), and the provenance $prov_{T_Q}$. The algorithm output is a T_Q-compatible factorization f. Starting from *prov*, the progress of the algorithm is made in steps, where at each step, the algorithm traverses the circuit induced by *prov* in a BFS manner from top to bottom and takes out a variable that would lead to a minimal expression out of the valid options that keep the current factorization T-compatible. Naturally, the algorithm does not guarantee an optimal factorization (in terms of length), but performs well in practice (see Section 5).

In more detail, we start by choosing the largest nodes according to $\leq_{T_Q}$ which have not been processed yet (Line 2). Afterwards, we sort the corresponding variables in a greedy manner based on the number of appearances of each variable in the expression using the procedure $sortByFrequentVars$ (Line 3). In Lines 4–5, we iterate over the sorted variables and extract them from their sub-expressions. This is done while preserving the $\leq_{T_Q}$ order with the larger nodes, thus ensuring that the factorization will remain T_Q-compatible. We then add all the newly processed nodes to the set *Processed* which contains all nodes that have already been processed (Line 6). Lastly, we check whether there are no more nodes to be processed, *i*.e., if the set *Processed* includes all the nodes of T_Q (denoted $V(T_Q)$, see the condition in Line 7). If the answer is "yes", we return the factorization. Otherwise, we make a recursive call. In each such call, the set *Processed* becomes larger until the condition in Line 7 holds.

Algorithm 2: GreedyFactorization

input : T_Q - the query tree, $\leq_{T_Q}$ - the query partial order, *prov* - the provenance, τ, α - dependency-to-query-mapping and assignment from nodes in T_Q to provenance variables, *Processed* - subset of nodes from $V(T_Q)$ which were already processed (initially, $\emptyset$)
output: f - T_Q-compatible factorization of $prov_{T_Q}$

1 $f \leftarrow prov$;
2 $Frontier \leftarrow \{x \in V(T_Q) | \forall (y \in V(T_Q) \setminus Processed)\ s.t.\ x \not\leq_{T_Q} y\}$;
3 $vars \leftarrow sortByFrequentVars(\{\alpha(\tau(x)) | x \in Frontier\}, f)$;
4 **foreach** $var \in vars$ **do**
5 Take out var from sub-expressions in f not including variables from $\{x | \exists y \in Processed : x = \alpha(\tau(y))\}$;
6 $Processed \leftarrow Processed \cup Frontier$;
7 **if** $|Processed| = |V(T_Q)|$ **then**
8 **return** f;
9 **else**
10 **return** $GreedyFactorization(T_Q, f, \tau, \alpha, Processed)$;

EXAMPLE 4.11. *Consider the query tree T_Q depicted in Figure 4a, and provenance prov in Figure 5. As explained above, the largest node according to $\leq_{T_Q}$ is organization, hence "TAU" will be taken out from the brackets multiplying all summands that contain it. Afterwards, the next node according to the order relation will be author, therefore we group by author, taking out "Tova M.", "Slava N." etc. The following choice (between conference, year and paper name) is then done greedily* for each author*, based on its number of occurrences. For instance, $VLDB$ appears twice for $Tova.M.$ whereas each paper title and year appears only once; so it will be pulled out. The polynomial $[SlavaN.] \cdot [OASSIS...] \cdot [SIGMOD] \cdot [2014]$ will remain unfactorized as all values appear once. Eventually, the algorithm will return the factorization f_1 depicted in Figure 7, which is T_Q-compatible and much shorter than the initial provenance expression.*

PROPOSITION 4.12. *Let f be the output of Algorithm 2 for the input dependency tree T_Q, then f is T_Q-compatible.*

Complexity. Denote the provenance size by n. The algorithm complexity is $O(n^2 \cdot \log n)$: at each recursive call, we sort all nodes in $O(n \cdot \log n)$ (Line 3) and the we handle (in $Frontier$) at least one node (in the case of a chain graph) or more. Hence, in the worst case we would have n recursive calls, each one costing $O(n \cdot \log n)$.

4.3 Factorization to Answer Tree

The final step is to turn the obtained factorization into an NL answer. Similarly to the case of a single assignment (Section 3), we devise a recursive algorithm that leverages the mappings and assignments to convert the query dependency tree into an answer tree. Intuitively, we follow the

structure of a single answer, replacing each node there by either a single node, standing for a single word of the factorized expression, or by a recursively generated tree, standing for some brackets (sub-circuit) in the factorized expression.

In more detail, the algorithm operates as follows. We iterate over the children of *root* (the root of the current sub-circuit), distinguishing between two cases. First, for each leaf child, p, we first (Line 4) assign to *val* the database value corresponding to the first element of p under the assignment α (recall that p is a pair (variable,value)). We then lookup the node containing the value mapped to p's variable in the answer tree T_A and change its value to *val* in Lines 5, 6 (the value of p). Finally, in Line 7 we reorder nodes in the same level according to their order in the factorization (so that we output a semantically correct NL answer). Second, for each non-leaf child, the algorithm performs a recursive call in which the factorized answer subtree is computed (Line 9). Afterwards, the set containing the nodes of the resulting subtree aside from the nodes of T_A are attached to T_F under the node corresponding to their LCA in T_F (Lines 10 – 13). In this process, we attach the new nodes that were placed lower in the circuit in the most suitable place for them semantically (based on T_A), while also maintaining the structure of the factorization.

Algorithm 3: ComputeFactAnswerTree

input : α - an assignment to the NL query, T_A - answer dependency tree based on α, *root* - the root of the circuit induced by the factorized provenance
output: T_F - tree of the factorized answer

1 $T_F \leftarrow copy(T_A)$;
2 **foreach** $p \in children_f(root)$ **do**
3 **if** *p is a leaf* **then**
4 $val \leftarrow \alpha(var(p))$;
5 $node \leftarrow Lookup(var(p), \alpha, T_A)$;
6 $ReplaceVal(val, node, T_F)$;
7 $Rearrange(node, T_A, T_F)$;
8 **else**
9 $T_F^{rec} = ComputeFactAnswerTree(\alpha, T_A, p)$;
10 $RecNodes = V(T_F^{rec}) \setminus V(T_A)$;
11 $parent_F^{rec} \leftarrow LCA(recNodes)$;
12 $parent_F \leftarrow$ Corresponding node to $parent_F^{rec}$ in T_F;
13 Attach *recNodes* to T_F under the $parent_F$;
14 **return** T_F;

Example 4.13. *Consider the factorization f_1 depicted in Figure 7, and the structure of single assignment answer depicted in Figure 4b which was built based on Algorithm 1. Given this input, Algorithm 3 will generate an answer tree corresponding to the following sentence:*

```
TAU is the organization of
    Tova M. who published
        in VLDB
           'Querying...' in 2006 and
           'Monitoring...' in 2007
        and in SIGMOD in 2014
            'OASSIS...' and  'A sample...'
    and Slava N. who published
        'OASSIS...' in SIGMOD in 2014.
UPENN is the organization of Susan D. who published
'OASSIS...' in SIGMOD in 2014.
```

Note that the query has two results: "TAU" and "UPENN". "UPENN" was produced with a single assignment, but there are 5 different assignments producing "TAU". We now focus on this sub-circuit depicted in Figure 8. After initializing T_F, in Lines 3 – 7 the algorithm finds the value TAU and the node corresponding to it in T_A (which originally contained the variable $organization$). It then copies this node to T_F and assigns it the value "TAU". Next the algorithm handles the + node with a recursive call in Line 9. This node has the two sub-circuits rooted at the two · nodes (Line 8); one containing $[authors, TovaM.]$ and the other $[authors, SlavaN.]$. When traversing the sub-circuit containing "Slava N.", the algorithm simply copies the subtree rooted at the $authors$ node with the values from the circuit and arranges the nodes in the same order as the corresponding variable nodes were in T_A (Line 7) as they are all leaves on the same level. Those values will be attached under the LCA "of" (Lines 9 – 13). The sub-circuit of "Tova M." also has nested sub-circuits. Although the node paper appears before the nodes year and $conference$ in the answer tree structure, the algorithm identifies that f_1 extracted the variables "VLDB", "SIGMOD" and "2014", so it changes their location so that they appear earlier in the final answer tree. Finally, recursive calls are made with the sub-circuit containing $[authors, TovaM.]$.

Why require compatibility? We conclude this part of the paper by revisiting our decision to require compatible factorizations, highlighting difficulties in generating NL answers using non-compatible factorizations.

Example 4.14. *Consider factorization f_2 from Figure 7. "TAU" should be at the beginning of the sentence and followed by the conference names "SIGMOD" and "VLDB". The second and third layers of f_2 are composed of author names ("Tova M.", "Slava N."), paper titles ("OASSIS", "A sample...", "Monitoring...") and publication years (*2007*,* 2014*). Changing the original order of the words such that the conference name "SIGMOD" and the publication year "2014" will appear before "Tova M." breaks the sentence structure in a sense. It is unclear how to algorithmically translate this factorization into an NL answer, since we need to patch the broken structure by adding connecting phrases. One hypothetical option of patching f_2 and transforming it into an NL answer is depicted below. The bold parts of the sentence are not part of the factorization and it is not clear how to generate and incorporate them into the sentence algorithmically. Even if we could do so, it appears that the resulting sentence would be quite convoluted:*

```
TAU is the organization of authors who published in
    SIGMOD 2014
        'OASSIS...' which was published by
            Tova M. and Slava N.
        and Tova M. published 'A sample...'
    and Tova M. published in VLDB
        'Querying...' in 2014
        and 'Monitoring...' in 2007.
UPENN is the organization of Susan D. who published
'OASSIS...' in SIGMOD in 2014
```

Observe that the resulting sentence is much less clear than the one obtained through our approach, even though it was obtained from a shorter factorization f_2; the intuitive reason is that since f_2 is not T-compatible, it does not admit a structure that is similar to that of the question, thus is not guaranteed to admit a structure that is coherent in Natural Language. Interestingly, the sentence we would obtain in such a way also has an edit distance from the question [18] that is shorter than that of our answer, demonstrating that edit distance is not an adequate measure here.

(A) [TAU] · *Size*([Tova M.],[Slava N.]) · *Size*([VLDB],[SIGMOD]) ·
Size([Querying...],[Monitoring...],
[OASSIS...],[A Sample...]) · *Range*([2006],[2007],[2014])
(B) [TAU]·(
[Tova M.] ·
Size([VLDB],[SIGMOD]) ·
Size([Querying...],[Monitoring...],
[OASSIS...],[A Sample...]) · *Range*([2006],[2007],[2014])
[Slava N.] · [OASSIS...] · [SIGMOD] · [2014])

Figure 9: Summarized Factorizations

(A) TAU is the organization of 2 authors who published 4 papers in 2 conferences in 2006 - 2014.
(B) TAU is the organization of Tova M. who published 4 papers in 2 conferences in 2006 - 2014 and Slava N. who published 'OASSIS...' in SIGMOD in 2014.

Figure 10: Summarized Sentences

4.4 From Factorizations to Summarizations

So far we have proposed a solution that factorizes multiple assignments, leading to a more concise answer. When there are many assignments and/or the assignments involve multiple distinct values, even an optimal factorized representation may be too long and convoluted for users to follow.

EXAMPLE 4.15. *Reconsider Example 4.13; if there are many authors from TAU then even the compact representation of the result could be very long. In such cases we need to summarize the provenance in some way that will preserve the "essence" of all assignments without actually specifying them, for instance by providing only the number of authors/papers for each institution.*

To this end, we employ *summarization*, as follows. First, we note that a key to summarization is understanding which parts of the provenance may be grouped together. For that, we use again the mapping from nodes to query variables: we say that two nodes are of the same *type* if both were mapped to the same query variable. Now, let n be a node in the circuit form of a given factorization f. A summarization of the sub-circuit of n is obtained in two steps. First, we group the descendants of n according to their type. Then, we summarize each group separately. The latter is done in our implementation simply by either counting the number of distinct values in the group or by computing their range if the values are numeric. In general, one can easily adapt the solution to apply additional user-defined "summarization functions" such as "greater / smaller than X" (for numerical values) or "in continent Y" for countries.

EXAMPLE 4.16. *Re-consider the factorization f_1 from Figure 7. We can summarize it in multiple levels: the highest level of authors (summarization "A"), or the level of papers for each particular author (summarization "B"), or the level of conferences, etc. Note that if we choose to summarize at some level, we must summarize its entire sub-circuit (e.g. if we summarize for Tova. M. at the level of conferences, we cannot specify the papers titles and publication years).*

Figure 9 presents the summarizations of sub-trees for the "TAU" answer, where "size" is a summarization operator that counts the number of distinct values and "range" is an operator over numeric values, summarizing them as their range. The summarized factorizations are further converted to NL sentences which are shown in Figure 10. Summarizing at a higher level results in a shorter but less detailed summarization.

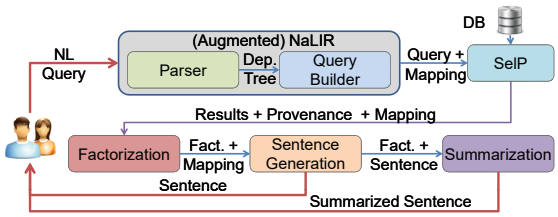


Figure 11: System Architecture

Table 1: NL queries

Num.	Queries
1	Return the homepage of SIGMOD
2	Return the papers whose title contains 'OASSIS'
3	Return the papers which were published in conferences in database area
4	Return the authors who published papers in SIGMOD after 2005
5	Return the authors who published papers in SIGMOD before 2015 and after 2005
6	Return the authors who published papers in database conferences
7	Return the organization of authors who published papers in database conferences after 2005
8	Return the authors from TAU who published papers in VLDB
9	Return the area of conferences
10	Return the authors who published papers in database conferences after 2005
11	Return the conferences that presented papers published in 2005 by authors from organization
12	Return the years of papers published by authors from IBM

5. IMPLEMENTATION AND EXPERIMENTS

5.1 System Architecture

NLProv is implemented in JAVA 8, extending NaLIR. Its web UI is built using HTML, CSS and JavaScript. It runs on Windows 8 and uses MySQL server as its underlying database management system (the source code is available in [21]). Figure 11 depicts the system architecture. First, the user enters a query in Natural Language. This NL sentence is fed to the augmented NaLIR system which interprets it and generates a formal query. This includes the following steps: a parser [37] generates the dependency tree for the NL query. Then, the nodes of the tree are mapped to attributes in the tables of the database and to functions, to form a formal query. As explained, to be able to translate the results and provenance to NL, NLProv stores the mapping from the nodes of the dependency tree to the query variables. Once a query has been produced, NLProv uses the SelP system [16] to evaluate it while storing the provenance, keeping track of the mapping of dependency tree nodes to parts of the provenance. The provenance information is then factorized (see Algorithm 2) and the factorization is compiled to an NL answer (Algorithm 3) containing explanations.

Finally, the factorized answer is shown to the user. If the answer contains excessive details and is too difficult to understand, the user may choose to view summarizations.

5.2 User Study

We have examined the usefulness of the system through a user study, involving 15 non-expert users. We have presented to each user 6 NL queries, namely No. 1–4, 6, and 7 from Table 1 (chosen as a representative sample). We have also allowed each user to freely formulate an NL query of

Table 2: Sample use-cases and results

Query	Single Assignment	Multiple Assignments - Summarized
Return the homepage of SIGMOD	http://www.sigmod2011.org/ is the homepage of SIGMOD	
Return the authors who published papers in SIGMOD before 2015 and after 2005	Tova M. published "Auto-completion..." in SIGMOD in 2012	Tova M. published 10 papers in SIGMOD in 2006-2014
Return the authors from TAU who published papers in VLDB	Tova M. from TAU published "XML Repository..." in VLDB	Tova M. from TAU published 11 papers in VLDB
Return the authors who published papers in database conferences	Tova M. "published Auto-completion..." in SIGMOD	Tova M. published 96 papers in 18 conferences
Return the organization of authors who published papers in database conferences after 2005	TAU is the organization of Tova M. who published 'OASSIS...' in SIGMOD in 2014	TAU is the organization of 43 authors who published 170 papers in 31 conferences in 2006 - 2015

her choice, related to the MAS publication database [38]. 2 users have not provided a query at all, and for 5 users the query either did not parse well or involved aggregation (which is not supported), leading to a total of 98 successfully performed tasks. For each of the NL queries, users were shown the NL provenance computed by `NLProv` for cases of single derivations, factorized and summarized answers for multiple derivations (where applicable). Multiple derivations were relevant in 50 of the 98 cases. Examples of the presented results are shown in Table 2.

We have asked users three questions about each case, asking them to rank the results on a 1–5 scale where 1 is the lowest score: (1) is the answer relevant to the NL query? (2) is the answer understandable? and (3) is the answer detailed enough, *i*.e. supply all relevant information? (asked only for answers including multiple assignments).

The results of our user study are summarized in Table 3. In all cases, the user scores were in the range 3–5, with the summarized explanation receiving the highest scores on all accounts. Note in particular the difference in understandability score, where summarized sentences ranked as significantly more understandable than their factorized counterparts. Somewhat surprisingly, summarized sentences were even deemed by users as being more detailed than factorized ones (although technically they are of course less detailed), which may be explained by their better clarity (users who ranked a result lower on understandability have also tended to ranked it low w.r.t. level of detail).

Table 3: Users ranking

Category	3	4	5	Average
Single				
Relevant	4	10	84	4.82
Understandable	7	25	66	4.60
Multiple				
Relevant	0	7	43	4.86
Understandable	4	13	33	4.58
Detailed	3	7	40	4.74
Summarized				
Relevant	2	2	46	4.88
Understandable	3	3	44	4.82
Detailed	2	5	43	4.82

5.3 Scalability

Another facet of our experimental study includes runtime experiments to examine the scalability of our algorithms. Here again we have used the MAS database whose total size is 4.7 GB, and queries No. 1–12 from Table 1, running the algorithm to generate NL provenance for each individual answer. The experiments were performed on a i7 processor and 32GB RAM with Windows 8. As expected, when the provenance includes a single assignment per answer, the runtime is negligible (this is the case for queries No. 1–3). We thus show the results only for queries No. 4–12.

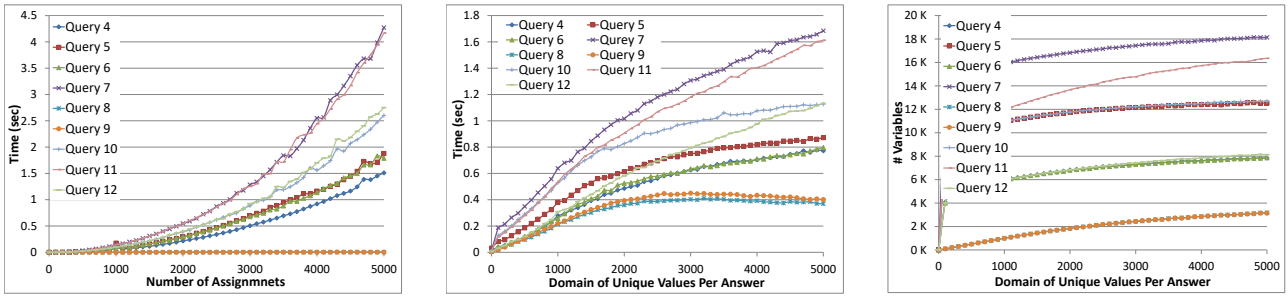
Table 4: Computation time (sec.), for the MAS database

Query	Query Eval. Time	Fact. Time	Sentence Gen. Time	`NLProv` Time
4	0.9	0.038	0.096	0.134
5	0.6	0.03	0.14	0.17
6	33	0.62	2.08	2.7
7	20.5	1.1	3.1	4.2
8	2.4	0.001	0.001	0.002
9	0.01	0.011	0.001	0.012
10	21.3	0.53	2.23	2.76
11	53.7	3.18	6.46	9.64
12	18.8	3.22	1.73	4.95

Table 4 includes, for each query, the runtime required by our algorithms to transform provenance to NL in factorized or summarized form, for all query results (as explained in Section 4, we can compute the factorizations independently for each query result). We show a breakdown of the execution time of our solution: factorization time, sentence generation time, and total time incurred by `NLProv` (we note that the time to compute summarizations given a factorization was negligible). For indication on the complexity level of the queries, we also report the time incurred by standard (provenance-oblivious) query evaluation, using the mySQL engine. We note that our algorithms perform quite well for all queries (overall `NLProv` execution has 16% overhead), even for fairly complex ones such as queries 7, 11, and 12.

Figure 12a (see next page) presents the execution time of NL provenance computation for an increasing number of assignments *per answer* (up to 5000, note that the maximal number in the real data experiments was 4208). The provenance used for this set of experiments was such that the only shared value in all assignments was the result value, so the factorization phase is negligible in terms of execution time, taking only about one tenth of the total runtime in the multiple assignments case. Most computation time here is incurred by the answer tree structuring. We observe that the computation time increased moderately as a function of the number of assignments (and is negligible for the case of a single assignment). The execution time for 5K assignments with unique values was 1.5, 2, 1.9, 4.9, 0.006, 0.003, 2.6, 5.3, and 3.7 seconds resp. for queries 4–12. Summarization time was negligible, less than 0.1 seconds in all cases.

For the second set of experiments, we have fixed the number of assignments per answer at the maximum 5K and changed only the domain of unique values from which provenance expressions were generated. The domain size *per answer, per query variable* varies from 0 to 5000 (this cannot

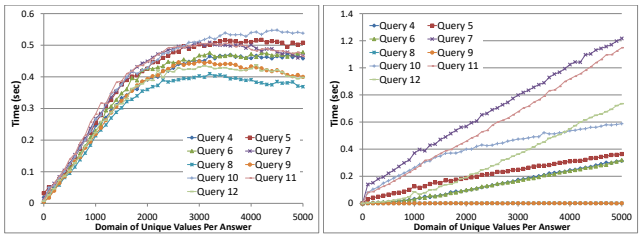


(a) Computation time as a function of the number of assignments

(b) Computation time as a function of the number of unique values

(c) Factorization size as a function of the number of unique values

Figure 12: Results for synthetic data



(a) Factorization time **(b)** Sentence gen. time

Figure 13: Breakdown for synthetic experiments

exceed the number of assignments). Note that the running time increases as a function of the number of unique values: when there are more unique values, there are more candidates for factorization (so the number of steps of the factorization algorithm increases), each factorization step is in general less effective (as there are more unique values for a fixed size of provenance, *i*.e. the degree of value sharing across assignments decreases), and consequently the resulting factorized expression is larger, leading to a larger overhead for sentence generation. Indeed, as our breakdown analysis (Figure 13) shows, the increase in running time occurs both in the factorization and in the sentence generation time. Finally, Figure 12c shows the expected increase in the factorized expression size w.r.t the number of unique values.

6. RELATED WORK

NL interfaces. Multiple lines of work (*e*.g. [34, 4, 35, 48, 43, 47, 1]) have proposed NL interfaces to formulate database queries, and additional works [20] have focused on presenting the *answers* in NL, typically basing their translation on the schema of the output relation. Among these, works such as [4, 35] also harness the dependency tree in order to make the translation form NL to SQL by employing *mappings* from the NL query to formal terms. The work of [33] has focused on the complementary problem of translating SQL *queries* (rather than their answers or provenance) to NL. To our knowledge, no previous work has focused on formulating the *provenance* of output tuples in NL. This requires fundamentally different techniques (*e*.g. that of factorization and summarization, building the sentence based on the input question structure, etc.) and leads to answers of much greater detail.

Provenance. The tracking, storage and presentation of provenance have been the subject of extensive research in the context of database queries, scientific workflows, and others (see *e*.g. [7, 28, 25, 9, 26, 14, 13, 24]) while the field of provenance applications has also been broadly studied (*e*.g. [16, 39, 45]). A longstanding challenge in this context is the complexity of provenance expressions, leading to difficulties in presenting them in a user-comprehensible manner. Approaches in this respect include showing the provenance in a graph form [46, 40, 29, 19, 14, 12, 2], allowing user control over the level of granularity ("zooming" in and out [11]), or otherwise presenting different ways of provenance visualization [28]. Other works have studied allowing users to query the provenance (*e*.g. [31, 30]) or to a-priori request that only parts of the provenance are tracked (see for example [16, 22, 23]). Importantly provenance factorization and summarization have been studied (*e*.g., [8, 5, 42, 44]) as means for compact representation of the provenance. Usually, the solutions proposed in these works aim at reducing the size of the provenance but naturally do not account for its presentation in NL; we have highlighted the different considerations in context of factorization/summarization in our setting. We note that *value-level provenance* was studied in [41, 10] to achieve a fine-grained understanding of the data lineage, but again do not translate the provenance to NL.

7. CONCLUSION AND LIMITATIONS

We have studied in this paper, for the first time to our knowledge, provenance for NL queries. We have devised means for presenting the provenance information again in Natural Language, in factorized or summarized form.

There are two main limitations to our work. First, a part of our solution was designed to fit `NaLIR`, and will need to be replaced if a different NL query engine is used. Specifically, the "sentence generation" module will need to be adapted to the way the query engine transforms NL queries into formal ones; our notions of factorization and summarization are expected to be easier to adapt to a different engine. Second, our solution is limited to Conjunctive Queries. One of the important challenges in supporting NL provenance for further constructs such as union and aggregates is the need to construct a concise presentation of the provenance in NL (e.g. avoiding repetitiveness in the provenance of union queries, summarizing the contribution of individual tuples in aggregate queries, etc.).

Acknowledgements. This research was partially supported by the Israeli Science Foundation (ISF, grant No. 1636/13), and by ICRC - The Blavatnik Interdisciplinary Cyber Research Center.

8. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[2] A. Ailamaki, Y. E. Ioannidis, and M. Livny. Scientific workflow management by database management. In *SSDBM*, pages 190–199, 1998.

[3] E. Ainy, P. Bourhis, S. B. Davidson, D. Deutch, and T. Milo. Approximated summarization of data provenance. In *CIKM*, pages 483–492, 2015.

[4] Y. Amsterdamer, A. Kukliansky, and T. Milo. A natural language interface for querying general and individual knowledge. *VLDB*, pages 1430–1441, 2015.

[5] N. Bakibayev, D. Olteanu, and J. Zavodny. FDB: A query engine for factorised relational databases. *PVLDB*, pages 1232–1243, 2012.

[6] P. Brgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Springer Publishing Company, Incorporated, 2010.

[7] P. Buneman, S. Khanna, and W. chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.

[8] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, pages 993–1006, 2008.

[9] J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, pages 379–474, 2009.

[10] L. Chiticariu, W. C. Tan, and G. Vijayvargiya. Dbnotes: a post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.

[11] S. Cohen-Boulakia, O. Biton, S. Cohen, and S. Davidson. Addressing the provenance challenge using zoom. *Concurr. Comput. : Pract. Exper.*, pages 497–506, 2008.

[12] D. Cohn and R. Hull. Business artifacts: A data-centric approach to modeling business operations and processes. *IEEE Data Eng. Bull.*, pages 3–9, 2009.

[13] S. B. Davidson, S. C. Boulakia, A. Eyal, B. Ludäscher, T. M. McPhillips, S. Bowers, M. K. Anand, and J. Freire. Provenance in scientific workflow systems. *IEEE Data Eng. Bull.*, pages 44–50, 2007.

[14] S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, pages 1345–1350, 2008.

[15] D. Deutch, N. Frost, and A. Gilad. Nlprov: Natural language provenance. *Proc. VLDB Endow.*, pages 1900–1903, 2016.

[16] D. Deutch, A. Gilad, and Y. Moskovitch. Selective provenance for datalog programs using top-k queries. *PVLDB*, pages 1394–1405, 2015.

[17] K. Elbassioni, K. Makino, and I. Rauf. On the readability of monotone boolean formulae. *JoCO*, pages 293–304, 2011.

[18] M. Emms. Variants of tree similarity in a question answering task. In *Proceedings of the Workshop on Linguistic Distances*, pages 100–108, 2006.

[19] I. Foster, J. Vockler, M. Wilde, and A. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. *SSDBM*, pages 37–46, 2002.

[20] E. Franconi, C. Gardent, X. I. Juarez-Castro, and L. Perez-Beltrachini. Quelo Natural Language Interface: Generating queries and answer descriptions. In *Natural Language Interfaces for Web of Data*, 2014.

[21] `https://github.com/navefr/NL_Provenance/`.

[22] B. Glavic. Big data provenance: Challenges and implications for benchmarking. In *Specifying Big Data Benchmarks - First Workshop, WBDB*, pages 72–80, 2012.

[23] B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.

[24] B. Glavic, R. J. Miller, and G. Alonso. Using sql for efficient generation and querying of provenance information. In *In Search of Elegance in the Theory and Practice of Computation*, pages 291–320. Springer, 2013.

[25] T. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.

[26] T. J. Green. Containment of conjunctive queries on annotated relations. In *ICDT*, pages 296–309, 2009.

[27] E. Hemaspaandra and H. Schnoor. Minimization for generalized boolean formulas. In *IJCAI*, pages 566–571, 2011.

[28] M. Herschel and M. Hlawatsch. Provenance: On and behind the screens. In *SIGMOD*, pages 2213–2217, 2016.

[29] D. Hull et al. Taverna: a tool for building and running workflows of services. *Nucleic Acids Res.*, pages 729–732, 2006.

[30] Z. G. Ives, A. Haeberlen, T. Feng, and W. Gatterbauer. Querying provenance for ranking and recommending. In *TaPP*, pages 9–9, 2012.

[31] G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, pages 951–962, 2010.

[32] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Annual Meeting on Association for Computational Linguistics*, pages 423–430, 2003.

[33] G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Explaining structured queries in natural language. In *ICDE*, pages 333–344, 2010.

[34] D. Küpper, M. Storbel, and D. Rösner. Nauda: A cooperative natural language interface to relational databases. SIGMOD, pages 529–533, 1993.

[35] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, pages 73–84, 2014.

[36] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, pages 313–330, 1993.

[37] M. Marneffe, B. Maccartney, and C. Manning. Generating typed dependency parses from phrase structure parses. In *LREC*, pages 449–454, 2006.

[38] MAS. `http://academic.research.microsoft.com/`.

[39] A. Meliou, Y. Song, and D. Suciu. Tiresias: a demonstration of how-to queries. In *SIGMOD*, pages 709–712, 2012.

[40] P. Missier, N. W. Paton, and K. Belhajjame. Fine-grained and efficient lineage querying of collection-based workflow provenance. In *EDBT*, pages 299–310, 2010.

[41] T. Müller and T. Grust. Provenance for SQL through abstract interpretation: Value-less, but worthwhile. *PVLDB*, pages 1872–1875, 2015.

[42] D. Olteanu and J. Závodný. Factorised representations of query results: Size bounds and readability. In *ICDT*, pages 285–298, 2012.

[43] A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 149–157, 2003.

[44] C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *Proc. VLDB Endow.*, pages 797–808, 2008.

[45] S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, pages 1579–1590, 2014.

[46] Y. L. Simmhan, B. Plale, and D. Gannon. Karma2: Provenance management for data-driven workflows. *Int. J. Web Service Res.*, pages 1–22, 2008.

[47] D. Song, F. Schilder, and C. Smiley. Natural language question answering and analytics for diverse and interlinked datasets. In *NAACL*, pages 101–105, 2015.

[48] D. Song, F. Schilder, C. Smiley, C. Brew, T. Zielund, H. Bretz, R. Martin, C. Dale, J. Duprey, T. Miller, and J. Harrison. TR discover: A natural language interface for querying and analyzing interlinked datasets. In *ISWC*, pages 21–37, 2015.