Noname manuscript No.
(will be inserted by the editor)

Explaining Natural Language Query Results

Daniel Deutch · Nave Frost · Amir Gilad

Received: date / Accepted: date

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

In the context of databases, **data provenance** captures the way in which data is used, combined and manipulated by the system. Provenance information can for instance be used to reveal whether data was illegitimately used, to reason about hypothetical data modifications, to assess the trustworthiness of a computation result, or to explain the rationale underlying the computation.

As database interfaces constantly grow in use, in complexity and in the size of data they manipulate, provenance tracking becomes of paramount importance. In its absence, it is next to impossible to understand the system's operation and to follow the flow of data through the system, which in turn may be extremely harmful for the quality of results.

A setting where the lack of provenance – and consequently lack of explanations – is of particular concern, is that of database interfaces geared to be used by non-experts. Such non-expert users lack understanding of the system inner workings, and are unable to verify that it has operated correctly. Indeed, an important component of such systems is the interface through which the non-expert communicates her needs/query to the system. But then, how does the system communicate its results back to the non-expert user? And how does it justify it in a manner that the non-expert can understand? For each system, developers currently need to develop dedicated solutions, if at all, and we are lacking a generic framework for explanations in this setting.

A particularly flourishing line of work for allowing non-experts to interact with a database, is that of Natural Language Interfaces to Databases (NLIDBs). Multiple such interfaces have been developed in recent years (see *e*.g. [55, 7, 52, 76]). The accuracy of translation is constantly improving. Still, it is far from perfect – in general, automated translation of free text to a formal language is an extremely difficult task. Since the users of such systems are typically non-experts, they may have a hard time understanding the result or verifying its correctness. Consider for example a complex NL

Daniel Deutch
Tel Aviv University
E-mail: danielde@post.tau.ac.il

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

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

query over a publication database, of the form "return all organizations of authors who published in database conferences after 2005". After translating this query to SQL and running it using a query engine, the answer is a list of qualifying organizations. By looking at the answer, the user has no way of knowing whether the retrieved organizations really satisfy her specified constraints; a slight error in the translation process, e.g. misunderstanding "database conferences" or erroneously associating "after 2005" with the conference inauguration date, could result in a list of organizations that is completely wrong for the question asked.

In this work we complement the efforts of developing high-quality NLIDBs, by developing a generic framework that *explains* the results of queries posed to NLIDBs. Explanations are based on provenance, but current provenance models are far too complex to allow for their direct presentation to non-experts. The novelty of our work is that we "translate" provenance into *NL explanations to the 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 [1] and consider the NL query in Figure 1a. A state-of-the-art NL query engine, NaLIR [55], 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* [41, 14, 17, 38, 39] 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. An understanding of the results also allows users to conclude whether their query was translated correctly and reproduce the results if needed. There are several models of provenance previously suggested in the literature. The *tuple-based model* [38, 39] tracks the source tuples which participated in the computation of the results, while the *value-based model* [61, 18] is a more fine grained model and follows the values of these tuples.

We propose a novel approach of presenting *provenance information for answers of NL queries, again as sentences in Natural Language*. There are several aspects to account for towards a solution, as follows.

- The provenance model needs to be very detailed. For example, the NL explanations that we aim for require storing not only which input tuples have contributed to the answer in the above example these may e.g. be the author, organization and publication entries but also the way in which they contributed to the answer. In our example, for generating the required explanations we need to store that the organization entry has matched the query head and was returned, that the author entry has been joined with it to find authors of the specific organization, that the publication entry was joined with the author entry, etc. Naturally, once the query is compiled from NL/examples to e.g. SQL, one could in principle track provenance as if the query was described in SQL to begin with. As we next explain, this would be sub-optimal.
- As we shall demonstrate, the way in which the user has phrased the question has a significant impact on both which parts of the computation needs to be tracked and on the way in which users expect provenance information to be presented to them. In general, in works on provenance, there is a typically a tight coupling between the query model and the provenance model. In particular, as we already observed, a suitable way for presenting the explanations is again as NL sentences, so that we obtain an end-to-end system where questions, answers and explanations are all expressed in Natural Language. Thus, the provenance model needs to keep track of which parts of the NL question have contributed to which parts of the computation. Furthermore, We use the value-based model of provenance as it is the implicitly enforced by the NLIDB which maps words to variables. Once we have this mapping, we store the mappings between variables to values to be able to reverse it. A major challenge in this respect is to design the model so that it correctly captures those parts of the provenance that are "important" based on the user question. As the basis for our provenance model, we use the value-based model of provenance (as opposed to tuple-based) as it is the implicitly enforced by the NLIDB which maps words to variables in the query. Once we have this mapping, we store the mappings between variables to values to be able to assemble an explanation sentence.

```
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

Fig. 1: NL Query and CQ Q

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

Fig. 2: Answer for a Single Assignment

- Then, given the tracked provenance, we further need to translate it back from the formal model to an NL sentence. Generating NL sentences is a difficult task in general - but importantly, here we have the NL question that can guide us. A challenge is then how to "plug-in" different parts of the provenance back into the NL question, to obtain a coherent, well-formed answer.
- Last, we need to address the challenge of provenance size. In particular, full information about the manner in which a result is obtained from the input data (and even full description of the input data itself) is typically exhaustively long to present, especially to a non-expert.

The end result for our running example is demonstrated in Figure 2, which shows one of the explained answers outputted by our system in response to the NL query in Figure 1a.

Having explained the overall approach and challenges, we next provide more details on each of our key contributions.

Provenance Tracking Based on the NL Query Structure As mentioned above, a first key idea in our solution is to leverage the *NL query structure* in constructing NL provenance. Our solution is generic in that it is not specific to a concrete NL interface (we do have some requirements on the operation of the underlying interface, as we detail below). In our implementation, we use and 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 [26]), 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 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 [16,62], 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.* [5,66]), 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

[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.

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 [23], forming an end-to-end NL interface to database querying where the NL queries, answers and provenance information are all expressed in NL. 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.

This paper is an extended version of our PVLDB 2017 paper [24] and includes a new section on the translation of provenance to NL for UCQs, a new section on a generalized solution that is not specific to NaLIR and a discussion of the use of other provenance models, new and comprehensive experiments, and an extended in-depth review of related work.

2 Preliminaries

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

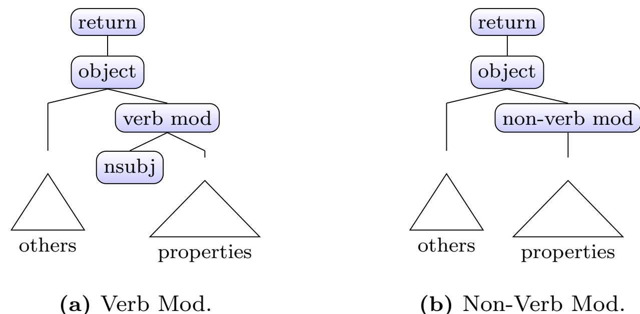
2.1 NL and Formal Queries

We start by recalling some basic notions from NLP, as they pertain to the translation process of NL queries to a formal query language. We further recall a particular formal query language of interest, namely Union of Conjunctive Queries.

A key notion that we will use is that of the *syntactic dependency tree* of NL queries:

Definition 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 [49,57] ; (2) Relationship (REL): the grammatical relationship between the word and its parent in the dependency tree [58].

We focus on a sub-class of queries handled by NaLIR, namely that of Union of Conjunctive Queries, possibly with comparison operators $(=, >, <)$ and logical combinations thereof (NaLIR further supports nested queries and aggregation). Formally, fix a database schema, i.e. a set of relation names along with their arities (number of attributes). A query is then defined with respect to a schema.



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

Fig. 3: Abstract Dependency Trees

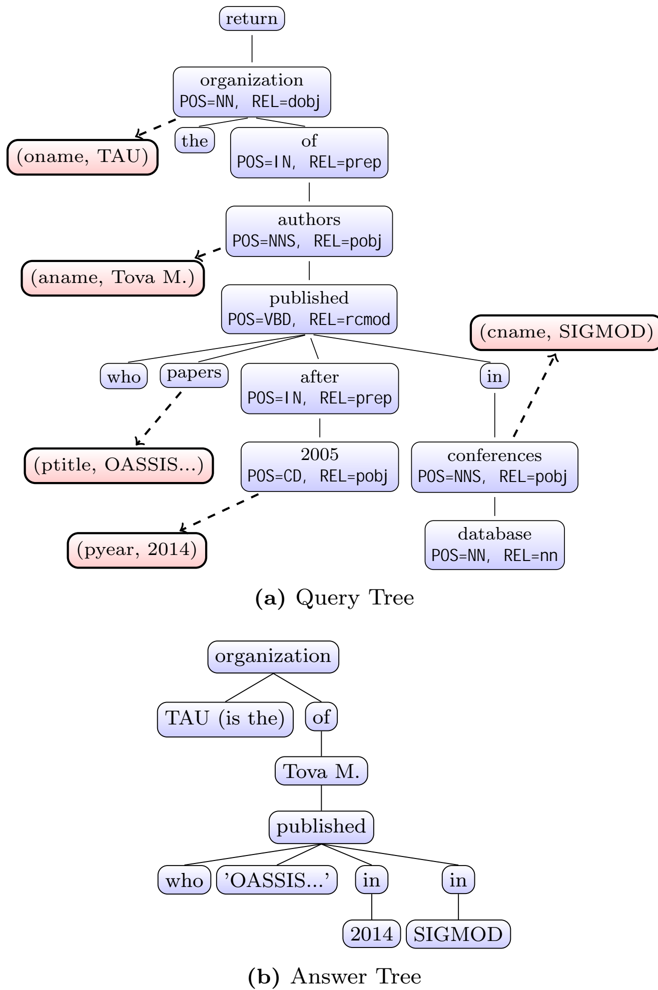
Definition 2 (From [2]) A Union of Conjunctive Queries Q is a set of Conjunctive Queries Q_i.

In turn, a conjunctive query is an expression $ans(u) \longleftarrow R1(u1), ..., Rn(un), C$ where $R1, ..., Rn$ are relation names in the database schema, and $u, u1, ..., un$ are tuples with either variables or constants, with ui conforming to the schema of Ri. Variables in u must appear in at least one of $u1, ...un$. Finally, C is a sequence of comparison constraints $(=, >, <)$ over variables in $u1, ...un$ and constants.

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 in our algorithms that follow).

Example 1 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 subsentence. 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 Union of Conjunctive Queries (UCQs), and discuss extensions to aggregates and nested queries below.



(a) Query Tree

(b) Answer Tree

Fig. 4: Question and Answer Trees

Example 2 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. This mapping is not a novel contribution of this paper, but we will employ this mapping to generate the NL explanation.

Definition 3 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.

As explained in Section 1, there are essentially two explanation models that will come into play here. The first is a provenance model for the underlying formal query, in our case Union of Conjunctive Queries. We next discuss existing provenance models, then choose a particular model that fits our construction. The second, which we will discuss below, is coupled with the Natural Language model.

In terms of provenance for formal database queries, previous work has proposed a large number of different models (see Section 7 for an overview of related work). A basic distinction that we already need to make is between fine-grained and coarse-grained provenance. Generally speaking, the former keeps track of which tuples (or even cells) have contributed to each result, while the latter keeps track of the general input and output of each query/workflow operator, without necessarily connecting each input and output pieces. Here our goal is to explain individual query results, and so a fine-grained provenance model is sought for.

In context of database queries, capturing fine-grained provenance means that we keep track of the *assignments* of database tuples to query atoms. Assignments are the basic building block of query evaluation, and for UCQs they are defined as follows:

Definition 4 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 $\alpha(x_i)$. The set of assignments to Q with respect to D is denoted by $\mathit{\Gamma}(Q, D)$. Note that a single tuple in the query result may have been obtained by multiple assignments in $\mathit{\Gamma}(Q, D)$.

Then, for a UCQ Q whose CQs are $Q_1, ..., Q_n$, the set of assignments to Q is defined as the union of the sets of assignments to its CQs, namely $\mathit{\Gamma}(Q) = \bigcup_{i=1}^{n} \mathit{\Gamma}(Q_i)$.

```
(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')
```

Fig. 5: Value-level Provenance

The notion of a tuple being obtained from an assignment and the α notation immediately extend, noting that a single tuple may be obtained from assignments to multiple conjunctive queries.

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 3 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).

Assignments may be used in provenance in multiple ways, varying in their granularities. For instance, the lineage [10] of a result tuple t is the set of input tuples appearing in some assignment yielding t; the why-provenance of t is the set of sets of tuples participating in such assignments, i.e. the contributing tuples are grouped based on the assignments they are used in. These approaches were shown in [38] to be concrete examples of a general algebraic construction, termed semiring provenance. At a high-level, the idea there is that we introduce two symbolic operations " + " and " · ", and use them to form algebraic representations of the provenance. Concretely, "+" is used for alternative derivations and "·" is used for combined derivation: for a given output tuple, we sum over the assignments that have yielded it, and each assignment is represented via a multiplication over the terms that has contributed to it. 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).

In [38], the basic atomic units that appear in a provenance expression are the "annotations" (intuitively identifiers, or meta-data associated with the tuples) of the *tuples* that contribute to an assignment. Here, in order to form a detailed explanation of the result of an NL query, we need to keep track of a finer-grained resolution. 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.

Definition 5 Let $A(Q, D)$ be the set of assignments for a UCQ Q and a database instance D. We define the *value-level provenance* of Q w.r.t. D as

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

.

The reason for our use of a value-based rather than tuple-based provenance model is that, as we will next show, we wish to connect different pieces of the provenance back to the NL question, in order to form a detailed NL explanation.

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	dname
18	Databases

Fig. 6: DB Instance

Example 4 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.

It is important to note that we refer to provenance as the mapping between the variables in the query to the values in the database which occurs during the evaluation process. This process is completely separate from NaLIR's framework. The provenance is stored as part of the evaluation of the formal query inferred by NaLIR over the database, and is therefore performed after NaLIR has completed the query inference process.

3 First step: Conjunctive Queries and a Single Assignment

We now start describing our transformation of provenance to an NL sentence, leveraging the structure of the original question. We focus in this section on the case of a Conjunctive Query and a single assignment to its clauses. In subsequent sections we show how to extend the solution to multiple assignments and unions of conjunctive queries, where the solution presented in this section will serve as a building block.

3.1 Mapping NL to Provenance and Back

Our first important observation is that *words* in the NL question can be connected to *(variable,value) pairs* in the provenance polynomial. For instance, "conference" corresponds to the assignment of *cname* to $SIGMOD$ or $VLDB$, "author" corresponds to the assignment of *aname* to $TovaM.$, and so on. The reason this connection is important is that it gives us strong hints on how to form a detailed answer in Natural Language: given this information we know for instance that $SIGMOD$ should replace/reside next to "database conferences" (the decision of which of the two options to follow will be discussed below based on the sentence structure). Fortunately, the choice of models we have made in the preliminaries gives us relatively straightforward means to derive this mapping. The idea is to marry the two mappings discussed in the previous section as a step towards generating an NL explanation: the dependency-to-query-mapping performed by NaLIR and the value-based provenance to get a direct mapping from words to database values. First, we have the dependency-to-query-mapping mapping (Definition 3) from the NL query's dependency tree (e.g. "author") to query variables (e.g. "aname"), which we get from the NLIDB. Second, we have, in the value-based provenance, a detailed account of the assignments of query variables to values from the database (e.g. "aname" to "Tova. M."). If we compose this mapping, we get a (partial) mapping from words in the NL question to data values.

Example 5 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".

3.2 Building an Answer Tree

Having established the mapping from words in the NL query to values in the provenance, we are ready to form a basic tree for the provenance-aware answer. The idea is now to 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 6 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, under some plausible assumptions on the structure of the question tree.

Recall that we have assumed 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 an-

swer 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 1).

Example 7 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).

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 ∈ 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;
```

3.3 From Answer Tree to an Answer Sentence

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. When generating the sentence, we have two different scenarios; when a word or phrase in the original dependency tree was replaced by the value to which it was mapped to, we replace the word/phrase in the NL query with the value mapped to it. Otherwise, the value mapped to it was added as its child, and in this case we add it either before or after the mapped word/phrase according to its POS with the appropriate connecting word taken from a stored table.

Example 8 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.

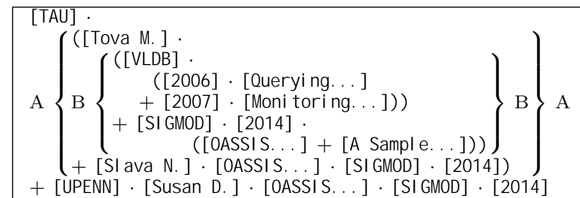
3.4 Logical Operators

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 (we consider here logical operators that are compiled by NaLIR into a single CQ, the UCQ case is considered in Section 6). 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).

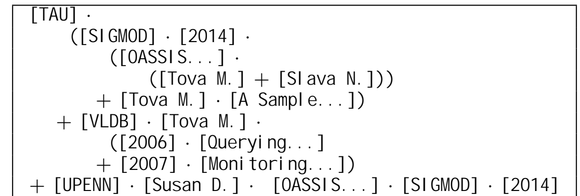
Procedure HandleLogicalOps

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

Fig. 7: Provenance Factorizations

4 Factorized Explanations for Multiple Assignments

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. This is the case already for Conjunctive Queries, as illustrated in Section 2. We next generalize the construction to account for multiple assignments. 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 [62, 16], and the second (in the following section) leveraging factorization to provide a summarized form.

4.1 NL-Oriented Factorization

Provenance size and complexity is a known and well-studied issue, and various solutions were presented to reduce it [26, 8]. Observing that different assignments in the provenance expression typically share significant parts, one prominent approach [8, 62, 16] suggests using algebraic factorization. The idea is to regard the provenance as a polynomial (see Figure 5) and use distributivity to represent it in a more succinct way. For instance, the expression $x \cdot y + x \cdot z$ can be factorized to the equivalent expression $x \cdot (y + z)$.

The main purpose of classical provenance factorization, as in algebraic factorization, is to reduce the size of the provenance by removing duplicate records and nodes. In our setting, different considerations come into play, as we shall show.

We start by defining the notion of factorization in a standard way (see *e*.g. [62,27]).

Definition 6 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 9 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 [62,27] 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 7 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 10 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* [13] of a given factorization:

Example 11 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.

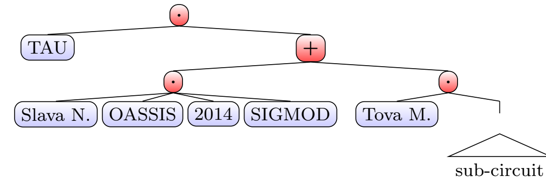


Fig. 8: Sub-Circuit of f_1

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 3, 4, respectively).

Definition 8 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 9 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 12 Recall the partial order $\leq_T$ imposed by our running example query, shown in part in Example 10. 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.

As a sanity check, note that the identity factorization that simply keeps the provenance intact is T-compatible. Further, T-compatible factorizations are factorizations that keep the answer (the *object* node in Figure 3) at the start of the sentence and only then refer to the provenance. When many answers and assignments are involved, it is thus possible to obtain a T-compatible factorization by factorizing each answer with its provenance by itself and then combining all of them under a joint root.

Example 13 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 [27]). 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.

As mentioned above, 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 factorization so that its 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) [40]. 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 12) 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 [40] as well). In general, obtaining the minimum size T-compatible factorization of $prov_T$ is coNP-hard by a reduction from [40].

Greedy Algorithm. Despite this result, the constraint of compatibility does help in practice, in that we can avoid examining choices that violate it. For choices that do maintain compatibility, 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 8).

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 ← prov;
2  Frontier ← {x ∈ V(T_Q) | ∀(y ∈ V(T_Q) \ Processed) s.t. x ≰_{T_Q} y};
3  vars ← sortByFrequentVars({α(τ(x)) | x ∈ Frontier}, f);
4  foreach var ∈ vars do
5      Take out var from sub-expressions in f not
       including variables from
       {x | ∃y ∈ Processed : x = α(τ(y))};
6  Processed ← Processed ∪ Frontier;
7  if |Processed| = |V(T_Q)| then
8      return f;
9  else
10     return
       GreedyFactorization(T_Q, f, τ, α, Processed);
```

Example 14 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.

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 $O(n \cdot \log n)$.

Optimization Since T-compatible factorizations keep the answer (the *object* node in Figure 3) at the start of the sentence, we can utilize the *abstract factorization structure* for one answer in the factorization of all other answers. For this, we need to augment Algorithm 2 in the following manner. First, only the monomials that contain the first answer will be factorized using Algorithm 2. Then, an abstract factorization structure f_a can be inferred from this factorization by replacing some of the values with the variables mapped to them. The values that are replaced with variables are those that have a clear hierarchy between them in T_Q while values that were mapped to words in the same level of T_Q are not part of the abstract factorization structure as the hierarchy between them may vary based on the nature of the assignments each results has. Namely, if $x \neq y \in V(T_Q)$ satisfy $x \leq_{T_Q} y$, the variables that x and y are mapped to will be part of f_a and will hold $var(x) \leq_{f_a} var(y)$ where $var(x)$ is the variable x is mapped to. Intuitively, the circuit induced by f_a maintains the partial order of nodes in T_Q. Finally, given the provenance polynomial of another answer, we replace the variables in f_a with the corresponding constants and greedily factorize only the parts of the polynomial that are not included in f_a.

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 15 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 $\cdot$ 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 $[authors, TovaM.]$.

Intuitively, “of” is indeed the root of a sub-tree specifying the authors in an institution in the structure of our answers.

5 From Factorized to Summarized Answers

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 16 Reconsider Example 15; 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

```
(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])
```

Fig. 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.
```

Fig. 10: Summarized Sentences

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 17 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.

6 Unions of Conjunctive Queries

So far our solution has been limited to Conjunctive Queries, and we next extend it to account for Unions thereof (UCQs). We next describe the necessary augmentations of the algorithms, illustrating them through examples. Recall that in the first step, the system takes a natural language query and translates it to a dependency tree, while maintaining the dependency-to-query-mapping mapping. The difference here is that a tree node can now be mapped to several variables. This implies a generalization of Definition 3:

Definition 10 Given a dependency tree $T = (V, E, L)$ and a UCQ $Q_1, \ldots, Q_m$, a dependency-to-UCQ-mapping is a set of dependency-to-query-mapping $\{\tau_1, \ldots, \tau_m\}$, where $\tau_i : T \to Q_i$.

Example 18 Consider the NL query "return the organization of authors who published papers in database conferences before 2005 or after 2015", whose dependency tree is depicted in Figure 11. The "or" here defines two different CQs (depicted in Figure 12). Since the two numerical values cannot form a conjunctive condition and thus cannot be compiled into a single boolean condition, NaLIR translates this NL query into two CQs. The two CQs define two different dependency-to-query-mapping that map nodes from the single dependency tree to two different sets of variables. Consider an organization (e.g., TAU) which appears as an answer. It is mapped both to $oname1$ and to $oname2$. Thus, we would like to present the assignments to both queries as explanations. Here, the dependency-to-UCQ-mapping is $\{\tau_1, \tau_2\}$ where τ_1 maps the nodes from the dependency tree to the variables of the first query in Figure 12 and τ_2 maps the nodes from the dependency tree to the variables of the second query. Thus, the dependency-to-UCQ-mapping captures the assignments from both queries. Note that τ_1 differs from τ_2 for some words. Specifically, τ_1 maps the nodes "before" and "2005" to $pyear1 < 2005$ and τ_2 maps the nodes "after" and "2015" to $pyear2 > 2005$.

We further give an unique integer identifier to each mapped word in the dependency tree as exemplified by the superscript in Figure 11 for reasons we explain in the sequel.

After determining the dependency-to-UCQ-mapping, we rely on an augmentation of Definition 5. The following definition essentially generalizes the definition for CQs by also summing the pairs of (word identifier, value) from all the CQs participating in the union.

Definition 11 Let $A(Q, D)$ be the set of assignments for a UCQ $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ and a database instance

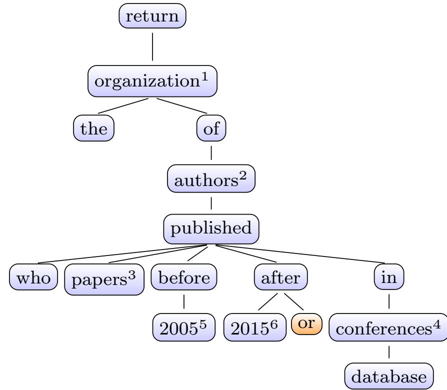


Fig. 11: Dependency Tree With "Or" Condition

```
query(oname1) :- org(oid1, oname1), author(aid1, aname1,
 oid1), pub(wid1, cid1, ptitle1, pyear1), conf(cid1,
cname1), domainConf(cid1, did1), domain(did1, dname1),
writes(aid1, wid1), dname1 = 'Databases', pyear1 < 2005
```

```
query(oname2) :- org(oid2, oname2), author(aid2, aname2,
 oid2), pub(wid2, cid2, ptitle2, pyear2), conf(cid2,
cname2), domainConf(cid2, did2), domain(did2, dname2),
writes(aid2, wid2), dname2 = 'Databases', pyear2 > 2015
```

Fig. 12: Two CQs from the Same NL Query

D, and let $\{\tau_1, \ldots, \tau_m\}$ be the dependency-to-UCQ-mapping of $\mathcal{Q}$. We define the *NL value-level provenance* of $\mathcal{Q}$ w.r.t. D as

$$\sum_{Q_i \in \mathcal{Q}} \sum_{\alpha \in A(Q_i, D)} \Pi_{\{x_i, a_i | \alpha(x_i) = a_i\}} (\tau_i^{-1}(x_i), a_i).$$

Rel. *org*

oid	oname
2	TAU

Rel. *author*

aid	aname	oid
4	Tova M.	2

Rel. *pub*

wid	cid	ptitle	pyear
6	10	"Positive Active XML"	2004
7	11	"Rudolf..."	2016

Rel. *writes*

aid	wid
4	6
4	7

Rel. *conf*

cid	cname
10	PODS
11	VLDB

Rel. *domainConf*

cid	did
10	18
11	18

Rel. *domain*

did	dname
18	Databases

Fig. 13: DB Instance for Example 19

Example 19 Reconsider the UCQ defined by the union of the two CQs depicted in Figure 12 and the database in Figure 13 with tuples standing for two more publications by the author Tova Milo: "Positive Active XML" published in PODS in 2004 and "Rudolf: Interactive Rule Refinement System for Fraud Detection" published in VLDB in 2016. The first of the two summands in Figure 14 (in the "A" brackets) stands for an assignment to the top query in Figure 12, while the second summand (in the "B" brackets) stands for an assignment for the bottom query. Assignments are represented as multiplication of pairs of the form (id, val) so that id is the unique identifier of a word in the NL query mapped to the variable var in a specific query Q_i that is assigned val in the particular assignment.

```
A { (1,TAU)·(2,Tova M.)·(3,Positive Active XML)·
  { (4,PODS)·(5,04')+                              } A
B { (1,TAU)·(2,Tova M.)·(3,Rudolf...)·
  { (4,VLDB)·(6,16')+...                } B
```

Fig. 14: Value-level Provenance for Example 19

We now have a polynomial containing sets of pairs where the first element is the unique word in the NL query and the second is the value from the database mapped to it. This allows us to consider explanations for the same answer *regardless of the query from which they originated.*

By replacing the variable name in each pair with the unique word identifier from the NL query, we are able to treat the assignment of different variable names as relating to the same word or phrase in the NL query. This allows us to factorize the provenance of the different queries in the union in the context of a single NL query to which we will build a single NL answer. Now, we can use the procedure described in Section 4 to produce a T-compatible factorization and summarization of the provenance. The only change needed in Algorithm 3 is to replace all nodes that form the logical "or" condition with the words mapped to them. In our example, replacing the subtrees rooted at "before" and "after" with the year from the provenance assignments.

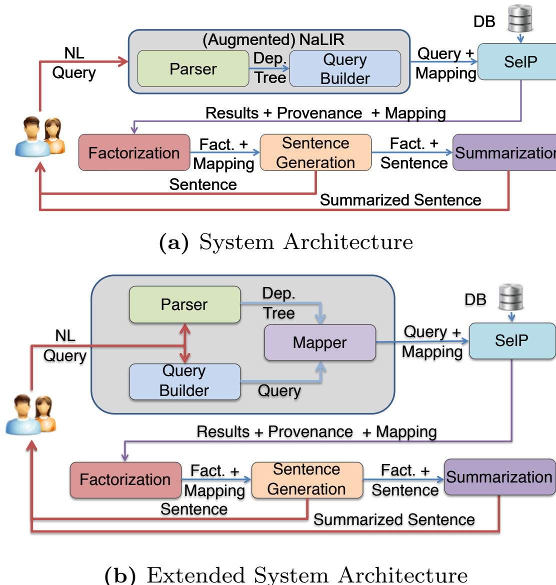
7 Implementation and Generalizations

7.1 Implementation

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 [34]). Figure 15a 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 [58] 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. In fact, NaLIR may generate several candidate queries, from which it will choose the one that is ranked highest according to an internal ranking function. We use the highest ranked as the chosen 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 [26] 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.

User Interface We now discuss the user interface NLProv. First the user writes a natural language question in the web interface. The question is inputted to the augmented NaLIR box, converted to an SQL query while storing the mapping from words to variables and evaluated over the database, where the query results . All results are then shown to the user, where each result can be further explored by viewing its natural language provenance, in each of the three forms described earlier: an explanation formed by a single assignment, an explanation which encapsulates all assignments as a factorized representation of the provenance, and a summarized explanation based on the factorization.



(a) System Architecture

(b) Extended System Architecture

Fig. 15: NLProv and General Architectures

7.2 Replacing the Black Boxes

Our solution "marries", for the first time to our knowledge, two fields: (1) Natural Language Interfaces to Databases, and (2) Data Provenance. For each of these two, we have made choices in our implementation: NaLIR for the NLIDB, as well as a semiring-like value-level provenance model. We next revisit these choices and discuss alternatives in detail.

7.2.1 Alternatives NLIDBs

As mentioned above, NaLIR is a prominent interface for querying relational databases in Natural Language. Yet the problem of transforming an NL query into a formal query has been researched extensively, by both the database and NLP communities, and it includes a variety of different approaches for the solution. As the field keeps progressing, the question rises: how flexible is our approach of Natural Language Provenance, with respect to NLIDB development? Namely, if an improved NLIDB is developed, can it be incorporated in our framework?

To address this question, we will analyze our requirements and briefly discuss state-of-the-art algorithms for the problem of translating text to SQL or similar formal languages, reviewing their compatibility to these requirements and consequently the possibility of their binding to NLProv. Further in-depth discussion of the works themselves appears in our review of related work in Section 10.

The DB community has been studying *Natural Language Interfaces to Databases* for several decades. Many solutions focus on matching the query parts to the DB schema, and infer the SQL based on this mappings, Obtaining a matching from natural language query to the DB schema in various ways, such as pattern matching, grammar matching, or intermediate representations language (see Section 9 for more details).

The NLP community has also extensively studied the translation of natural language question to logical representations that query a knowledge base [79,56,11, 9]. One of the earliest statistical models for mapping text to SQL was the PRECISE system [65,64]; it was able to achieve high precision on specific class of queries that were able to be linked tokens and database values, attributes, and relations. However, PRECISE did not attempt to generate SQL for questions outside this class. Later work considered generating queries based on relations extracted by a syntactic parser [33] and applying techniques from logical parsing research [63]. Recently there is a flourish of work on generating SQL [78,45, 80], typically applying Machine Learning methods such as seq2seq networks and reinforcement learning.

NLProv architecture, as depicted in Figure 15a and explained previously, is coupled with an augmented version of NaLIR in the following sense: we get from NaLIR both its translation to a formal query, along with the

dependency-to-query-mapping τ. These are the two essential factors for the operation of NLProv. That is, NLProv can use any existing system that transform natural language question to formal query and also return partial mapping from the dependency tree nodes to the query parts. Indeed, many of the other techniques for Natural Language Interface design mentioned above, could be adapted to return τ and support our requirements. For example PRECISE has a component called "matcher", that generates mapping from tokens and database values, attributes, and relations. However, not all of the methods described above are designed in a way that allows generation of this mapping. E.g., a semantic parser that relies on seq2seq Deep Neural Network may be unable to return this mapping. The DNN would be trained on a large corpus of natural language questions along with their relevant SQL queries, and its objective would be to generalize to new questions. Due to the network complex representation it may be hard to extract the desired mapping.

To this end, we propose an alternative architecture, depicted in Figure 15b. This architecture does not rely on the query builder to also generate the partial mapping τ from the dependency tree nodes to the query parts. Instead, we have added an additional block, *Mapper*, that receives as input the dependency tree along with the generated query and outputs the mapping τ. Note that generating the dependency tree may be done using existing tools such as the Stanford Parser, independently of whether the NLIDB generates it (as NaLIR does) or not (as is the case with semantic parsers).

Algorithm 4: Mapper

```
   input : Dependency tree nodes V,
           Conjunctive Query Q,
           Similarity Threshold β
   output: Partial Mapping τ
1  G_vertices := V ∪ VAR(Q);
2  G_edges := ∅;
3  foreach v ∈ V do
4      foreach q ∈ VAR(Q) do
5          if Sim(v, q) ≥ β then
6              e := (v, q);
7              e_weight := Sim(v, q);
8              G_edges := G_edges ∪ {e};
9  return MaximalMatching(G);
```

We then present Algorithm 4 responsible for the mapping generation. The algorithm is similar in spirit to the default mapping algorithms of NaLIR and PRECISE, but could be used as a stand-alone component without these systems. It generates a bipartite graph, with the dependency tree nodes at one side, and the query parts in the second side. For each pair the algorithm calculates a similarity between the two, and in case they are similar enough (similarity is higher than the input constant β) an edge will be generated with the corresponding weight. Eventually, the algorithm will perform maximal matching, and will return the mapping τ with the highest match score.

Note that the similarity threshold β balance between the mapping precision and recall. Low β values will enable more edges in the bipartite graph, which results in higher recall. However, more edges may introduce noise, which in turn will be harmful to the precision. For our use case it is crucial to have a mapping with high precision, hence high β values will be used.

Example 20 Recall our running example, and consider the two mapping functions presented in Figure 16. τ_1 depicted in the orange nodes has high recall, as all of the relevant tree nodes mapped to query parts, however it does not have perfect precision as *organization* node is incorrectly mapped to *aname* and *authors* node is mapped to *oname*. As a result our answer will be:

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

This answer makes no sense, and will cause the user to mistrusts the answer and the system. On the other hand τ_2, depicted in the green nodes, has perfect precision but low recall as *papers* and *conferences* nodes are not mapped to any variable; this will result in:

```
TAU is the organization of Tova M. who
published papers in database conferences
in 2014
```

Even though the answer does not supply all relevant information, it is a coherent sentence, and clearly a better answer than the previous one.

Since the dependency tree can be artificially made by our system from the NL query, the only recommended component of this NLIDB is a mapping from words of the NL query to the parts of the formal query. Therefore, any NLIDB with such a component could work well with our system (e.g. PRECISE [65] and ATHENA [68]). But even this component can be replaced by Algorithm 4 which artificially generates such a mapping. If we do use this algorithm, any NLIDB can be fitted to the system.

7.2.2 Alternative Provenance Models

We have used a detailed value-level provenance model for UCQs, which we have leveraged to connect different pieces of the provenance with different parts of the NL question, eventually resulting in a detailed answer. We next briefly discuss alternatives and extensions.

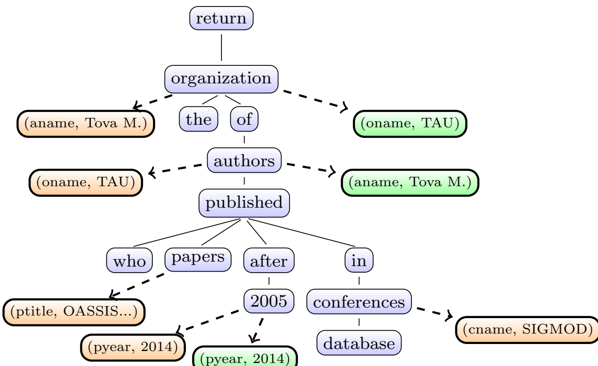


Fig. 16: Dependency-To-Query Partial Mappings

Tuple-level provenance Using value-level provenance has been essential in our construction of the NL representation of provenance, and consequently in the generation of answers with NL explanations.

If the system is connected to a system that only allows coarser-grained provenance, then the mapping between words and values needs to be otherwise constructed. Namely, one could use provenance that is only at a tuple-level, as is typically done in provenance models (including the standard semiring model [38], why-provenance [14], lineage [66], etc.). Then, considering all values in the tuple participating in the provenance, we can reconstruct the mapping to NL query words in alternative means, such as word embeddings and semantic similarity.

Operator-level provenance An even coarser grain view of the provenance is at the *operator level*. For instance, in the context of relational queries one may consider tracking tuples that are the input and output of each operator in the query plan, while not necessarily keeping track of which input contributed to which output. Can such form of provenance be useful in our setting?

One use-case of marrying operator-level provenance with NL queries is in the context of provenance for non-answers. Examining the set of input and output tuples of each operator in the query plan, the work of [15] defines the notion of a "picky" operator with respect to a tuple, as one that is responsible for its omission from the output. This opens up possibilities for explaining non-answers. In a recent preliminary work [25] we have combined NaLIR's mapping of words to query operators, with the work of [15] to identify the words that map to picky operators. Then, for each requested non-answer, we can highlight this word as "responsible".

Example 21 Reconsider our running example, but this time assume it is executed on a smaller dirty DB as depicted in Figure 17, where the papers "OASSIS: ..." and "A sample ..." are erroneously associated with the publication year 2004 instead of 2014. Due to the errors in the database "TAU" will not return as an answer to the query, and a user who expects to see "TAU" in the results screen will be interested to understand the reason for its absence (and fix the database accordingly). Consider the query evaluation plan in Figure 18, for the query in Figure 1a. The frontier picky operator for "TAU" is $\sigma_{pyear} > 2005$, thus the system depicted in [25] will highlight the relevant part in the NL query, and return

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

Indicating "TAU" is a non-answer because the authors associated with it did not published papers after 2005.

Similarly to the approach discussed here, the system utilizes the mappings from words to operators, constructed by the NLIDB, and highlights the relevant term which filtered the queried tuple. A challenge arises when the filtering operator has no direct word or phrase in the NL query mapped to it.

Example 22 Continuing Example 21, if the query was about an organization whose authors did not publish in any database conference after 2005, the filtering operator would have been the join between the *author* and *writes* tables. Since there is no direct mapping between a word in the NL query and the join operator, it is unclear which word/phrase to highlight.

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. *writes*

aid	wid
4	6
3	6
5	6
4	7

Rel. *pub*

wid	cid	ptitle	pyear
6	10	"OASSIS..."	2004
7	10	"A sample..."	2004

Rel. *conf*

cid	cname
10	SIGMOD

Rel. *domainConf*

cid	did
10	18

Rel. *domain*

did	dname
18	Databases

Fig. 17: Faulty DB Instance

Provenance Beyond UCQs A limitation of our work is that it is limited to the SPJU fragment of SQL (UCQs), while NLIDBs have considerable success in handling questions that compile to far more expressive formalisms. NaLIR in particular also supports nesting and aggregation, both lacking support in our solution.

Provenance models for such formalisms do exist, from [6] and [62] for aggregate queries, [53] for nested

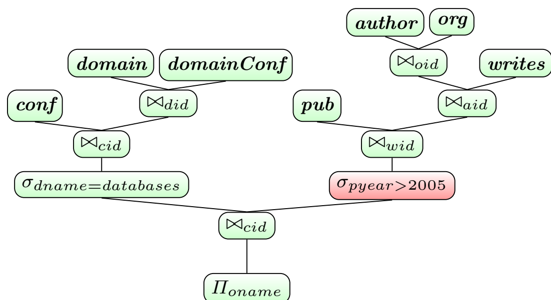


Fig. 18: Query Plan with Frontier Picky

queries, [54] for queries with negation to [35] for full SQL, just as few examples. By and large, these solutions are intended as internal representations. Presenting them as explanations is an important task that lack satisfactory solutions. For instance, the model of [6] for aggregate results includes in a sense a record of all tuples participating in the aggregate computation, which may be far too many to show to the user. The work of [62] discusses a factorized circuit-based form, yet it is also too complicated to allow its presentation to a user. We view devising an effective way of showing such provenance instances to users – e.g. summarizing the contribution of individual tuples in aggregate queries – as an important goal for future work.

8 Experiments

We have performed an experimental study to assess NLProv through two prisms: (1) the quality of answers produced by the system, and (2) the efficiency of the algorithms in terms of execution time.

8.1 User Study

We have examined the usefulness of the system through a user study, involving 22 non-expert users. The user study was conducted in two phases, first we asked 15 users to evaluate the solution for SPJ queries, where in the second phase 7 different users were requested to evaluate the solution for union queries. For the SPJ evaluation we presented to each user 6 NL queries, namely No. 1–4, 6, and 7 from Table 1 (chosen as a representative sample), where for the union evaluation users were presented with queries 13–15. We have also allowed each user to freely formulate an NL query of her choice, related to the MAS database [1]. 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 119 successfully performed tasks. For each of the NL queries, users

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 paper published by authors from IBM
13	Return the authors who published papers in VLDB or SIGMOD after 2005
14	Return the authors from TAU or HUJI who published papers in VLDB or SIGMOD
15	Return the papers published by authors from TAU or HUJI

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 71 of the 119 cases. Examples of the 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 Figure 19. 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).

8.2 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

Table 2: Sample use-cases and results

Query	Single Assignment	Multiple Assignments - Summarized
Return authors from TAU	Tova M. from TAU	
Return the homepage of SIGMOD	http://www.sigmod2011.org/ is the homepage of SIGMOD	
Return the domain of VLDB	Databases is the domain of VLDB	
Return the domain of conferences	Databases is the domain of VLDB	Databases is the domain of 260 conferences
Return the year of VLDB paper	2007 is the year of VLDB "Graph Partitioning..." paper	2007 is the year of VLDB 152 papers
Return authors who published in papers in a journal	Tova M. published "Putting lipstick on Pig..." in CORR	Tova M. published 60 papers in 17 journals
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
Return the authors who published papers in VLDB or SIGMOD after 2005	Tova M. published "Auto-completion..." in SIGMOD in 2012	Tova M. published 12 papers in VLDB or SIGMOD in 2006-2014

	SPJ Queries				Union Queries			
Category	**3**	**4**	**5**	**Avg.**	**3**	**4**	**5**	**Avg.**
Single								
Relevant	4	10	84	**4.82**	0	5	16	**4.76**
Understandable	7	25	66	**4.60**	0	4	17	**4.81**
Multiple								
Relevant	0	7	43	**4.86**	0	6	15	**4.71**
Understandable	4	13	33	**4.58**	1	7	13	**4.57**
Detailed	3	7	40	**4.74**	0	7	14	**4.67**
Summarized								
Relevant	2	2	46	**4.88**	0	4	17	**4.81**
Understandable	3	3	44	**4.82**	0	3	18	**4.86**
Detailed	2	5	43	**4.82**	0	6	15	**4.71**

Fig. 19: Users ranking

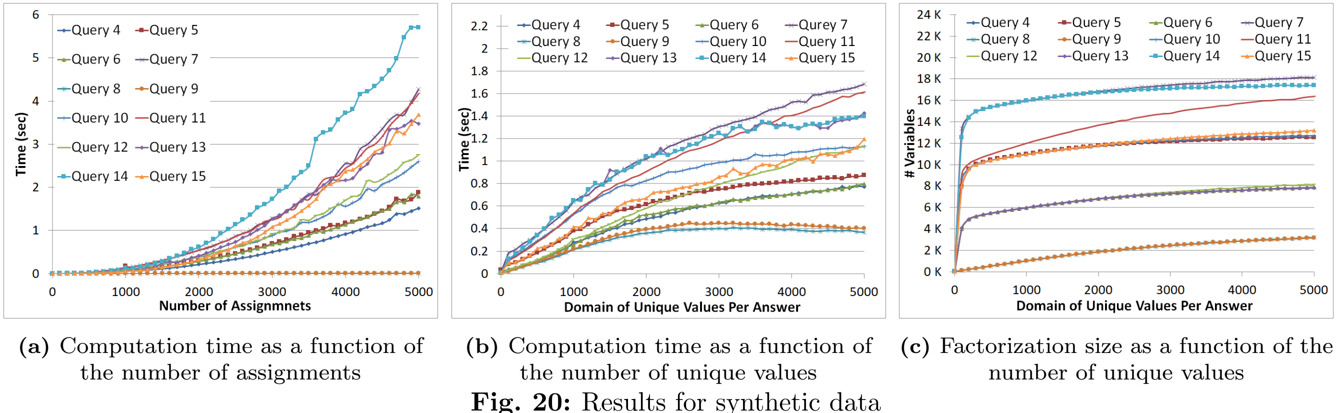
Table 3: Computation time (sec.), for the MAS database

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
13	1.4	0.07	0.33	0.4
14	14.4	0.001	0.004	0.005
15	5.5	0.1	0.41	0.51

whose total size is 4.7 GB, and queries No. 1–15 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. 4–15.

Table 3 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 15% overhead), even for fairly complex ones such as queries 7, 11, and 12.

Figure 20a (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, 3.7, 3.5, 5.7, and 3.7 seconds resp. for queries 4–15. Summarization time was negligible, less than 0.1 seconds in all cases.

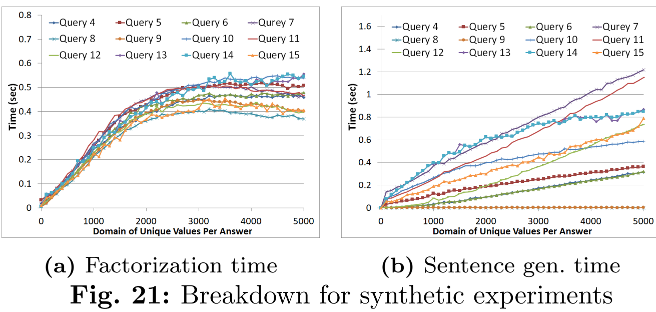


(a) Computation time as a function of the number of assignments

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

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

Fig. 20: Results for synthetic data



(a) Factorization time

(b) Sentence gen. time

Fig. 21: Breakdown for synthetic experiments

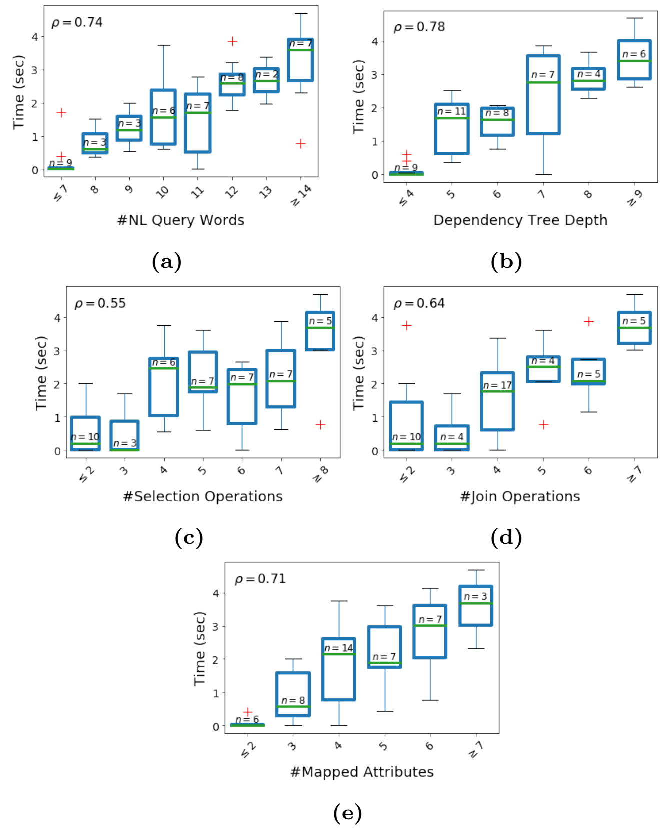


Fig. 22: Computation time as function of **(a)** NL query length **(b)** depth of the NL query dependency tree **(c)** number of query selection operations **(d)** number of query join operations **(e)** number of provenance attributes

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 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 21) shows, the increase in running time occurs both in the factorization and in the sentence generation time. Finally, Figure 20c shows the expected increase in the factorized expression size w.r.t the number of unique values.

For the third scalability experiment we evaluated the computation time for different classes of queries. In this experiment we have used a larger set of queries, consisting of 45 different NL queries (available in [34]) which vary in their size, structure, and complexity. For each query we have fixed both the number of assignments per answer and the domain of unique values at 5K. `NLProv` computation time varied between 0.005 to 4.69 seconds, where the mean and median computation times were 1.73 and 1.8 seconds respectively. Figure 22 depict aggregation of the computation times with respect to different query aspects. Figures 22a and 22b explore the influence of the original NL query sentence structure, recall that both Algorithms 1 and 3 utilize

the query tree in order to generate the answer sentence, hence the computation time increase as function of the query sentence length \ tree depth, with Pearson correlation of 0.74 and 0.78 respectively. The impact of the formal query complexity on NLProv running time is presented in Figures 22c and 22d, notice that the query complexity influence the query evaluation time, but does not have direct impact on the explanation generation, hence a lower correlation was measured (0.55 and 0.64 for number of selection and join operations respectively). Finally, Figure 22e depict the impact of provenance size on the computation time, the number of provenance attributes is crucial for the factorization step, hence it is influential on Algorithm 3 running time as exhibited by the 0.71 correlation.

9 Related Work

In this section we review and compare our work to existing approaches in the context of database theory and database interfaces.

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. [14,41,38,17,39,22,21,37,48]) while the field of provenance applications has also been broadly studied (*e*.g. [26,59,67]). 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 [74,60, 43,31,22,20,4], allowing user control over the level of granularity ("zooming" in and out [19]), or otherwise presenting different ways of provenance visualization [41]. Other works have studied allowing users to query the provenance (*e*.g. [47,44]) or to a-priori request that only parts of the provenance are tracked (see for example [26,35,36]). Importantly provenance factorization and summarization have been studied (*e*.g., [16,8,62, 66]) 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 [61,18] to achieve a fine-grained understanding of the data lineage, but again do not translate the provenance to NL.

Detailed Answers to Keyword Queries There is an extensive line of work on answering keyword queries which focuses on providing not just the query answer (tuples that contain the queried value), but also comprehensive details about it. Works such as [3,12,42] focus on answering keyword queries over a relational database, by outputting tuples that are related to one or more of the queried keywords. In particular, [50,73] studies the subject of prećis queries over relational database. These queries are logical combinations of keywords. The query along with constraints on the schema is inputted to the system, and the answer returned should include the most relevant tuples to the keyword(s), according to the constraints, as relations that form a logical subset of the original database (i.e., contain not only items directly related to the given query terms but also items implicitly related to them). Still in the field of answering keyword queries, [28,30,29] deal with snippets of a database object, which is an entity that has its identity in the result tuple. In this scenario, the system provides a snippet which is a summary of the relevant information related to these objects. This information is taken from tuples that relate to the queried object's tuple and is prioritized in different manners (e.g., diversity and proportionality). All of these works, similarly to ours, provide the query answer along with further details about it, these details stem from tuples that relate in some defined manner to the answer. While there is a commonality between these works and ours, our work supports complex CQs formulated in NL as opposed to keyword queries. Answers to keyword queries are not always explicitly specified in the query, e.g., we can ask about and author and get the name of her organization, or get tuples that contain this author from different tables. For CQs, users explicitly specify the form of answer they would like, from which relation they would like it, and what conditions it has to satisfy. Additionally, our work defines the related tuples by their membership in the provenance of the result, i.e, the query structure (formulated by the user) is a major factor in determining which tuples will be included in the explanation. Furthermore, our system generates the NL explanation based on the NL query given by the user, and not a textual template.

Summarization of Database Content There have been previous works that proposed a summarized presentation of the query results. In the context of keyword queries, the approach of [30] gives short summaries on information regarding data object by limiting the number of related tuples (according to the schema), showing only the highest ranked. The summary is represented as a tree where the root is a tuple containing the keywords and the neighboring tuples are the related ones. In the context of top aggregate queries, [77] presents an approach that summarizes the results using clustering

of the top ranked results, by formulating the clustering problem as an optimization problem. The framework of this work is interactive and allows users to choose the number of clusters and other parameters. After observing the results, users can update the parameters and get different results. The summarization aims to serve as an overview of all query results through a summarized relation. Another related approach is [46] which devised the smart drill-down operator. the operator allows users to obtain interesting summarizations of the tuples in the relation. The summary is essentially the top-k clusters, according to a goal function, of tuples with dont-care values. Similarly to these approaches, we focus on tuples with the same values or similar values and The SaintEtiQ system [69] summarizes entire database relations using background knowledge with a vocabulary to translate raw tuple values. Summarization is done using a clustering approach. Like our system, there is a notion of getting a more detailed and precise summary containing more information, and a less detailed one which is more compact. Provenance summarization has also been proposed by [5], yet it offers an approximated summary of the provenance based on distance, semantic constraints and size, with a possible loss of information. Our summarization technique compacts all the tuples in the provenance, through functions like SUM and RANGE, as opposed to showing/summarizing a few representative tuples. We base this summarization on the factorization of the provenance done as an initial step. Furthermore, we focus here on UCQs and do not cover aggregate queries. Finally, we do not rely on background knowledge of a vocabulary, or other external constraints to summarize, but rather use the provenance factorization. Additionally, we translate the summarization into NL which is geared towards non-expert users.

NL Interfaces Multiple lines of work (*e.g.* [52, 7, 55, 76, 75, 3, 65]) have proposed NL interfaces for the formulation of database queries, and additional works [32] have focused on presenting the *answers* in NL, typically basing their translation on the schema of the output relation. Among these, works such as [7, 55] 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 [51] has focused on the complementary problem of translating SQL *queries* (rather than their answers or provenance) to NL. Another work has devised an interactive chatbot interface to drill down and zoom-in on a specific part of the database which the user is interested in [70]. This work helps guide the user with NL but does not show the query answers and their explanations in natural language. In the context of answering keyword queries, [71] shows an approach that presents the results of prećis queries as a narrative text so that the output is more user friendly. To do so, there is a need for predefined textual templates to embed the relevant tuples in. The templates are predefined by a designer or the administrator of the database. Synthesizing text directly from databases has also been explored in [72] which extended [71]. This work revolves around the generation of textual representation for database subsets. The text is generated based on templates rather than on user formulated queries in NL. Moreover, the explanation is composed of tuples from related database tables, as opposed to tuples from the provenance which provide a targeted explanation tailored to the specific details provided by a user in an NL query. 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.

10 Conclusion

We have studied in this paper, for the first time to our knowledge, provenance for NL queries. We have devised a novel model of “word-to-provenance” mapping, thereby leveraging the structure of the original NL question for the generation of a new NL sentence that captures the answers along with their provenance-based explanations. Since there may be many explanations, even for a single answer, we have developed factorization and summarization techniques that are geared towards sentence generation, showing that they result in new criteria for preferring one factorized/summarized form over another. We have implemented the approach and demonstrated its effectiveness through use cases and experiments.

Our work presented a simple yet effective approach of generating NL explanations based on the user NL query. We have demonstrated that by applying basic transformations on the original question we are able to get understandable and relevant NL explanations. Usage of more advanced *Natural Language Generation* techniques can farther improve the explanations quality; this is an interesting direction for future work.

Our implementation is based on a particular NL interface to Databases and on a particular provenance model for UCQs, but we have also discussed at some depth the extension of our solution beyond these settings. This discussion provides indication of the generic nature of the approach, but further research is required

to fully realize its potential in these other settings. In particular, we believe that the need to handle more complex queries with nesting, aggregation etc. may lead to new and exciting research avenues.

Acknowledgements This research has been funded by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (Grant agreement No. 804302), the Israeli Science Foundation (ISF) Grant No. 978/17, and the Google Ph.D. Fellowship. The contribution of Amir Gilad is part of a Ph.D. thesis research conducted at Tel Aviv University.

References

1. Mas. http://academic.research.microsoft.com.
2. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.
3. S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.
4. A. Ailamaki, Y. E. Ioannidis, and M. Livny. Scientific workflow management by database management. In *SSDBM*, pages 190–199, 1998.
5. E. Ainy, P. Bourhis, S. B. Davidson, D. Deutch, and T. Milo. Approximated summarization of data provenance. In *CIKM*, pages 483–492, 2015.
6. Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen. Putting lipstick on pig: Enabling database-style workflow provenance. *Proc. VLDB Endow.*, 2011.
7. Y. Amsterdamer, A. Kukliansky, and T. Milo. A natural language interface for querying general and individual knowledge. *VLDB*, pages 1430–1441, 2015.
8. N. Bakibayev, D. Olteanu, and J. Zavodny. FDB: A query engine for factorised relational databases. *PVLDB*, pages 1232–1243, 2012.
9. I. Beltagy, K. Erk, and R. Mooney. Semantic parsing using distributional semantics and probabilistic logic. In *Proceedings of the ACL 2014 Workshop on Semantic Parsing*, pages 7–11, 2014.
10. O. Benjelloun, A. Sarma, A. Halevy, M. Theobald, and J. Widom. Databases with uncertainty and lineage. *VLDB J.*, 2008.
11. J. Berant and P. Liang. Semantic parsing via paraphrasing. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 1415–1425, 2014.
12. G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440, 2002.
13. P. Brgisser, M. Clausen, and M. A. Shokrollahi. *Algebraic Complexity Theory*. Springer Publishing Company, Incorporated, 2010.
14. P. Buneman, S. Khanna, and W. chiew Tan. Why and where: A characterization of data provenance. In *ICDT*, pages 316–330, 2001.
15. A. Chapman and H. V. Jagadish. Why not? In *SIGMOD*, pages 523–534, 2009.
16. A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *SIGMOD*, pages 993–1006, 2008.
17. J. Cheney, L. Chiticariu, and W. C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, pages 379–474, 2009.
18. 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.
19. S. Cohen-Boulakia, O. Biton, S. Cohen, and S. Davidson. Addressing the provenance challenge using zoom. *Concurr. Comput. : Pract. Exper.*, pages 497–506, 2008.
20. 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.
21. 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.
22. S. B. Davidson and J. Freire. Provenance and scientific workflows: challenges and opportunities. In *SIGMOD*, pages 1345–1350, 2008.
23. D. Deutch, N. Frost, and A. Gilad. Nlprov: Natural language provenance. *Proc. VLDB Endow.*, pages 1900–1903, 2016.
24. D. Deutch, N. Frost, and A. Gilad. Provenance for natural language queries. *PVLDB*, 10(5):577–588, 2017.
25. D. Deutch, N. Frost, A. Gilad, and T. Haimovich. Nlprovenans: natural language provenance for non-answers. *Proceedings of the VLDB Endowment*, 11(12):1986–1989, 2018.
26. D. Deutch, A. Gilad, and Y. Moskovitch. Selective provenance for datalog programs using top-k queries. *PVLDB*, pages 1394–1405, 2015.
27. K. Elbassioni, K. Makino, and I. Rauf. On the readability of monotone boolean formulae. *JoCO*, pages 293–304, 2011.
28. G. J. Fakas. Automated generation of object summaries from relational databases: A novel keyword searching paradigm. In *ICDE*, pages 564–567, 2008.
29. G. J. Fakas, Z. Cai, and N. Mamoulis. Versatile size-l object summariesfor relational keyword search. *IEEE Trans. on Knowl. and Data Eng.*, 26(4):1026–1038, 2014.
30. G. J. Fakas, Z. Cai, and N. Mamoulis. Diverse and proportional size-l object summaries using pairwise relevance. *VLDB J.*, 25(6):791–816, 2016.
31. 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.
32. 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.
33. A. Giordani and A. Moschitti. Translating questions to sql queries with generative parsers discriminatively reranked. *Proceedings of COLING 2012: Posters*, pages 401–410, 2012.
34. https://github.com/navefr/NL_Provenance/.
35. B. Glavic. Big data provenance: Challenges and implications for benchmarking. In *Specifying Big Data Benchmarks - First Workshop, WBDB*, pages 72–80, 2012.
36. B. Glavic and G. Alonso. Perm: Processing provenance and data on the same data model through query rewriting. In *ICDE*, pages 174–185, 2009.
37. 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.

38. T. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, pages 31–40, 2007.
39. T. J. Green. Containment of conjunctive queries on annotated relations. In *ICDT*, pages 296–309, 2009.
40. E. Hemaspaandra and H. Schnoor. Minimization for generalized boolean formulas. In *IJCAI*, pages 566–571, 2011.
41. M. Herschel and M. Hlawatsch. Provenance: On and behind the screens. In *SIGMOD*, pages 2213–2217, 2016.
42. V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, pages 670–681, 2002.
43. D. Hull et al. Taverna: a tool for building and running workflows of services. *Nucleic Acids Res.*, pages 729–732, 2006.
44. Z. G. Ives, A. Haeberlen, T. Feng, and W. Gatterbauer. Querying provenance for ranking and recommending. In *TaPP*, pages 9–9, 2012.
45. S. Iyer, I. Konstas, A. Cheung, J. Krishnamurthy, and L. Zettlemoyer. Learning a neural semantic parser from user feedback. *arXiv preprint arXiv:1704.08760*, 2017.
46. M. Joglekar, H. Garcia-Molina, and A. G. Parameswaran. Smart drill-down: A new data exploration operator. *PVLDB*, 8(12):1928–1931, 2015.
47. G. Karvounarakis, Z. G. Ives, and V. Tannen. Querying data provenance. In *SIGMOD*, pages 951–962, 2010.
48. B. Kenig, A. Gal, and O. Strichman. A new class of lineage expressions over probabilistic databases computable in p-time. In *SUM*, pages 219–232, 2013.
49. D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *Annual Meeting on Association for Computational Linguistics*, pages 423–430, 2003.
50. G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Précis: The essence of a query answer. In *ICDE*, pages 69–78, 2006.
51. G. Koutrika, A. Simitsis, and Y. E. Ioannidis. Explaining structured queries in natural language. In *ICDE*, pages 333–344, 2010.
52. D. Küpper, M. Storbel, and D. Rösner. Nauda: A cooperative natural language interface to relational databases. SIGMOD, pages 529–533, 1993.
53. N. Kwasnikowska and J. V. den Bussche. Mapping the NRC dataflow model to the open provenance model. In *IPAW*, pages 3–16, 2008.
54. S. Lee, S. Köhler, B. Ludäscher, and B. Glavic. A sql-middleware unifying why and why-not provenance for first-order queries. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 485–496, 2017.
55. F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *Proc. VLDB Endow.*, pages 73–84, 2014.
56. P. Liang, M. I. Jordan, and D. Klein. Learning dependency-based compositional semantics. *Computational Linguistics*, 39(2):389–446, 2013.
57. 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.
58. M. Marneffe, B. Maccartney, and C. Manning. Generating typed dependency parses from phrase structure parses. In *LREC*, pages 449–454, 2006.
59. A. Meliou, Y. Song, and D. Suciu. Tiresias: a demonstration of how-to queries. In *SIGMOD*, pages 709–712, 2012.
60. 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.
61. T. Müller and T. Grust. Provenance for SQL through abstract interpretation: Value-less, but worthwhile. *PVLDB*, pages 1872–1875, 2015.
62. D. Olteanu and J. Závodný. Factorised representations of query results: Size bounds and readability. In *ICDT*, pages 285–298, 2012.
63. H. Poon. Grounded unsupervised semantic parsing. In *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, volume 1, pages 933–943, 2013.
64. A.-M. Popescu, A. Armanasu, O. Etzioni, D. Ko, and A. Yates. Modern natural language interfaces to databases: Composing statistical parsing with semantic tractability. In *Proceedings of the 20th international conference on Computational Linguistics*, page 141, 2004.
65. A.-M. Popescu, O. Etzioni, and H. Kautz. Towards a theory of natural language interfaces to databases. In *IUI*, pages 149–157, 2003.
66. C. Ré and D. Suciu. Approximate lineage for probabilistic databases. *Proc. VLDB Endow.*, pages 797–808, 2008.
67. S. Roy and D. Suciu. A formal approach to finding explanations for database queries. In *SIGMOD*, pages 1579–1590, 2014.
68. D. Saha, A. Floratou, K. Sankaranarayanan, U. F. Minhas, A. R. Mittal, and F. Özcan. ATHENA: an ontology-driven system for natural language querying over relational data stores. *PVLDB*, 9(12):1209–1220, 2016.
69. R. Saint-Paul, G. Raschia, and N. Mouaddib. Database summarization: The saintetiq system. In *ICDE*, pages 1475–1476, 2007.
70. T. Sellam and M. L. Kersten. Have a chat with clustine, conversational engine to query large tables. In *HILDA*, page 2, 2016.
71. A. Simitsis and G. Koutrika. Comprehensible answers to précis queries. In *CAiSE*, pages 142–156, 2006.
72. A. Simitsis, G. Koutrika, Y. Alexandrakis, and Y. E. Ioannidis. Synthesizing structured text from logical database subsets. In *EDBT*, pages 428–439, 2008.
73. A. Simitsis, G. Koutrika, and Y. E. Ioannidis. Précis: from unstructured keywords as queries to structured databases as answers. *VLDB J.*, 17(1):117–149, 2008.
74. Y. L. Simmhan, B. Plale, and D. Gannon. Karma2: Provenance management for data-driven workflows. *Int. J. Web Service Res.*, pages 1–22, 2008.
75. D. Song, F. Schilder, and C. Smiley. Natural language question answering and analytics for diverse and interlinked datasets. In *NAACL*, pages 101–105, 2015.
76. 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.
77. Y. Wen, X. Zhu, S. Roy, and J. Yang. Interactive summarization and exploration of top aggregate query answers. *PVLDB*, 11(13):2196–2208, 2018.
78. S. W.-t. Yih, M.-W. Chang, X. He, and J. Gao. Semantic parsing via staged query graph generation: Question answering with knowledge base. 2015.
79. L. S. Zettlemoyer and M. Collins. Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. *arXiv preprint arXiv:1207.1420*, 2012.
80. V. Zhong, C. Xiong, and R. Socher. Seq2sql: Generating structured queries from natural language using reinforcement learning. *arXiv preprint arXiv:1709.00103*, 2017.