Reverse-Engineering Conjunctive Queries from Provenance Examples

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

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

ABSTRACT

The *provenance* of a query result details relevant parts of the input data as well as the computation leading to each output tuple. Multiple lines of work have studied the tracking and presentation of provenance, showing its effectiveness in explaining and justifying query results. The willingness of application owners to share provenance information for these purposes may however be hindered by the resulting exposure of the underlying query logic, which may be proprietary and confidential. We therefore formalize and study the following problem: when a (small) subset of the query results along with their provenance is given, what information is revealed about the underlying query? Our model is based on the provenance semiring framework and applies to many previously proposed provenance models. We analyze two flavors of the problem: (1) how many queries may be consistent with a given provenance example? and (2) what is the complexity of inferring a consistent query, or one that is a "best fit"? Our theoretical analysis shows that there may be many (for some models, even infinitely many in presence of self-joins) consistent queries, yet we provide practically efficient algorithms to find (best-fit) such queries. We experimentally show that the algorithms are generally successful in correctly reverse engineering queries, even when given only a few output examples and their provenance.

1 INTRODUCTION

Provenance information is a form of meta-data that is associated with query results, to describe the origin of each piece of data and the computational process leading to its generation. In particular, provenance was shown to be useful as means of providing explanations for query results and allowing for their validation [6, 8, 12, 15, 16, 21, 23, 29, 30, 36]. On the other hand, query owners who publish this information for their users may wish to keep the original query private due to its proprietary logic or due to criteria they wish to keep obfuscated. Thus, owners may wonder if their query can be reverse engineered from a few leaked outputs and their provenance w.r.t this query.

Example 1.1. Consider a bank whose clients must meet certain criteria to have their loan requests approved: they must have a positive credit score and one guarantor associated with the private banking division to co-sign the loan. If a client is already associated with the private banking division, she does not need a guarantor.

The bank's database contains the tables shown in Figures 1a, 1b, 1c, and 1d. The tables correspond to clients and their balance (table B), pairs of possible guarantors and borrowers (table G), clients with positive credit score (table PCS) and clients associated with the private banking division (table PB), respectively (ignore the *prov.* column for now).

The following Conjunctive Query (CQ), Q_{real}, captures the bank's loan conditions as specified above:

$$Q_{real}(id_1, b_2) :- B(id_1, b_1), B(id_2, b_2), G(id_2, id_1), PCS(id_1), PB(id_2)$$

It takes the balance of two clients, specifies that the second client is the guarantor of the first, and makes sure that the borrowing client has a positive credit score and that the guarantor is associated with the private banking division. The output is a table of ids and balances where for each borrower id, we show how much security the bank has for the loan, represented as the balance of the guarantor.

The bank may wish to explain to each applicant the reason underlying the answer to its own request, but to avoid exposure of the general criteria, i.e. of the query Q_{real}, since these criteria are part of the bank's confidential business strategy. If other clients obtain some of these explanations, they may understand the general criteria. The question then arises: given examples of the output and explanations provided for multiple clients, can one infer the underlying query?

In the above example, the answers and explanations can be formally defined as output and provenance examples. This leads us to define and study for first time on the modeling and algorithmic challenges stemming from the problem of reverse engineering queries from output and provenance examples.

Modeling. We propose a novel formal model for the problem, as follows. We are given output tuples, and their provenance, represented as instances of the previously proposed semiring model [30]. We have chosen the semiring model due to its generality: as shown in [29], many previously proposed provenance models may be captured via corresponding choices of semirings. Intuitively, different semirings represent different granularities of explanations given to users (see Example 1.2 in this section). We then formally define what it means for a CQ to be consistent with output examples and their provenance, thus defining the *query-by-provenance* problem. Intuitively, we look for a CQ that, when evaluated with respect to the input database *does not only yield the specified example tuples, but also its derivation of these tuples (i.e. their provenance) is "consistent" with the prescription made by the provenance.* We do not expect the full provenance to be provided (i.e. all explanations for a loan being approved); instead, we define "consistency" of provenance in a less restrictive manner, by leveraging the inclusion property of provenance from [29]. Last, we define the notion of inclusion-minimality to capture the idea of the query not only matching the provenance, but rather being a "best-fit" for it.

Theoretical Analysis. We then study in depth the query-by-provenance problem, for five different choices of semirings, namely, provenance polynomials [29, 30], trio provenance [36], positive

prov.	ID_1	Amount
a	1	1000
b	2	1000
c	3	300

(a) Balance B

prov.	ID_1	ID_2
d	1	2
e	2	2
f	3	1

(b) Guarantor G

prov.	ID_1
h	1
i	2

(c) Positive Credit Score PCS

prov.	ID
j	1
k	2
l	3

(d) Private Banking PB

A	B	$\mathbb{N}[X]$	$\mathbb{B}[X]$	$Trio(X)$	$Why(X)$	$PosBool(X)$
2	1000	$2b^2eik + badij$	$b^2eik + badij$	$2beik + badij$	$beik + badij$	$beik + badij$
1	300	$acfhl$	$acfhl$	$acfhl$	$acfhl$	$acfhl$

(e) Example Data and Provenance Via Different Semirings

Figure 1: Database Tables and Provenance Example

boolean expressions [32], and why-provenance [12]. Each semiring corresponds to an explanation with a different granularity. We next demonstrate two such provenance semirings, namely provenance polynomials ($\mathbb{N}[X]$) and why-provenance (Why(X)), in the context of our running example.

Example 1.2. Reconsider the query Q_{real} and the database in Example 1.1. Tuples in the database are associated with annotations (a to l), intuitively serving as the tuple identifiers (note that the annotations are not part of the relations). Figure 1e presents two output tuples of the query when evaluated on the database. Consider for instance the first row describing the provenance of the tuple $(2, 1000)$ (the id of the borrower with the balance of the guarantor). The $\mathbb{N}[X]$ column shows the polynomial $2b^2eik + badij$, which consists of three monomials (a monomial with coefficient x is considered as x monomials), each corresponding to a different assignment of input tuples to the atoms of Q_{real}, or different explanations for approving the loan request of the client with id 2. For instance, $2b^2eik$ stems from two assignments (implied by the coefficient 2) that use the tuple annotated by b twice and the tuples annotated by e, i, k once. This explanation immediately exposes the number of query atoms (5, as is the number of annotations in the monomial) and the number atoms from each relation. In the context of loan conditions, this explanation exposes that a borrower can be her own guarantor (from the double use of the tuple b). In contrast, in the $Why(X)$ model, we represent each explanation as the **set** of tuples that contributed to the formation of the output tuple, thus "dropping" both coefficients and exponents: $beik + badij$. Considering this kind of explanation, we no longer know the number of query atoms and the number of atoms from each relation. In particular, we no longer expose that a client associated with the private banking division does not need a guarantor.

We study two factors that determine to what extent does a set of output examples and their provenance reveal a hidden underlying query, as follows. The first factor is the number of possible consistent queries, and the second is the complexity of reverse-engineering a consistent query. In more detail, when the provenance is given in $\mathbb{N}[X]$ or $\mathbb{B}[X]$, we provide a bound on the number of consistent queries which is exponential in the sum of arities of all relations whose tuples participate in a single provenance monomial, and provide an example where there are indeed exponentially many consistent queries. We then show that finding a consistent query w.r.t $\mathbb{N}[X]\backslash\mathbb{B}[X]$-examples is NP-complete in the number of attributes of the output tuple, and we design a practically efficient algorithm.

Reconstructing queries from examples of why, trio or PosBool provenance is more cumbersome: there may be infinitely many consistent and inclusion-minimal queries that lead to the same provenance (with different exponents, due to self-joins, that are abstracted away in these models). To this end, we prove a small world property, namely that if a consistent query exists, then there exists such query of size bounded by some parameters of the input. The bound by itself does not suffice for an efficient algorithm (trying all detailed expressions of sizes up to the bound would be inefficient), but we leverage it in devising an efficient algorithm for these provenance semirings. The algorithm is similar to the $\mathbb{N}[X]$ and $\mathbb{B}[X]$ case but also includes an option to expand the monomials, gradually, generating self joins when necessary.

Experimental Analysis. We have also conducted an experimental study of our algorithms, assessing their ability to reverse engineer the correct TPC-H [40] queries as well as highly complex join queries presented as a baseline in [43]. We have executed the queries while tracking provenance, and then showed our algorithms randomly sampled portions of the output and provenance. We report how many examples were required for the algorithms to correctly identify the underlying query, and what were the differences between the actual and inferred query when incorrect. In the vast majority of the cases, our algorithms converged to the underlying query after having viewed only a small number of examples; when this was not the case, the inferred query was typically similar to the actual one, e.g. containing additional constants due to the same value recurring in the viewed examples. Last, further experiments indicate the computational efficiency of our algorithms. Our experiments show that, although theoretically provenance examples may correspond to a large or even infinite number of queries, in practice, publishing provenance information, even for a small number of output tuples, reveals the full logic of the query. Hence, in this respect, there is no advantage in publishing a less detailed form of provenance.

2 RELATED WORK

Reverse Engineering Queries from Partial/Full Output. There is a large body of literature on learning queries from examples, in different variants. A first axis of these variants concerns learning a query whose output *precisely* matches the example (e.g. [33, 41, 43]), versus one whose output contains the example tuples and possibly more (e.g. [33–35, 38] and the somewhat different problem in [44]). The first is mostly useful e.g. in a use-case where an actual query was run and its result, but not the query itself, is available. This may be the case if e.g. the result was exported and sent. The second, that we adopt here, is geared towards examples provided manually by a user, who may not be expected to provide a full account of the output. Another distinguishing factor between works in this area is the domain of queries that are inferred; due to the complexity of the problem, it is typical (e.g. [10, 35, 43]) to restrict the attention to join queries, and many works also impose further restrictions on the join graph [17, 41]. We do not impose such restrictions and are able to infer complex CQs. Last, there is a prominent line of work on query-by-example in the context of *data exploration* [3, 9, 10, 24, 37]. Here users typically provide an initial set of examples, leading to the generation of a consistent query (or multiple such queries); the queries and/or their results are presented to users, who may in turn provide feedback used to refine the queries, and so on. In our settings, the number of examples required for convergence to the actual intended query was typically small. In cases where more examples are needed, an interactive approach is expected to be useful in our setting as well. Works along the lines of [2, 7, 42] explored the problem of reverse engineering queries from positive and negative examples and the general complexity of different variations of the problem.

The fundamental difference between our work and previous work in this field in this area is the assumed input - output examples and provenance information (in particular no foreign keys are known; in fact, we do not even need to know the entire input database, but rather just tuples used in explanations). *While our approach requires more input, it greatly reduces the search space for the query. The advantage becomes more significant as the database size increases and the schema becomes more complex.* Furthermore, we are able to leverage the provenance information and reconstruct the original query, or a very similar one (1) in a highly complex setting where the underlying queries includes multiple joins and self-joins, (2) with only very few examples (up to 5 were typically sufficient to obtain over 90% recall, and less than 20 in all but one case were sufficient to converge to the actual underlying query), and (3) in split-seconds for a small number of examples, and in 1.3 seconds even with 500 examples. No previous work, to our knowledge, has exhibited the combination of these characteristics.

Data Provenance. Data Provenance has been extensively studied, for different formalisms including relational algebra, XML query languages, Nested Relational Calculus, and functional programs (see e.g. [11, 15, 25, 26, 28, 30, 36, 39]). In contrast to these lines of work that focus on provenance tracking and usages, we have focused on learning queries from output examples and their partial provenance, based on the semiring framework. This includes quite a few of the models proposed in the literature, but by no means all of them. Investigating query-by-provenance for other provenance models is an intriguing direction for future work.

The high-level question of what can be learned from provenance has been extensively studied in the context of workflow privacy, e.g. [13, 18–20, 27]. In contrast, we do not focus on black-box modules, but rather on detailed fine-grained provenance obtained from queries. This makes the technical results of these works inapplicable to our setting. [14] describes a general framework for provenance security, but, for the relational database case, focuses on what parts of the *data* are disclosed, while the underlying query is assumed to be known.

Last, we note that in [4] we have leveraged provenance information for designing a user-interactive SPARQL interface, including a component of inferring SPARQL queries from examples and provenance. There are many differences in the model, and consequently none of our results here follow from [4]. In particular, our focus in [4] was on single output nodes which if translated to the relational settings means $k = 1$ (recall that k is the output arity); in that setting it is consequently PTIME to find a consistent query (compare to our NP-hardness result). SPARQL is also bounded to 2 attributes per relation, implying that $r = 2n$, as opposed to the relational setting where the number of attributes is not necessarily 2. Indeed, a prominent challenge in our experimental study stemmed from the number of variables, in particular for the TPC-H queries, where $n << r$.

3 PRELIMINARIES

3.1 Conjunctive Queries

We will focus in this paper on CQs (see e.g. [31]). Fix a database schema $\mathcal{S}$ with relation names $\{\mathcal{R}_1, ..., \mathcal{R}_n\}$ over a domain C of constants. Further fix a domain $\mathcal{V}$ of variables. A *CQ* Q over $\mathcal{S}$ is an expression of the form $T(\vec{u}) : -\mathcal{R}_1(\vec{v_1}), \ldots, \mathcal{R}_l(\vec{v_l})$ where T is a relation name not in $\mathcal{S}$. For all $1 \leq i \leq n$, $\vec{v_i}$ is a vector of the form $(x_1, \ldots, x_k)$ where $\forall 1 \leq j \leq k.\ x_j \in \mathcal{V} \cup C$. $T(\vec{u})$ is the query head, denoted *head*(Q), and $\mathcal{R}_1(\vec{v_1}), \ldots, \mathcal{R}_l(\vec{v_l})$ is the query body and is denoted *body*(Q). The variables appearing in $\vec{u}$ are called the *head variables* of Q, and each of them must also appear in the body. We use *CQ* to denote the class of all CQs, omitting details of the schema when clear from context.

We next define the notion of *derivations* for CQs. A derivation 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 arguments, 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 $\alpha(x_i) = a_i$). We require that a variable x_i will not be mapped to multiple distinct values, and a constant x_i will be mapped to itself. We define $\alpha(head(Q))$ as the tuple obtained from $head(Q)$ by replacing each occurrence of a variable x_i by $\alpha(x_i)$.

Example 3.1. Reconsider our example query Q_{real} presented in Example 1.2 (the database is depicted in Figures 1a, 1b, 1c, and 1d). Now, consider the result tuple (1, 300). It is obtained through the derivation that maps the atoms to six distinct tuples from the database to the three atoms (in order of the atoms). These are the tuples annotated by **a, c, f, h, l**. The tuple (2, 1000) is obtained through three derivations: the first (second and third) maps the tuple annotated **b** (**b**) to the first atom, the tuple annotated by **a** (**b**) to the second atom, the tuple annotated by **d** (**e**) to the third atom, the tuple annotated by **i** (**i**) to the fourth, and **j** (**k**) to the remaining atom.

3.2 Provenance

The tracking of *provenance* to explain query results has been extensively studied in multiple lines of work, and [29] has shown that different such models may be captured using the *semiring approach* (originally proposed in [30]). We next overview several aspects of the approach that we will use in our formal framework.

Commutative Semirings. A *commutative monoid* is an algebraic structure $(M, +_M, 0_M)$ where $+_M$ is an associative and commutative binary operation and 0_M is an identity for $+_M$. A *commutative semiring* is then a structure $(K, +_K, \cdot_K, 0_K, 1_K)$ where $(K, +_K, 0_K)$ and $(K, \cdot_K, 1_K)$ are commutative monoids, $\cdot_K$ is distributive over $+_K$, and $a \cdot_K 0_K = 0 \cdot_K a = 0_K$.

Annotated Databases. We will capture provenance through the notion of databases whose tuples are associated ("annotated") with elements of a commutative semiring. For a schema $\mathcal{S}$ with relation names $\{\mathcal{R}_1, ..., \mathcal{R}_n\}$, denote by $Tup(\mathcal{R}_i)$ the set of all (possibly infinitely many) possible tuples of $\mathcal{R}_i$.

Definition 3.2 (adapted from [30]). A *K-relation* for a relation name $\mathcal{R}_i$ and a commutative semiring K is a function R :

$Tup(\mathcal{R}_i) \mapsto K$ such that its *support* defined by $\text{supp}(R) \equiv \{t \mid R(t) \neq 0\}$ is finite. We say that $R(t)$ is the annotation of t in R. A K-database D over a schema $\{\mathcal{R}_1, ..., \mathcal{R}_n\}$ is then a collection of K-relations, over each $\mathcal{R}_i$.

Intuitively a K-relation maps each tuple to its annotation. We will sometimes use $D(t)$ to denote the annotation of t in its relation in a database D. We furthermore say that a K-relation is *abstractly tagged* if each tuple is annotated by a distinct element of K (intuitively, its identifier).

Provenance-Aware Query Results. We then define CQs as mappings from K-databases to K-relations. Intuitively we define the annotation (provenance) of an output tuple as a combination of annotations of input tuples. This combination is based on the query derivations, via the intuitive association of alternative derivations with the semiring "+" operation, and of joint use of tuples in a derivation with the "·" operation.

Definition 3.3 (adapted from [30]). Let D be a K-database and let $Q \in CQ$, with T being the relation name in $head(Q)$. For every tuple $t \in T$, let α_t be the set of derivations of Q w.r.t. D that yield t. $Q(D)$ is defined to be a K-relation T s.t. for every t, $T(t) = \sum_{\alpha \in \alpha_t} \prod_{t' \in Im(\alpha)} D(t')$. $Im(\alpha)$ is the image of α.

Summation and multiplication in the above definition are done in an arbitrary semiring K. Different semirings give different interpretations to the operations [29].

Provenance Polynomials ($\mathbb{N}[X]$). The most general form of provenance for positive relational algebra (see [30]) is the *semiring of polynomials with natural numbers as coefficients*, namely $(\mathbb{N}[X], +, \cdot, 0, 1)$. The idea is that given a set of basic annotations X (elements of which may be assigned to input tuples), the output of a query is represented by a sum of products as in Def. 3.3, with only the basic equivalence laws of commutative semirings in place. Coefficients serve in a sense as "shorthand" for multiple derivations using the same tuples, and exponents as "shorthand" for multiple uses of a tuple in a derivation. In Example 3.1, we see that a tuple may have several derivations, serving as the explanations for it. The two monomials are separated by a + sign, as opposed to a single monomial. We address this as part of our solution presented in Section 5.

Many additional forms of provenance have been proposed in the literature, varying in their level of abstraction and the details they reveal on the derivations. [29] showed that these can be captured by congruence relations. Formally, consider the function $f_K : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ as a congruence relation defined by $P_1 \equiv P_2$ if $f_K(P_1) = f_K(P_2)$, where f_K varies based on the semiring K. We exemplify these notions using our running example of the third row in Figure 1e.

Boolean Provenance Polynomials ($\mathbb{B}[X]$). The boolean provenance semiring, $(\mathbb{B}[X], +, \cdot, 0, 1)$, is the semiring of polynomials over variables X, where coefficients are either 1 or 0 (intuitively corresponding to boolean coefficients). We can think of the $\mathbb{B}[X]$ semiring as the $\mathbb{N}[X]$ semiring after applying the homomorphism $f_{\mathbb{B}}[X] : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ which maps all non-zero coefficients to 1. In the third row of our example, the polynomial $2b^2eik + badij$ becomes $b^2eik + badij$ after mapping the two coefficients 2, 1 to 1.

In the context of our loan example, if we were to obtain the provenance as $\mathbb{N}[X]$ or $\mathbb{B}[X]$, we could infer that a client can get a loan without needing a guarantor.

Trio. This semiring can again be modeled as the image of the surjective homomorphism $f_{trio} : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ dropping all exponents in the provenance polynomial. Consider again the third row of our example. For instance, the polynomial $2b^2eik + badij$ becomes $2beik + badij$ after the function f_{trio} maps all exponents to 1.

Why. A natural approach to provenance tracking, referred to as *why*-provenance [12], capturing each derivation as a *set* of the annotations of tuples used in the derivation. The overall why-provenance is thus a *set of such sets*. As shown (in a slightly different way) in [29], this corresponds to using provenance polynomials but without "caring" about exponents and coefficients. Formally, consider the function $f_{why} : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ that *drops all coefficients and exponents* of its input polynomial. Using our example again, the polynomial $2b^2eik + badij$ is converted to $beik + badij$ under f_{why}.

Positive Boolean Expressions. This is a semiring of positive boolean expressions over variables X, i.e., the expressions are composed of disjunction, conjunction, and constants which are true or false. Formally, we define $f_{PosBool} : \mathbb{N}[X] \mapsto \mathbb{N}[X]$ that dropping all exponents and coefficients in the provenance polynomial, and considers + as $\vee$ and · as $\wedge$, i.e., allowing for the absorption of monomials. For the purpose of the demonstration, consider the abstract polynomial $2b^2eik + baeik$. Under $f_{PosBool}$ it is converted to $beik$ since all exponents and coefficients, and the monomial $baeik$ are absorbed into $beik$ $((b \wedge b \wedge e \wedge i \wedge k) \vee (b \wedge a \wedge e \wedge i \wedge k) \equiv b \wedge e \wedge i \wedge k)$.

In our running example, trio, why and PosBool provenance reveals less information about the loan conditions, e.g., who is the guarantor of the client with id 2 in Figure 1e.

As we demonstrated, each provenance model represents the provenance in a specific level of detail, and all models other than $\mathbb{N}[X]$ incur some loss of information in the provenance description.

4 QUERY-BY-PROVENANCE

We define in this section the problem of learning queries from examples and their provenance. We first introduce the notion of such examples, using provenance.

Definition 4.1 (Examples with provenance). Given a semiring K, a K-*example* is a pair (I, O) where I is an abstractly-tagged K-database called the *input* and O is a K-relation called the *output*.

Intuitively, annotations in the input only serve as identifiers, and those in the output serve as explanations – combinations of annotations of input tuples contributing to the output.

We next define the notion of a query being consistent with a K-example. In the context of query-by-example, a query is consistent if its evaluation result includes all example tuples (but maybe others as well). We resort to [29] for the appropriate generalization to the provenance-aware settings:

Definition 4.2. Let $(K, +_K, \cdot_K, 0, 1)$ be a semiring and define $a \leq_K b$ iff $\exists c.\ a +_K c = b$. If $\leq_K$ is a (partial) order relation then we say that K is naturally ordered.

Given two K-relations R_1, R_2 we say that $R_1 \subseteq_K R_2$ iff $\forall t. R_1(t) \leq_K R_2(t)$.

Note that if $R_1 \subseteq_K R_2$ then in particular $supp(R_1) \subseteq supp(R_2)$, so the notion of containment w.r.t. a semiring is indeed a faithful extension of "standard" relation containment. In terms of provenance, we note that for $\mathbb{N}[X]$ and $Why(X)$, the natural order corresponds to inclusion of monomials: $p_1 \leq p_2$ if every monomial in p_1 appears in p_2. The order relation has different interpretations in other semirings.

We are now ready to define the notion of consistency with respect to a K-example, and introduce our problem statement. Intuitively, we look for a query whose output is contained in the example output, and for each example tuple, the provenance is "reflected" in the computation of the tuple by the query.

Definition 4.3 (Problem Statement). Given a K-example (I,O) and a CQ Q we say that Q is consistent with respect to the example if $O \subseteq_K Q(I)$. QUERY-BY-PROVENANCE is the problem of finding a consistent query for a given K-example.

The above definition allows multiple CQs to be consistent with a given K-example. This is in line with the conventional wisdom in query-by-example. Throughout the paper we will always assume that the provenance is non-empty, since if it is empty, we return to the setting of the classic query-by-example problem.

A consistent query can be very general, as we demonstrate in the following example. A natural desideratum (employed in the context of "query-by-example"), is that the query is "inclusion-minimal". This notion extends naturally to K-databases.

Definition 4.4. A consistent query Q (w.r.t. a given K-example Ex) is inclusion-minimal if for every query Q' such that $Q' \subsetneq_K Q$ (i.e. for every K-database D it holds that $Q'(D) \subseteq_K Q(D)$, but not vice-versa), Q' is not consistent w.r.t. Ex.

We next demonstrate the notion of consistent and inclusion-minimal queries with respect to a given K-example.

Example 4.5. We now treat Figure 1e as an $\mathbb{N}[X]$-example. Consistent queries must derive the example tuples in the ways specified in the polynomials (and possibly in additional ways). The query Q_{real} from Example 1.2 is of course a consistent query with respect to it, since it generates the example tuples and the provenance of each of them according to Q_{real} is the same as that provided in the example. Q_{real} is not the only consistent query, since the following query, $Q_{general}$ (which simply performs a Cartesian product), is also consistent:

$$Q_{general}(id_1, b_2) : -B(id_1, b_1), B(id_2, b_2), G(id_3, id_4), \\ PCS(id_5), PB(id_6)$$

However, Q_{real} is also an inclusion-minimal query, as opposed to $Q_{general}$ since $Q_{real} \subsetneq Q_{general}$.

In the following section we study the complexity of the above computational problems for the different models of provenance. We will analyze the complexity with respect to the different facets of the input, notations for which are provided in Table 1.

Table 1: Notations

Ex	K-example
I	Input database
O	Output relation and its provenance
m	Total number of monomials
k	Number of attributes of the output relation
n	(Maximal) Number of elements in a monomial
r	Sum of arities in the atoms of a query body
d	Number of distinct relation names in the provenance

We list our results for the QUERY-BY-PROVENANCE problem in Table 2. The columns of the table list: the bound for the number of possible consistent queries for each of the semirings, the maximal length of a consistent query if one exists, and the lower and upper bounds of finding a consistent query. Each result has a reference to the relevant proposition. Some of the proofs are omitted for brevity and can be found in the full version [22].

5 LEARNING FROM $\mathbb{N}[X]$, $\mathbb{B}[X]$-EXAMPLES

We start our study with the case where the given provenance consists of $\mathbb{N}[X]$ expressions. This is the most informative form of provenance under the semiring framework, and thus the most informative explanation. In particular, we note that given the $\mathbb{N}[X]$ provenance, the number of query atoms (and the relations occurring in them) are trivially identifiable. What remains is the choice of variables to appear in the query atoms (body and head), and as a consequence, decide how to order the tuples in each monomial.

We first bound the number of possible consistent queries:

Proposition 5.1. *For every choice of $r, k \in \mathbb{N}$, every $\mathbb{N}[X]$-example has at most $O(B_r r^k 2^r)$ consistent queries, where B_r is the Bell number of r. Furthermore, there exists an $\mathbb{N}[X]$-example whose output tuples have arity of k, and monomials have arity of r, for which there exists exponentially many (in both r and k) consistent queries.*

Proof. We start by analyzing the case for boolean queries, i.e., $k = 0$. Consider the set of indexes $\{1, \ldots, r\}$ in the query body. Every partition of this set of indexes defines a different consistent query, since the partition determines exactly which indexes will have the same variable, assuming we impose an order on the query atoms (e.g., the query $Q() : -R(x, x, x), R(x, x, x), T(x, y)$ is determined by the partition $\{1, 2, 3, 4, 5, 6, 7\}, \{8\}$). We can thus say that the maximum number of consistent boolean queries w.r.t an $\mathbb{N}[X]$-example is the number of partitions of the set $\{1, \ldots, r\}$ into disjoint non-empty subsets. This number is the Bell number B_r. Each subset of the partition can then either be instantiated with a constant or with a distinct variable. As there can be at most r subsets in the partition, the number of options is bounded by 2^r. In the general case, when queries are not boolean but have k output attributes, we further have to choose which variables will be projected to the head. Since the maximum number of unique variables in the query body is r, there are at most r^k options to do so. Therefore the number of consistent queries can be bounded by $B_r r^k 2^r$. For the second part of the claim, consider an $\mathbb{N}[X]$-example for which a query with a single constant at every index, both in the head of the query and its body, is consistent (e.g. $Q(1, 1) : -R(1, 1, 1), R(1, 1, 1), T(1, 1)$). In this case, replacing every subset of 1s with different variables, will also result in a consistent query. As there are exponentially many subsets of attributes to the query, there are exponentially many consistent queries w.r.t the $\mathbb{N}[X]$-example. □

We have shown a bound for the number of different consistent queries, but it is also important to note that there can be multiple inclusion-minimal queries. It is easy to show an example where there are may be exponentially many inclusion-minimal queries. We next show an upper bound

Table 2: Results for the QUERY-BY-PROVENANCE Problem

Semiring	Number of consistent queries	Small world query length	Lower bound	Upper bound
$\mathbb{N}[X]$	$O(B_r r^k 2^r)$ (Prop. 5.1)	n (Trivial)	NP-complete in k (Prop. 5.4)	$O(n^{2k}m + krn^k)$ (Prop. 5.2)
$\mathbb{B}[X]$				
$Trio(X)$	∞ (Prop. 6.1)	$k + d(n-1)$ (Prop. 6.5)	Find minimal sized query is NP-complete in k (Prop. 6.4)	$O(n^{O(k)}mkr)$ (Prop. 6.7)
$PosBool(X)$				
$Why(X)$				

5.1 An Efficient Algorithm for Finding a Query

Although the number of query atoms is known from the given example, finding a consistent query efficiently is non-trivial. An important observation in this respect is that we can focus on finding atoms that "cover" the attributes of the output relation (i.e. that include the right values of the output tuples, in the right order), and the number of required such atoms is at most k (the arity of the output relation). We may need further atoms so that the query realizes all provenance tokens (eventually, these atoms will also be useful in imposing e.g. join constraints), and this is where care is needed to avoid an exponential blow-up with respect to the provenance size. To this end, we observe that we may generate a "most general" part of the query simply by generating atoms with fresh variables, and without considering all permutations of parts that do not contribute to the head. This will suffice to guarantee a consistent query, but may lead to the generation of a too general query; this issue will be addressed in Section 5.2.

Algorithm 1: FindConsistentQuery (N[X])

```
input  : An $\mathbb{N}[X]$ example $Ex = (I, O)$
output : A consistent query $Q$ or an answer that none exists
1  Let $(t_1, M_1), ..., (t_m, M_m)$ be the tuples and
     corresponding provenance monomials of $O$ ;
2  $(V, E) \leftarrow BuildLabeledGraph((t_1, M_1), (t_2, M_2))$ ;
3  foreach consistent matchings $E' \subseteq E$ s.t. $|E'| \leq k$ do
4      if $\cup_{e \in E'} label(e) = \{1, \ldots, k\}$ then
5          $Q \leftarrow BuildQueryFromMatch(E', Ex)$ ;
6          foreach $1 < j < m$ do
7              if not consistent$(Q, t_j, M_j)$ then
8                  Go to next matching ;
9          return $Q$ ;
10 Output "No consistent query exists";
```

We next detail the construction, shown in Algorithm 1. We separate (line 1) monomials so that each (t_i, M_i) is a tuple along with a single monomial of its provenance expression. We then start by picking two tuples and monomials (see below a heuristic for making such a pick) and denote the tuples by t_1 and t_2 and their provenance by $M_1 = a_1 \cdot ... \cdot a_n$ and $M_2 = b_1 \cdot ... \cdot b_n$ respectively. Our goal is to find all "matches" of *parts* of the monomials so that all output attributes are covered. To this end, we define (line 2) a full bipartite graph $G = (V_1 \cup V_2, E)$ where each of V_1 and V_2 is a set of n nodes labeled by $a_1, \ldots, a_n$ and $b_1, \ldots, b_n$ respectively. We also define labels on each edge, with the label of (a_i, b_j) being the set of all attributes that are *covered* by a_i, b_j, in the following sense: an output attribute A is covered if there is an input attribute A' whose value in the tuple corresponding to a_i (b_j), matches the value of the attribute A in t_1 (respectively t_2).

We then (lines 3-4) find all matchings, *of size k or less*, that cover all output attributes; namely, that the union of sets appearing as labels of the matching's edges equals $\{1, ..., k\}$. As part of the matching, we also specify which subset of the edge label attributes is chosen to be covered by each edge (the number of such options is exponential in k). It is easy to observe that if such a cover (of any size) exists, then there exists a cover of size k or less. We further require that the matching is consistent in the sense that the permutation that it implies is consistent.

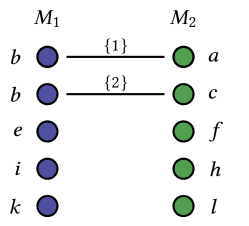
For each such matching we generate (line 5) a "most general query" Q corresponding to it, as follows. We first consider the matched pairs a_i, b_j one by one, and generate a query atom for each pair. This is done by assigning the same variable to the head attribute A covered by the edge and to the attribute covering it in the new atom A'. Note that the query generation is done here based only on k pairs of provenance atoms, rather than all n atoms, since we only examine the k edges of the matching.

To this end, we further generate for each provenance token a_i that was not included in the matching a new query atom with the relation name of the tuple corresponding to a_i, and fresh variables. Intuitively, we impose minimal additional constraints, while covering all head attributes and achieving the required query size of n.

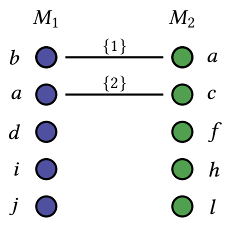
Each such query Q is considered as a "candidate", and its consistency needs to be verified with respect to the other tuples of the example (line 7). One way of doing so is simply by evaluating the query with respect to the input, checking that the output tuples are generated, and their provenance includes those appearing in the example. As a more efficient solution, we test for consistency of Q with respect to each example tuple by first assigning the output tuple values to the head variables, as well as to the occurrences of these variables in the body of Q (by our construction, they can occur in at most k query atoms). For query atoms corresponding to provenance annotations that have not participated in the cover, we only need to check that for each relation name, there is the same number of query atoms and of provenance annotations relating to it. A subtlety here is in handling coefficients; for the part of provenance that has participated in the cover, we can count the number of assignments. This number is multiplied by the number of ways to order the other atoms (which is a simple combinatorial computation), and the result should exceed the provided coefficient.

Proposition 5.2. *Given a $\mathbb{N}[X]$-example, Algorithm 1 finds a consistent query if one exists.*

Choosing the two tuples. For correctness and worst case complexity guarantees, any choice of tuples as a starting point for the algorithm (line 1) would suffice. Naturally, this choice still affects the practical performance, and we aim at minimizing the number of candidate matchings. A simple but effective heuristic is to choose two tuples and monomials for which the number of distinct values (both in the output tuple and in input tuples participating in the derivations) is maximal.



(a) Matching A



(b) Matching B

Figure 2: Matchings for Example 5.3

Example 5.3. Reconsider our running example depicted in Figure 1e. Assume that the monomials b^2eik and $acfhl$ were picked for graph generation. Each monomial forms a side in the bipartite graph. The algorithm now traverses all partial matchings of size ≤ 2, and checks whether aligning two tuples that are connected by an edge, one below the other, forms a vector of constants that also appears in an attribute of the two corresponding output tuples. Figure 2a depicts a matching of size 2 which consists of the edges $(b, a), (b, c)$ where the first attribute of input tuples **b** and **a** covers the first output attribute (aligning the tuple b below the tuple a creates the constants vector $\binom{2}{1}$ that appear in the first attribute of the output tuple), and the second attribute of input tuples **b** and **c** covers the second output attribute (aligning the tuple b below the tuple c creates the constants vector $\binom{1000}{300}$ that appears in the second attribute of the output tuple).

Generating a query based on this matching results in the query $Q_{general}$ from Example 4.5, since the first and second projected variables have been set to the first and third attributes of the most general atoms $B(id_1, b_1)$ and $B(id_2, b_2)$, respectively and the other atoms remain most general with no joins and projected attributes. The algorithm now verifies the consistency of $Q_{general}$ with respect to the other monomials by assigning the output tuple to the head, e.g. assigning id_1 and b_2 to 2 and 1000 (for the output tuple (2, 1000)), and returns $Q_{general}$. There are other valid partial matchings of these two tuples that will yield a somewhat different query to $Q_{general}$. If we were to choose the monomials $badij$ and $acfhl$, we would have the different matching shown in Figure 2b (where the edges (b, a) and (a, c) cover the head attributes). Note that, in general, a single edge can cover several attributes of the output.

$Q_{general}$ is a very general query, and is probably not the original one. We will adapt our solution so that it finds a more "precise" query, i.e., an inclusion-minimal query (Definition 4.4 in Section 4) in the next subsection.

Complexity. The algorithm's complexity is $O(n^{2k}m + krn^k)$: at most n^k matchings are considered; for each matching, a single query is generated, and consistency is validated in $O(n^k)$ for each of the m example tuples. Furthermore, for each of the matchings, we scan the attributes of the tuples in the matching in $O(kr)$. The exponential factor only involves k, which is much smaller than n and m in practice. Can we do even better? We can show that if $P \neq NP$, there is no algorithm running in time polynomial in k.

Proposition 5.4. *Deciding the existence of a consistent query with respect to a given $\mathbb{N}[X]$-example is NP-complete in k.*

5.2 Achieving a tight fit

To find inclusion-minimal queries, we next refine Algorithm 1 as follows. We do not halt when finding a single consistent query, but instead find all of those queries obtained for some matching. For each consistent query Q, we examine queries obtained from Q by (i) equating variables and (ii) replacing variables by constants where possible (i.e. via an exact containment mapping [1]). We refer to both as *variable equating*. To explore the possible combinations of variable equatings, we use an algorithm inspired by data mining techniques (e.g., [5]): in each iteration, the algorithm starts from a minimal set of variable equatings that was not yet determined to be (in)consistent with the example. E.g., in the first iteration it starts by equating a particular pair of variables. The algorithm then tries, one-by-one, to add variable equatings to the set, while applying transitive closure to ensure the set reflects an equivalence relation. If an additional equating leads to an inconsistent query, it is discarded. Each equatings set obtained in this manner corresponds to a homomorphism h over the variables of the query Q, and we use $h(Q)$ to denote the query resulting from replacing each variable x by $h(x)$.

Importantly, by equating variables or replacing variables by constants we only impose further constraints and obtain no new derivations. In particular, the following result holds, as a simple consequence of Theorem 7.11 in [29] (note that we must keep the number of atoms intact to be consistent with the provenance):

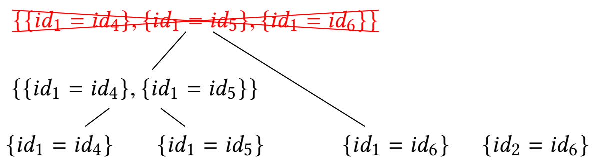
Proposition 5.5. *Let Q be a CQ over a set of variables $\mathcal{V}$. Let $h : \mathcal{V} \mapsto \mathcal{V} \cup C$ be a homomorphism. For every $\mathbb{N}[X]$-example Ex, if Q is not consistent with Ex, then neither is $h(Q)$.*

We can model the homomorphism process between queries, achieved by variable equating, as a lattice whose leaves represent a single variable equating in the query and its top element is the query with a single variable in all atoms. Every variable equating implies a move from the current lattice node to a parent of that node (see next an illustration and example). By this model, the proposition above actually determines that if a query represented by a node in the lattice is consistent, then all of the queries represented by the descendants of this node are also consistent and furthermore, all queries represented by the frontier of the lattice are consistent. We next use these observations to establish an algorithm for finding an inclusion-minimal query.

Consequently, the algorithm finds a *maximal set of variable equatings* that is consistent with the query, by attempting to add at most $O(r^2)$ different equatings, since there are $\binom{r}{2}$ pairs of variables (r is the sum of arities of the atoms in the body of Q, see Table 1). We record every query that was found to be (in)consistent – in particular, every subset of a consistent set of equatings is also consistent – and use it in the following iterations (which again find maximal sets of equatings).

Checking for consistency. This check may be done very efficiently for query atoms that contribute to the head, since we only need to check that equality holds for the provenance annotations assigned to them. For other atoms we no longer have their consistency as a given and in the worst case we would need to examine all matchings of these query atoms to provenance annotations.

Example 5.6. Reconsider our running example query $Q_{general}$ in Example 4.5. A part of the lattice is depicted in Figure 3. The algorithm starts by considering individually each pair of variables as well as pairs of variables and constants co-appearing in the two output tuples or in the tuples used in their provenance. In our example, when considering the lattice element $\{id_1 = id_4\}$, the



~~$\{\{id_1 = id_4\}, \{id_1 = id_5\}, \{id_1 = id_6\}\}$~~
$\{\{id_1 = id_4\}, \{id_1 = id_5\}\}$
$\{id_1 = id_4\}$
$\{id_1 = id_5\}$
$\{id_1 = id_6\}$
$\{id_2 = id_6\}$

Figure 3: Part of the lattice in Example 5.6

algorithm will find that the query $Q_{id_1=id_4}$ (i.e. $Q_{general}$ after equating id_1 and id_4), is still consistent. Next, the algorithm will find that equating id_1, id_5 in $Q_{id_1=id_5}$ also yields a consistent query so it will proceed with $Q_{id_1=id_4, id_1=id_5}$. However, if we try to add the equality $id_1 = id_6$, the consistency check will discover that $Q_{id_1=id_4, id_1=id_5, id_1=id_6}$ is not consistent anymore, so we will not continue on this path of the lattice (marked with a red "x" in Figure 3). Of course, multiple steps may yield the same equivalence classes in which case we perform the computation only once. The resulting query is Q_{real} shown in Example 1.2. Any further step with respect to Q_{real} leads to an inconsistent query, and so it is returned as output.

Proposition 5.7. *Given a consistent query w.r.t an* $\mathbb{N}[X]$*-example, the procedure finds an inclusion-minimal query.*

For each consistent query found by Algorithm 1, there may be multiple inclusion-minimal queries obtained in such a manner (though the number of such queries observed in practice was not very large, see Section 7). If we wish to provide a single query as output, when multiple inclusion-minimal queries are obtained, a natural heuristic that we employ is to prefer a query with the least number of unique variables (this was implemented in our experimental study).

Adapting the Solution for $\mathbb{B}[X]$. In the $\mathbb{B}[X]$ semiring, exponents are kept but coefficients are not, so we can adapt the algorithm of $\mathbb{N}[X]$ (Algorithm 1), omitting the treatment of coefficients. In $\mathbb{N}[X]$, for the part of the provenance that has participated in the cover, we count the number of assignments and multiply by the number of ways to order the other atoms (which is a simple combinatorial computation), and verify that the result exceeds the provided coefficient. In $\mathbb{B}[X]$, we remove this step of the algorithm.

6 LEARNING FROM WHY, TRIO, AND POSBOOL EXAMPLES

We next study the problem of learning queries from $Why(X)$-examples. This semiring is less detailed than $\mathbb{N}[X]$ and thus often easier to store and present, but is in turn more challenging for query inference. We will adapt the solution to the $Trio(X)$ and $PosBool(X)$ semirings.

A natural approach is to reduce the problem of learning from a $Why(X)$-example to that of learning from an $\mathbb{N}[X]$-example (note that a $Why(X)$-example without self-joins is equivalent to an $\mathbb{N}[X]$-example). Recall that the differences are the lack of coefficients and the lack of exponents. The former is trivial to address (see solution for $\mathbb{B}[X]$ in the previous section), but the latter means that we do not know the number of query atoms. In particular, for $Why(X)$-examples we have:

Proposition 6.1. *There exists a* $Why(X)$*-example with an infinite number of non-equivalent consistent queries.*

A first plausible idea is to consider the $Why(X)$-example as an $\mathbb{N}[X]$-example, i.e., assume the number of query atoms is equal to the number of tuples in the largest monomial of the example. Surprisingly, we cannot be sure that this suffices:

Proposition 6.2. *There exists a* $Why(X)$ *example for which there is no consistent CQ with n atoms (recall that n is the length of the largest monomial, see Table 1), but there exists a consistent CQ with more atoms.*

prov.	A	B	C
a	1	2	3
b	3	4	5
c	6	7	8
d	7	6	8

(a) Relation R

A	B	C	prov.
1	1	5	$a \cdot b$
6	7	8	$c \cdot d$

(b) $Why(X)$ **Example**

Figure 4: Source and $Why(X)$ **Example for Prop. 6.2**

Proof. Let *Ex* denote the example depicted in Figure 4. A consistent query with two atoms can impose two possible orderings on the tuples in each monomial. We show that both orderings do not form a consistent query. Consider the first ordering of the tuples. A consistent query needs to be satisfied by the assignments:

$$Q(1,1,5) : -R(1,2,3), R(3,4,5)$$
$$Q(6,7,8) : -R(6,7,8), R(7,6,8)$$

But no query can be satisfied by these two assignments because the first assignment requires that the second variable in the head will be bound by the first variable in the first atom in the body. But, the second assignment requires that the second variable in the head will be different from the variable in the mentioned position. Thus, there is no query that is satisfied by both assignments.

Consider the second ordering. A consistent query needs to be satisfied by the assignments:

$$Q(1,1,5) : -R(3,4,5), R(1,2,3)$$
$$Q(6,7,8) : -R(6,7,8), R(7,6,8)$$

Again, observe that in the first assignment, the first variable in the head must be bound by the first variable in the second atom in the body. But, in the second assignment, the first variable in the head is different from the variable in the mentioned position. Thus, again, there is no query that is satisfied by both assignments, establishing that there is no consistent query of length 2 w.r.t *Ex*.

Now, consider the monomials are $a \cdot a \cdot b$ and $c \cdot d \cdot d$. A consistent query w.r.t these monomials needs to satisfy the assignments:

$$Q(1,1,5) : -R(1,2,3), R(1,2,3), R(3,4,5)$$
$$Q(6,7,8) : -R(6,7,8), R(7,6,8), R(7,6,8)$$

Hence, the following query is consistent:

$$Q(x,y,z) : -R(x,w,u), R(y,v,u), R(k,m,z)$$

□

However, finding one consistent query is not difficult, using a method that takes all possible tuple alignments into account:

Proposition 6.3. *If there exists a consistent query w.r.t a given* $Why(X)$*-example, there is a consistent query of length* n^m.

Proof. Denote the example as *Ex* and the consistent query as Q'. Consider the query Q which has at most n^m atoms and is built in the following manner: have each tuple in *Ex* participate in all possible permutations of tuples in the rows above and below it with the same relation, imposing an order on the tuples in a monomial and then treat the resulting monomials as an $\mathbb{N}[X]$-example, and create a query by replacing each unique constant column with a variable (see previous example). We first show that Q is safe (note that Q' is consistent, so it is in particular safe). Suppose the atoms that have attributes projected to the head in Q' are $a'_1, \ldots, a'_i$. Take the tuples to which they are mapped to in each row j, $t^j_1, \ldots, t^j_i$, then Q also contains atoms $a_1, \ldots, a_i$ mapped to $t^j_1, \ldots, t^j_i$, respectively in each row j (this is because in part of the ordering imposed on the monomials, $t^1_1, \ldots, t^m_1$ appear one below another and the same goes for $t^1_x, \ldots, t^m_x$ for every $2 \leq x \leq i$). Therefore, the same attributes of $a'_1, \ldots, a'_i$ exist in $a_1, \ldots, a_i$ and can also be projected to the head in Q. Furthermore, Q is consistent by the way it was built. Every atom was created for a specific combination of tuples, so every atom of Q can be mapped to a specific tuple appearing in every row. In addition, for every row j, the atoms of Q cover all tuples in this row since an atom was created for each permutation of tuples, and in particular, there is at least one atom that was created to match each tuple in row j. □

Based on this result, we can find a consistent query in a straightforward way. However, a query that has n^m atoms is very long and probably overfits the example. In general, in the $\mathbb{N}[X]$ case, the number of query atoms is set by the provenance and the only degree of freedom is variable equatings. On the other hand, for the $Why(X)$ case, the number of query atoms is also flexible. This calls for a different criterion of "tight fit" for $Why(X)$ provenance, which is minimizing the number of atoms in a consistent query. In general, we can show the following result regarding this criterion:

Proposition 6.4. *Given a $Why(X)$-example, deciding the existence of a consistent query with $\leq n$ atoms is NP-complete in k.*

We can however show a "small world" property, that will guide our solution.

Proposition 6.5. *For any $Why(X)$-example, if there exists a consistent query then there exists a consistent query with $k+d\cdot(n-1)$ atoms or less, where d is the number of distinct relation names occurring in the provenance monomials (see Table 1).*

Intuitively, there are at most k atoms contributing to the head. The worst case is when only one "duplicated" annotation contributes to the head, and then in each provenance monomial there are at most $n-1$ remaining annotations. If the query includes a single relation name ($d = 1$), then a query with at most $n-1$ more atoms would be consistent. Otherwise, as many atoms may be needed for each relation name.

Together with our algorithm for $\mathbb{N}[X]$, Proposition 6.5 dictates a simple algorithm that exhaustively goes through all $\mathbb{N}[X]$ expressions that are compatible with the $Why(X)$ expressions appearing in the example, and whose sizes are up to $k+d\cdot(n-1)$. This, however, would be highly inefficient.

Instead of the inefficient exhaustive algorithm, we next present a much more efficient algorithm for finding a consistent query of minimal length. The algorithm will operate greedily by duplicating atoms in the query body when a self-join is implied by one of the provenance monomials.

The pseudo-code for an efficient algorithm for finding CQs consistent with a given $Why(X)$-example is given in Algorithm 2. The idea is to traverse the examples one by one, trying to "expand" (by adding atoms) candidate queries computed thus far to be consistent with the current example. We start (line 1), as in the $\mathbb{N}[X]$ case, by "splitting" monomials if needed so that each tuple is associated with a single monomial. We maintain a map $\boldsymbol{Q}$ whose values are candidate queries, and keys are the parts of the query that contribute to the head, in a canonical form (e.g. atoms are lexicographically ordered). This will allow us to maintain only a single representative for each such "contributing part", where the representative is consistent with all the examples observed so far. For the first step (line 2) we initialize $\boldsymbol{Q}$ so that it includes only (t_1, M_1) (just for the first iteration, we store an example rather than a query). We then traverse the remaining examples one by one (line 3). In each iteration i, we consider all queries in $\boldsymbol{Q}$; for each such query Q, we build a bipartite graph (line 6) whose one side is the annotations appearing in M_i, and the other side is the *atoms of* Q. The label on each edge is the set of head attributes covered jointly by the two sides: in the first iteration this is exactly as in the $\mathbb{N}[X]$ algorithm, and in subsequent iterations we keep track of covered attributes by each query atom. Then, instead of looking for *matchings* in the bipartite graph, we find (line 7) all *sub-graphs* whose edges cover all head attributes (again specifying a choice of attributes subset for each edge). Intuitively, having e edges adjacent to the same provenance annotation corresponds to the same annotation appearing with exponent e, so we "duplicate" it e times (lines 8-10). On the other hand, if multiple edges are adjacent to a single query atom, we also need to "split" (Lines 11-13) each such atom, i.e. to replace it by multiple atoms (as many as the number of edges connected to it). Intuitively each copy will contribute to the generation of a single annotation in the monomial. Now (line 14), we construct a query Q' based on the matching and the previous query "version" Q: the head is built as in Algorithm 1, and if there were x atoms not contributing to the head with relation name R in Q, then the number of such atoms in Q' is the maximum of x and the number of annotations in M_i of tuples in R that were not matched. Now, we "combine" Q' with Q'' which is the currently stored version of a query with the same contributing atoms (lines 15- 16). Combining means setting number of atoms for each relation name not contributing to the head to be the maximum of this number in Q' and Q''.

Algorithm 2 stores queries according to the atoms that have the same variables as the ones in the head. Each matching that does not stem from the existing matchings in $\boldsymbol{Q}$ (i.e., splitting some of the atoms in an existing cover) will not result in a consistent query since it will not be consistent with the previous rows. The number of combinations of size k can there be out of n different relations is n^k. $\boldsymbol{Q}$ does not store sets of contributing atoms that are subsets of other existing keys.

Complexity. The number of keys in $\boldsymbol{Q}$ is exponential only in k; the loops thus iterate at most $m \cdot n^k \cdot n^k \cdot (n + n^2)$ times, so the overall complexity is $O(n^{O(k)} \cdot mkr)$.

Achieving a tight fit. Algorithm 2 produces a set of candidate queries, which may not be syntactically minimal. To discard atoms that are "clearly" redundant, we first try removing atoms not contributing to the head, and test for consistency. We then perform the process of variable equating as in Section 5.2.

Example 6.6. Reconsider our running example, but now with the why-provenance given in Figure 1e. If we start from the first monomials of the tuples (2, 1000) and (1, 300) then we generate a bipartite graph with $V_1 = \{b, e, i, k\}$ and $V_2 = \{a, c, f, h, l\}$, and obtain the cover E' (seen in Figure 5a) where the edge (b, a)

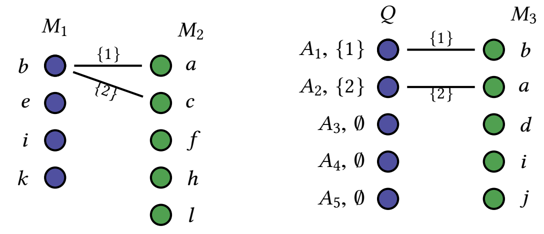
Algorithm 2: FindConsistentQuery (Why(X))

```
input  : A $Why(X)$ example Ex
output : A set of consistent queries (possibly empty, if
         none exists)
1  Let $(t_1, M_1), ..., (t_m, M_m)$ be the tuples and
     corresponding provenance monomials of $Ex$ ;
2  $\mathcal{Q} \leftarrow \{NULL : (t_1, M_1)\}$ ;
3  foreach $2 \leq i \leq m$ do
4      foreach $Q \in$ values($\mathcal{Q}$) do
5          $\mathcal{Q} \leftarrow \mathcal{Q} - \{Q\}$ ;
6          $(V_1 \cup V_2, E) \leftarrow BuildGraph(Q, (t_i, M_i))$ ;
7          foreach sub-graph $E' \subseteq E$ s.t. $|E'| \leq$
             $k$ and $\cup_{e \in E'}$ label$(e) = \{1, \ldots, k\}$ do
8              foreach provenance annotation $a$ in $M_i$ do
9                  if $a$ is an endpoint of more than one edge
                     in $E'$ then
10                     $E' \leftarrow split(E', a)$ ;
11             foreach atom $C \in Q$ do
12                 if $C$ is an endpoint of more than one edge
                     in $E'$ then
13                     $E' \leftarrow split(E', C)$ ;
14             $Q' \leftarrow BuildQuery(E', Q)$;
15             $Q'' \leftarrow \mathcal{Q}.get(contribs(Q'))$ ;
16             $\mathcal{Q}.put(contribs(Q'), combine(Q', Q''))$ ;
17 return $values(\mathcal{Q})$ ;
```

covers the first head attribute and (b, c) covers the second. The fact that the tuple **b** is connected to two edges will lead to the split of this tuple to the atoms $B(id_1, b_1)$ and $B(id_2, b_2)$. When we continue with E', no duplication is performed, and we get a query Q with the two atoms $B(id_1, b_1)$, $B(id_2, b_2)$ contributing to the head, and three most general atoms. Then, we match Q to the monomial $badij$, resulting in a sub-graph matching both $B(id_1, b_1)$ and $B(id_2, b_2)$ to **b** and **a**, respectively. After variables equating (see Subsection 5.2), we obtain the query Q_{real} shown in Example 1.2 (other covers are possible, but they will also result in Q_{real} after variable equating). If instead we start with the monomials $badij$ and $acfhl$, the partial matching chosen will be the edge (b, a) which covers the first head attribute and the edge (a, c) which covers the second and forming the atoms $B(id_1, b_1)$, $B(id_2, b_2)$ again. Continuing to the monomial $beik$, we would split the tuple b into two tuples since the matching will include the edges $(B(id_1, b_1), b)$ that covers the first head attribute and $(B(id_2, b_2), b)$ which covers the second.

Proposition 6.7. *Given a Why(X)-example, Algorithm 2 finds a consistent query if one exists.*

Adapting the Solution for $Trio(X)$ and PosBool(X). In the $Trio(X)$ semiring coefficients are kept but exponents are not. To handle this case we employ Algorithm 2, with a simple modification: upon checking consistency of a candidate query with a tuple, then we further check that there are as many derivations that use the tuples of the monomial as dictated by the coefficient (as done in Section 5). In the semiring of positive boolean expressions (*PosBool*(X)) + and · are interpreted as disjunction and conjunction, respectively. If the expressions are given in DNF, Algorithm 2 may be used here as well. The only difference is the possible



(a) 1st iteration in 6.6 **(b) 2nd iteration in 6.6**

Figure 5: Subgraphs for Example 6.6

absorption of monomials ($a + a \cdot b \equiv a$), but we already assume that only a subset of the monomials are given. If the expressions are given in an arbitrary form there is an additional (exponential time) pre-processing step of transforming them into DNF.

7 IMPLEMENTATION AND EXPERIMENTS

Our experimental study is composed of experiments based on the actual output and provenance of the benchmark queries in [40, 43] measuring both accuracy and scalability. All experiments were performed on Windows 8, 64-bit, with 8GB of RAM and Intel Core Duo i7 2.59 GHz processor. We have implemented our algorithms in JAVA with MS SQL server as the underlying database management system.

As we have shown in Sections 5 and 6, the algorithms for the $\mathbb{N}[X]$ and $Why(X)$ semirings also capture the other semirings we have studied, with slight modifications. Namely, the algorithms for $\mathbb{N}[X]$ also apply for the $\mathbb{B}[X]$ semiring, ignoring the treatment of coefficients, and the algorithms presented for $Why(X)$ also capture the *PosBool*(X) semiring (since there is only an absorption of monomials), and the $Trio(X)$ semiring with the handling of coefficients as done for $\mathbb{N}[X]$ in Algorithm 1. Hence, our experiments focus on the $\mathbb{N}[X]$ and $Why(X)$ models. In particular, Algorithm 1 will behave in the same manner for $\mathbb{B}[X]$-examples as for $\mathbb{N}[X]$-examples, and Algorithm 2 will behave in the same manner for *PosBool*(X)\$Trio(X)$-examples as for $Why(X)$-examples.

To examine our approach, we have used multiple queries with varying complexity. Namely, Q1–Q6 from [43] as well as (modified, to drop aggregation and arithmetics) the TPC-H queries TQ2–TQ5, TQ8 and TQ10. The queries have 2–8 atoms, 18–60 variables, and multiple instances of self-joins (we show Q6 for illustration in Figure 6; the reader is referred to [40, 43] for the other queries).

7.1 Accuracy

We have used the system to "reverse engineer" the queries mentioned above. This part of the experiments had two objectives: (1) understanding the number of examples needed to infer the exact query (2) measure the precision and recall of the query inferred by the system with comparison to the original. We have evaluated each query using a provenance-aware query engine, and have then sampled random fragments (of a given size that we vary) of the output database and its provenance (we have tried both $\mathbb{N}[X]$ and Why(X)), feeding it to our system. In each experiment we have gradually added random examples until our algorithm has retrieved the original query. This was repeated 3 times. We report (1) the *worst-case* (as observed in the 3 executions) number of examples needed until the original query is inferred, and (2) for fewer examples (i.e. before convergence to the actual query),

Table 3: Results for the TPC-H query set and the queries from [43] with $\mathbb{N}[X]$ provenance

Query	Worst-case number of examples to learn the original query	Difference between original and inferred queries for fewer examples
Q1 (TQ3)	14	The inferred Query includes an extra join on a "status" attribute of two relations. Only 2–3 values are possible for this attribute, and equality often holds.
Q2	2	
Q3	5	For 2 examples, the inferred query contained an extra constant. For 3 and 4 examples, it included an extra join.
Q4	19	For 2 examples, the inferred query included an extra constant. For 3–18, it included an extra join on a highly skewed "status" attribute.
Q5	11	The inferred query included an extra join on a "name" attribute.
Q6	3	The inferred query included an extra constant.
TQ4	234	The inferred query included an extra join on "orderstatus" and "linestatus" attributes of two relations (they have two possible values). One of the original join conditions has led to the occurrence of the same value in these attributes in the vast majority of joined tuples.
TQ10	4	The inferred query contained an extra constant.
TQ2	3	The inferred query contained an extra constant.
TQ5	3	The inferred query contained an extra constant.
TQ8	18	For 2 examples, the inferred query contained an extra constant. For 4-17 exam––ples, the query had an extra join between a "status" attribute of two relations.

$$\boxed{\begin{array}{l} ans(a,b) :- supplier(c,a,add,k,p,d,c_1), \\ partsupp(h,c,v,j,c_2),\ nation(w,na_2,r,c_8), \\ part(h,i,z,q,t,s,e,rp_2,c_3),\ region(r,u,c_7), \\ partsupp(h,o,x,n,c_4),\ nation(k,na1,r,c_6), \\ supplier(o,b,y,w,p_2,d_2,c_5) \end{array}}$$

Figure 6: $Q6$

the differences between the inferred queries and the actual one in the worst-case run of the experiment.

The results are reported in Table 3. For some queries the convergence is immediate, and achieved when viewing only 2–5 examples. For other queries, more examples are needed, but with one exception (TQ4), we converge to the original query after viewing at most 19 tuples for the different queries. For TQ4 only a very small fraction of the output tuples reveal that an extra join should not have appeared, and so we need one of these tuples to appear in the sample. Furthermore, even for smaller sets of examples, the inferred query was not "far" from the actual query. The most commonly observed difference involved extra constants occurring in the inferred query (this typically happened for a small number of examples, where a constant co-occurred by chance). Another type of error was an extra join in the inferred query; this happened often when two relations involved in the query had a binary or trinary attribute (such as the "status" attribute occurring in multiple variants in TPC-H relations), which is furthermore skewed (for instance, when other join conditions almost always imply equality of the relevant attributes). We have also measured the precision and recall of the output of the inferred query w.r.t. that of the original one. Obviously, when the original query was obtained, the precision and recall were 100%. Even when presented with fewer examples, in almost all cases already with 5 examples, the precision was 100% and the recall was above 90%. The only exception was Q5 with 75% recall for 5 examples.

The results for $Why(X)$ are shown in Table 4. For queries with no self-join, the observed results were naturally the same as in the $\mathbb{N}[X]$ case; we thus report the results only for queries with self-joins (some queries included multiple self joins). When presented with a very small number of examples, our algorithm was not always able to detect the self-joins (see comments in Table 7); but the overall number of examples required for convergence has only marginally increased with respect to the $\mathbb{N}[X]$ case.

7.2 Execution Times

As we have established in the accuracy experiments, a small number of examples will usually suffice to infer the original query. To also account for the execution time of our algorithms, we have increased the number of examples up to 500, which is about twice as many examples needed to infer each of the queries. The results for $\mathbb{N}[X]$ provenance and Q1–Q6 exhibit good scalability: the computation time for 500 examples was 0.4 seconds for Q1 (TQ3), 0.1 seconds for Q2, 0.7 seconds for Q3, 1 second for Q4 and 0.4 and 1.1 seconds for Q5 and Q6 respectively. The performance for the TPC-H queries was similarly scalable: for 500 examples, the computation time of TQ2 and TQ10 (which are the queries with the maximum number of head attributes: 8 and 7, resp.) was 0.2 and 0.4 seconds respectively. The runtimes for TQ4 and TQ8 were 0.1, 0.9 seconds resp. The number of example for TQ5 was limited due to the query output size. For 15 examples, the runtime for this query was 0.2 seconds. We have repeated the experiment using $Why(X)$ provenance. The computation time was generally fast, and only slightly slower than the $\mathbb{N}[X]$ case, with a max runtime of 1.3 seconds for Q4 and TQ8. This is consistent with our complexity analysis.

Effect of Tuples Choice. Recall that Algorithm 1 starts by finding consistent queries w.r.t two example tuples and explanations. In Section 5, we have described a heuristic that chooses the two tuples with the least number of shared values. We have measured the effect of this optimization on Q6 and found that using the optimization leads to a single matching in the graph, as oppose to a random choice of tuples that has led to 4 matchings. Making such a random choice, instead of using our optimization led to a runtime which was more than 1.3 times slower on average.

8 DISCUSSION AND CONCLUSIONS

We have formalized and studied the problem of "query-by-provenance", where queries are inferred from example output tuples and their

Table 4: Results for the TPC-H query set and the queries from [43] containing self-joins with $Why(X)$ provenance

Query	Worst-case number of examples to learn the original query	Difference between original and inferred queries for fewer examples
Q2	2	
Q3	5	The inferred query for 2–4 examples did not include self-joins.
Q4	19	For 2–3 examples, the inferred query did not include self-joins. For 4–18 examples, the query had an extra join on a “status" attribute.
Q5	13	The inferred query for 2–12 examples did not include self-joins.
Q6	3	The inferred query included an extra constant.
TQ8	18	For 2–3 examples, the inferred query contained an extra constant. For 4-17 the query had an extra join between a “status" attribute of two relations.

provenance. We have theoretically analyzed and experimentally demonstrated the effectiveness of the approach in inferring highly complex CQs, including ones with multiple self-joins, based on a small number of output and provenance examples and their provenance. Note that for UCQs, a stronger definition is needed.

A notable provenance model that we have not discussed so far is that of lineage [16], which entails a set representation of the provenance, i.e., the contributing input tuples' annotations for a given output tuple are represented as a set. The semiring model can support lineage, and the interpretation of Definition 4.2 in this case is simply that each output example is associated with a subset of its contributing tuples. In general, this means that without looking at the full input database or at least its schema, we gain very little information (e.g. we are not guaranteed that the given annotations come from a relation that is projected to the head). In particular it is straightforward to show a reduction from the classic reverse engineering query problem, which is intractable by [42].

Acknowledgments. This research has been funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 804302). The contribution of Amir Gilad is part of a Ph.D. thesis research conducted at Tel Aviv University.

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases.* Addison-Wesley.
[2] Azza Abouzied, Dana Angluin, Christos H. Papadimitriou, Joseph M. Hellerstein, and Avi Silberschatz. 2013. Learning and verifying quantified boolean queries by example. In *PODS.* 49–60.
[3] Azza Abouzied, Joseph M. Hellerstein, and Avi Silberschatz. 2012. Playful Query Specification with DataPlay. *PVLDB* 5, 12 (2012), 1938–1941.
[4] Efrat Abramovitz, Daniel Deutch, and Amir Gilad. 2018. Interactive Inference of SPARQL Queries Using Provenance. In *ICDE.* 581–592.
[5] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications. In *SIGMOD.* 94–105.
[6] Tarun Arora et al. 1993. Explaining Program Execution in Deductive Systems. In *DOOD.* 101–119.
[7] Pablo Barceló and Miguel Romero. 2017. The Complexity of Reverse Engineering Problems for Conjunctive Queries. In *ICDT.* 7:1–7:17.
[8] Nicole Bidoit, Melanie Herschel, and Katerina Tzompanaki. 2014. Query-Based Why-Not Provenance with NedExplain. In *EDBT.* 145–156.
[9] Angela Bonifati, Radu Ciucanu, Aurélien Lemay, and Slawek Staworko. 2014. A Paradigm for Learning Queries on Big Data. In *Data4U@VLDB.*
[10] Angela Bonifati, Radu Ciucanu, and Slawek Staworko. 2014. Interactive Join Query Inference with JIM. *PVLDB* 7, 13 (2014), 1541–1544.
[11] P. Buneman, J. Cheney, and S. Vansummeren. 2008. On the expressiveness of implicit provenance in query and update languages. *ACM Trans. Database Syst.* (2008), 28:1–28:47.
[12] P. Buneman, S. Khanna, and W.C. Tan. 2001. Why and Where: A Characterization of Data Provenance. In *ICDT.* 316–330.
[13] Artem Chebotko, Seunghan Chang, Shiyong Lu, Farshad Fotouhi, and Ping Yang. 2008. Scientific Workflow Provenance Querying with Security Views. In *WAIM.* 349–356.
[14] James Cheney. 2011. A Formal Framework for Provenance Security. In *CSF.* 281–293.
[15] J. Cheney, L. Chiticariu, and W. C. Tan. 2009. Provenance in Databases: Why, How, and Where. *Foundations and Trends in Databases* (2009), 379–474.
[16] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Trans. Database Syst.* (2000), 179–227.
[17] Anish Das Sarma, Aditya Parameswaran, Hector Garcia-Molina, and Jennifer Widom. 2010. Synthesizing View Definitions from Data *(ICDT).* 89–103.
[18] Susan B. Davidson, Sanjeev Khanna, Tova Milo, Debmalya Panigrahi, and Sudeepa Roy. 2011. Provenance views for module privacy. In *PODS.* 175–186.
[19] Susan B. Davidson, Sanjeev Khanna, Sudeepa Roy, Julia Stoyanovich, Val Tannen, and Yi Chen. 2011. On provenance and privacy. In *ICDT.* 3–10.
[20] Susan B. Davidson, Sanjeev Khanna, Val Tannen, Sudeepa Roy, Yi Chen, Tova Milo, and Julia Stoyanovich. 2011. Enabling Privacy in Provenance-Aware Workflow Systems. In *CIDR.* 215–218.
[21] Daniel Deutch, Nave Frost, and Amir Gilad. 2017. Provenance for Natural Language Queries. *PVLDB* 10, 5 (2017), 577–588.
[22] D. Deutch and A. Gilad. 2018. Full Version. http://www.cs.tau.ac.il/~amirgilad/full/RevEngFull.pdf. (2018).
[23] Daniel Deutch, Amir Gilad, and Yuval Moskovitch. 2015. Selective Provenance for Datalog Programs Using Top-K Queries. *PVLDB* 8, 12 (2015), 1394–1405.
[24] Kyriaki Dimitriadou, Olga Papaemmanouil, and Yanlei Diao. 2014. Explore-by-example: An Automatic Query Steering Framework for Interactive Data Exploration. In *SIGMOD.* 517–528.
[25] R. Fink, L. Han, and D. Olteanu. 2012. Aggregation in Probabilistic Databases via Knowledge Compilation. *PVLDB* 5, 5 (2012), 490–501.
[26] F. Geerts and A. Poggi. 2010. On database query languages for K-relations. *J. Applied Logic* (2010), 173–185.
[27] Yolanda Gil and Christian Fritz. 2010. Reasoning about the Appropriate Use of Private Data through Computational Workflows. In *AAAI.*
[28] B. Glavic, J. Siddique, P. Andritsos, and R. J. Miller. 2013. Provenance for Data Mining. In *TaPP.*
[29] T. J. Green. 2009. Containment of conjunctive queries on annotated relations. In *ICDT.* 296–309.
[30] T. J. Green, G. Karvounarakis, and V. Tannen. 2007. Provenance semirings. In *PODS.* 31–40.
[31] Garcia-Molina Hector, Jeffrey D Ullman, and Jennifer Widom. 2002. *Database systems: The complete book.* Prentice-Hall.
[32] Tomasz Imieliński and Witold Lipski, Jr. 1984. Incomplete Information in Relational Databases. *J. ACM* (1984), 761–791.
[33] Dmitri V. Kalashnikov, Laks V.S. Lakshmanan, and Divesh Srivastava. 2018. FastQRE: Fast Query Reverse Engineering. In *SIGMOD.* 337–350.
[34] Davide Mottin, Matteo Lissandrini, Yannis Velegrakis, and Themis Palpanas. 2014. Exemplar Queries: Give Me an Example of What You Need. *PVLDB* 7, 5, 365–376.
[35] Fotis Psallidas, Bolin Ding, Kaushik Chakrabarti, and Surajit Chaudhuri. 2015. S4: Top-k Spreadsheet-Style Search for Query Discovery *(SIGMOD).* 2001–2016.
[36] Anish Das Sarma, Martin Theobald, and Jennifer Widom. 2008. Exploiting Lineage for Confidence Computation in Uncertain and Probabilistic Databases. In *ICDE.* 1023–1032.
[37] Thibault Sellam and Martin L. Kersten. 2013. Meet Charles, big data query advisor *(CIDR).*
[38] Yanyan Shen, Kaushik Chakrabarti, Surajit Chaudhuri, Bolin Ding, and Lev Novik. 2014. Discovering Queries Based on Example Tuples. In *SIGMOD.* 493–504.
[39] Wang Chiew Tan. 2003. Containment of Relational Queries with Annotation Propagation. In *DBPL.* 37–53.
[40] TPC. [n. d.]. TPC benchmarks. ([n. d.]). http://www.tpc.org/
[41] Quoc Trung Tran, Chee-Yong Chan, and Srinivasan Parthasarathy. 2014. Query Reverse Engineering. *The VLDB Journal* 23, 5 (2014), 721–746.
[42] Yaacov Y. Weiss and Sara Cohen. 2017. Reverse Engineering SPJ-Queries from Examples. In *PODS.* 151–166.
[43] Meihui Zhang, Hazem Elmeleegy, Cecilia M. Procopiuc, and Divesh Srivastava. 2014. Reverse Engineering Complex Join Queries. In *SIGMOD.* 809–820.
[44] Moshé M. Zloof. 1975. Query by Example. In *AFIPS NCC.* 431–438.