Contribution Maximization in Probabilistic Datalog

Tova Milo
Tel Aviv University
Tel Aviv, Israel
milo@post.tau.ac.il

Yuval Moskovitch
Tel Aviv University
Tel Aviv, Israel
moskovitch1@post.tau.ac.il

Brit Youngmann
Tel Aviv University
Tel Aviv, Israel
brity@mail.tau.ac.il

Abstract—**The use of probabilistic datalog programs has been recently advocated for applications that involve recursive computation and uncertainty. While using such programs allows for a flexible knowledge derivation, it makes the analysis of query results a challenging task. Particularly, given a set O of output tuples and a number k, one would like to understand which k-size subset of the input tuples have *contributed the most* to the derivation of O. This is useful for multiple tasks, such as identifying the critical sources of errors and understanding surprising results. Previous works have mainly focused on the quantification of tuples contribution to a query result in *non-recursive* SQL queries, very often *disregarding probabilistic inference*. To quantify the contribution in probabilistic datalog programs, one must account for the recursive relations between input and output data, and the uncertainty. To this end, we formalize the Contribution Maximization (CM) problem. We then reduce CM to the well-studied Influence Maximization (IM) problem, showing that we can harness techniques developed for IM to our setting. However, we show that such naïve adoption results in poor performance. To overcome this, we propose an optimized algorithm which injects a refined variant of the classic Magic Sets technique, integrated with a sampling method, into IM algorithms, achieving a significant saving of space and execution time. Our experiments demonstrate the effectiveness of our algorithm, even where the naïve approach is infeasible.**

I. Introduction

Real-life applications often rely on an underlying database in their operation. While many of these applications employ conventional SQL as their query language, the use of *probabilistic datalog* has been recently advocated for applications that involve recursive computation and uncertainty [1], [2], [3], [4]. To illustrate, consider AMIE [2], an information extraction system that mines logical rules from Knowledge Bases (e.g., YAGO [5]), based on correlations in the data. The mined rules are then treated as a datalog program, which may be evaluated w.r.t. the knowledge base, e.g., to address data incompleteness and derive new facts. Since the program rules are automatically mined, there is an inherent uncertainty w.r.t. their validity. AMIE thus associates corresponding probabilities to the rules, yielding a probabilistic datalog program.

The use of probabilistic datalog allows for a flexible and expressive knowledge derivation, yet introduces intricate relationships between the data items. This makes the analysis of a query results a challenging task. In particular, given a set O of output tuples, one would like to understand which (bounded size) subset of the input tuples have *contributed the most* to the derivation of O. This is useful for multiple tasks, such as identifying the critical sources of errors and understanding surprising results [6], [7]. Acquiring information about the most influential tuples may also serve users in obtaining detailed explanations for output tuples using selective provenance tracking systems (e.g., [3]). These systems provide explanations based on user-defined patterns, however, defining the patterns may be challenging without prior knowledge on the tuples that have contributed the most to the results.

exports

Country	Product
France	wine
France	vinegar
France	oil
Cuba	tobacco
Cuba	sugar
Cuba	nickel
Russia	gas

imports

Country	Product
Germany	wine
USA	vinegar
Pakistan	oil
India	tobacco
Denmark	sugar
Iran	nickel
Ukraine	gas

dealsWith (edb copy)

Country	Country
France	Cuba

TABLE I: Example Database.

To illustrate the problem that we study in this paper consider the following example.

Example 1.1: We consider a small probabilistic datalog program consisting of 3 rules, mind by AMIE over YAGO[1]. A sample database instance is depicted in Table I. This program outputs a binary relation *dealsWith*, including information on international trade relationships.

$r_1(0.8)$ `dealsWith(a, b):- dealsWith(b, a)`
$r_2(0.7)$ `dealsWith(a, b):- exports(a, c),imports(b, c)`
$r_3(0.5)$ `dealsWith(a, b):- dealsWith(a, f), dealsWith(f, b)`

Focusing on a set of derived trade relations of interest (e.g., `dealsWith(USA, Iran)`, `dealsWith(Pakistan, India)` `dealsWith(Russia, Ukraine)`), one may wish to understand which database facts have led to this inference. Presenting all tuples that took part in the computation as an explanation may be cumbersome and not informative. For instance, nearly 36% of the database tuples are used to derive solely the fact that USA trades with Iran. Thus, it is of great importance to be able to focus on a *small set* of facts that have *contributed the most* to this inference.

To facilitate such an analysis, one needs first to formally *quantify the notion of contribution*, while considering the following three issues: First, as opposed to the simple SQL setting, the *recursive relationship* between input and output data, and the *uncertainty* induced by the rules' probabilities are need to be taken into account. For instance, in the above recursive program, the rules' probabilities (in parenthesis) model the fact that transitive trade relations are considered less trustworthy (likely to happen) than direct relations. Second, note that selecting the top k tuples with the highest individual contribution is not the same as finding a k-size set with highest contribution, as two input tuples may contribute to exactly the same derived facts. Therefore, one must consider the *joint* contribution of a set of input tuples to a set of output

[1]This program is a strict subset of the actual set of rules generated by AMIE

tuples. As we demonstrate, our definition, which considers such joint contribution of sets of tuples, ensures a greater coverage of the output tuples of interest. Last, the rules' probabilities are independent, and so are the instantiations of each rule. Thus, the *independence* of the choice to fire a rule instantiation or not, from other rule firing choices, needs to be considered. For instance, among many other tuples, the tuple t_1 = `dealsWith(France, Cuba)` takes part in the derivation of t_2 = `dealsWith(USA, Iran)`. To properly asses its own contribution to t_2, we focus in our definition on the marginal contribution of t_1 (or more generally, of a k-size set of tuples), regardless of other parts in the derivation.

While the quantification of tuples contribution to a query results have been studied in the literature (e.g., [8], [6], [7], [9]), previous works have mainly focused on *non-recursive* SQL queries, very often *disregarding probabilistic inference*. Furthermore, a key difference is that we aim to quantify the marginal contribution of input tuple(s) to output tuple(s), independently of other derivations in the program, whereas previous works have focused on quantifying the contribution of a *single* database tuple to a *single* output tuple, focusing on dependencies. See further details in the Related Work Section.

Interestingly, an analogous problem of selecting influential k-size items set was studied in the context of social networks analysis. Particularly, the classic Influence Maximization (IM) problem [10] is the problem of finding a k-size set of users in a social network (called a seed-set), so that their aggregated influence (which may propagate transitively in the network) on other users is maximized. Reducing our problem to this well-studied problem allows to harness techniques for IM to our setting. However, as we show, there are several challenges that need to be overcome to facilitate an efficient execution.

We next outline our main contributions.

Problem formulation: It is common to describe the process of (probabilistic) datalog evaluation through the notion of derivation trees. A derivation tree of a tuple t specifies the rule instantiations and intermediate facts jointly used in the gradual process of deriving t. However, multiple derivation trees of the same or different tuples may share common parts. To simultaneously capture all trees, we merge them into a single Weighted Derivation (WD) graph, where edge weights record the corresponding derivation probabilities. This graph essentially captures the provenance of the derived tuples [11], [3], [12]. Intuitively, to quantify the contribution of the derivations involving a set of databases tuples T to a set of output tuples O, *independently of other derivations in the graph*, we consider a random draw of fire-or-not for all rules derivations (edges) in the WD graph. We measure the contribution as the expected number of output tuples from O reachable from the tuples in T in such a randomly generated subgraph. As we explain in Section III-B, this captures the desired properties expected from a contribution function as described above. We then define the Contribution Maximization (CM) problem as the problem of finding, among a set of database tuples of interest T, a k-size subset with the highest contribution to the target set of output tuples O (see Section III).

Algorithms: **A naïve approach**. We show that our contribution function matches nicely the influence function from the classic *Influence Maximization* (IM) problem [10]. Indeed, CM can be formulated as a variant of IM, and therefore top performing IM algorithms can be adjusted to solve our problem. A naïve solution for the CM problem is to first fully materialize the WD graph, then to run an (adjusted) IM algorithm over it, finding a k-size set of tuples with the maximal contribution. However, while this naïve algorithm provides, in expectation, the optimal approximation for the CM problem, the WD graph may be prohibitively large, and thus this approach is impractical.

An optimized algorithm. We, therefore, devise an optimized algorithm that avoids the materialization of the full WD graph (while maintaining the same optimal approximation factor), and allows for a significant saving in both memory and running time. Specifically, this algorithm takes as an input an IM algorithm and employs two optimizations. The first harnesses principles from the classical Magic-Sets transformation [13], to construct only the graph parts that are essential for the computation of the (adjusted) IM algorithm. Intuitively, we rewrite the input program so that we can materialize, on-the-fly, only a small subgraph of the WD graph that is required for the computation. The key challenge is to assign probabilities to the rules of the transformed program s.t. its corresponding WD graph is essentially isomorphic to the desired subgraph of the original WD graph. We further refine the above process, allowing for additional space and runtime saving: We bundle the Magic-Sets graph construction with the sampling process employed by the IM algorithm, to further reduce the size of the materialized subgraphs. The key challenge here is to inject the IM sampling into the Magic-Sets graph construction, yielding a subgraph that is still essentially isomorphic to the relevant portion of the original WD graph (see Section IV).

Experimental study: We complement our algorithmic development with an experimental study that evaluates the runtime performance and memory consumption of our algorithm. We experimentally compare the algorithm performance in multiple real-life scenarios, highlighting the savings and trade-offs achieved by each proposed optimization. Our experimental results indicate the effectiveness of our algorithm, even where the naïve algorithm is infeasible. We conclude with a case study, demonstrating both the quality results of our contribution measure, as well as the quality of approximation of the proposed algorithms (see Section V).

Related work is presented in Section VI and we conclude in Section VII. For space constraints, all proofs are deferred to our technical report [14].

II. Preliminaries

A. Probabilistic Datalog

We assume that the reader is familiar with standard (probabilistic) datalog concepts [13]. Here we review the terminology and illustrate it using Example 1.1 from the Introduction, which will serve as our running example throughout the paper.

Datalog programs: A *datalog program* is a finite set of datalog *rules*. A datalog rule is an expression of the form $R_1(\mathbf{u_1})$:- $R_2(\mathbf{u_2}), \ldots, R_n(\mathbf{u_n})$ where R_i's are relation names, and $\mathbf{u_1}, ..., \mathbf{u_n}$ are sets of variables with appropriate arities. $R_1(\mathbf{u_1})$ is called the rule's head, and $R_2(\mathbf{u_2}), \ldots, R_n(\mathbf{u_n})$ is the rule's body. Every variable occurring in $\mathbf{u_1}$ must occur in at least one of $\mathbf{u_2}, \ldots, \mathbf{u_n}$. We distinguish between extensional (edb) and intentional (idb) database relations. The former are the input database relations while the latter are derived by the program and appear in the rule heads. We use interchangeably the words fact and tuple for the edb/idb relations. A datalog program is then a mapping from edb instances to idb instances, whose semantics may be defined via the notion of the *consequence operator*. First, the *immediate* consequence operator induced by a program P maps a database instance D to an instance $D \bigcup \{A\}$ if there exists an *instantiation* of some rule in P (i.e., a consistent replacement of variables occurring in the rule with constants) s.t. the body of the instantiated rule includes only atoms in D and the head of the instantiated rule is A. Then the consequence operator is defined as the *transitive closure* of the immediate consequence operator, i.e. the fixpoint of the repeated application of the immediate consequence operator. Given a database D and a program P we use $P(D)$ to denote the restriction to idb relations obtained by applying to D the consequence operator induced by P. We use $r(inst)$ to denote an instantiation of a rule r.

It is common to describe the process of datalog evaluation through the notion of derivation trees. A derivation tree of a tuple t w.r.t. a datalog program P and a database D is a finite tree whose root is labeled by t, the leaves are labeled by edbs, and the internal nodes by idb facts. The tree structure is dictated by the consequence operator of the program: the labels set of the children of a node n corresponds to an instantiation of the body of some rule r, s.t. the label of n is the corresponding instantiation of r's head. To make clear which instantiated rule is used, we use a refined variant of the traditional derivation tree, where we add an auxiliary node labeled $r(inst)$ between each derived idb tuple and the tuples in the rule's body.

Example 2.1: Consider the datalog program depicted in Example 1.1 (ignore, for now, the numbers in parenthesis), which outputs the binary relation *dealsWith* (an edb "copy" of this relation appears as well, with rules for copying its content that are omitted for brevity). To illustrate the derivation process, consider again the database shown in Table I. Two possible derivations of idb facts, in a form of derivation trees, are depicted in the gray and blue areas in Figure 1. The idb tuples are colored in red, the edb tuples are colored in yellow, and the auxiliary rule instantiations nodes are colored in purple (also ignore, for now, the edge weights).

Probabilistic Datalog program: To model scenarios with uncertainty, it is common to associate probabilities to edb tuples and program rules, capturing the confidence in the correctness of the tuples/rules [2], [15]. Since probabilities on edb tuples can be modeled by probabilistic rules[2], for simplicity, we consider here only rules probabilities.

Let (P, w) denote a probabilistic datalog program, where P is the set of rules and w is the probabilities function associating each rule r with a probability $w(r)$. We assume that the interpretation of the probability associated with a rule is that a rule instantiation is trusted with probability of $w(r)$. Note that this interpretation matches the definition of rules confidence values introduced in [2], where the rules weights are based on the correlations in the database.

[2]E.g., by simply making each edb fact a rule (that copies the original fact into an auxiliary replica relation to be used by the program), with the corresponding probability.

The semantics of a probabilistic datalog program. In the execution of a probabilistic datalog program, each rule instantiation is drawn to fire in a probability that corresponds to the rule's probability $w(r)$. That is, the derivation process in a probabilistic datalog program is the same as the derivation process in a regular datalog program, except that for each rule instantiation *that has not been considered yet*, we draw if it should be fired or not according to the rule's probability. Only instantiations for which the result of the draw was positive are fired. That is, the semantics of a probabilistic datalog program is also defined via the notion of the consequence operator, while each rule r instantiation occur with probability of $w(r)$. The process halts where no new facts were derived. Note that here there is no single fixpoint, as each run of the probabilistic program may derive a different set of facts. Therefore, the semantics of probabilistic datalog programs assigns a probability to each idb fact in $P(D)$, capturing its likelihood to be derived in a random program execution.

The derivation trees have the same shape as before, except that we now also annotate the outgoing edges of the rule-instantiation nodes with their firing probability. In what follows we will overload the notation $P(D)$ to denote also all the idb facts that can be derived by some probabilistic execution of the program. Note that $P(D)$ includes the same set of facts derived in the regular (non probabilistic) execution of the program.

Example 2.2: Consider again the probabilistic program presented in Example 1.1. Each rule mind by AMIE is associated with a probability, capturing the fact that the rules' validity is questionable. The rule weights are represented in the derivation trees by corresponding edge weights, as depicted in Figure 1. Observe that a random execution of the program will generate the fact `dealsWith(Pakistan,India)` (resp. `dealsWith(USA,Iran)`), if all rules instantiation in the blue (resp. gray) area (i.e., derivation tree) were drawn to fire.

B. Influence Maximization

As mentioned, we will employ concepts from Influence Maximization (IM) for our Contribution Maximization (CM) problem. Here we review the classic IM problem and its state-of-the-art algorithm scheme.

Let $\mathcal{G} = (V, E, W)$ be a weighted directed graph, where V is the set of nodes and each edge $(u, v) \in E$ is associated with a weight $0 \leq W(u, v) \leq 1$, which models the probability that a node u will *influence* its neighbor v. Given a function $I(\cdot)$ dictating how influence is propagated in the graph and a number $k \leq |V|$, the goal is to find a k-size node set with maximal expected influence. Formally,

Definition 2.3 (IM [10]): Given a graph $\mathcal{G}$ and a number $k \leq |V|$, let OPT_k denote the maximum expected influence spread of any k-size node set S, i.e. $OPT_k = max_{S \subseteq V, |S|=k} E[I(S)]$. The IM problem finds an optimal k-size seed set S s.t $E[I(S)] = OPT_k$.

The function $I(\cdot)$ is defined by the influence propagation model. The majority of existing IM algorithms apply for the two most researched Information Cascade (IC) model [10], [16]. In the IC model the propagation is carried out in discrete steps, s.t. each node influenced in the preceding step attempts to influence its uninfluenced neighbors, with an independent probability indicated by the edge weights.

Subgraphs generation. The propagation process under the IC model is modeled using sampled subgraphs, which are used to estimate the influence of a seed-set, and are constructed as follows. Independently, for each edge (u, v) with the weight $W(u, v)$ we flip a coin indicating whether this edge is active or not. The edges in G for which the coin flip indicated an activation will be successful are declared to be live; the remaining edges are declared to be blocked. If we fix the outcomes of the coin flips and then initially activate a seed-set S, a node $v \in V$ ends up influenced if there is a path from some node in S to v consisting entirely of live edges.

The IM problem is hard to approximate beyond a factor of $(1 - \frac{1}{e})$ [10]. In the sequel, whenever we refer to an IM algorithm, we in fact refer to a probabilistic algorithm, which, given the input parameters $0 \leq \epsilon, \delta \leq 1$, achieves, with probability $\geq (1 - \delta)$, the optimal approximation factor up to an additive error of ϵ. To ease the presentation, we omit the discussion of ϵ and δ whenever possible.

The RIS framework: State-of-the-art IM algorithms are based on the Reverse Influence Sampling (RIS) framework [16], which reduces the problem to the classic *Maximum Coverage* (MC) problem [17]. Formally, given a graph $\mathcal{G} = (V, E, W)$, RIS captures the influence of a given seed-set by generating a set of random Reverse Reachable (*RR*) sets, where each *RR* set is a subset of V and constructed as follows. Given $\mathcal{G}$, a random *RR* set is generated from $\mathcal{G}$ by (1) selecting a random node $v \in V$ (2) generating a sample graph g from $\mathcal{G}$ (as described above) and (3) returning the set of nodes that can reach v in g. The RIS framework consists of two phases: First, nodes are sampled independently and uniformly, then, for each sampled node u, we construct its *RR* set. Next, each node is associated with a set whose members are the *RR* sets containing it, then, using the greedy algorithm for MC [17], k nodes are selected. Top performing IM algorithms are based on this scheme, with [18] achieving nearly optimal time complexity of $\tilde{\Theta}(k \cdot (|V| + |E|))$. The main difference between existing RIS IM algorithms is the number of generated *RR* sets θ. Generally, these algorithms dynamically decide how many *RR* sets to generate during their run, where the number of sets is a function of the graph size, the required error rate (ϵ) and the success probability (δ) [19], [20].

III. Problem Definition

A. The Weighted Derivation (WD) Graph

Given a probabilistic datalog program, consider all its possible executions, and recall that in each such execution every instantiation of a rule $r(inst)$ is drawn to fire in a probability that corresponds to the rule's probability $w(r)$. We next devise a directed weighted graph that captures all these possible executions, referred to as the Weighted Derivation (WD) graph. Intuitively, this graph integrates all derivation trees of the program, merging their common parts.

Definition 3.1: Let (P, w) be a probabilistic datalog program and D be a database instance. The WD graph of (P, w) and D is a directed weighted graph $\mathcal{G} = (V, E, W)$ where:

Nodes V consists of a distinct node per each edb in D, each idb in $P(D)$, and each rule instantiation $r(inst)$.

Edges E is a set of edges where for every rule instantiation $r(inst) = h\text{:-}b_1, \ldots, b_n$, the node $r(inst)$ has n incoming edges $(b_i, r(inst))$, $i = 1 \ldots n$, from the edb/idb facts in its body, as well as one outgoing edge $(r(inst), h)$ to the idb fact h in its head.

Weights W is a weight function assigning weights to the graph edges s.t. each edge $(r(inst), h)$ outgoing a rule instantiation is assigned with the rule's weight $w(r)$, and all other edges have a weight of 1.

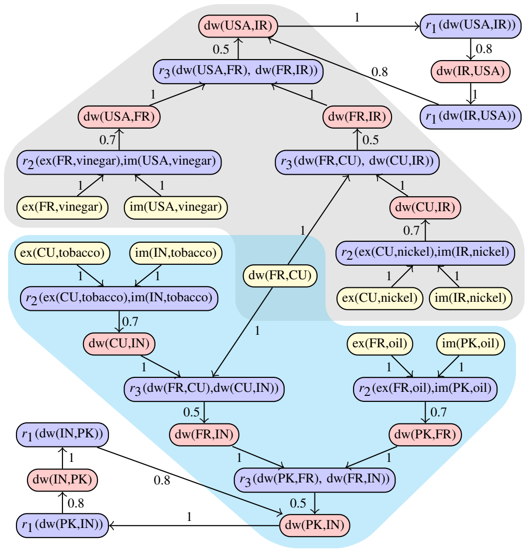


Fig. 1: A partial WD graph

Example 3.2: Consider again the probabilistic datalog program given in Example 1.1, and the database instance shown in Table I. Figure 1 depicts a partial WD graph. Observe that the two derivation trees had been merged into a single graph.

We can show that the size of the WD graph is polynomial in the database size (see [14] for proof).

Proposition 3.3: The size of the WD graph of a database D and probabilistic datalog program (P, w) is polynomial in D (with the exponent depending on the size of P).

However, in our experimental study we demonstrate an empirical evidence for its practical blowup (see Section V). Thus, one of our goals is to avoid fully constructing the WD graph.

B. Problem Formulation

As discussed in the Introduction, to properly quantify the notion of contribution, one must consider: (1) The *recursive relationship* between input and output data, and the *uncertainty* induced by the rules' probabilities; (2) The *joint* contribution of a set of input tuples to a set of output tuples; (3) the *independence* of the choice to fire a rule instantiation or not, from other rule firing choices.

Intuitively, a tuple $t_1 \in D$ contributes to the derivation of a tuple $t_2 \in P(D)$ if t_1 is a part of (at least) one of the derivation trees of t_2. In other words, the WD graph contains some directed path(s) from t_1 to t_2. The probabilities of the rules instantiations along the path(s) capture the potential involvement of t_1 in this derivation. Analogously, when considering a set T_1 of input tuples and a set T_2 of output tuples, the weights along the paths in the WD graph from nodes in T_1 to nodes in T_2, capture the involvement of the former tuples in the derivation of the latter. To quantify the joint contribution of tuples in T_1

independently of other facts that participated in the derivations, we consider a subgraph generated from the WD graph by drawing all edges, independently at random, with probability corresponding to their respective weights (as described in Section II-B). Note that such a random subgraph represents a random execution of the probabilistic program, independently flipping a coin to decide if each rule instantiation was fired or not. We measure the contribution of T_1 to T_2 as the expected number of nodes in T_2 reachable for nodes in T_1 in a randomly generated such subgraph.

Definition 3.4 (Tuple Sets Contribution): Given the WD graph $\mathcal{G}$ of a database D and a probabilistic program (P, w), let g be a random subgraph generated from $\mathcal{G}$. We define the contribution of a set $T_1 \subseteq D$ to a set $T_2 \subseteq P(D)$, denoted $c(T_1 \rightsquigarrow T_2)$, as the expected number of nodes in T_2 reachable from nodes in T_1 in the subgraph g.

Observe that this definition captures the desired properties. The recursive relations and the uncertainty are captured via the WD graph. The more paths from T_1 to T_2 that the WD graph contains and the higher the probabilities of rules they include, the greater is the contribution of T_1 to T_2. We capture the joint contribution of tuples in T_1 to tuples in T_2 by considering the expected number of nodes in T_2 reachable from nodes in T_1, in a random subgraph g, i.e., a random program execution. To account for the independence of the rules' probabilities and rules instantiations, we measure the involvement of tuples in T_1 in the derivation of tuples in T_2, independently of other parts of the derivations, thereby, capturing the marginal contribution of T_1 to T_2. We further demonstrate the quality of our measure to capture tuples contribution via a case study in Section V-C.

Example 3.5: Continuing with our example, let T_1 = {`dealsWith(France,Cube)`, `exports(France,vineger)`}, and T_2 = {`dealsWith(USA,Iran)`, `delasWith(Pakistan, India)`}. As can be seen in Figure 1, the tuple `dealsWith(France,Cuba)` contributes to both tuples in T_2, as it is a part of the derivations of both idbs. In contrast, the tuple `exports(France, vineger)` is a part of the derivation of only `dealsWith(USA, Iran)`. Indeed, the contribution scores of the each tuple {`dealsWith(France,Cube)`, `exports(France,vineger)`} to T_2 are $\approx$ 0.5 and 0.35, resp. Note that to assess the marginal contribution of `exports(France, vineger)` to `dealsWith(USA, Iran)`, we consider only paths connecting these tuples, ignoring other parts of the derivation (e.g., we ignore the part corresponding to `dealsWith(France,Cuba)`). Finally, the contribution score of T_1 to T_2 is ≈ 0.6. This score is lower than the sum of the separated contribution scores because of the shared sub-paths.

C. The Contribution Maximization (CM) problem

We are now ready to define the CM problem. We conclude with a complexity analysis of our problem.

Definition 3.6 (CM): Given a probabilistic program (P, w), a database D, a set $T_1 \subseteq D$, a set $T_2 \subseteq P(D)$, and a natural number $k \leq |D|$, let OPT_k denote the maximum expected contribution of any k-size set to the set T_2, i.e. $OPT_k = max_{S \subseteq T_1, |S|=k} E[c(S \rightsquigarrow T_2)]$, where $E[c(S \rightsquigarrow T_2)]$ is the expected contribution of S to T_2. The CM problem is to find an optimal k-size set $S' \subseteq T_1$ such that $E[c(S' \rightsquigarrow T_2)] = OPT_k$.

Example 3.7: The derived facts in Example 1.1 include some surprising results: t_1 ={`dealsWith(USA,Iran)`,t_2 =`delasWith(Pakistan, India)`, and t_3 = `delasWith(Russia, Ukraine)`}. Over 73% of the database tuples are taking part in the derivation of these facts. To find a small set of edbs contributing the most to the derivation of these facts, one may set $T_1 = D$, and $T_2 = \{t_1, t_2, t_3\}$. For brevity, let $k = 2$. Observe that each edb except of t=`dealsWith(France,Cuba)` contributes to the derivation of a single idb, while t is involved in the derivation of both t_1 and t_2. Intuitively, a 2-size set with the highest contribution to T_2 should contain at least one edb that contributes to each idb in T_2. Indeed, a set containing t and one tuple contributing to t_3 (e.g., `exports(Russia, gas)`) yields the maximal contribution score. In contrast, selecting the top-2 edbs with the highest individual contribution scores may results in selecting t and any other tuple that also contributes to either t_1 or t_2 (e.g., `exports(France, oil)`), hence with overall smaller joint contribution.

At this point we note that our problem can be seen as a special case of the classic IM problem. Indeed, as we explain in Section IV-A, our contribution function matches nicely the influence function with IC as the propagation model. Our problem can thus be formulated as a variant of IM with the following restrictions: (i) the selected seed nodes must belong to the set T_1, and (ii) the only target nodes that we count are those belong to T_2. Therefore, as we explain in Section IV-A, existing IM algorithms can be adjusted to solve our problem. A naïve approach would be to first build the WD graph, then to run the adjusted IM algorithm over it, finding a k-size set of tuples with the highest contribution. However, as we shall see, such a naïve approach is highly inefficient.

Complexity Analysis: We show that it is NP-hard to determine the optimum for CM, and the optimum value is hard to approximate beyond a constant factor, even if the program contains a single rule. As in IM, we prove so by reducing from the classic *Set Cover* problem. Nonetheless, the key difference is that the complexity of IM is measured in terms of the graph size (which is given as part of the input), whereas in our case the complexity is measured in terms of the input database size (and the graph is not provided). The proof is available in [14].

Theorem 3.8: It is NP-hard in the database size to determine the optimum for CM, and the optimum is hard to approximate beyond a factor of $(1 - \frac{1}{e})$.

IV. Algorithms

We next describe the naïve algorithm, called NaïveCM, then present our optimized algorithm, named $Magic^S$CM.

A. The NaïveCM Algorithm

We begin by providing a naïve procedure for constructing the WD graph, then explain the required adjustments for an RIS-based IM algorithm to solve CM.

Algorithm 1 depicts a naïve procedure for incrementally building the WD graph. Similarly to the semi-naïve algorithm for datalog evaluation [13], it builds the graph by iteratively applying the rules of a given program (P, w) on the facts derived on the previous iteration. Specifically, first the algorithm initializes the node set with all edb facts $t \in D$ (lines 1–2).

Algorithm 1: Building the WD graph.

```
input : A probabilistic datalog program $(P, w)$ and a
        database $D$.
output: The WD graph $\mathcal{G}$ of $(P, w)$ and $D$.
1  foreach edb tuple $t \in D$ do
2      Add a new node $t$ to $V$
3  while $\mathcal{G} = (V, E, W)$ changes do
4      foreach rule $r \in P$ do
5          foreach instantiation $h$:-$b_0, \ldots, b_n$ of $r$ s.t.
           $\forall 0 \le i \le n\ b_i \in V$ do
6              if $h \notin V$ then
7                  $V \leftarrow V \cup \{h\}$
8              if $\nexists r(inst)$-node $v \in V$ s.t
               $\forall 0 \le i \le n (b_i, v) \in E$ and $(v, h) \in E$ then
9                  Add an $r(inst)$-node $v$ to $V$
10                 $E \leftarrow E \cup \{(b_i, v) \mid 0 \le i \le n\} \cup \{v, h\}$
11                 $\forall i \in [0, n] : W(b_i, v) = 1,\ W(v, h) = w(r)$
12 return $\mathcal{G} = (V, E, W)$
```

Then, while new facts and derivations are discovered (line 3), for every rule $r \in P$, and every instantiation h:-$b_0, \ldots, b_n$ of r such that all the body atoms are in V (lines 4–5), the algorithm adds h to V if h is a new idb (lines 6–7), and otherwise uses the existing node. Then it adds the corresponding $r(inst)$-node and edges (lines 8–10) to E, if not there already. The incoming edges of the $r(inst)$-node are set to be of weight 1, and its outgoing edge's weight is equal to the weight of the rule $w(r)$ (line 11). The algorithm terminates when no new nodes or edges are added.It is easy to show that Algorithm 1 always halts and computes the WD graph correctly. As it follows similar lines as the semi-naive algorithm for datalog evaluation [13], its (polynomial) time complexity analysis is omitted.

To employ an existing IM algorithm over the WD graph, we need to adjust it to comply to the following restrictions: (i) all selected seed nodes belong to T_1, and (ii) instead of maximizing the contribution to all nodes, we maximize the contribution to only nodes in T_2. Given an IM algorithm $\mathcal{A}$, a set $T_1 \subseteq D$, and a set $T_2 \subseteq P(D)$, we define $\mathcal{A}_{T_2}^{T_1}$ as its analogous algorithm that chooses only seed nodes from T_1 and maximizes the contribution only to T_2. Any such algorithm $\mathcal{A}$ can be adapted to $\mathcal{A}_{T_2}^{T_1}$ via two modification. First, instead of including all nodes that are reachable from the sampled starting node in its corresponding *RR* set, the *RR* set contains only reachable nodes from T_1. Second, following a previous work on *targeted IM* [21] (where the goal is to maximize influence over a targeted group of users), the *RR* sets are generated from nodes in T_2. We can prove that $\mathcal{A}_{T_2}^{T_1}$ outputs a seed set contributing to at least $(1 - \frac{1}{e})$-fraction from the optimal solution (which is optimal [10]), while maintaining the same computational complexity as $\mathcal{A}$ (see [14]).

The complete NaïveCM (shown in Algorithm 2) operates as follows. Given a program (P, w), a database D, a set $T_1 \subseteq D$, a set $T_2 \subseteq P(D)$, and a number $k \le |D|$, it first computes the WD graph $\mathcal{G} = (V, E, W)$. Then, given an IM RIS-based algorithm $\mathcal{A}$, it applies the $\mathcal{A}_{T_2}^{T_1}(\mathcal{G}, k)$ algorithm as described above. Note that the "while" loop in line 2 depends on the input IM algorithm, which, as mentioned in Section II-B,

Algorithm 2: The NaïveCM algorithm.

```
input : A probabilistic datalog program $(P, w)$, a
        database $D$, a set $T_1 \subseteq D$, a set $T_2 \subseteq P(D)$, a
        number $k$, an IM algorithm $\mathcal{A}$, and the
        parameters $\delta$ and $\epsilon$.
output: An $(1 - \frac{1}{e} - \epsilon)$ optimal solution with
        probability $\ge (1 - \delta)$.
1  $\mathcal{G} \leftarrow$ build_wd_graph($(P, w), D$)
   Apply $\mathcal{A}_{T_2}^{T_1}$:
2  while Some condition over the approximation holds
   do
3      i. Generate a collection of RR sets from $\mathcal{G}$
4      ii. Use the greedy algorithm to find a $k$-size seed
        set $S$ that covers the maximum number of RR
        sets;
5  return $S$
```

dynamically decides how many *RR* sets to generate (depending on the required error rate ϵ and the success probability δ).

Proposition 4.1: The NaïveCM algorithm provides a $(1 - \frac{1}{e} - \epsilon)$-approximation to the CM problem, with probability $\ge (1 - \delta)$, where ϵ and δ are given as input parameters.

While the complexity of top performing IM algorithms is (nearly) linear in the graph size (as mentioned in Section II-B), the complexity of NaïveCM is dominated by the procedure for building the full WD graph, which is polynomial in D (Proposition 3.3). However, as indicated by our experiments, this naïve approach is impractical for real-life scenarios.

B. The Magic^{S}CM Algorithm

As mentioned in the Introduction, our optimized algorithm employs two optimizations. We next detail our optimizations, then provide the full *Magic^S*CM algorithm.

1) On-the-fly subgraph constriction: Recall that NaïveCM first constructs the full WD graph $\mathcal{G}$, then runs an (adjusted) IM algorithm. Also recall that the IM algorithm samples some idb nodes, then constructs for each sampled node its corresponding *RR* set, which consists of the edb nodes reachable from it (in a reversed walk). In our first optimization, rather then constructing the full WD graph $\mathcal{G}$, we build, on-the-fly, for each sampled idb tuple t, only the subgraph of $\mathcal{G}$ that is reachable from t. For that we consider, for each sampled tuple t, a top-down evaluation of the program P with the query $q = t$. We apply the Magic-Sets transformation to obtain a new program P_t^m that computes only the facts derived in this evaluation, then assign probabilities to the rules, obtaining a program (P_t^m, w_t^m) having the following property: For every tuple $t \in P(D)$ and every possible *RR* generated from t, its probability to be sampled in $\mathcal{G}$ equals to its probability to be sampled (for t^m) in the WD graph $\mathcal{G}^m$ generated using (P_t^m, w_t^m) and D. For brevity, in the following whenever t is clear from the context we omit it and simply use P^m and w^m. For completeness of this paper, we first provide a short overview of the classical Magic-Sets transformation technique, then explain how and why it works here.

Magic-Sets. In a (semi-)naïve datalog evaluation, the actual query (e.g., testing if a given tuple was derived) is considered only at the very end of program execution, after all idb facts where derived. The Magic-Sets transformation rewrites the program s.t. the rules can "fire" only when the derived idb fact is relevant for the query. This is done by making the notion of relevant facts explicit, encoding them as facts of new "magic predicates" that now "adorn" the original program rules. This transformation produces a program which is equivalent to the original program *for the given query*, but often the number of derived facts is much smaller than in the original program, and includes only idb tuples that are relevant for the given query.

Given a datalog program P and a query $\mathtt{Q(\mathbf{u})\text{:-}Q_1(\mathbf{u_1}),\ldots,Q_n(\mathbf{u_n})}$, the Magic-Sets transformation generates a program P^m as follows. For each idb relation name A, P^m contains adorned predicates A^β, and magic predicates m_A^β, where β is an adornment annotation, i.e., a string of b (for bound) and f (for free), with the same length as the arity of A, describing the binding pattern in a top-down evaluation. For each rule r, A:-$B_0, \ldots, B_m$ in P the following rules are generated:

$$A^{\beta_A}\text{:-}m\_A^{\beta_A}, B_0^{\beta_0}, \ldots, B_m^{\beta_m} \tag{1}$$

$$m\_B_i^{\beta_i}\text{:-}m\_A^{\beta_A}, B_0^{\beta_0}, ..., B_{i-1}^{\beta_{i-1}} \quad \forall \text{idb } B_i,\ i \in [0, m] \tag{2}$$

where $B_i^{\beta_i} = B_i$ is B if an edb relation. We call the rules of type (1) and (2) *modified rules* and *magic rules*, resp. We denote by $origin(r^m)$, the rule $r \in P$ from which r^m was generated. In addition, the rules $\mathtt{m\_Q^\beta(\mathbf{u})\text{:-}}$, and $\mathtt{Q^\beta(\mathbf{u})\text{:-}Q_1^{\beta_1}(\mathbf{u_1}),\ldots,Q_n^{\beta_n}(\mathbf{u_n})}$ are generated for the query rule. The first triggers the evaluation process, and the latter represents the query result. We refer to those rules as query rules. (for more details see [13]).

Example 4.2: Consider a simple probabilistic TC program, that computes the transitive closure of a directed graph.

(1.0)	r_1	`TC(X,Y):- E(X,Y)`
(0.8)	r_2	`TC(X,Y):- TC(X,Z), TC(Z,Y)`

Let a and b two nodes in the graph, represented by a database. The program generated by the Magic Sets transformation (after removing redundant predicates and rules) for the query `Q():-TC(a,b)` (a Boolean query that returns true if the fact $TC(a, b)$ is derived) is (ignore, for now, the rules wights):

(1.0)	m_1	$\mathtt{TC^{bb}(X,Y)\text{:- }m\_TC^{bb}(X,Y),E(X,Y)}$
(0.8)	m_2	$\mathtt{TC^{bb}(X,Y)\text{:- }m\_TC^{bb}(X,Y),TC^{bf}(X,Z),\ TC^{bb}(Z,Y)}$
(1)	m_3	$\mathtt{m\_TC^{bf}(X)\text{:- }m\_TC^{bb}(X,Y)}$
(1)	m_4	$\mathtt{m\_TC^{bb}(Z,Y)\text{:- }m\_TC^{bb}(X,Y),TC^{bf}(X,Z)}$
(1.0)	m_5	$\mathtt{TC^{bf}(X,Y)\text{:- }m\_TC^{bf}(X),E(X,Y)}$
(0.8)	m_6	$\mathtt{TC^{bf}(X,Y)\text{:- }m\_TC^{bf}(X),TC^{bf}(X,Z),\ TC^{bf}(Z,Y)}$
(1)	m_7	$\mathtt{m\_TC^{bf}(Z)\text{:- }m\_TC^{bf}(X),TC^{bf}(X,Z)}$
(1)	m_8	$\mathtt{Q()\text{:- }TC^{bb}}(a,\ b)$
(1)	m_9	$\mathtt{m\_TC^{bb}}(a,\ b)\mathtt{\text{:-}}$

where m_1, m_2, m_5 and m_6 are modified rules, m_8 and m_9 are query rules, and the rest are magic rules. $origin(m_i) = r_1$ for $i \in \{1, 5\}$, and $origin(m_i) = r_2$ for $i \in \{2, 3, 4, 6, 7\}$. Intuitively, the facts derived by the above program are facts that used in the derivation of $TC(a, b)$.

Employing Magic-Sets in our setting. Given a tuple $t \in P(D)$, let $P_t^m(D)$ be the program generated by the Magic-Sets transformation for the program P and the query `Q():-` t, and let $t^m \in P_t^m(D)$ be the tuple corresponding to t in $P_t^m(D)$ (i.e., the tuple t in the adorned predicate in $P_t^m(D)$, where all variables are bounded). We apply Algorithm 1 on $P_t^m(D)$ to generate a graph which is analogous (though not identical) to the subgraph of $\mathcal{G}$ which is reachable by a reversed walk from t^m. We note that while the program resulting form the transformation contains a larger number of rules and relations, the number of derived idbs is usually smaller (as typically, not all edbs and rules are being used to derive a given idb). We further show that ignoring the magic predicates, the graph generated for (P^m, w^m) is essentially isomorphic to the corresponding subgraph of (P, w), with the ancestor relationship (resp., weights of the outgoing edges) of the original $r(inst)$ nodes being identical to that of the corresponding modified adorned rules. In fact, the size of $\mathcal{G}^m$ may be reduced by not materializing the $r(inst)$-nodes of the magic rules instantiations. To see why the graphs are not identical, note that the derivation of a fact t^m, using the program P_t^m, also includes instantiations to new magic predicates, which are absent from the original program. Nevertheless, we can define the probabilities of the new rules of P_t^m so that the probability of each RR set to be sampled remains intact.

Definition 4.3: Given a probabilistic datalog program (P, w) and an idb tuple t, let P^m be the program generated from P by the above Magic-Sets transformation. The probabilistic transformed program (P^m, w^m) (for (P, w) and t) contains the rules of P^m with $w^m(r) = w(origin(r))$ for each modified rule r, and $w^m(r) = 1$ for all the magic and query rules.

We can prove that for every tuple $t \in P(D)$ and every RR set generated from t, its probability to be sampled in the WD graph, equals to its probability to be sampled (for t^m) in the WD graph generated using (P^m, w^m) and D (Proposition 4.4). This critical property allows us to sample RR sets on-the-fly, instead of materializing the full WD graph.

Proposition 4.4: Let $\mathcal{G}$ be the corresponding WD graph of a datalog program (P, w) and a database D. For every tuple $t \in P(D)$ and every possible RR set of edb fact of t, its probability to be sampled in $\mathcal{G}$ equals to its probability to be sampled (for t^m) in the WD graph $\mathcal{G}^m$ generated using (P^m, w^m) and D.

Putting it all together. We refer to NaïveCM enhanced with this optimization as the *Magic*CM algorithm. To avoid graph materialization, our adjusted IM-based algorithm $\mathcal{A}_{T_2}^{T_1}$, computes the RR set of each sampled idb tuple $t \in T_2$ using Algorithm 3. That is, Algorithm 3 applies the Magic-Sets transformation to obtain the transform program (P^m, w^m) (line 1), then, using Algorithm 1, generates the WD graph $\mathcal{G}^m$ of (P^m, w^m) and D (line 2). Finally, it generates a subgraph g from $\mathcal{G}^m$, and returns the set of edb nodes in T_1 from which t is reachable (lines 3–4), i.e., the desired RR set for t.

The differences between *Magic*CM and NaïveCM are: (1) the full WD graph is not generated (line 1 of Algorithm 2 is removed) and (2) each call of the IM algorithm for an RR set generation (line 3) is replaced by a call to Algorithm 3. Namely, the only difference between NaïveCM and *Magic*CM (the algorithm which employs only the Magic-sets optimization) is how they access the WD graph, and hence they both compute the same (approximated) solution to a given CM instance (as they both run an IM-based algorithm over the WD graph). We note that, in the worst case (i.e., where all edbs and rules are being used to derive every idb in T_2) *Magic*CM builds the full WD graph for every sampled idb. Hence, in the worst case, it could be equivalent of running NaïveCM θ times, where θ is the number of generated RR sets. Nonetheless,

Algorithm 3: On-the-fly *RR* set generation.

```
input : A probabilistic datalog program $(P, w)$, a
        database $D$, a set $T_1 \subseteq D$, and a tuple $t \in T_2$
output: RR set of $t$
1 $(P^m, w^m) \leftarrow$ magic_set($(P, w), t$)
2 $\mathcal{G}^m \leftarrow$ build_wd_graph($(P^m, w^m), D$)
3 Generate a sample graph $g$ from $\mathcal{G}^m$
4 return the set of nodes in $T_1$ from which $t$ is
    reachable in $g$
```

as our experiments show, in real-life scenarios, *Magic*CM is significantly more efficient than NaïveCM in terms of both running times and memory consumption.

2) Sampled subgraph construction: Note that the subgraph $\mathcal{G}^m$ generated (using the Magic-Sets transformation) for the *RR* computation of a given idb tuple t, is sampled before usage, and thus only the sampled edges are in fact useful. We may thus further optimize *Magic*CM by incorporating this sampling already in the graph generation (i.e., in Algorithm 1). This may be done by considering the rules' weights (in line 4), generating the nodes and edges corresponding to instantiations of each rule r with the probability of $w(r)$. Namely, apply the loop in lines 5–11 in Algorithm 1 with the probability of $w(r)$. Importantly, note that for every rule $r \in P$ the magic transformation may generate more than a single modified rule. Therefore, to ensure consistency with the full WD graph, each rule instantiation is drawn to fire only once, and thus the set of all modified rules belong to the same rule are drawn to fire "together". Namely, we draw to fire-or-not the *origin* rules and apply accordingly the decision on all modified rules.

Here again, note that *Magic*^{S}CM computes the same (approximated) solution as NaïveCM, and hence *Magic*^{S}CM provides a $(1 - \frac{1}{e} - \epsilon)$-approximation to CM with probability $\geq (1 - \delta)$. Regarding *Magic*^{S}CM time and space complexity, in the worst case, no sampling was done (e.g., if all rules probabilities are equal to 1) and it operates as *Magic*CM. Nonetheless, as we show in our experiments, this simple modification in practice significantly reduces the size of the materialized subgraphs as well as the *RR* sets generation time.

We conclude this section with two remarks.

Remark 1: Recall that in *Magic*CM we generate a subgraph $\mathcal{G}^m$ for each sampled idb tuple. We note that these subgraphs may have common parts which are thus being re-generated multiple times. To avoid this repetition, one may build a single subgraph that is the union of the individual subgraphs. This can be done by employing the Magic-Sets transformation for the more general query that contains all sampled idb facts, and compute all *RR* sets directly from this subgraph. This would replace the per-tuple graph construction/*RR* set computation in line 2 of Algorithm 3. We refer to *Magic*CM enhanced with this grouping as *Magic*^{G}CM. However, our sampling optimization cannot be employed for this merged graph because the subgraphs sampling performed for each *RR* set must be done independently, which is not possible when the graphs are merged together. This, as we show in our experiment, results in poor performance since the program resulting from the Magic-Sets transformation for a set of tuples contains large number of rules, which significantly affects the graph size and its generation time.

Remark 2: Generally, the number of *RR* sets generated by RIS-based IM algorithms is a function of the graph size, the required error rate (ϵ), and the success probability (δ) [19], [20]. In our case, since we avoid materializing the WD graph and generate instead *RR* sets on demand, the actual graph size is unknown. Thus we use upper bounds on the number of edges and nodes, that we prove to hold (see proof in [14]). Note that generating more sets than the minimal number required only makes the result approximation tighter.

V. Experimental study

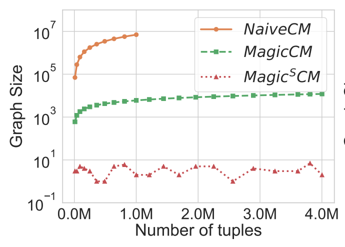
We experimentally examine the scalability of our algorithm, demonstrating the effectiveness of the proposed optimizations, in multiple real-life scenarios. We conclude with a case study, demonstrating the quality of our measure and the approximated algorithms. We have implemented all algorithms in JAVA 8 by extending IRIS [22], a JAVA- based system for datalog evaluation. The experiments were executed on a Linux server with a 2.1GHz CPU and 96GB memory.

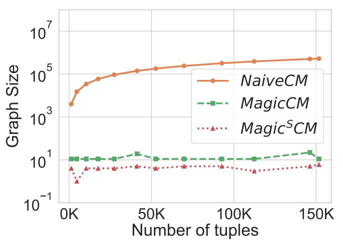
Datasets: We have experimented with multiple datasets, commonly used in the literature, which suitably include both datalog rules and an underlying database (based on the benchmark presented in [3]). These datasets include both non-recursive programs (e.g., IRIS) and (linear/bilinear) recursive programs (e.g., AMIE). Full details are provided in [14]. Unless stated otherwise, all rules have been randomly assigned with probabilities in the range of $[0, 1]$.
IRIS [22]. A non-recursive program, consists of 8 rules and generates up to 4.26M tuples.
AMIE. A recursive datalog program consisting of rules mined by AMIE [2], with weights reflecting the rule confidence. The input database is that of YAGO [5]. The program consists of 23 rules that generate up to 1.45M tuples.
Explain. The recursive datalog program, consists of 3 rules, as described in [23]. The database was randomly populated and gradually growing so that the output size is up to 0.5M tuples.
Transitive Closure. We have used a recursive program consisting of 3 rules, computing Transitive Closure in an undirected graph. The database was randomly populated to represent fully connected graphs, yielding output sizes of up to 4M tuples.

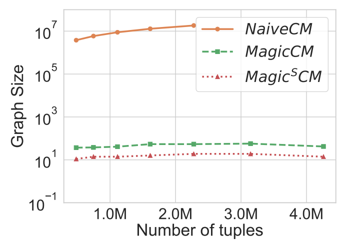
A. Experimental setup

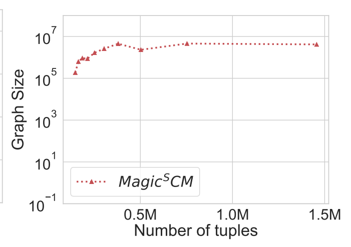
To quantify the usefulness of each proposed optimization, we examine the following baselines. (i) NaïveCM, the algorithm which employs no optimization. (ii) *Magic*CM, which employs only our Magic-Sets optimization; (iii) *Magic*^{S}CM, our proposed algorithm which enhances the Magic-Sets optimization with sampling. For completeness, we also consider the *Magic*^{G}CM algorithm (described at Section IV), which employs both the Magic-Sets and the grouping optimizations, demonstrating it to be less effective than *Magic*^{S}CM.

All examined algorithms employ an RIS-based IM algorithm and therefore, they all first generate *RR* sets. Then, to find a k-size set of tuples with the heights contribution, the same instance of the Maximum Coverage (MC) problem is solved (as IM algorithms). The value of k affects only this last phase of solving the MC instance, and hence, have the same effect on this (identical) last phase of all algorithms. Thus, to evaluate the differences between the algorithms, we compare only the



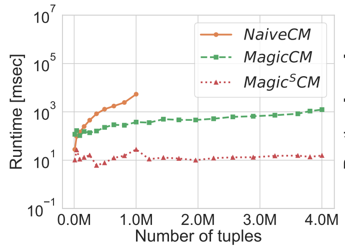


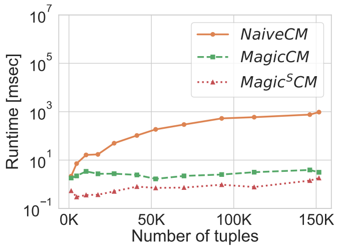


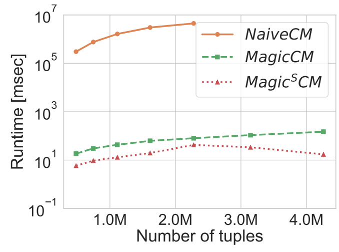


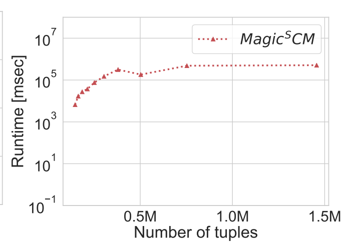
(a) TC dataset (b) Explain Dataset (c) IRIS Dataset (d) AMIE Dataset

Fig. 2: Memory consumption (amortized per *RR* set for NaïveCM) as a function of number of output tuples.









(a) TC dataset (b) Explain Dataset (c) IRIS Dataset (d) AMIE Dataset

Fig. 3: *RR* set generation time (including amortized graph generation for NaïveCM) as a function of number of output tuples.

generation time for the *RR* sets (which is what impacts the difference in running time), and the generated WD (sub)graph(s) size(s) (which reflects the memory consumption).

We note that the main difference between the algorithms is reflected in the size of the (sub)graphs each algorithm generates for the sake of *RR* sets computation. Therefore, we report the (averaged) constructed graph dimensions (the same trends hold for the maximum). Furthermore, we point out that for the same input *Magic*CM, *Magic*^{G}CM and *Magic*^{S}CM generate the same number of *RR* sets (as explained in Section IV), whereas NaïveCM may generate less *RR* sets, as the exact dimensions of the graph are known. Nonetheless, according to our experiments, the number of *RR* sets generated by the optimized algorithms was in the same order of magnitude of the number of *RR* sets NaïveCM had generated.

Last, note that even though *Magic*CM and *Magic*^{S}CM generate multiple subgraphs of the WD graph (one per each *RR* set), each such subgraphs is being used only once (for the sake of an *RR* set computation) and thus there is no need to keep it in memory. In contrast, NaïveCM and *Magic*^{G}CM keep one (sub)graph in memory throughout the algorithm run, and generate all *RR* sets from it. Furthermore, NaïveCM generates the same full WD graph, regardless of the number of *RR* sets.

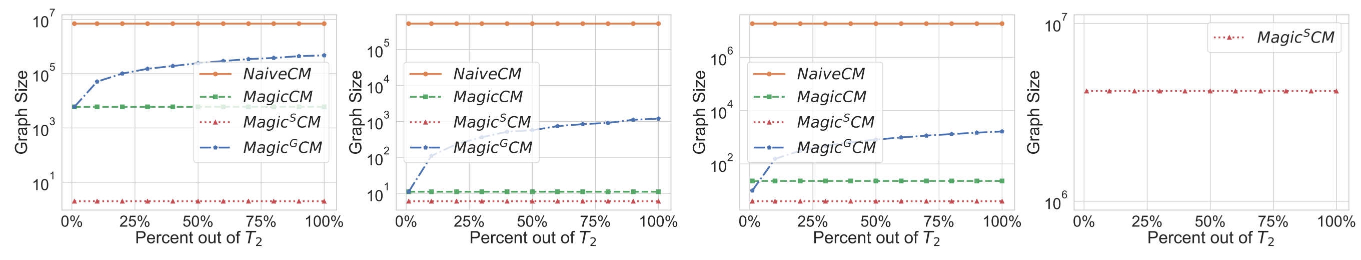
Parameter Settings: We use, as a default setting, $k = 10$ and $\epsilon = 0.1$, which are commonly used values for RIS-based IM algorithms [19], [20]. We set T_1 to be the set of all input tuples, and randomly select 100 output tuples as T_2 - the set of output tuples of interest. We note that in a real-life scenario, the user often wishes to focus on surprising/unexpected output tuples, and that the number of tuples that are investigated simultaneously is typically small [3], [6]. The examined output tuples were derived from multiple rules and different portions of the input tuples (See full details in [14]). As we shell see, in all cases, our optimized algorithms materialize only small parts of the WD graph, which demonstrates the robust nature of our optimizations. We report that the results obtained over other choices of these parameters demonstrated similar trends, and thus are omitted from presentation. Unless mentioned otherwise, the number of *RR* sets sampled for each experiment was set to be 30% of $|T_2|$, which is a typical number of samples for top performing IM algorithms.

B. Experimental results

Varying data size: Here we aim at studying the algorithms' performance as a function of number of derived tuples. To this end, we used varying input data sizes to generate increasing number of output tuples. Figures 2 and 3 show the resulting averaged WD (sub)graph size for a single *RR* set computation, and its generation time, resp. (in log scale). We amortized the WD graph generation time of NaïveCM to compute the generation time for a single *RR* set, by dividing the total generation time by the number of generated *RR* sets. We state the in these experiments, as considering a single *RR* set, *Magic*^{G}CM and *Magic*CM are completely identical, and thus we omit the results of *Magic*^{G}CM here.

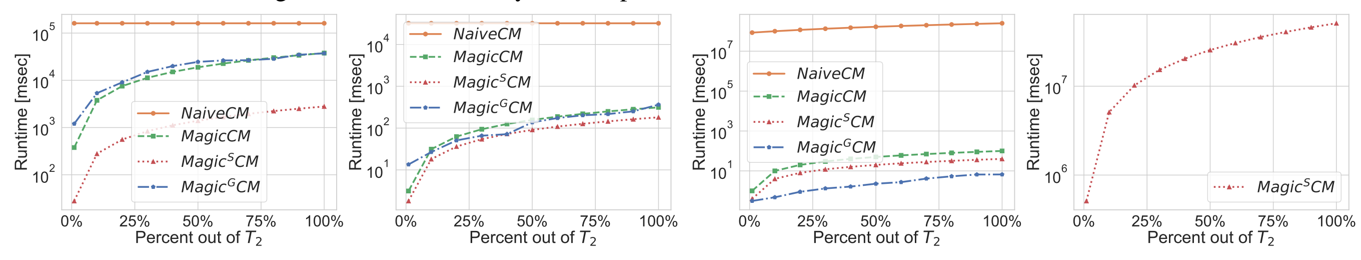
We first consider Figure 2, where the x-axis represents the number of output tuples and the y-axis the generated (sub)graph(s) size (i.e., the sum of nodes and edges). Figure 2a shows the results for TC. Generating the WD graph for NaïveCM was infeasible beyond 1M tuples (that were generated using only 1K input tuples). The memory consumption of *Magic*CM was less than 1% compared to NaïveCM. For *Magic*^{S}CM, the largest graph generated contained only 4 nodes and 3 edges. The WD graph was feasible for NaïveCM only for Explain (Figure 2b). The largest WD graph contained 225, 957 nodes and 299, 193 edges. The memory consumption of *Magic*CM was less than 0.02% compared to NaïveCM. Generating the WD graph beyond 2.27M tuples was infeasible for NaïveCM for IRIS (Figure 2c). The WD subgraph size generated by *Magic*^{S}CM for AMIE is depicted in Figure 2d. Here, generating the WD (sub)graph for NaïveCM and *Magic*CM was infeasible, even for small number of input tuples, because of the high complexity of the program that leads to a huge number of rule instantiations.

We next consider Figure 3, where the x-axis is the number of derived tuples and the y-axis are the running times in



(a) TC dataset (b) Explain Dataset (c) IRIS Dataset (d) AMIE Dataset

Fig. 4: Maximal memory consumption as a function of the number of *RR* sets.



(a) TC dataset (b) Explain Dataset (c) IRIS Dataset (d) AMIE Dataset

Fig. 5: Total running time as a function of number of *RR* sets.

milliseconds (including the amortized graph generation time for NaïveCM). For TC (Figure 3a), again NaïveCM was feasible for up to 1M output tuples. Using *Magic*CM, the *RR* set generation time was around one second for the largest number of tuples, and 15ms using *Magic^S*CM. For Explain (Figure 3b), the total computation time was about 0.5 sec, 150ms and less than 20ms for the NaïveCM, *Magic*CM and *Magic^S*CM resp. For IRIS (Figure 3c), generating the graph for NaïveCM took over 23 hours for the largest feasible data size, and computing a single *RR* set took 27 minutes. The total computation time was roughly 8 minutes for the largest number of tuples in AMIE, using *Magic^S*CM (Figure 3d).

Number of RR Sets: Figures 4 and 5 present the memory consumption and running times as a function of the number of generated *RR* sets, resp. (in log scale). To this end, we varied the number of *RR* sets from 1% to 100%. We examine the performance for various number of output tuples, and we report the results for the largest number of output tuples where all algorithms were feasible (1M tuples for TC, 151,700 for Explain, and 2.27M for IRIS). We observed similar trends for other numbers of tuples. For AMIE, only *Magic^S*CM was feasible and thus, NaïveCM, *Magic*CM and *Magic^G*CM are omitted from the graphs. The results presented are for $1.45M$ tuples. Recall that *Magic*CM and *Magic^S*CM generate multiple subgraphs of the WD graph, each of those subgraphs is being used only once. Thus, the memory consumption does not depend on the number of *RR* sets generated. *Magic^G*CM, on the other hand, keeps one subgraph in memory throughout the algorithm run, and the subgraph generated depends on the number of *RR* sets, thus we expect to see a moderate growth in the memory consumption as a function of the number of *RR* sets. Finally, NaïveCM generates the same full WD graph regardless of the number of *RR* sets.

We consider first Figure 4, where the x-axis represents the number of *RR* sets (as the percentage of T_2), and the y-axis is the averaged graph size (sum of nodes and edges). Here again, *Magic^S*CM outperformed the competitors in all datasets. The results for TC are presented in Figure 4a. The WD subgraph generated by *Magic^G*CM is relatively large here, since the graph represented by the database is fully connected. The results for Explain (4b) and IRIS (Figure 4c) showed similar trends with moderate growth on the memory consumption of *Magic^G*CM. The WD subgraph generated by *Magic^S*CM for AMIE contained (on average) around 1M nodes and 3M edges.

Last, consider Figure 5, where the x-axis represents the percent of T_2 tuples selected to generate the *RR* sets, and the y-axis is the runtime in milliseconds. Not surprisingly, as NaïveCM generates the full WD regardless of the number of *RR* sets, and the graph generation is the dominant time of its computation, the overall runtime of NaïveCM is much higher than the alternatives. We can also see that the performance of *Magic^G*CM is highly depended on the WD graph structure and the sampled nodes in T_2. It performs well when the derivation trees for the facts in T_2 overlap in the WD graph is large (i.e., they share the same nodes and edges). In TC (Figure 5a) *Magic^S*CM outperformed *Magic*CM and *Magic^G*CM, that showed similar results. The results for Explain (Figure 5b) were roughly the same, with differences of milliseconds. For IRIS (Figure 5c *Magic^G*CM is a bit faster because of the WD graph structure and large overlap in the derivation trees, however the differences are in milliseconds. Finally, as noted above, for AMIE, only *Magic^S*CM was feasible.

C. A case study

We next present a case study whose goal is twofold: First, we demonstrate the intuition underlying our proposed contribution measure, illustrating how it indeed captures the most influential tuples for a program output. Second, we quantitatively evaluate the accuracy of the approximation algorithms. Recall that all of our algorithms compute the same solution for a given CM instance (as they all run an RIS-based IM algorithm over the WD graph). Therefore, here we focus only on the *Magic^S*CM algorithm, comparing its results to the optimal solution for a given CM instance, referred to as OPT. In this set of experiments we consider a standard TC program, whose computation is easy to follow. We note that for the scenarios examined in Section V-B, as they involve complex

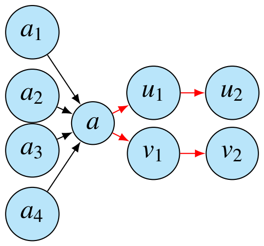
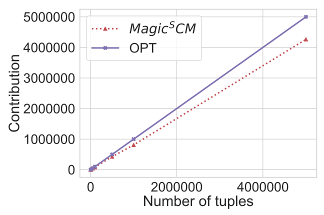
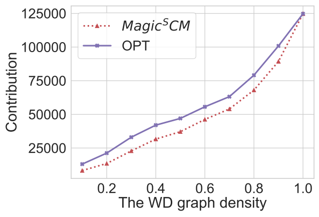


Fig. 6: An example graph with a star-subgraph of size 5 and two “sink” nodes.



(a) Connected star-subgraphs.



(b) Random graphs.

Fig. 7: The contribution sizes of $Magic^S$CM and OPT.

datalog programs and/or large databases, computing OPT is infeasible (as well as following the full derivation process).

Consider again the probabilistic TC problem presented in Example 4.2, and a directed graph (representation by an edge relation) consisting of a star subgraph of size l, where the internal node is connected to m additional “sink” nodes with paths of length 2. Figure 6 depicts such graph with a star subgraphs of size $l = 5$, and $m = 2$ sink nodes. Assume that one is wishes to find which two edb tuples (i.e., edges in the graph) contribute the most to the derivation of the tuples of the form $TC(a_i, u_2)$ and $TC(a_i, v_2), i = 1, ...4$ (namely the reachability from the a_i's to u_2 and v_2). This can be useful, e.g., for finding a small-size “bottle neck” in the paths leading from the a_i's to u_2 and v_2. Observe that the edges $(a, u1)$ and (u_1, u_2) participate in all derivations of the $TC(a_i, u_2)$ tuples, whereas the edges (a, v_1) and (v_1, v_2) participate in all derivations of the $TC(a_i, v_2)$ tuples. Intuitively, to maximally contribute to both sets, we need to pick one of the $(a, v_1), (v_1, v_2)$ edges and one of the $(a, u_1), (u_1, u_2)$ edges. Indeed, any such pair yields the maximal contribution score. Note that all four edges $((a, v_1), (v_1, v_2), (a, v_1), (v_1, v_2))$ have the same individual contribution scores. Thus, selecting the top 2 tuples with the highest individual contribution score (instead of selecting a 2-size set with the maximal joint contribution) may yield the pair $(a, u_1), (u_1, u_2)$ (by breaking equality arbitrarily), which fails to capture the essence of a “bottle neck” pair for both u_2 and v_2.

In this small example, $Magic^S$CM also returns the optimal solution. However, when it comes to larger database (i.e., graphs), it may return an approximated solution. To measure the quality of approximation, Figure 7 depicts the results of OPT and of $Magic^S$CM over gradually growing instances of CM. We report the averaged results of $Magic^S$CM of 10 runs. As expected, in all cases, the contribution size of $Magic^S$CM was at least $(1 - \frac{1}{e})$-fraction of the contribution size OPT. Generally, as in IM algorithms, the approximation ratio of $Magic^S$CM depends on the WD graph structure: the more dense the WD graph is, the better is the approximation that $Magic^S$CM (and IM algorithms) achieves. For example, in fully connected WD graphs (i.e., in CM instances where all edbs are used to derive every idb), we get a perfect solution, whereas in sparse WD graphs (i.e., in CM instances where a distinct set of edbs is used to derive each idb) the approximation could be as worse as $(1 - \frac{1}{e})$.

To illustrate, consider first Figure 7a, where the x-axis represents the number of idbs using star-like graphs while varying the parameters m and l[3], and the y-axis is the contribution size of the optimal solution/the solution returned by $Magic^S$CM. In these star-like graphs, the contribution size of OPT (nearly) equals to the number of idbs, and the worst approximation ratio measured was 0.8. This relatively high ratio achieved due to the fact that the corresponding WD graphs are dense graphs, and the edges to the sink nodes (i.e., red edges in Figure 6) are hub nodes the WD graphs (i.e., nodes with a number of edges that greatly exceeds the average). Figure 7b depicts the results of $Magic^S$CM and OPT of the probabilistic TC program executed over random graphs with 500 nodes[4]. The x-axis is the WD graph density $d = \frac{|E|}{|V| \cdot (|V|-1)}$, and the y-axis is the contribution size. One can see that on fully connected WD graphs (i.e., where $d = 1$), $Magic^S$CM gets a perfect approximation ratio of 1, while on sparse WD graphs (e.g., $d = 0.1$), the approximation ratio is ≈ 0.63.

VI. Related Work

There is a wealth of work on query results explanation. Closest to our work is a line of works providing alternative explanations on how a result was derived, by pointing on database facts that significantly affect the results [8], [6], [7], [9]. However, those works have focused on *non-recursive* SQL queries, very often *disregarding probabilistic inference*. To analyze probabilistic datalog programs, one must consider the potentially involved transitive relationship between input and output data, and the uncertainty should be reflected in the computation. Another important difference is the fact that previous works have mainly focused on quantifying the contribution of a *single* database tuple to a *single* output tuple, whereas we consider the the *joint* contribution of a set of input tuples to a set of output tuples. Therefore, the relationship between individual tuple contributions is also considered.

Among the variety of works in this direction, our approach is most similar to the work presented in [6], where Meliou et al. have defined the notion of causality and degree of responsibility for SQL queries, for a single input/output tuple. Responsibility serves to rank potentially many causes by their relative contributions to a query outcome. Naturally, the setting of SQL queries can be seen as a simple case of probabilistic datalog programs. However, when considering the contribution of a single database tuple to a single output fact in this setting, our contribution function provides an alternative definition to the responsibility function, as we quantify the marginal contribution of a tuple to an output tuple *independently of other derivations in the program*. Contrarily, the responsibility definition in [6] is affected by the dependencies among the database facts, as it quantifies causality of a tuple t to an answer a by considering the size of a set Γ such that after removing Γ from the database, we bring it to a state where

[3]We focus on such graphs since on these graphs OPT can be computed directly (avoiding an exhaustive search which is infeasible for large graphs).

[4]We note that even on such small graphs (i.e., databases), the corresponding WD graph contains $\approx$ 100K nodes, and finding OPT requires to examine all $\binom{|V|}{k}$ seed sets, which was infeasible for larger databases.

removing/inserting t causes a to switch between an answer and non-answer to the query. Our work and theirs are thus complementary, providing two alternative angles of involvement in the result computations.

Another line of works utilize the *data provenance*, providing an explanation for a query results, as it captures the essence of the computation performed by queries. Provenance information was shown to be useful for query results explanation [8], [3], [24], [25], [26], [27]. Particularly, provenance for datalog was studied in [11], [3], [12]. Our WD graph captures the conventional datalog provenance, as defined in [12]. However, while theoretical analysis of the provenance size shows that it is polynomial in that of the input database, there is a practical blowup of it. To overcome this, different works have studied techniques for the factorization or summarization of provenance [12], [28], [29], [30], which differs then the goal of identifying important input tuples. Provenance summarization/approximation for explanation for non-recursive programs was studied in [31]. Selective provenance tracking for Nested Relational Calculus was presented in [32] and for datalog in [3]. The method presented in [3] provides explanations for datalog programs output, based on derivation tree, using user-defined patterns. As noted, [3] assumes that the user has prior knowledge on the database tuples and their contribution to the output, thus our framework is complementary to [3], as it can provide such information. Integrating the two solutions into a single system is left for future work.

Our work harnesses algorithms developed for the Influence Maximization (IM) problem as a component of our solution. The seminal work of [10], the first to formulate the IM problem, has motivated extensive research [16], [20], which can be classified into two main approaches for solving IM: (i) The greedy framework [10], [33], which iteratively adds nodes to the seed set s.t. each addition maximizes the expected marginal influence gain; (ii) More recently, the Reverse Influence Sampling (RIS) framework has been proposed in [16], where, while retaining optimal accuracy, running times were gradually improved, resulting in highly scalable algorithms of near-optimal time complexity [18], [20], [19]. As explained in Section IV, any given RIS-based IM algorithm can be adjusted to solve the CM problem, retaining the same properties.

VII. Conclusions and Future Work

In this work we have presented the CM problem for probabilistic datalog programs, and devised an efficient algorithm which integrates IM algorithms with the Magic-Sets transformation technique. We are currently pursuing an implementation of a graphical interface, allowing users to easily specify their input/output tuple-set of interest, using patterns. Further directions for future research include the development of additional index-based optimization and the incorporation of user constraints for the properties of the returned output. For instance, for diversification, require that every selected database tuple will come from a different table in the database.

Acknowledgment: This work has been partially funded by the Israel Innovation Authority, MDM, the Israel Science Foundation, the Binational US-Israel Science foundation, Len Blavatnik, the Blavatnik Family foundation, and the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (grant No. 804302).

References

[1] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, “Big data analytics with datalog queries on spark,” in *SIGMOD*. ACM, 2016.

[2] L. A. Galárraga, C. Teflioudi, K. Hose, and F. M. Suchanek, “Amie: association rule mining under incomplete evidence in ontological knowledge bases,” in *WWW*, 2013.

[3] D. Deutch, A. Gilad, and Y. Moskovitch, “Selective provenance for datalog programs using top-k queries,” *PVLDB*, 2015.

[4] V. Bárány, B. T. Cate, B. Kimelfeld, D. Olteanu, and Z. Vagena, “Declarative probabilistic programming with datalog,” *TODS*, 2017.

[5] F. M. Suchanek, G. Kasneci, and G. Weikum, “Yago: a core of semantic knowledge,” in *WWW*. ACM, 2007.

[6] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu, “The complexity of causality and responsibility for query answers and non-answers,” *PVLDB*, 2010.

[7] B. Kanagal, J. Li, and A. Deshpande, “Sensitivity analysis and explanations for robust query evaluation in probabilistic databases,” in *SIGMOD*, ser. SIGMOD ’11, 2011.

[8] S. Roy and D. Suciu, “A formal approach to finding explanations for database queries,” in *SIGMOD*, 2014.

[9] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu, “Why so? or why no? functional causality for explaining query answers,” *arXiv*, 2009.

[10] D. Kempe, J. Kleinberg, and E. Tardos, “Maximizing the spread of influence through a social network,” in *SIGKDD*, 2003.

[11] T. J. Green, G. Karvounarakis, and V. Tannen, “Provenance semirings,” in *PODS*, 2007.

[12] D. Deutch, T. Milo, S. Roy, and V. Tannen, “Circuits for datalog provenance.” in *ICDT*, 2014.

[13] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.

[14] “Technical report,” 2019.

[15] S. Bistarelli, F. Martinelli, and F. Santini, “Weighted datalog and levels of trust,” in *ARES*. IEEE, 2008.

[16] C. Borgs, M. Brautbar, J. Chayes, and B. Lucier, “Maximizing social influence in nearly optimal time,” in *SODA*. Society for Industrial and Applied Mathematics, 2014.

[17] V. V. Vazirani, *Approximation algorithms*. Springer Science & Business Media, 2013.

[18] Y. Tang, Y. Shi, and X. Xiao, “Influence maximization in near-linear time: A martingale approach,” in *SIGMOD*, 2015.

[19] K. Huang, S. Wang, G. Bevilacqua, X. Xiao, and L. V. S. Lakshmanan, “Revisiting the stop-and-stare algorithms for influence maximization,” *PVLDB*, 2017.

[20] H. T. Nguyen, M. T. Thai, and T. N. Dinh, “Stop-and-stare: Optimal sampling algorithms for viral marketing in billion-scale networks,” in *SIGMOD*. ACM, 2016.

[21] Y. Li, D. Zhang, and K.-L. Tan, “Real-time targeted influence maximization for online advertisements,” *PVLDB*, 2015.

[22] “Iris reasoner,” http://www.iris-reasoner.org, 2018.

[23] T. Arora, R. Ramakrishnan, W. G. Roth, P. Seshadri, and D. Srivastava, “Explaining program execution in deductive systems,” in *DOOD*. Springer, 1993.

[24] D. Deutch, N. Frost, and A. Gilad, “Provenance for natural language queries,” *PVLDB*, 2017.

[25] S. Lee, B. Ludäscher, and B. Glavic, “Provenance summaries for answers and non-answers,” *PVLDB*, 2018.

[26] P. Bourhis, D. Deutch, and Y. Moskovitch, “Analyzing data-centric applications: Why, what-if, and how-to,” in *ICDE*, 2016.

[27] P. Buneman, S. Khanna, and W. C. Tan, “Why and where: A characterization of data provenance,” in *ICDT*, 2001.

[28] R. Fink, L. Han, and D. Olteanu, “Aggregation in probabilistic databases via knowledge compilation,” *PVLDB*, 2012.

[29] C. Ré and D. Suciu, “Approximate lineage for probabilistic databases,” *PVLDB*, 2008.

[30] E. Ainy, P. Bourhis, S. B. Davidson, D. Deutch, and T. Milo, “Approximated summarization of data provenance,” in *CIKM*, 2015.

[31] S. Lee, X. Niu, B. Ludäscher, and B. Glavic, “Integrating approximate summarization with provenance capture,” in *TaPP*, 2017.

[32] J. Cheney, A. Ahmed, and U. A. Acar, “Database queries that explain their work,” in *PPDP*, 2014.

[33] A. Goyal, W. Lu, and L. V. Lakshmanan, “Celf++: Optimizing the greedy algorithm for influence maximization in social networks,” in *WWW*. ACM, 2011.