Computing the Shapley Value of Facts in Query Answering

Daniel Deutch
Tel Aviv University
Blavatnik School of Computer Science, Tel Aviv, Israel
danielde@post.tau.ac.il

Nave Frost
Tel Aviv University
Blavatnik School of Computer Science, Tel Aviv, Israel
navefrost@mail.tau.ac.il

Benny Kimelfeld
Technion - Israel Institute of Technology
Faculty of Computer Science, Haifa, Israel
bennyk@cs.technion.ac.il

Mikaël Monet
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL, F-59000 Lille, France
mikael.monet@inria.fr

arXiv:2112.08874v2 [cs.DB] 2 Jan 2022

ABSTRACT

The Shapley value is a game-theoretic notion for wealth distribution that is nowadays extensively used to explain complex data-intensive computation, for instance, in network analysis or machine learning. Recent theoretical works show that query evaluation over relational databases fits well in this explanation paradigm. Yet, these works fall short of providing practical solutions to the computational challenge inherent to the Shapley computation. We present in this paper two practically effective solutions for computing Shapley values in query answering. We start by establishing a tight theoretical connection to the extensively studied problem of query evaluation over probabilistic databases, which allows us to obtain a polynomial-time algorithm for the class of queries for which probability computation is tractable. We then propose a first practical solution for computing Shapley values that adopts tools from probabilistic query evaluation. In particular, we capture the dependence of query answers on input database facts using Boolean expressions (data provenance), and then transform it, via Knowledge Compilation, into a particular circuit form for which we devise an algorithm for computing the Shapley values. Our second practical solution is a faster yet inexact approach that transforms the provenance to a Conjunctive Normal Form and uses a heuristic to compute the Shapley values. Our experiments on TPC-H and IMDB demonstrate the practical effectiveness of our solutions.

1 INTRODUCTION

Explaining query answers has been the objective of extensive research in recent years [12, 15, 20, 29, 30]. A prominent approach is to devise an explanation based on the facts that were used for deriving the answers; these facts are often termed *provenance* or *lineage* of the query answer. For illustration, consider a query asking whether there exists a route from the USA to France with at most one connection over a database of airports and flights. The answer is Boolean, and upon receiving a positive answer one may seek explanations of why it is so. The basis for such explanations would include all details of qualifying routes that are used in the derivation of the answer. Unfortunately, the number of relevant routes might be huge (specifically, quadratic in the database size). Moreover, it is conceivable that different facts differ considerably in their importance to the answer at hand; for instance, some flights may be crucial to enabling the USA-France connection, while others may be easily replaced by alternatives.

To address these issues, there have been several proposals for principled ways of *quantifying* the contribution of input facts to query answers [20, 24, 25, 30]. We focus here on the recent approach of Livshits et al. [20] that applies to this setting the notion of *Shapley values* [32]—a game-theoretic function for distributing the wealth of a team in a cooperative game. This function has strong theoretical justifications [28], and indeed, it has been applied across various fields such as economics, law, environmental science, and network analysis. It has also been used for explanations in data-centric paradigms such as knowledge representation [16, 39] and machine learning [21, 22]. In the context of relational databases, given a query $q(\bar{x})$, a database D, an input fact $f \in D$ and a tuple $\bar{t}$ of same arity as $\bar{x}$, *the Shapley value of f in D for query $q(\bar{x})$ and tuple $\bar{t}$* intuitively represents the contribution of f to the presence (or absence) of $\bar{t}$ in the query result.

Livshits et al. [20, 27] initiated the study of the computational complexity of calculating Shapley values in query answering. They showed mainly lower bounds on the complexity of the problem, with the exception of the sub-class of self-join free SPJ queries called *hierarchical*, where they gave a polynomial-time algorithm. The results are more positive if imprecision is allowed, as they showed that the problem admits a tractable approximation scheme (FPRAS, to be precise) via Monte Carlo sampling. The state of affairs is that the class of known tractable cases (namely the hierarchical conjunctive queries) is highly restricted, and the approximation algorithms with theoretical guarantees are impractical in the sense that they require a large number of executions of the query over database subsets (the samples). Hence, the theoretical analysis of Livshits et al. [20, 27] does not provide sufficient evidence of practical feasibility for adopting the Shapley value as a measure of responsibility in query answering. Moreover, the results of Livshits et al. [20] imply that, for self-join free SPJ queries the class of tractable queries for computing Shapley values coincides with the class of tractable queries in probabilistic tuple-independent databases [7]. Yet, no direct connection has been made between these two problems and, theoretically speaking, it has been left unknown whether algorithms for probabilistic databases can be used for Shapley computation.

Recently, Van den Broeck et al. [36] and Arenas et al. [2, 3] investigated the computational complexity of the *SHAP-score* [22], a notion used in machine learning for explaining the predictions of a model. While both are based on the general notion of Shapley value, the SHAP-score for machine learning and Shapley values for databases are different. In the latter case, the players are the tuples of the database and the game function that is used is simply the value of the query on a subset of the database, while in the former case, the players are the features of the model and the game function is a conditional expectation of the model's output (see Section 6.2 for a more formal definition of the SHAP-score). Remarkably, Van den Broeck et al. [36] have shown that computing the SHAP-score is equivalent (in terms of polynomial-time reductions) to the problem of computing the expected value of the model. One of our

contributions is to show that the techniques developed by [2, 3, 36] can be adapted to the context of Shapley values for databases. For instance, by adapting to our context the proof of Van den Broeck et al. [36] that computing the SHAP-score reduces to computing the expected value of the model, we resolve the aforementioned open question affirmatively: we prove that Shapley computation can be efficiently (polynomial-time) reduced to probabilistic query answering. Importantly, this applies not only to the restricted class of SPJ queries without self-joins, but to *every* database query. Hence, extending theory to practice, one can compute the Shapley values using a query engine for probabilistic databases.

In turn, a common approach that was shown to be practically effective for probabilistic databases is based on *Knowledge Compilation* [11, 18]. In a nutshell, the idea is to first compute the Boolean provenance of a given output tuple in the sense of Imielinski and Lipski [17], and then to "compile" the provenance into a particular circuit form that is more favorable for probability computation. Specifically, the target class of this compilation is that of *deterministic and decomposable circuits* (d-D). In our case, rather than going through probabilistic databases, we devise a more efficient approach that computes the Shapley values directly from the d-D circuit. This is similar to how Arenas at al. [2, 3] directly prove that the SHAP-score can be computed efficiently over such circuits, without using the more general results of [36]. By adapting the proof of [2, 3], we show how, given a d-D circuit representing the provenance of an output tuple, we can efficiently compute the Shapley value of every input fact. While the aforementioned properties of the circuit are not guaranteed in general (beyond the class of hierarchical queries), we empirically show the applicability and usefulness of the approach even for non-hierarchical queries.

Our experimental results (see below) indicate that our exact computation algorithm is fast in most cases, but is too costly in others. For the latter cases, we propose a heuristic approach to retrieve the relative order of the facts by their Shapley values, without actually computing these values. Indeed, determining the most influential facts is in many cases already highly useful, even if their precise contribution remains unknown. The solution that we propose to this end is termed *CNF Proxy*; it is based on a transformation of the provenance to Conjunctive Normal Form (CNF) and using it to compute proxy values intuitively based on (1) the number of clauses in which a variable occur and (2) its alternatives in each clause. These are two aspects that are correlated with Shapley values. The proxy values may be very different from the real Shapley values, and yet, when we order facts according to their proxy values we may intuitively get an ordering that is similar to the order via Shapley. Our experiments validate that this intuition indeed holds for examined benchmarks.

We have experimented with multiple queries from the two standard benchmarks TPC-H and IMDB. Our main findings are as follows. In most cases (98.67% of the IMDB output tuples and 83.83% for TPC-H), our exact computation algorithm terminates in 2.5 seconds or less, given the provenance expression. In the vast majority of remaining cases the execution is very costly, typically running out of memory already in the Knowledge Compilation step. By contrast, our inexact solution *CNF Proxy* is extremely fast even for these hard cases – it typically terminates in a few milliseconds with the worst observed case (an outlier) being 4 seconds. In fact, it is faster by several orders of magnitude than sampling-based approximation techniques (the Monte Carlo sampling proposed in [20] as well as a popular sampling-based solution for Shapley values in Machine Learning (Kernel SHAP [22]). To measure quality, we use *CNF Proxy* to rank the input tuples, and compared the obtained ranked lists to ranking by actual Shapley values (in cases where exact computation has succeeded), using the standard measures of nDCG and Precision@k. Our solution outperforms the competitors in terms of quality as well.

We then propose a simple hybrid approach: execute the exact algorithm until it either terminates or a timeout elapses. If we have reached the timeout, resort to executing *CNF Proxy* and rank the facts based on the obtained values. We show experiments with different timeout values, justifying our choice of 2.5 seconds.

Hence, our contributions are both of a theoretical and practical nature and can be summarized as follows.

- By adapting the proof technique of [36], we establish a fundamental result about the complexity of computing Shapley values over relational queries: Shapley values can be computed in polynomial time–in data complexity–whenever the query can be evaluated in polynomial time over tuple-independent probabilistic databases (Proposition 3.1). This holds for *every* query.
- By adapting the proof technique of [2, 3], we devise a novel algorithm for computing Shapley values for query evaluation via compilation to a deterministic and decomposable circuit (Proposition 4.4). We show that this algorithm is practical and has the theoretical guarantee of running in polynomial time in the size of the circuit.
- We present a novel heuristic, CNF Proxy, that is fast yet inexact, and is practically effective if we are interested in ranking input facts by their contribution rather than computing exact Shapley values (Section 5).
- We describe a thorough experimental study of our algorithms over realistic data and show their efficiency (Section 6).

Related work. Existing models for explaining database query results may roughly be divided in two categories: (1) models that are geared for tracking/presenting provenance of output tuples, e.g., the set of all input facts participating in their computation [6], possibly alongside a description of the ways they were used, in different granularity levels (e.g., [4, 5, 14]); (2) models that quantify contributions of input facts [20, 24, 25, 30], which is the approach that we follow here. Works in the latter context often have connections with the influential line of work on probabilistic databases [33], and we show that this is the case for Shapley computation as well.

As already mentioned, an important point of comparison is the work of Van den Broeck et al. [36] and that of Arenas et al. [2, 3] on the *SHAP-score*. While we show that the proof techniques developed in this area can be adapted to the context of relational databases, we point out that the two sets of results obtained (for SHAP-score and for Shapley values for databases) seem incomparable, as we do not see a way of proving results for Shapley value for query answering using the results on the SHAP-score, or vice-versa. In fact, this adaptation only works up to a certain point. For instance, the *efficiency axiom* of the Shapley value immediately implies that

computing the expected value of a model can be reduced in polynomial time to computing the SHAP-score of its features; in contrast, this axiom does not seem to yield any clear such implication in our context (see our Open Problem 1 and the discussion around it).

Paper organization. We formalize the notion of Shapley values for query answering in Section 2. In Section 3 we present the theoretical connection to probabilistic databases and its implications. Our exact computation algorithm is presented in Section 4 and our heuristic in Section 5. Experimental results are presented in Section 6 and we conclude in Section 7.

2 THE SHAPLEY VALUE OF FACTS

We define here the main notion and illustrate it with an example.

Relational databases and queries. Let $\Sigma = \{R_1, \ldots, R_n\}$ be a *signature*, consisting of *relation names* R_i each with its associated *arity* $\mathsf{ar}(R_i) \in \mathbb{N}$, and Const be a set of *constants*. A *fact* over (Σ, Const) is simply a term of the form $R(a_1, \ldots, a_{\mathsf{ar}(R)})$, for $R \in \Sigma$ and $a_i \in \mathsf{Const}$. A *(Σ, Const)-database* D, or simply a *database* D, is a finite set of facts over (Σ, Const). We assume familiarity with the most common classes of query languages and refer the reader to [1] for the basic definitions. In particular, we recall the equivalence between relational algebra and relational calculus [1], and the fact that Select-Project-Join-Union (SPJU) queries are equivalent to unions of conjunctive queries (UCQs). Depending of the context and for consistency with relevant past publications, we will use terminology of either relational calculus or relational algebra. What we call a *Boolean query* is a query q that takes as input a database D and outputs $q(D) \in \{0, 1\}$. If $q(\bar{x})$ is a query with free variables $\bar{x}$ and $\bar{t}$ is a tuple of constants of same length as $\bar{x}$, we denote by $q[\bar{x}/\bar{t}]$ the Boolean query defined by: $q[\bar{x}/\bar{t}](D) = 1$ if and only if $\bar{t}$ is in the output of $q(\bar{x})$ on D.

Shapley values of facts. Following [20, 27], we use the notion of Shapley values [32] to attribute a contribution to facts of an input database. In this context, the database D is traditionally partitioned into two sets of facts: a set D_{x} of so-called *exogenous* facts, and a set D_{n} of *endogenous* facts. The idea is that exogenous facts are considered as given, while endogenous facts are those to which we would like to attribute contributions. Let q be a Boolean query and $f \in D_{\mathrm{n}}$ be an endogenous fact. *The Shapley value of f in D for query q*, denoted $\mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, f)$, is defined as

$$\mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, f) \stackrel{\text{def}}{=} \sum_{E \subseteq D_{\mathrm{n}} \setminus \{f\}} \frac{|E|!(|D_{\mathrm{n}}| - |E| - 1)!}{|D_{\mathrm{n}}|!} \big(q(D_{\mathrm{x}} \cup E \cup \{f\}) - q(D_{\mathrm{x}} \cup E)\big). \tag{1}$$

Notice that here, $|E|!(|D_{\mathrm{n}}| - |E| - 1)!$ is the number of permutations of D_{n} with all endogenous facts in E appearing first, then f, and finally, all the other endogenous facts. Intuitively then, the value $\mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, f)$ represents the contribution of f to the query's output: the higher this value is, the more f helps in satisfying q.

For non-Boolean queries $q(\bar{x})$, we are interested in the Shapley value of the fact f for every individual tuple $\bar{t}$ in the output [20]. The extension to non-Boolean $q(\bar{x})$ is then straightforward: the Shapley value of the fact f *for the answer* $\bar{t}$ *to* $q(\bar{x})$ is the value $\mathsf{Shapley}(q[\bar{x}/\bar{t}], D_{\mathrm{n}}, D_{\mathrm{x}}, f)$. Therefore, the computational challenge reduces to that of the Boolean query $q[\bar{x}/\bar{t}]$. Hence, in the theoretical analysis we focus on Boolean queries, and we go back to considering non-Boolean queries when we study the implementation aspects (starting in Section 4.2).

Example 2.1. Consider the database D and the Boolean query q from Figures 1a and 1c. All facts in table Flights are endogenous, while facts in Airports are exogenous. To alleviate the notation we write, e.g., a_1 for Flights(JFK, CDG). The query q checks if there are routes from "USA" to "FR" with one or less connecting flights. Let us compute the Shapley value of all endogenous facts. First, we notice that fact a_8 is not part of any valid route, so $\mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, a_8) = 0$ by Equation (1). Next, let us focus on a_1. Since a_1 is a valid route on its own, adding it to any subset of (endogenous) facts E such that E does not contain a valid route results in $q(D_{\mathrm{x}} \cup E \cup \{a_1\}) - q(D_{\mathrm{x}} \cup E) = 1$ (for all other subsets the difference will be 0). The relevant subsets are the empty set, all singletons $\{a_i\}$ for $2 \leq i \leq 8$ (7 singletons), all the pairs of tuples from $a_2, \ldots, a_8$ excluding the pairs $\{a_2, a_4\}$, $\{a_2, a_5\}$, $\{a_3, a_4\}$, $\{a_3, a_5\}$, and $\{a_6, a_7\}$ (so $\binom{7}{2} - 5 = 16$ pairs), the quadruples $\{a_2, a_3, a_6, a_8\}$, $\{a_2, a_3, a_7, a_8\}$, $\{a_4, a_5, a_6, a_8\}$, and $\{a_4, a_5, a_7, a_8\}$, and overall 14 triplets (left to the reader). Summing it all up results in

$$\mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, a_1) = 1 \cdot \frac{0! \cdot 7!}{8!} + 7 \cdot \frac{1! \cdot 6!}{8!} + 16 \cdot \frac{2!5!}{8!} + 14 \cdot \frac{3!4!}{8!} + 4 \cdot \frac{4!3!}{8!} = \frac{43}{105} \approx 0.4095.$$

Similarly one can compute the Shapley value of the remaining facts, and find that for $a_i \in \{a_2, a_3, a_4, a_5\}$ it holds that $\mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, a_i) = \frac{23}{210} \approx 0.1095$, and that for $a_i \in \{a_6, a_7\}$ we have $\mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, a_i) = \frac{8}{105} \approx 0.0762$.

3 REDUCTION TO PROBABILISTIC DATABASES

In this section we investigate the complexity of computing Shapley values. As explained in the previous section, the non-Boolean setting of the problem may be reduced to that of Boolean queries, so we will study the following problem for a given Boolean query q.

PROBLEM:	$\mathsf{Shapley}(q)$
INPUT:	A database $D = D_{\mathrm{x}} \cup D_{\mathrm{n}}$ and an endogenous fact $f \in D_{\mathrm{n}}$.
OUTPUT:	The value $\mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, f)$.

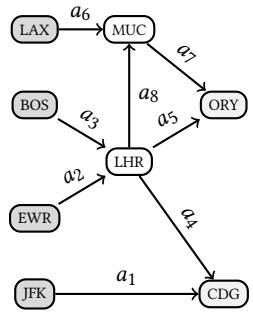
Note that for the sake of complexity analysis, the query q is assumed to be fixed, so that each query gives rise to a different computational problem; we are then considering what is called the *data complexity* [37]. This assumption is motivated by the fact that in practice, the queries are much smaller than the databases.

The complexity of this problem has been studied in [20, 27], where in particular a dichotomy has been obtained for *self-join–free Boolean conjunctive queries* (sjfbcqs). There, the authors show that, for every sjfbcq q, either q is *hierarchical* (we will not need to define this notion here) and $\mathsf{Shapley}(q)$ can be solved in polynomial time, or q is not hierarchical and then $\mathsf{Shapley}(q)$ is intractable (specifically, $\mathrm{FP}^{\#\mathrm{P}}$-hard). It turns out that the tractability criterion that

	Flights (endo)	
	Src	**Dest**
a_1	JFK	CDG
a_2	EWR	LHR
a_3	BOS	LHR
a_4	LHR	CDG
a_5	LHR	ORY
a_6	LAX	MUC
a_7	MUC	ORY
a_8	LHR	MUC

	Airports (exo)	
	Name	**Country**
b_1	JFK	USA
b_2	EWR	USA
b_3	BOS	USA
b_4	LAX	USA
b_5	LHR	EN
b_6	MUC	GR
b_7	ORY	FR
b_8	CDG	FR

(a) Database of flights and airports



(b) Flights in graph view. Dark and light gray depict "USA" and "FR" airports respectively

Routes from "USA" to "FR" with one or less connecting flights

$$q_1 = \exists x, y : \textsc{Airports}(x, \text{"USA"}) \land \textsc{Airports}(y, \text{"FR"}) \land \textsc{Flights}(x, y)$$

$$q_2 = \exists x, y, z : \textsc{Airports}(x, \text{"USA"}) \land \textsc{Airports}(z, \text{"FR"}) \land \textsc{Flights}(x, y) \land \textsc{Flights}(y, z)$$

$$q = q_1 \lor q_2$$

(c) q is a Boolean union of conjunctive queries (UCQ)

$$\underbrace{(a_1 \land b_1 \land b_8)}_{q_1 \text{ provenance}} \lor \underbrace{(a_2 \land a_4 \land b_2 \land b_8) \lor (a_2 \land a_5 \land b_2 \land b_7) \lor (a_3 \land a_4 \land b_3 \land b_8) \lor (a_3 \land a_5 \land b_3 \land b_7) \lor (a_6 \land a_7 \land b_5 \land b_7)}_{q_2 \text{ provenance}}$$

(d) The lineage $\mathsf{Lin}(q, D)$ as a formula in Disjunctive Normal Form (DNF)

Figure 1: The database, query q and its lineage used for our running example

is obtained—being hierarchical—is exactly the same as in the context of probabilistic query evaluation (PQE); see, e.g., [7, 8]. In fact, the main result of this section is that this is not a coincidence: we prove that, for *every* Boolean query q (not just for sjfbcqs), if PQE is tractable for q then so is the problem $\mathsf{Shapley}(q)$. Since PQE has been intensively studied already, our result allows us to vastly extend the tractable cases identified in [20, 27]. We now proceed with the definitions and proof of this result, and explain its consequences.

Probabilistic query evaluation. A *tuple-independent (TID) database* is a pair consisting of a database D and a function π mapping each fact $f \in D$ to a probability value $\pi(f) \in [0, 1]$. The TID (D, π) defines a probability distribution $\Pr_\pi$ on $D' \subseteq D$, where $\Pr_\pi(D') \stackrel{\text{def}}{=} \prod_{f \in D'} \pi(f) \times \prod_{f \in D \setminus D'} (1 - \pi(f))$. Given a Boolean query q, *the probability that q is satisfied by* (D, π) is $\Pr(q, (D, \pi)) \stackrel{\text{def}}{=} \sum_{D' \subseteq D \text{ s.t. } q(D')=1} \Pr_\pi(D')$. The *probabilistic query evaluation problem for q*, $\mathrm{PQE}(q)$ for short, is then defined as follows.

PROBLEM:	$\mathrm{PQE}(q)$
INPUT:	A tuple-independent database (D, π).
OUTPUT:	The value $\Pr(q, (D, \pi))$.

For two computational problems A and B, we write $A \leq_{\mathrm{T}}^{\mathrm{p}} B$ to assert the existence of a polynomial-time Turing reduction from A to B. We are ready to state the main result of this section.

Proposition 3.1. *For every Boolean query q, we have that* $\mathsf{Shapley}(q) \leq_{\mathrm{T}}^{\mathrm{p}} \mathrm{PQE}(q)$.

This result implies that for any query q for which $\mathrm{PQE}(q)$ is tractable then so is $\mathsf{Shapley}(q)$. Dalvi and Suciu [8] showed a dichotomy for unions of conjunctive queries: for every such query q, either $\mathrm{PQE}(q)$ is solvable in polynomial time, in which case q is called *safe*[1], or $\mathrm{PQE}(q)$ is $\mathrm{FP}^{\#\mathrm{P}}$-hard (and q is called *unsafe*). Therefore, we obtain as a direct corollary of Proposition 3.1 that $\mathsf{Shapley}(q)$ can be solved in polynomial time for all safe queries.

Corollary 3.2. *If q is a safe UCQ then* $\mathsf{Shapley}(q)$ *can be solved in polynomial time.*

In particular, this corollary generalizes the tractability result obtained in [20], to account for CQs with self-joins and even unions of such queries. We now prove Proposition 3.1.

Proof of Proposition 3.1. For a Boolean query q, database $D = D_{\mathrm{x}} \cup D_{\mathrm{n}}$, and integer $k \in \{0, \ldots, |D_{\mathrm{n}}|\}$, define

$$\#\mathsf{Slices}(q, D_{\mathrm{x}}, D_{\mathrm{n}}, k) \stackrel{\text{def}}{=} |\{E \subseteq D_{\mathrm{n}} \mid |E| = k \text{ and } q(D_{\mathrm{x}} \cup E) = 1\}|.$$

Then, by grouping by size the terms E from Equation 1 we obtain

$$\mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, f) = \sum_{k=0}^{|D_{\mathrm{n}}|-1} \frac{k!(|D_{\mathrm{n}}| - k - 1)}{|D_{\mathrm{n}}|} \Big(\#\mathsf{Slices}(q, D_{\mathrm{x}} \cup \{f\}, D_{\mathrm{n}} \setminus \{f\}, k) - \#\mathsf{Slices}(q, D_{\mathrm{x}}, D_{\mathrm{n}} \setminus \{f\}, k) \Big). \tag{2}$$

All arithmetical terms (such as $k!$ or $|D_{\mathrm{n}}|!$) can be computed in polynomial time. Therefore, to prove that $\mathsf{Shapley}(q) \leq_{\mathrm{T}}^{\mathrm{p}} \mathrm{PQE}(q)$, it is enough to show that, given an oracle to the problem $\mathrm{PQE}(q)$, we can compute in polynomial time the quantities $\#\mathsf{Slices}(q, D_{\mathrm{x}}, D_{\mathrm{n}}, k)$, for some arbitrary $D = D_{\mathrm{x}} \cup D_{\mathrm{n}}$ and $k \in \{0, \ldots, |D_{\mathrm{n}}|\}$. This is what we do next.

The proof is similar to that of [36, Theorem 2] in the context of SHAP-score for machine learning (but, as explained in the Introduction, the two results seem to be incomparable).

We wish to compute $\#\mathsf{Slices}(q, D_{\mathrm{x}}, D_{\mathrm{n}}, k)$, for some database $D = D_{\mathrm{x}} \cup D_{\mathrm{n}}$ and integer $k \in \{0, \ldots |D_{\mathrm{n}}|\}$. Let $n = |D_{\mathrm{n}}|$ be the number of endogenous facts of D. For $z \in \mathbb{Q}$, we define a TID database (D_z, π_z)

[1] This notion of safety is distinct from the "usual" notion of query safety [1] that ensures domain independence.

as follows: D_z contains all the facts of D, and for an exogenous fact f of D we define $\pi_z(f) \stackrel{\text{def}}{=} 1$ while for an endogenous fact f of D we define $\pi_z(f) \stackrel{\text{def}}{=} \frac{z}{1+z}$. It is then routine to show that the following relation holds:

$$(1+z)^n \Pr(q, (D_z, \pi_z)) = \sum_{i=0}^{n} z^i \, \#\mathrm{Slices}(q, D_\mathrm{x}, D_\mathrm{n}, i).$$

This suffices to conclude the proof. Indeed, we now call an oracle to $\mathrm{PQE}(q)$ on $n+1$ databases $D_{z_0}, \ldots, D_{z_n}$ for $n+1$ arbitrary distinct values $z_0, \ldots, z_n$, forming a system of linear equations as given by the relation above. Since the corresponding matrix is a Vandermonde with distinct coefficients, it is invertible, so we can compute in polynomial time the value $\#\mathrm{Slices}(q, D_\mathrm{x}, D_\mathrm{n}, k)$. □

A intriguing natural question is whether the converse of Proposition 3.1 is true, that is, whether we also have $\mathrm{PQE}(q) \leq_\mathrm{T}^\mathrm{p} \mathsf{Shapley}(q)$. This is true when q is a self-join–free conjunctive query: indeed, by the results of [20], either q is hierarchical and then both $\mathrm{PQE}(q)$ and $\mathsf{Shapley}(q)$ can be solved in polynomial time (hence $\mathrm{PQE}(q) \leq_\mathrm{T}^\mathrm{p} \mathsf{Shapley}(q)$), or q is not hierarchical and then $\mathsf{Shapley}(q)$ is $\mathrm{FP}^{\#\mathrm{P}}$-hard (which, together with the fact that $\mathrm{PQE}(q) \in \mathrm{FP}^{\#\mathrm{P}}$, implies $\mathrm{PQE}(q) \leq_\mathrm{T}^\mathrm{p} \mathsf{Shapley}(q)$). However, to the best of our knowledge, the existence of such a reduction in the general case is unknown.

Open problem 1. *Do we have* $\mathrm{PQE}(q) \leq_\mathrm{T}^\mathrm{p} \mathsf{Shapley}(q)$ *for every Boolean query* q*?*

Interestingly, we note that this direction is trivial in the setting of SHAP-scores. Indeed, this is directly implied by the *efficiency axiom* of the Shapley value; see, e.g., [2, Lemma 4.2] or [36, Equation 5]. In our case, this axiom only gives us the following equality:

$$\sum_{f \in D_\mathrm{n}} \mathsf{Shapley}(q, D_\mathrm{n}, D_\mathrm{x}, f) \;=\; q(D_\mathrm{n} \cup D_\mathrm{x}) - q(D_\mathrm{x}),$$

which does not seem to help in showing $\mathrm{PQE}(q) \leq_\mathrm{T}^\mathrm{p} \mathsf{Shapley}(q)$.

4 EXACT COMPUTATION THROUGH KNOWLEDGE COMPILATION

Motivated by the connection to PQE that we have seen in Section 3, we now investigate whether an approach using knowledge compilation can be used for computing Shapley values. Indeed, a common method to compute the probability that a probabilistic database (D, π) satisfies a Boolean query q is to first compute the *lineage* of q on D in a formalism from knowledge compilation, and then to use the good properties of said formalism to compute $\Pr(q, (D, \pi))$ in linear time [18, 26, 33]. Recently, Arenas and others [2, 3] showed that this approach is also viable for the notion of SHAP-score used in machine learning, by proving that SHAP-scores can be computed in polynomial time when the models are given as circuits from knowledge compilation. By reusing some of these techniques, we can show that this method can also be used in our setting for computing Shapley values of database facts. Again, to the best of our knowledge, the two results are incomparable, i.e., we are not aware of a reduction in either direction between the two problems. We start by formally defining the notions of lineage and the relevant circuit classes from knowledge compilation.

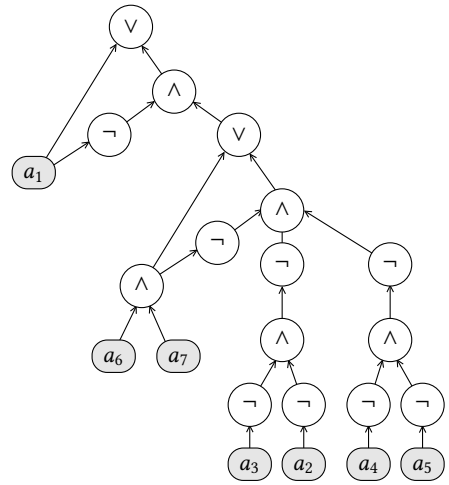


Figure 2: Deterministic and decomposable circuit for $\mathsf{ELin}(q, D_\mathrm{x}, D_\mathrm{n})$ from the running example.

Boolean functions and query lineages. Let X be a finite set of variables. An *assignment* ν *of* X is a subset $\nu \subseteq X$ of X. We denote by 2^X the set of all assignments of X. A *Boolean function* φ over X is a function $\varphi : 2^X \to \{0, 1\}$. An assignment $\nu \subseteq X$ is *satisfying* if $\varphi(\nu) = 1$. We denote by $\mathsf{SAT}(\varphi) \subseteq 2^X$ the set of all satisfying assignments of φ, and by $\#\mathsf{SAT}(\varphi)$ the size of this set. For $k \in \mathbb{N}$, we define $\mathsf{SAT}_k(\varphi) \stackrel{\text{def}}{=} \mathsf{SAT}(\varphi) \cap \{\nu \subseteq X \mid |\nu| = k\}$, that is, the set of satisfying assignments of φ of Hamming weight k, and let $\#\mathsf{SAT}_k(\varphi)$ be the size of this set.

Let q be a Boolean query and D be a database. The *lineage* $\mathsf{Lin}(q, D)$ is the (unique) Boolean function whose variables are the facts of D, and that maps each sub-database $D' \subseteq D$ to $q(D')$. This definition extends straightforwardly to queries with free variables as follows: if $q(\bar{x})$ is a query with free variables $\bar{x}$ and $\bar{t}$ is a tuple of constants of the appropriate size, then $\mathsf{Lin}(q[\bar{x}/\bar{t}]), D)$ is the *lineage for the tuple* $\bar{t}$.

Example 4.1. Consider again the database D and the Boolean query q from Figures 1a and 1c. In Figure 1d, the lineage $\mathsf{Lin}(q, D)$ is represented as a formula in *disjunctive normal form* (DNF).

For our purposes, we will use a refinement of this lineage that accounts for the nature of exogenous tuples; specifically, these tuples should be considered as always being part of the database. Let $D = D_\mathrm{n} \cup D_\mathrm{x}$ be a database with endogenous tuples D_n and exogenous tuples D_x, and let q be a Boolean query. Then the *endogenous lineage* $\mathsf{ELin}(q, D_\mathrm{x}, D_\mathrm{n})$ is the (unique) Boolean function whose variables are D_n and that maps every set E of endogenous facts to $q(D_\mathrm{x} \cup E)$. In other words, $\mathsf{ELin}(q, D_\mathrm{x}, D_\mathrm{n})$ can be obtained from $\mathsf{Lin}(q, D)$ by fixing all variables in D_x to the value 1. Again, we extend this definition to queries with free variables by using the function $\mathsf{ELin}(q[\bar{x}/\bar{t}], D_\mathrm{x}, D_\mathrm{n})$.

Example 4.2. Continuing the previous example, the endogenous lineage $\mathsf{ELin}(q, D_\mathrm{n}, D_\mathrm{x})$ can be represented as a DNF by

$$a_1 \vee (a_2 \wedge a_4) \vee (a_2 \wedge a_5) \vee (a_3 \wedge a_4) \vee (a_3 \wedge a_5) \vee (a_6 \wedge a_7)\,.$$

In the last two examples, lineages were represented with Boolean formulas in DNF. Since a lineage is a Boolean function, it can be represented with any formalism that allows to represent Boolean functions. We next review some classes of circuits from the field of knowledge compilation that will be relevant for our work.

Knowledge compilation classes. Let C be a Boolean circuit, featuring $\wedge$, $\vee$, $\neg$, and variable gates, with the usual semantics.[2] For a gate g of C, we denote by $\mathsf{Vars}(g)$ the set of variables that have a directed path to g. An $\wedge$-gate g of C is *decomposable* if for every two input gates $g_1 \neq g_2$ of g we have $\mathsf{Vars}(g_1) \cap \mathsf{Vars}(g_2) = \emptyset$. We call C *decomposable* if all $\wedge$-gates are. An $\vee$-gate g of C is *deterministic* if the Boolean functions captured by each pair of distinct input gates of g are pairwise disjoint; i.e., no assignment satisfies both. We call C *deterministic* if all $\vee$-gates in it are. A *deterministic and decomposable (d-D [26]) Boolean circuit* is a Boolean circuit that is both deterministic and decomposable. If C is a Boolean circuit we write $\mathsf{Vars}(C)$ to denote the set of variables that appear in it.

Example 4.3. Recall $\mathsf{ELin}(q, D_\mathrm{n}, D_\mathrm{x})$ from Example (4.2) represented as a DNF. Figure 2 depicts a d-D circuit for $\mathsf{ELin}(q, D_\mathrm{n}, D_\mathrm{x})$. The output gate, for instance, is a deterministic $\vee$-gate: indeed, its left child requires a_1 to be 1, whereas its right child requires a_1 to be 0. The right child of the output gate is a decomposable $\wedge$-gate: indeed, for its left child g_1 we have $\mathsf{Vars}(g_1) = \{a_1\}$, whereas for its right child g_2 we have $\mathsf{Vars}(g_2) = \{a_2, a_3, a_4, a_5, a_6, a_7\}$, and these are indeed disjoint. The reader can easily check that all other $\vee$-gates are deterministic, and that all other $\wedge$-gates are decomposable.

4.1 Algorithm

The main result of this section is then the following.

Proposition 4.4. *Given as input a deterministic and decomposable circuit C representing $\mathsf{ELin}(q, D_\mathrm{n}, D_\mathrm{x})$ for a database $D = D_\mathrm{x} \cup D_\mathrm{n}$ and Boolean query q, and an endogenous fact $f \in D_\mathrm{n}$, we can compute in polynomial time (in $|C|$) the value $\mathsf{Shapley}(q, D_\mathrm{n}, D_\mathrm{x}, f)$.*

Next, we prove Proposition 4.4 and present the algorithm, and then explain in Section 4.2 the architecture of the implementation.

Proof of Proposition 4.4. Let C be a deterministic and decomposable circuit representing $\mathsf{ELin}(q, D_\mathrm{n}, D_\mathrm{x})$, and let $f \in D_\mathrm{n}$. First, we complete the circuit C so that all variables of D_n appear in C. Indeed, it could be the case that $\mathsf{Vars}(C) \subsetneq D_\mathrm{n}$: this happens for instance with the deterministic and decomposable circuit in Figure 2, where the endogenous fact a_8 does not appear in the circuit. To do this, we conjunct C with the conjunction $\bigwedge_{f' \in D_\mathrm{n} \setminus \mathsf{Vars}(C)} (f' \vee \neg f')$. Note that this does not change the semantics of the circuit (as this conjunction always evaluates to 1) and that the resulting circuit is still deterministic and decomposable. Now, let C_1 (resp., C_2) be the Boolean circuit obtained from C by replacing all variable gates corresponding to the fact f by a constant 1-gate (resp., by a constant 0-gate). Observe then that the variables of C_1 and C_2 are exactly $D_\mathrm{n} \setminus \{f\}$, and moreover that C_1 and C_2 are still deterministic and decomposable. By definition of the endogenous lineage, we can rewrite Equation (2) into the following.

$$\mathsf{Shapley}(q, D_\mathrm{n}, D_\mathrm{x}, f) = \sum_{k=0}^{|D_\mathrm{n}|-1} \frac{k!(|D_\mathrm{n}| - k - 1)}{|D_\mathrm{n}|} \big(\#\mathsf{SAT}_k(C_1) - \#\mathsf{SAT}_k(C_2) \big). \tag{3}$$

Proposition 4.4 will thus directly follow from the next lemma.

Lemma 4.5. *Given as input a deterministic and decomposable Boolean circuit C and an integer $k \in \{0, \ldots, |\mathsf{Vars}(C)|\}$, we can compute in polynomial time the quantity $\#\mathsf{SAT}_k(C)$.*

Proof. Our proof is similar to that of [3, Section 3.2]. Let $X \stackrel{\text{def}}{=} \mathsf{Vars}(C)$ and $n \stackrel{\text{def}}{=} |X|$. First of all, we preprocess C so that the fanin of every $\vee$- and $\wedge$-gate is exactly 0 or 2; this can simply be done by rewriting every $\wedge$-gate of fanin $m > 2$ with $m - 1$ $\wedge$-gates of fanin 2 (same for $\vee$-gates), and adding a constant gate of the appropriate type to every $\vee$- and $\wedge$-gate of fan-in 1. We then compute, for every gate g of C, the set of variables $\mathsf{Vars}(g)$ upon which the value of g depends. For a gate g of C, let us denote by φ_g the Boolean function over the variables $\mathsf{Vars}(g)$ that is represented by this gate. For a gate g and an integer $\ell \in \{0, \ldots, |\mathsf{Vars}(g)|\}$, we define $\alpha_g^\ell \stackrel{\text{def}}{=} \#\mathsf{SAT}_\ell(\varphi_g)$, i.e. the number of assignments of size ℓ to $\mathsf{Vars}(g)$ that satisfy φ_g. We will show how to compute all the values α_g^ℓ for every gate g of C and $\ell \in \{0, \ldots, |\mathsf{Vars}(g)|\}$ in polynomial time. This will conclude the proof since, for the output gate g_output of C, we have that $\alpha_{g_\mathsf{output}}^k = \#\mathsf{SAT}_k(f)$. We will need the following notation: for two disjoint sets of variables X_1, X_2 and two subsets $S_1 \subseteq 2^{X_1}$, $S_2 \subseteq 2^{X_2}$ of assignments to X_1 and X_2, we denote by $S_1 \otimes S_2 \subseteq 2^{X_1 \cup X_2}$ the set of assignments of $X_1 \cup X_2$ defined by $S_1 \otimes S_2 \stackrel{\text{def}}{=} \{\nu_1 \cup \nu_2 \mid \nu_1 \in S_2, \nu_2 \in S_2\}$. We next show how to compute the values α_g^ℓ by bottom-up induction on C.

Variable gate. If g is a variable gate corresponding to some variable y, then $\mathsf{Vars}(g) = \{y\}$. Then, α_g^0 is 0 and α_g^1 is 1.

$\neg$-gate. If g is a $\neg$-gate with input gate g', then $\alpha_g^\ell = \binom{|\mathsf{Vars}(g)|}{l} - \alpha_{g'}^\ell$ for every $\ell \in \{0, \ldots, |\mathsf{Vars}(g)|\}$.

Deterministic $\vee$-gate. If g is a deterministic $\vee$-gate with no input then φ_g is the Boolean function on variables $\mathsf{Vars}(g) = \emptyset$ that is always false, hence $\alpha_g^0 = 0$. Otherwise g has exactly two input gates; let us denote them g_1 and g_2. Observe that $\mathsf{Vars}(g) = \mathsf{Vars}(g_1) \cup \mathsf{Vars}(g_2)$ by definition. Define $S_1 \stackrel{\text{def}}{=} \mathsf{Vars}(g_2) \setminus \mathsf{Vars}(g_1)$ and similarly $S_2 \stackrel{\text{def}}{=} \mathsf{Vars}(g_1) \setminus \mathsf{Vars}(g_2)$. Since g is deterministic, we have:

$$\mathsf{SAT}(\varphi_g) = (\mathsf{SAT}(\varphi_{g_1}) \otimes 2^{S_1}) \cup (\mathsf{SAT}(\varphi_{g_2}) \otimes 2^{S_2})$$

with the union being disjoint. By intersecting with the assignments of $\mathsf{Vars}(g)$ of size ℓ, we obtain:

$$\begin{aligned}\mathsf{SAT}_\ell(\varphi_g) = &\big[(\mathsf{SAT}(\varphi_{g_1}) \otimes 2^{S_1}) \cap \{\nu \subseteq \mathsf{Vars}(g) \mid |\nu| = \ell\}\big] \\ &\cup \big[(\mathsf{SAT}(\varphi_{g_2}) \otimes 2^{S_2}) \cap \{\nu \subseteq \mathsf{Vars}(g) \mid |\nu| = \ell\}\big]\end{aligned}$$

with again the middle union being disjoint, therefore:

$$\begin{aligned}\#\mathsf{SAT}_\ell(\varphi_g) = &|(\mathsf{SAT}(\varphi_{g_1}) \otimes 2^{S_1}) \cap \{\nu \subseteq \mathsf{Vars}(g) \mid |\nu| = \ell\}| \\ &+ |(\mathsf{SAT}(\varphi_{g_2}) \otimes 2^{S_2}) \cap \{\nu \subseteq \mathsf{Vars}(g) \mid |\nu| = \ell\}|\end{aligned}$$

[2] We allow unbounded-fanin $\wedge$- and $\vee$-gates, and also allow constant 1-gates and constant 0-gates as, respectively, $\wedge$-gates with no input and $\vee$-gates with no inputs.

We now explain how to compute the first term, that is, $|(\mathsf{SAT}(\varphi_{g_1}) \otimes 2^{S_1}) \cap \{\nu \subseteq \mathsf{Vars}(g) \mid |\nu| = \ell\}|$; the second term is similar. This is equal[3] to

$$\sum_{i=\max(0,\,\ell-|S_1|)}^{\min(\ell,\,|\mathsf{Vars}(g_1)|)} \alpha_{g_1}^{i} \times \binom{|S_1|}{\ell-i}.$$

Decomposable $\wedge$-gate. If g is a decomposable $\wedge$-gate with no input then φ_g is the Boolean function on variables $\mathsf{Vars}(g) = \emptyset$ that is always true, hence $\alpha_g^0 = 1$. Otherwise, let g_1 and g_2 be the two input gates of g. Since g is decomposable we have $\mathsf{Vars}(g) = \mathsf{Vars}(g_1) \cup \mathsf{Vars}(g_2)$ with the union being disjoint. But then we have:

$$\mathsf{SAT}(\varphi_g) = \mathsf{SAT}(\varphi_{g_1}) \otimes \mathsf{SAT}(\varphi_{g_2})$$

We now intersect with the set of assignments of $\mathsf{Vars}(g)$ of size l to obtain

$$\alpha_g^{\ell} = \#\mathsf{SAT}_{\ell}(\varphi_g) = \sum_{i=\max(0,\,\ell-|\mathsf{Vars}(g_2)|)}^{\min(\ell,\,|\mathsf{Vars}(g_1)|)} \alpha_{g_1}^{i} \times \alpha_{g_2}^{\ell-i}$$

This concludes the proof of the lemma, as well as the proof of Proposition 4.4. □

Algorithm. Algorithm 1 depicts the solution underlying Proposition 4.4. The subroutine ComputeAll#SAT$_k$ takes as input a d-D circuit C and outputs all the values $\#\mathsf{SAT}_0(C), \ldots, \#\mathsf{SAT}_{|\mathsf{Vars}(C)|}(C)$. This function computes values α_g^{ℓ} by bottom-up induction on C just as in the proof of Lemma 4.5, by using the appropriate equations depending on the type of each gate. Then, Lines 1–5 in the algorithm simply follow the part of the proof that starts at the beginning of this section until Lemma 4.5. For instance, the returned value on Line 5 corresponds to Equation (3). A quick inspection of Algorithm 1 reveals that, if one ignores the complexity of performing arithmetic operations (i.e., considering that additions and multiplications take constant time), the running time is $O(|C| \cdot |D_{\mathrm{n}}|^2)$. If one wishes to compute the Shapley value of every endogenous fact (as will be done in the experiments), then the overall complexity is $O(|C| \cdot |D_{\mathrm{n}}|^3)$. Last, we point out that, in the case of non-Boolean queries, this cost is incurred for each potential output tuple that one wants to analyze.

Algorithm 1: Shapley values from deterministic and decomposable Boolean circuits

Input : Deterministic and decomposable Boolean circuit C with output gate g_{output} representing $\mathsf{ELin}(q, D_{\mathrm{x}}, D_{\mathrm{n}})$ and an endogenous fact $f \in D_{\mathrm{n}}$.

Output: The value $\mathsf{Shapley}(q, D_{\mathrm{x}}, D_{\mathrm{n}}, f)$.

1 Complete C so that $\mathsf{Vars}(g_{\mathsf{output}}) = D_{\mathrm{n}}$;

2 Compute $C_1 = C[f \to 1]$ and $C_2 = C[f \to 0]$;
`// Partial evaluations of C by setting f to 1 and to 0`

3 $\Gamma = \mathrm{ComputeAll\#SAT}_k(C_1)$; `// As an array`

4 $\Delta = \mathrm{ComputeAll\#SAT}_k(C_2)$; `// As an array`

5 **return** $\displaystyle\sum_{k=0}^{|D_{\mathrm{n}}|-1} \frac{k!\,(|D_{\mathrm{n}}|-k-1)!}{|D_{\mathrm{n}}|!} \cdot (\Gamma[k] - \Delta[k])$;

6 **Def** `ComputeAll#SAT`$_k(C)$**:**

7 Preprocess C so that each $\vee$-gate and $\wedge$-gate has fan-in exactly 0 or 2;

8 Compute the set $\mathsf{Vars}(g)$ for every gate g in C;

9 Compute values α_g^{ℓ} for every gate g in C and $\ell \in \{0, \ldots, |\mathsf{Vars}(g)|\}$ by bottom-up induction on C using the inductive relations from the proof of Lemma 4.5;

10 **return** $[\alpha_{g_{\mathsf{output}}}^{0}, \ldots, \alpha_{g_{\mathsf{output}}}^{|\mathsf{Vars}(C)|}]$;

4.2 Implementation Architecture

In this section, we present our architecture for implementing the knowledge compilation approach over realistic datasets. The relevant parts, for now, are the middle and top part of Figure 3, which we next explain. Given a database $D = D_{\mathrm{x}} \cup D_{\mathrm{n}}$, a query $q(\bar{x})$, a tuple $\bar{t}$ of the same arity as $\bar{x}$, and an endogenous fact $f \in D_{\mathrm{n}}$, we want to compute $\mathsf{Shapley}(q[\bar{x}/\bar{t}], D_{\mathrm{x}}, D_{\mathrm{n}}, f)$. We use two existing tools to help us with this task: ProvSQL [31] and the knowledge compiler c2d [10]. ProvSQL is a tool integrated into PostgreSQL that can perform provenance (lineage) computation in various semirings. For our purposes, a knowledge compiler is a tool that takes as input a Boolean function in CNF and outputs an equivalent Boolean function into another formalism. The target formalism that we will use is the so-called "d-DNNF". A d-DNNF is simply a deterministic and decomposable Boolean circuit such that negation gates are only applied to variables (NNF stands for *negation normal form*).[4]

In our case, we use ProvSQL as follows: we feed it the database D, query $q(\bar{x})$ and tuple $\bar{t}$, and ProvSQL computes $\mathsf{Lin}(q[\bar{x}/\bar{t}], D)$ as a Boolean circuit, called C in Figure 3. We note here that for SPJU queries, C can be computed in polynomial-time data complexity. We then set to 1 all the exogenous facts to obtain a Boolean circuit C' for $\mathsf{ELin}(q[\bar{x}/\bar{t}], D_{\mathrm{x}}, D_{\mathrm{n}})$. Then, ideally, we would like to use the knowledge compiler to transform C' into an equivalent d-DNNF, in order to be able to apply Algorithm 1. Unfortunately, every knowledge compiler that we are aware of takes as input Boolean formulas in conjunctive normal form (CNF), and not arbitrary Boolean circuits. To circumvent it, we use the *Tseytin transformation* [35] to transform the circuit C' into a CNF $\varphi \stackrel{\text{def}}{=} \textsc{Tseytin}(C')$, whose size is linear in that of C'. This CNF φ has the following properties: (1) its variables are the variables of C' plus a set Z of additional variables; (2) for every valuation $\nu \subseteq \mathsf{Vars}(C')$ that satisfies C', there exists exactly one valuation $\nu' \subseteq Z$ such that $\varphi(\nu \cup \nu') = 1$; and (3) for every valuation $\nu \subseteq \mathsf{Vars}(C')$ that does not satisfy C', there is no valuation $\nu' \subseteq Z$ such that $\varphi(\nu \cup \nu') = 1$. We then

[3] This comes from the fact that, for disjoint X_1, X_2 and assignments ν_1 of X_1 and ν_2 of X_2 we have $|\nu_1 \cup \nu_2| = |\nu_1| + |\nu_2|$, and $\mathsf{Vars}(\varphi_{g_1})$ and S_1 are disjoint.

[4] This additional NNF restriction is not important here, but, as far as we know, no knowledge compiler has the more general "deterministic and decomposable circuits" (without NNF) as a target. It is currently unknown whether d-DNNFs and deterministic and decomposable circuits are exponentially separated or not [11, Table 7].

feed φ to the knowledge compiler, which produces a d-DNNF C'' equivalent to φ (the variables of C'' are again $\mathsf{Vars}(C') \cup Z$). We note here that there is no theoretical guarantee that this step is efficient; indeed, the task of transforming a CNF into an equivalent d-D circuit is $\mathrm{FP}^{\#\mathrm{P}}$-hard in general; see Section 6 for an experimental analysis of its tractability in practice. Next, we need to eliminate the additional variables Z in order to be able to apply Algorithm 1. To this end, we use the following Lemma.

Lemma 4.6. *Given as input a d-DNNF C'' that is equivalent to $\textsc{Tseytin}(C')$ for a Boolean circuit C', we can compute in time $O(|C''|)$ a d-DNNF C''' that is equivalent to C' (in particular, the variables of C''' are the same as the variables of C'; in our case, they consist only of endogenous facts).*

Proof sketch. Let Z be the additional variables coming from the Tseytin transformation. First, we remove all the gates of C'' that are not satisfiable, and then we remove all the gates that are not connected to the output gate. Now, let C''' be the circuit that is obtained from this intermediate circuit by replacing every literal z or $\neg z$ for $z \in Z$ by a constant 1-gate. We return C'''. The proof that this algorithm is correct uses the properties (1–3) of the Tseytin transformation, and is omitted due to lack of space. □

Using this lemma, we obtain a d-DNNF C''' for the endogenous lineage $\mathsf{ELin}(q[\bar{x}/\bar{t}], D_{\mathrm{x}}, D_{\mathrm{n}})$, to which we can finally apply Algorithm 1 to obtain the value $\mathsf{Shapley}(q[\bar{x}/\bar{t}], D_{\mathrm{x}}, D_{\mathrm{n}}, f)$.

5 INEXACT COMPUTATION

As we will show in the experimental section, the exact computation algorithm that we have proposed performs well in most cases but is too costly in others. In the latter cases, we may wish to resort to methods that do not necessarily compute exact Shapley values, if their results still typically suffice to determine the *order* of facts according to their Shapley contribution. In this section we propose *CNF Proxy*, a heuristic solution that is very efficient, and we will experimentally show that the ranking of facts based on CNF Proxy tends to match the ranking based on the exact Shapley values.

At a high level, CNF Proxy is based on the observation that having a high Shapley score is correlated (albeit in a complex manner) with (1) appearing many times in the provenance and (2) having few "alternatives," that is, facts that could compensate for the absence of the given fact. The first factor (number of occurrences) may be directly read from the CNF obtained by applying the Tseytin transformation to the provenance circuit (C' in Figure 3). It is also easy to read from the CNF partial information about the second factor (number of alternatives), namely the number of alternatives in each clause (ignoring intricate dependencies between clauses). Next, we present the details of CNF Proxy.

We will start with an auxiliary definition, denoting the Shapley value of a general function $h : 2^X \to \mathbb{R}$ and a variable $x \in X$ as

$$\mathsf{Shapley}(h, x) \stackrel{\text{def}}{=} \sum_{S \subseteq X \setminus \{x\}} \frac{|S|!(|X| - |S| - 1)!}{|X|!} \big(h(S \cup \{x\}) - h(S)\big).$$

Naturally, if $h = \mathsf{ELin}(q, D_{\mathrm{n}}, D_{\mathrm{x}})$, i.e., the endogenous lineage, and x is a fact in D_{n}, then $\mathsf{Shapley}(h, x) = \mathsf{Shapley}(q, D_{\mathrm{n}}, D_{\mathrm{x}}, x)$.

Algorithm 2: CNF Proxy

Input : CNF φ and a set of endogenous facts D_{n}.
Output: The value $\mathsf{Shapley}(\widetilde{\varphi}, x)$ for each $x \in D_{\mathrm{n}}$.

```
n ← |φ.clauses()|;
v ← 0^{|D_n|};                                   // As an array
for ψ ∈ φ.clauses() do
    L ← ψ.literals();
    m ← |L|;
    pos ← {ℓ ∈ L | ℓ is positive};
    neg ← {ℓ ∈ L | ℓ is negative};
    for ℓ ∈ pos ∩ D_n do
        v[ℓ.var()] ← v[ℓ.var()] + 1/(nm·C(m−1, |neg|));
    end
    for ℓ ∈ neg ∩ D_n do
        v[ℓ.var()] ← v[ℓ.var()] − 1/(nm·C(m−1, |pos|));
    end
end
return v
```

Now, note that for a CNF formula $\varphi = \bigwedge_{i=1}^{n} \psi_i$ (where each ψ_i is a disjunction of literals) and an assignment ν it holds that $\varphi(\nu) = \prod_{i=1}^{n} \psi_i(\nu)$. Instead of calculating the Shapley values of a CNF formula φ (which may be a hard problem), CNF Proxy computes Shapley values with respect to a *proxy function*, denoted $\widetilde{\varphi}$. The proxy function of φ is defined as the sum (instead of the product) of the clauses of φ, i.e., $\widetilde{\varphi}(\nu) \stackrel{\text{def}}{=} \sum_{i=1}^{n} \frac{1}{n}\psi_i(\nu)$. Intuitively, a fact that appears in many clauses of the CNF φ will occur in many summands of $\widetilde{\varphi}$, and when we compute Shapley values with respect to $\widetilde{\varphi}$, the number of alternatives in each clause will be reflected in decreased value of the respective summands.

Example 5.1. Consider the CNF formula $\varphi = (x_1 \vee x_2) \wedge (x_1 \vee x_3 \vee x_4)$. The Shapley values of x_1, x_2, x_3, x_4 are $\frac{7}{12}, \frac{3}{12}, \frac{1}{12}, \frac{1}{12}$ respectively. Note that x_1 has the highest influence, which intuitively may be attributed to its appearance in two clauses whereas each other variable appears only in a single clause. The variable x_2 has more influence than x_3 and x_4, intuitively since it has less alternatives. These comparative features are preserved in $\widetilde{\varphi} = (x_1 \vee x_2) + (x_1 \vee x_3 \vee x_4)$, and indeed the Shapley values of x_1, x_2, x_3, x_4 with respect to $\widetilde{\varphi}$ are $\frac{5}{6}, \frac{1}{2}, \frac{1}{3}, \frac{1}{3}$ respectively. Observe that although the values assigned to the variables are very different from their actual Shapley values, their order remains intact in this case.

Due to the linearity of Shapley values, their computation with respect to $\widetilde{\varphi}$ is much more efficient, as implied by the following lemma (whose proof we omit for space reasons):

Lemma 5.2. *Let $h = \sum_{i=1}^{n} \frac{1}{n}\psi_i$, where each ψ_i is a Boolean function representing a disjunction of literals. Without loss of generality let us assume that for each ψ_i there is no variable that appears in more than one literal of ψ_i. Denote by a_i and b_i the number of positive and negative literals in ψ_i respectively. Then, for every variable x, it holds*

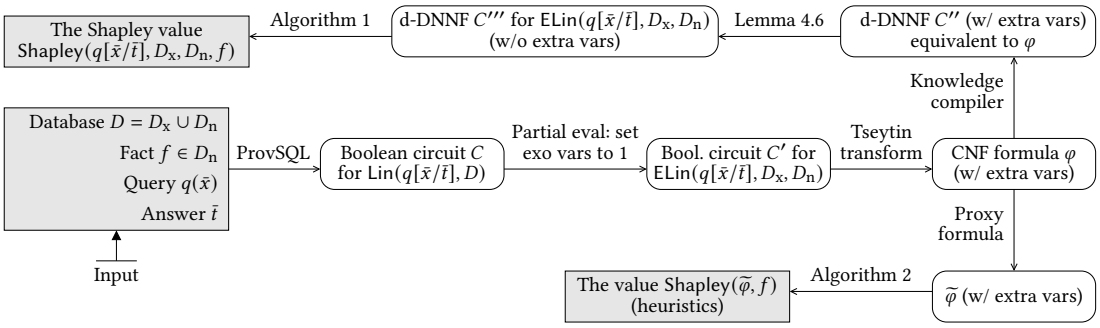


Figure 3: Our implementation architecture.

that $\mathsf{Shapley}(h, x) = \frac{1}{n}\sum_{i=1}^{n}\Phi(\psi_i, x)$, *where*

$$\Phi(\psi_i, x) = \begin{cases} \frac{1}{(a_i+b_i)\cdot\binom{a_i+b_i-1}{b_i}} & \textit{if } x \textit{ appears in } \psi_i \textit{ in positive form;} \\ \frac{-1}{(a_i+b_i)\cdot\binom{a_i+b_i-1}{a_i}} & \textit{if } x \textit{ appears in } \psi_i \textit{ in negative form;} \\ 0 & \textit{otherwise.} \end{cases}$$

Algorithm 2 then describes the operation of CNF Proxy, computing Shapley values of $\widetilde{\varphi}$ according to Lemma 5.2. The input to the algorithm is a CNF formula φ and the set of endogenous facts D_n. In Line 1, CNF Proxy counts the number n of clauses of φ, and in Line 2 it initializes the contribution of every variable of φ to zero. Then, it iterates over the clauses (Line 3). For each clause, it counts the number m of literals, and identifies the set of positive and negative literals, *pos* and *neg* respectively (Lines 4–7). In Lines 9 and 12 we add (resp., subtract) quantities for each variable in a positive (resp., negative) literal according to Lemma 5.2 (note that m corresponds to $a_i + b_i$ as denoted in the lemma). Finally, in Line 15 the algorithm returns the contribution value of each fact $x \in D_n$ based on $\mathsf{Shapley}(\widetilde{\varphi}, x)$. Observe that CNF Proxy runs in linear time in the size of φ (which itself, being the Tseytin transformation of the provenance circuit C' from Figure 3, is linear in C').

Example 5.3. Recall the queries q_1, q_2, q and their lineages depicted in Figure 1. Using endogenous lineages, we have:

- $\mathsf{ELin}(q_1, D_n, D_x) = a_1$
- $\mathsf{ELin}(q_2, D_n, D_x) = (a_2 \wedge a_4) \vee (a_2 \wedge a_5) \vee (a_3 \wedge a_4) \vee (a_3 \wedge a_5) \vee (a_6 \wedge a_7)$
- $\mathsf{ELin}(q, D_n, D_x) = a_1 \vee (a_2 \wedge a_4) \vee (a_2 \wedge a_5) \vee (a_3 \wedge a_4) \vee (a_3 \wedge a_5) \vee (a_6 \wedge a_7)$

The lineage of q_1 is a CNF with a single variable, thus the contribution of a_1 to q_1 as computed by Algorithm 2 is 1, which is indeed equal to $\mathsf{Shapley}(q_1, D_n, D_x, a_1)$. Applying the Tseytin transformation to the lineage of q_2 introduces 6 new variables ($\{z_i\}_{i=1}^{6}$) and results in the following equisatisfiable CNF:

$(z_1) \wedge (z_1 \vee \overline{z_2}) \wedge (z_1 \vee \overline{z_3}) \wedge (z_1 \vee \overline{z_4}) \wedge (z_1 \vee \overline{z_5}) \wedge (z_1 \vee \overline{z_6}) \wedge$
$(\overline{z_1} \vee z_2 \vee z_3 \vee z_4 \vee z_5 \vee z_6) \wedge$
$(\overline{z_2} \vee a_2) \wedge (\overline{z_2} \vee a_4) \wedge (z_2 \vee \overline{a_2} \vee \overline{a_4}) \wedge (\overline{z_3} \vee a_2) \wedge (\overline{z_3} \vee a_5) \wedge (z_3 \vee \overline{a_2} \vee \overline{a_5}) \wedge$
$(\overline{z_4} \vee a_3) \wedge (\overline{z_4} \vee a_4) \wedge (z_4 \vee \overline{a_3} \vee \overline{a_4}) \wedge (\overline{z_5} \vee a_3) \wedge (\overline{x_5} \vee a_5) \wedge (z_5 \vee \overline{a_3} \vee \overline{a_5}) \wedge$
$(\overline{z_6} \vee a_6) \wedge (\overline{z_6} \vee a_7) \wedge (z_6 \vee \overline{a_6} \vee \overline{a_7})$

Algorithm 2 iterates over the above clauses, and computes the contribution of the endogenous facts D_n over the proxy function. Note that the facts D_n appear in clauses of two forms. The first form is $(\overline{z_j} \vee a_i)$; appearance in this type of clause adds $\frac{1}{22\cdot 2\cdot\binom{1}{1}} = \frac{1}{44}$ to the contribution of a_i. The second form is $(z_j \vee \overline{a_i} \vee \overline{a_h})$, which adds $\frac{-1}{22\cdot 3\cdot\binom{2}{1}} = \frac{-1}{132}$. Note that each of a_2, a_3, a_4, a_5 has two appearances in clauses of the first form, and one appearance in clauses of the second form. Thus, according to Algorithm 2 the contribution of a_2, a_3, a_4, a_5 is $\frac{5}{132} \approx 0.038$. In contrast, a_6 and a_7 each have a single appearance in a clause of the first form and a single appearance in a clause of the second form, thus their contribution is $\frac{1}{66} \approx 0.015$. We note that the values calculated by Algorithm 2 are very different from the actual Shapley values, as $\mathsf{Shapley}(q_2, D_n, D_x, a_i) = \frac{11}{60} \approx 0.183$ for $a_i \in \{a_2, a_3, a_4, a_5\}$ and $\mathsf{Shapley}(q_2, D_n, D_x, a_i) = \frac{2}{15} \approx 0.133$ for $a_i \in \{a_6, a_7\}$. However, the facts a_2, a_3, a_4, a_5, are correctly determined to be more influential than a_6 and a_7.

Our experimental evaluation indicates that in most cases, the ordering of facts according to the values assigned to them by Algorithm 2 agrees with the order obtained by using the actual Shapley values. There is however no theoretical guarantee that this will always be the case, as shown by the following example.

Example 5.4. Applying the Tseytin transformation over the lineage of q will result in a CNF similar to that obtained for q_2, with a new variable (z_7) and the new clauses $(z_1 \vee \overline{z_7}) \wedge (z_7 \vee \overline{a_1}) \wedge (\overline{z_7} \vee a_1)$. In addition, the disjunct $\vee z_7$ will be added to the clause $(\overline{z_1} \vee z_2 \vee z_3 \vee z_4 \vee z_5 \vee z_6)$. Similarly to the case of q_2, the contributions of a_2, a_3, a_4, and a_5 are correctly determined to be larger than those of a_6 and a_7. As for a_1, its contribution according to Algorithm 2 is 0 while in fact it is the most influential fact.

6 EXPERIMENTS

Our system is implemented in Python 3.6 and using the PostgreSQL 11.10 database engine, and the experiments were performed on a Linux Debian 14.04 machine with 1TB of RAM and an Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz processor. ProvSQL [31] was used to capture the provenance. For knowledge compilation we have used the c2d compiler [9, 10]. The source code of our implementation is available in [13].

Since no standard benchmark for our problem exists, we have created such a benchmark of 40 queries over the TPC-H (1.4GB) and IMDB (1.2GB) databases.The TPC-H queries are based on the ones in [34], where we have only removed nested queries (which ProvSQL does not handle) and aggregation operations (for which

provenance is not Boolean). The queries for the IMDB database are based on the join queries in [19], where for each query we have added a (last) projection operation over one of the join attributes to make provenance more complex and thus more challenging for our algorithms. The resulting queries are quite complex: in particular, only 4 out of the 40 are hierarchical. See [13] for details of the obtained queries.

6.1 Exact computation

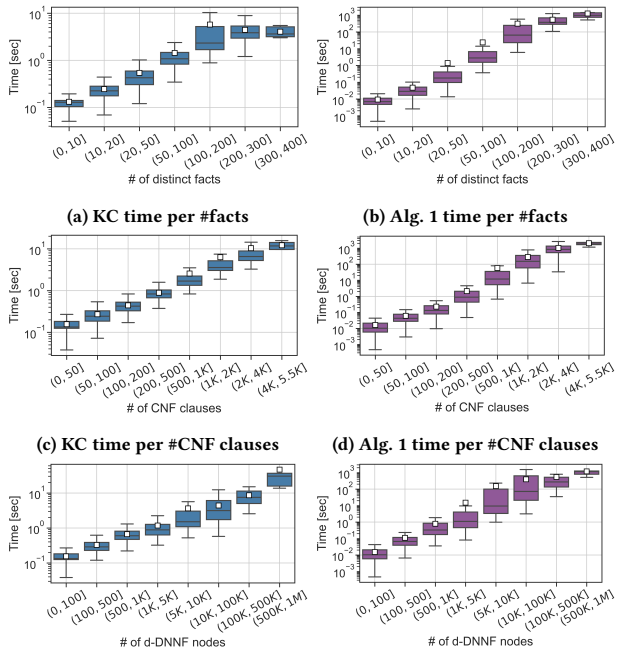
We have evaluated our solution for exact Shapley computation, presented in Section 4.2, on each of the 40 queries. In total, we have obtained 95,803 output tuples along with their provenance expressions (computed with ProvSQL). We have then transformed each provenance expression into a d-DNNF structure using the c2d [9, 10] knowledge compiler. In this experiment, for both the knowledge compilation (KC) and Shapley evaluation steps we have set a timeout of one hour (in fact, as we show below, a much shorter timeout of 2.5 seconds typically suffices). In case the compilation completed successfully within this timeframe, we have computed the Shapley values using Algorithm 1. Table 1 presents the execution times of our solution for 16 representative queries; next we will overview different aspects of the results.

Success rate. We report here the rate of successful executions. The IMDB queries resulted in 95,636 output tuples; the KC step completed successfully for 95,599 out of them, where all 37 failures were the result of insufficient memory. For each of the IMDB output tuples that were successfully compiled into a d-DNNF, we have executed Algorithm 1; only a single execution failed in this step (due to a timeout of one hour). Overall, the exact computation of Shapley values was successful for 95,598 out of the 95,636 IMDB output tuples (i.e., 99.96% success rate). The TPC-H queries resulted in 167 output tuples; the KC step has completed successfully for 141 out of them, again all 26 failures were the result of insufficient memory. For all TPC-H outputs that compiled successfully Algorithm 1 was successful, yielding an overall 84.43% success rate.

Execution time. For each of the 40 queries, we have measured the execution time of each step of the computation. First, we have measured (in the column "Execution time") the execution time in PostgreSQL, which includes provenance generation for every output tuple using ProvSQL. Then, for each tuple $\bar{t}$ in the output of the query, we have measured the KC execution time and the execution time of Algorithm 1 to compute the contribution of all input facts with respect to the output tuple $\bar{t}$. For the latter two algorithms, the execution times varied significantly for the different output tuples, and thus we report the execution times for different percentiles (mean, p25, p50, p75 and p99). Observe that the computation is typically efficient; outliers include q11d for which the execution time of Algorithm 1 was over 96 seconds in average.

Figure 4 depicts the running time of the KC step and of the computation of Shapley values from the d-DNNF as a function of different features of the provenance, such as the number of facts appearing in the provenance, the number of clauses in its CNF representation, and the number of gates in its d-DNNF representation.

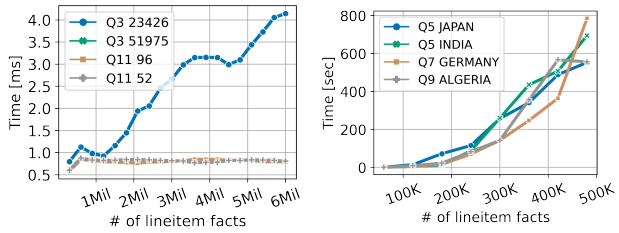
Scalability. To evaluate the scalability of Algorithm 1 we have further looked at different scales of the TPC-H database. Figure 5



(a) KC time per #facts
(b) Alg. 1 time per #facts
(c) KC time per #CNF clauses
(d) Alg. 1 time per #CNF clauses
(e) KC time per d-DNNF size
(f) Alg. 1 time per d-DNNF size

Figure 4: Running times of knowledge compilation and computation of Shapley values from the d-DNNF as function of the number of distinct facts, CNF clauses, and d-DNNF size

depicts the running time of Algorithm 1 over 8 representative query outputs as a function of the number of facts in the LINEITEM table. Figure 5a depicts 4 representative query outputs, for which the running time takes a few milliseconds, e.g., for "Q3 23426" (result 23426 of query $Q3$) and the full TPC-H dataset, the computation of Shapley values for all relevant input facts complete in 4.3ms. In contrast, Figure 5b depicts 4 query outputs for which the exact computation failed to complete over the full TPC-H database. We observe that the algorithm does succeed in these cases if we execute the queries over subsets of the input database, though its execution time may still be high: e.g., if we take a "slice" of the LINEITEM table consisting of 480,097 facts, then computation of the contribution of all input facts w.r.t. "Q9 ALGERIA" takes 556sec.



(a) Representative outputs
(b) "Difficult" outputs

Figure 5: Alg. 1 running time for various TPC-H query outputs as function of table LINEITEM size

Table 1: Statistics on the exact computation of Shapley values for 16 representative queries

Dataset	Query	#Joined tables	#Filter conditions	Execution time [sec]	#Output tuples	Success rate	KC execution times [sec]					Alg. 1 execution times [sec]				
							Mean	p25	p50	p75	p99	Mean	p25	p50	p75	p99
TPC-H	3	3	5	20980.71	100	100%	0.06	0.04	0.07	0.08	0.13	0.00	0.00	0.00	0.00	0.01
	5	6	9	48.67	5	0%	-	-	-	-	-	-	-	-	-	-
	7	6	8	30.57	4	0%	-	-	-	-	-	-	-	-	-	-
	10	4	6	4.72	10	100%	0.14	0.13	0.13	0.14	0.19	0.01	0.01	0.01	0.01	0.01
	11	6	7	0.13	10	100%	0.09	0.08	0.08	0.08	0.13	0.01	0.00	0.00	0.01	0.03
	16	3	5	1.25	10	100%	0.26	0.13	0.18	0.33	0.57	0.18	0.01	0.03	0.29	0.88
	18	4	3	37.31	10	100%	0.13	0.08	0.08	0.18	0.23	0.01	0.00	0.01	0.01	0.03
	19	2	21	2.04	1	100%	1.20	1.20	1.20	1.20	1.20	156.06	156.06	156.06	156.06	156.06
IMDB	1a	5	10	0.25	35	100%	0.17	0.08	0.08	0.13	2.10	5.92	0.01	0.01	0.01	206.72
	6b	5	8	2.61	1	100%	0.44	0.44	0.44	0.44	0.44	0.08	0.08	0.08	0.08	0.08
	7c	8	21	77.33	2415	99%	0.82	0.18	0.28	0.63	9.44	24.28	0.02	0.05	0.39	787.12
	8d	7	10	145.10	44517	99.9%	0.26	0.13	0.18	0.29	1.09	0.36	0.01	0.02	0.05	2.28
	11a	8	18	3.20	10	100%	0.75	0.18	0.48	1.14	2.15	23.34	0.01	0.13	1.27	151.99
	11d	8	16	56.99	210	98.1%	0.99	0.19	0.32	0.78	5.97	96.19	0.02	0.06	0.48	1650.72
	13c	9	19	2.44	14	100%	0.22	0.13	0.18	0.28	0.53	0.02	0.01	0.01	0.01	0.06
	15d	9	18	24.25	207	97.6%	1.89	0.24	0.48	1.28	10.01	70.05	0.06	0.30	3.52	1821.13
	16a	8	15	5.56	173	100%	0.18	0.13	0.14	0.20	0.53	0.02	0.01	0.01	0.02	0.12

6.2 Inexact computation

As observed above, computing exact Shapley values using our solution for a given output tuple is typically fast, but may be costly or even fail in some cases. In this section we evaluate inexact computation alternatives.

Algorithms. We compare three algorithms: CNF Proxy (Section 5) and two existing baselines: Monte Carlo, and Kernel SHAP.

Monte Carlo. This is a well-known sampling algorithm [23] for approximating Shapley values in general. To employ the Monte Carlo algorithm in our setting, we feed it a provenance expression h containing n distinct input facts, and a budget of $r \cdot n$ samples, for some $r \in \mathbb{Z}_+$. The Shapley value of each fact f is approximated by sampling r permutations $(\pi_1, \ldots, \pi_r)$ of the input facts, and then outputting $\frac{1}{r} \cdot \sum_{i=1}^{r} \Big(h(S_{\pi_i,<f} \cup \{f\}) - h(S_{\pi_i,<f})\Big)$, where $S_{\pi_i,<f}$ is the coalition of all facts preceding f in the permutation π_i.

Kernel SHAP. Lundberg and Lee [22] have defined the notion of SHAP values in the context of ML explainability. Given a function $h : \mathbb{R}^d \to \mathbb{R}$ (the model whose decisions we want to explain), a probability distribution $\mathcal{D}$ on $\mathbb{R}^d$ (the inputs), and an input vector $\bar{e} \in \mathbb{R}^d$, SHAP values were defined to measure the contribution of $\bar{e}$'s features to the outcome $h(\bar{e})$. To overcome the issue that h does not operate over subsets of features, the notion of SHAP-score has been defined as follows

$$\mathsf{SHAP}(h, \bar{e}, x) \stackrel{\text{def}}{=} \sum_{S \subseteq X \setminus \{x\}} \frac{|S|!(|X| - |S| - 1)!}{|X|!} (h_{\bar{e}}(S \cup \{x\}) - h_{\bar{e}}(S)),$$

where X is the set of d features, x is a specific feature whose contribution we wish to assess, and $h_{\bar{e}} : 2^X \to \mathbb{R}$ is defined by $h_{\bar{e}}(S) \stackrel{\text{def}}{=} \mathbb{E}_{\bar{z} \sim \mathcal{D}}[h(\bar{z}) \mid \bar{e}_S = \bar{z}_S]$, where $\bar{e}_S$ and $\bar{z}_S$ denote the vectors $\bar{e}$ and $\bar{z}$ restricted to the features in S.

In [22] the authors have proposed a method for approximating SHAP values, called Kernel SHAP. Kernel SHAP assumes feature independence and estimates the probability by multiplying the marginal distributions $\prod_{i \notin S} \Pr(z_i)$. The marginal probabilities $\Pr(z_i)$ in turn are estimated from a background data T. To approximate SHAP values, Kernel SHAP then samples m coalitions $S_1, \ldots, S_m$ of features, and trains a linear model $g : 2^X \to \mathbb{R}$ by minimizing the weighted loss $\sum_{i=1}^{m} w_i \cdot (g(S_i) - \hat{h}_{\bar{e}}(S_i))^2$, where w_i is proportional to the size of S_i and $\hat{h}_{\bar{e}}$ is the estimation of $h_{\bar{e}}$ using the feature independence assumption. The coefficient associated with a feature x in the trained model g is the approximated SHAP value of x.

Table 2: Median (resp., mean) performance. Monte Carlo and Kernel SHAP use $50 \cdot$ #facts samples

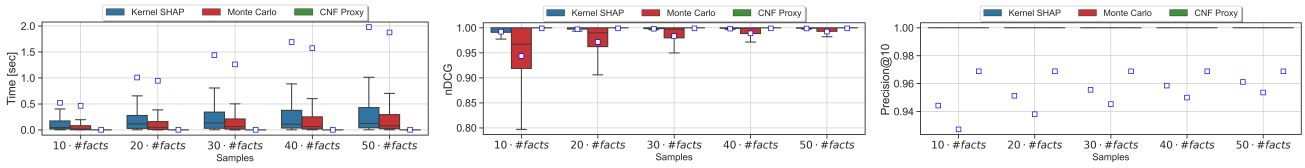
	Monte Carlo	Kernel SHAP	CNF Proxy
Execution time	0.079 (1.875)	0.127 (1.978)	**7e-4 (0.002)**
L1	0.439 (0.448)	**0.110 (0.109)**	0.317 (0.315)
L2	0.03 (0.034)	**0.001 (0.002)**	0.010 (0.014)
nDCG	1.0 (0.992)	1.0 (0.998)	**1.0 (0.999)**
Precision@5	1.0 (0.955)	1.0 (0.961)	**1.0 (0.989)**
Precision@10	1.0 (0.953)	1.0 (0.961)	**1.0 (0.968)**

We adapt Kernel SHAP to our setting and use it to approximate the Shapley values of facts, as follows. The features are the input facts of the database, we set h to be the Boolean function representing the (endogenous) provenance; the vector of interest, $\bar{e}$, has the value 1 for all facts, and the background data T contains a single example with value 0 in all its entries. Note that for such $\bar{e}$ and T, Kernel SHAP estimates $h_{\bar{e}}(S)$ as the result of applying h to a vector with ones in S features and zeros in the remaining entries.

The input of both Monte Carlo and Kernel SHAP includes a budget of m samples; for provenance expressions containing n distinct facts we have experimented with $m \in \{10n, 20n, 30n, 40n, 50n\}$.

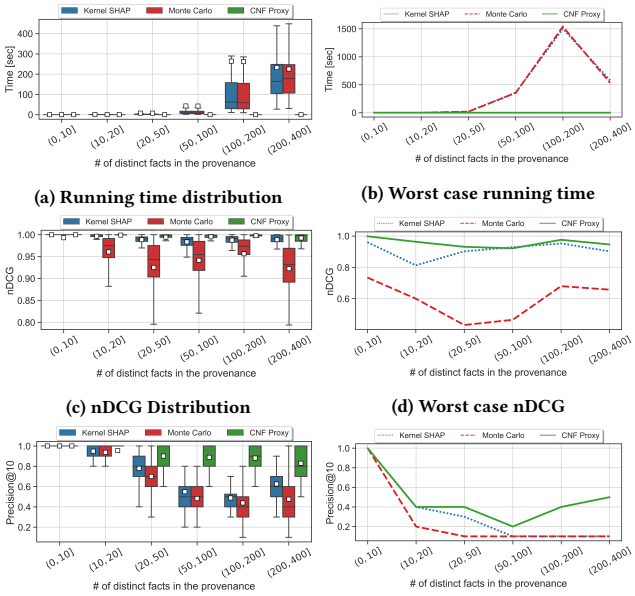
Accuracy metrics To evaluate the performance of the above methods we have used various metrics, specified below. All metrics were computed with respect to the ground truth values obtained by the knowledge compilation approach, and thus these experiments are confined to the cases where the exact computation succeeded.

- **nDCG** is the normalized discounted cumulative gain score [38], used to compare the ordering based on the inexact solution to the ordering based on the ground truth.
- **Precision@k** is the number of facts that appears in the top-k of both the inexact and exact solutions, divided by k. This was evaluated for $k \in \{1, 3, 5, 10\}$.
- **L1** and **L2** are the mean absolute error and squared error, respectively, of the results of an inexact computation method



(a) Execution time **(b) nDCG** **(c) Precision@10**

Figure 6: Comparing various metrics for inexact methods as a function of the sampling budget; CNF Proxy does not rely on sampling, thus remains constant for all budgets.



(a) Running time distribution **(b) Worst case running time**

(c) nDCG Distribution **(d) Worst case nDCG**

(e) Precision@10 distribution **(f) Worst case Precision@10**

Figure 7: Comparing inexact methods as a function the number of distinct facts in the provenance (n). Sampling methods (Kernel SHAP and Monte Carlo) presented with sampling budget of $m = 20n$; different budgets led to similar trends.

with respect to the ground truth (i.e. how different the results are from the actual Shapley values).

Execution time. Figure 6a depicts the execution time of the above methods as a function of the sampling budget. The execution times of Monte Carlo and Kernel SHAP are rather similar, while the CNF Proxy method (which does not rely on sampling) is substantially faster, with a median of 0.72 milliseconds and a mean of 2.06 milliseconds for a single query output. Figures 7a and 7b depict the distribution and worst-case running times of the three methods (for Monte Carlo and Kernel SHAP we used a budget of 20 samples per fact) as function of the number of distinct facts in the provenance expressions. CNF Proxy is substantially faster than its competitors: in most cases CNF Proxy completes in few milliseconds and 4 seconds in the worst case, whereas Monte Carlo and Kernel SHAP median execution time for circuits with 101–200 distinct input facts are 59 and 62 seconds respectively and may take up to 1,539 seconds.

Recall our analysis of the execution time of the exact solution in Section 6.1. The computation of Shapley values (KC plus Algorithm 1) takes 27,600 seconds if performed for all output tuples of query q8d in the IMDB dataset; for comparison, the CNF Proxy method has (inexactly, see quality analysis below) computed the Shapley value of the proxy functions for all query outputs within 95 seconds, which is 0.3% of the exact computation time. Surprisingly, the execution time of the exact computation was comparable to Monte Carlo and Kernel SHAP, and even faster for large sampling budgets; For example, Kernel SHAP with $m = 10n$ (where n is the number of distinct facts in the provenance) has completed computation for all output tuples of the query q8d in 16,447 seconds, which is 59% of the exact computation time. For $m = 50n$, Kernel SHAP required 58,161 seconds for completion of this computation, which is 210% of the exact computation time. This means that using such a sampling budget for Kernel SHAP is impractical in this setting, and we will use it to obtain upper bounds on Kernel SHAP's quality.

Quality analysis. Figures 6b and 6c depict the ranking quality of the above methods as a function of the sampling budget. The rankings were compared for output tuples where the exact computation succeeded, so we have the ground truth. Recall that CNF Proxy does not rely on sampling, and so it remains constant throughout the different budgets. Naturally, as the sampling budget grows, Monte Carlo and Kernel SHAP quality improves. Most notable is Monte Carlo improvement in terms of nDCG (Figure 6b), where its median (resp., mean) nDCG with budget of 10 samples per fact is 0.9669 (resp., 0.9435), while with budget of 50 samples per fact it is 1.0 (resp., 0.9923). Comparison between the two sampling methods (Monte Carlo and Kernel SHAP) reveals that Kernel SHAP is superior w.r.t all metrics, across the entire budgets range. In terms of nDCG, all methods achieve rather high scores, but CNF Proxy performs best. Indeed, the median (resp., mean) nDCG of CNF Proxy is 1.0 (resp., 0.9989), while Kernel SHAP requires 50 samples per fact to get a median (resp., mean) nDCG of 1.0 (resp., 0.9986); as previously noted such budget leads to slower execution than that of the exact computation. It is worth mentioning that also when looking at nDCG@k for $k \in \{1, 3, 5, 10\}$, CNF Proxy performs better than Monte Carlo and Kernel SHAP, even when allowing a budget of 50 samples per fact. In terms of identifying the top influential facts (i.e., Precision@k), CNF Proxy also outperforms the other methods. For example, CNF Proxy's median (resp., mean) Precision@10 are 1.0 (resp., 0.9688), while Kernel SHAP reaches 1.0 (resp., 0.9611) with 50 samples per fact. Similarly, CNF Proxy outperforms Monte Carlo and Kernel SHAP in terms of Precision@k for $k \in \{1, 3, 5\}$.

Finally, Table 2 zooms in on the results when fixing the budgets of Monte Carlo and Kernel SHAP to 50 samples per fact, which

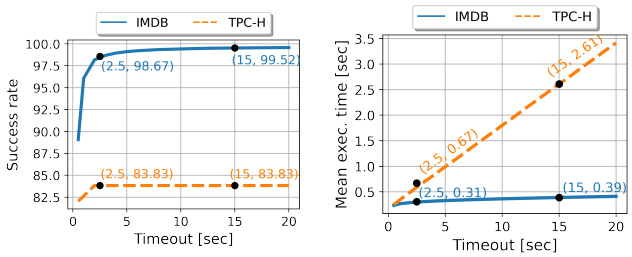
is the highest budget tested (already for this budget, computation of Monte Carlo and Kernel SHAP is slower than for the exact algorithm). Note that CNF Proxy is much faster then the other methods, while still being superior in terms of ranking (nDCG and Precision@k). As explained above, ranking of facts is indeed the use case we recommend for CNF Proxy. Unsurprisingly, Kernel SHAP achieves better distance from the exact Shapley values (L1 and L2), at the cost of being slower by several orders of magnitude.

Dependency on the provenance size. Figure 7 depicts the performance of Monte Carlo, Kernel SHAP, and CNF Proxy as a function of the number of facts in the provenance expression. The results are aggregated over all output tuples of all queries. Figure 7c presents the quality of facts ranking (nDCG) as a function of the number of distinct provenance facts. CNF Proxy performs the best, and its quality remains steady regardless of the number of facts in the provenance expression. For example, with 1–10 facts the CNF Proxy median (resp., mean) nDCG is 1.0 (resp., 0.9999), and with 201–400 facts it is 0.9977 (resp., 0.9924). Kernel SHAP has a minor deterioration, where it drops from median (resp., mean) nDCG of 1.0 (resp., 0.9998) with 1–10 facts to 0.9906 (resp., 0.9888) with 201–400 facts. Figure 7d zooms in on the aggregated results over the worst case expressions of all output tuples for all queries, and shows that even the worst case CNF Proxy is superior to the alternatives, and that the error in terms of nDCG is small (0.92 in the worst case evaluated). Figure 7e depicts the dependency of Precision@10 on the number of provenance facts. Both Monte Carlo and Kernel SHAP suffer from a massive drop of median (resp., mean) Precision@10, down to 0.4 (resp., 0.476) and 0.6 (resp., 0.6253) respectively with 201–400 facts, while CNF Proxy remains at 0.8 (resp., 0.8293) median and mean Precision@10. A similar trend is observed for Precision@5, whereas for Precision@3 and Precision@1 the drop is less significant. Here again, Figure 7f zooms in on the worst case and again shows the superiority of CNF Proxy.

6.3 Hybrid computation

Recall that in Section 6.1 we have measured the success rate of the exact computation: for IMDB the success rate was 99.96% and for TPC-H the success rate was 84.43%. We saw in Section 6.2 that the inexact method CNF Proxy is very efficient and that the ranking of tuples based on CNF Proxy is typically close to the ranking obtained based on the real Shapley values.

In this section we consider a hybrid approach that works as follows. First, we start by running the exact computation, that is, the knowledge compilation step and Algorithm 1. If the exact computation completes successfully within less than t seconds (where t is configurable) we return its result. Otherwise, we terminate the exact computation and execute the inexact method CNF Proxy, returning only a ranking of input tuples rather than their Shapley values. Figure 8a depicts the success rate of the exact computation given different timeouts. Note that given a timeout of 2.5 seconds, the exact computation succeed for 98.67% of the IMDB output tuples, and 83.83% for TPCH-H. Increasing the timeout has a rather minor impact on the success rate: having a 15 seconds timeout increases the success rate for IMDB to 99.52%, while the success rate for TPC-H remains unchanged. Recall that having a timeout of one hour results in success rates of 99.96% and 84.43% for IMDB and



(a) Exact computation success rate as function of the computation timeout

(b) Mean execution time of the hybrid approach as a function of the computation timeout

Figure 8: Hybrid approach performance

TPC-H respectively. Figure 8b depicts the mean execution time of the hybrid approach as a function of the chosen timeout t. Observe that given a timeout of 2.5 seconds the mean hybrid execution time is 0.31 seconds and 0.67 seconds for IMDB and TPC-H respectively. The mean execution time of the hybrid approach grows very moderately w.r.t. the timeout for IMDB (since most cases do not reach the timeout); it grows faster for TPC-H, where the difficult cases for which timeout is reached have a more significant effect on the overall mean execution time.

Main conclusions. Our experimental results indicate that for most output tuples (98.67% for IMDB and 83.83% for TPC-H) exact computation of Shapley values terminates within 2.5 seconds. When it does not, we propose an alternative of ranking facts according to CNF Proxy values, which typically only takes several milliseconds (up to 4 seconds for an outlier case). Experimental evidence shows that the ranking obtained via CNF Proxy is both much faster to compute and more accurate (in terms of nDCG and Precision@k, measured in cases where exact computation does succeed, and so we have the ground truth) than the alternative of only using sampling to approximate the actual Shapley values.

7 CONCLUSION

We have proposed in this paper a first practical framework for computing the contributions of database facts in query answering, quantified through Shapley values. The framework includes an exact algorithm that computes the contribution of input facts, and a faster algorithm that is practically effective in ranking contributions of input facts, while producing inexact Shapley values. Our practical implementation is currently designed for SPJU queries (that is, to the class of queries supported by ProvSQL). In addition to these practical contributions, we have also established a theoretical connection between the problem of computing Shapley values and that of probabilistic query evaluation, by showing that, for every query, the former can be reduced in polynomial time to the latter. We leave it open to determine whether there is also a reduction in the other direction (Open Problem 1). Other interesting directions would be to study further constructs such as aggregates and negation, or to extend the framework to bag semantics. Concerning bag semantics we observe that, by differentiating each copy of a same tuple in a bag database (for instance, adding an identifier attribute), our framework can be used as-is. Nevertheless, it would be interesting to see how one could adapt the definitions in order to consider fact multiplicities in a more elaborate way.

ACKNOWLEDGMENTS

This research has been partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 804302). The work of Benny Kimelfeld was supported by the Israel Science Foundation (ISF), Grant 768/19, and the German Research Foundation (DFG) Project 412400621 (DIP program).

REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Vol. 8. Addison-Wesley Reading. http://webdam.inria.fr/Alice/

[2] Marcelo Arenas, Pablo Barceló, Leopoldo Bertossi, and Mikaël Monet. 2021. On the complexity of SHAP-score-based explanations: Tractability via knowledge compilation and non-approximability results. *arXiv preprint* (2021). https://arxiv.org/abs/2104.08015

[3] Marcelo Arenas, Pablo Barceló, Leopoldo Bertossi, and Mikaël Monet. 2021. The tractability of SHAP-score-based explanations over deterministic and decomposable Boolean circuits. In *Proceedings of AAAI*. https://arxiv.org/abs/2007.14045

[4] Peter Buneman, James Cheney, Wang-Chiew Tan, and Stijn Vansummeren. 2008. Curated databases. In *Proceedings of PODS*. 1–12. https://homepages.inf.ed.ac.uk/opb/papers/inv.pdf

[5] Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. 2001. Why and where: A characterization of data provenance. In *ICDT*. Springer, 316–330. https://repository.upenn.edu/cgi/viewcontent.cgi?article=1209&context=cis_papers

[6] Yingwei Cui, Jennifer Widom, and Janet L Wiener. 2000. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems (TODS)* 25, 2 (2000), 179–227. http://ilpubs.stanford.edu:8090/252/1/1997-3.pdf

[7] Nilesh Dalvi and Dan Suciu. 2007. Efficient query evaluation on probabilistic databases. *VLDB J.* 16, 4 (2007), 523–544. https://homes.cs.washington.edu/~suciu/vldbj-probdb.pdf

[8] Nilesh Dalvi and Dan Suciu. 2013. The dichotomy of probabilistic inference for unions of conjunctive queries. *Journal of the ACM (JACM)* 59, 6 (2013), 1–87. https://homes.cs.washington.edu/~suciu/jacm-dichotomy.pdf

[9] Adnan Darwiche. 2001. On the tractable counting of theory models and its application to truth maintenance and belief revision. *J. Applied Non-Classical Logics* 11, 1-2 (2001). https://arxiv.org/abs/cs/0003044

[10] Adnan Darwiche. 2004. New advances in compiling CNF to decomposable negation normal form. In *Proceedings of ECAI*. Citeseer, 328–332. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.178.2262

[11] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17 (2002), 229–264. https://arxiv.org/abs/1106.1819

[12] Daniel Deutch, Nave Frost, and Amir Gilad. 2020. Explaining natural language query results. *The VLDB Journal* 29, 1 (2020), 485–508. https://arxiv.org/abs/2007.04454

[13] Daniel Deutch, Nave Frost, Benny Kimelfeld, and Mikaël Monet. 2021. Shapley for database facts source code. https://github.com/navefr/ShapleyForDbFacts.

[14] Todd J Green, Grigoris Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *Proceedings of PODS*. 31–40. https://repository.upenn.edu/cgi/viewcontent.cgi?article=1022&context=db_research

[15] Todd J Green and Val Tannen. 2017. The semiring framework for database provenance. In *Proceedings of PODS*. 93–99. https://dl.acm.org/doi/10.1145/3034786.3056125

[16] Anthony Hunter and Sébastien Konieczny. 2010. On the measure of conflicts: Shapley inconsistency values. *Artificial Intelligence* 174, 14 (2010), 1007–1026. http://www.cril.univ-artois.fr/~konieczny/papers/aij10a.pdf

[17] Tomasz Imielinski and Witold Lipski Jr. 1984. Incomplete Information in Relational Databases. *J. ACM* 31, 4 (1984), 761–791. https://doi.org/10.1145/1634.1886

[18] Abhay Jha and Dan Suciu. 2013. Knowledge compilation meets database theory: compiling queries to decision diagrams. *Theory of Computing Systems* 52, 3 (2013), 403–440. https://link.springer.com/article/10.1007/s00224-012-9392-5

[19] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. https://www.vldb.org/pvldb/vol9/p204-leis.pdf

[20] Ester Livshits, Leopoldo E. Bertossi, Benny Kimelfeld, and Moshe Sebag. 2020. The Shapley value of tuples in query answering. In *ICDT*, Vol. 155. Schloss Dagstuhl, 20:1–20:19. https://arxiv.org/abs/1904.08679

[21] Scott M Lundberg, Gabriel Erion, Hugh Chen, Alex DeGrave, Jordan M Prutkin, Bala Nair, Ronit Katz, Jonathan Himmelfarb, Nisha Bansal, and Su-In Lee. 2020. From local explanations to global understanding with explainable AI for trees. *Nature machine intelligence* 2, 1 (2020), 2522–5839. https://arxiv.org/pdf/1905.04610.pdf

[22] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *Advances in neural information processing systems*. 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

[23] Irwin Mann and LS Shapley. 1960. Values for large games, IV: Evaluating the Electoral College by Monte Carlo Techniques. The Rand Corporation. *Research Memorandum* 2651 (1960). https://www.rand.org/pubs/research_memoranda/RM2651.html

[24] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. 2010. The complexity of causality and responsibility for query answers and non-answers. *PVLDB* 4, 1 (2010), 34–45. https://www.vldb.org/pvldb/vol4/p34-meliou.pdf

[25] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. 2014. Causality and explanations in databases. *Proceedings of the VLDB Endowment (PVLDB)* 7, 13 (2014), 1715–1716. http://www.vldb.org/pvldb/vol7/p1715-meliou.pdf

[26] Mikaël Monet. 2020. Solving a Special Case of the Intensional vs Extensional Conjecture in Probabilistic Databases. In *Proceedings of PODS*. 149–163. https://arxiv.org/abs/1912.11864

[27] Alon Reshef, Benny Kimelfeld, and Ester Livshits. 2020. The impact of negation on the complexity of the Shapley value in conjunctive queries. In *Proceedings of PODS*. 285–297. https://arxiv.org/abs/1912.12610

[28] Alvin E Roth. 1988. *The Shapley Value: Essays in Honor of Lloyd S. Shapley*. Cambridge University Press. http://www.library.fa.ru/files/Roth2.pdf

[29] Sudeepa Roy, Laurel J. Orr, and Dan Suciu. 2015. Explaining query answers with explanation-ready databases. *Proceedings of the VLDB Endowment (PVLDB)* 9, 4 (2015), 348–359. http://www.vldb.org/pvldb/vol9/p348-roy.pdf

[30] Babak Salimi, Leopoldo E. Bertossi, Dan Suciu, and Guy Van den Broeck. 2016. Quantifying causal effects on query answering in databases. In *TaPP*. USENIX Association. http://web.cs.ucla.edu/~guyvdb/papers/SalimiTaPP16.pdf

[31] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. Provsql: Provenance and probability management in postgresql. *Proceedings of the VLDB Endowment (PVLDB)* 11, 12 (2018), 2034–2037. https://hal.inria.fr/hal-01851538/file/p976-senellart.pdf

[32] Lloyd S Shapley. 1953. A value for n-person games. *Contributions to the Theory of Games* 2, 28 (1953), 307–317. http://www.library.fa.ru/files/Roth2.pdf#page=39

[33] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. 2011. *Probabilistic Databases*. Morgan & Claypool. https://www.morganclaypool.com/doi/abs/10.2200/S00362ED1V01Y201105DTM016

[34] Transaction Processing Performance Council (TPC). 2017. TPC-H benchmark. http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.2.pdf

[35] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. In *Automation of reasoning*. Springer, 466–483. https://link.springer.com/chapter/10.1007/978-3-642-81955-1_28

[36] Guy Van den Broeck, Anton Lykov, Maximilian Schleich, and Dan Suciu. 2021. On the tractability of shap explanations. In *Proceedings of AAAI*. https://arxiv.org/abs/2009.08634

[37] Moshe Y Vardi. 1982. The complexity of relational query languages. In *STOC*. ACM, 137–146. http://www.dis.uniroma1.it/~degiacom/didattica/semingsoft/SIS05-06/materiale/1-query-congiuntive/riferimenti/vardi-1982.pdf

[38] Yining Wang, Liwei Wang, Yuanzhi Li, Di He, Wei Chen, and Tie-Yan Liu. 2013. A theoretical analysis of NDCG ranking measures. In *Proceedings of COLT*, Vol. 8. 6. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.680.490&rep=rep1&type=pdf

[39] Bruno Yun, Srdjan Vesic, Madalina Croitoru, and Pierre Bisquert. 2018. Inconsistency Measures for Repair Semantics in OBDA. In *IJCAI*. ijcai.org, 1977–1983. https://www.ijcai.org/proceedings/2018/0273.pdf