Advancing Fact Attribution for Query Answering: Aggregate Queries and Novel Algorithms

Omer Abramovich
Tel Aviv University
Tel Aviv, Israel

Daniel Deutch
Tel Aviv University
Tel Aviv, Israel

Nave Frost
eBay Research
Netanya, Israel

Ahmet Kara
OTH Regensburg
Regensburg, Germany

Dan Olteanu
University of Zurich
Zurich, Switzerland

ABSTRACT

In this paper, we introduce a novel approach to computing the contribution of input tuples to the answers of queries, quantified using the Banzhaf and Shapley values. In contrast to prior algorithmic work focusing on Select-Project-Join-Union queries, ours is the first practical approach for queries with aggregates.

Our approach relies on two novel optimizations that are essential for its practicality, and also significantly improve the performance even for queries without aggregates. The first optimization exploits the observation that many input tuples have the same contributions to the query answer, so it is enough to compute the contribution of one of them. The second optimization uses the gradient of the query lineage to compute the contributions of all tuples in the time needed for one of them. Experiments show that our approach achieves up to 3 orders of magnitude runtime improvements over the state-of-the-art for queries without aggregates, and that it is indeed practical for aggregate queries.

1 INTRODUCTION

Recent years have witnessed a surge of development that uses game theoretic measures to quantify the contribution of each database tuple to the query answer, also referred to as *attribution*. The Banzhaf [4, 27] and Shapley [31] values are two prominent examples of such measures. They originate in cooperative game theory, where they are used to quantify the contribution of a player by summing up their marginal contributions over the subsets (or permutations, for Shapley) of players. They are used in applications across several domains. For instance, in machine learning, they are used for feature importance and data valuation, feature selection, and explanations [29, 33]. Prior works used Banzhaf and Shapley values for data attribution in query answering: the contribution of a database tuple to the query answer is quantified by defining a game where the players are database tuples and each tuple in the query result is treated as the objective function [2, 11, 19]. Banzhaf and Shapley values are two different measurements, yet they are closely related. Previous work [2] has shown that they tend to agree on the relative order of fact contribution in query answering.

Example 1.1. Figure 1 depicts an IMDB-like database with information on some movies directed by Tarantino.

Query Q_1 asks whether Tarantino was nominated for the academy awards for "best director". The query answer is true, since Tarantino was nominated for 3 movies. We would like to understand which leading actor contributed the most to the query answer which in this case corresponds to Tarantino's nominated movies. To measure such contribution, we can use importance scores such as the Banzhaf or Shapley values. In our example, it turns out that the actor with the largest (Banzhaf and Shapley) score is Brad Pitt, having played a leading role in 2 of these 3 movies.

Query Q_2 asks for the maximum revenue of a movie directed by Tarantino. Again, we would like to understand which leading actor contributed the most to the query answer, i.e., to Tarantino's high-revenue movies. Using Banzhaf or Shapley values, we find that Samuel L. Jackson contributed the most, followed by Leonardo DiCaprio, Brad Pitt, Uma Thurman, and Christoph Waltz.

Multiple previous works have demonstrated the usefulness of Banzhaf and Shapley values [2, 6, 7, 16–19, 21, 28]. In this paper, we study the problem of *computing the contribution of database tuples to the answers of aggregate queries, quantified using the Banzhaf and Shapley values.* The computational problem is known to be highly intractable in general: already for Boolean non-hierarchical queries (which form a strict subclass of acyclic queries), the exact computation of Shapley and Banzhaf values is #P-hard [19]. Furthermore, ranking tuples by their Banzhaf or Shapley values remains intractable under standard theoretic assumptions [2]. While previous work has proposed multiple optimization strategies for computing Banzhaf and Shapley values in the context of SPJU queries, the execution time is far from negligible, especially for computing the Banzhaf/Shapley values for all input fact-output tuple pairs. *Furthermore, all practical algorithms and prototype implementations in this context are limited in their applicability, in that they have focused on SQL queries with select, project, join, and union.*

In this paper, we go beyond prior work by adding support to aggregates (COUNT, SUM, MIN, MAX) in the select clause of the query. Aggregation is key to practical OLAP and decision support workloads. All 22 TPC-H queries involve aggregation.

All prior practical approaches to fact attribution, which were designed for queries without aggregates, do not work for queries with aggregates. Such approaches compile the lineage (or provenance) of the query [8, 14, 25] into tractable circuits that support the efficient computation of the Banzhaf and Shapley values for the input tuples. Yet the lineage formalism necessarily differs between the two query classes: Whereas the lineage for the former is a *Boolean semiring* expression, for the latter it is a *semimodule* expression, which pairs an *aggregate monoid* expression with a Boolean semiring expression. *The first main contribution of this paper is an extension of the*

DirectingAwards (endo)

	Award
d_1	Academy
d_2	BAFTA

Movies (endo)

	Title	Director	Gross revenue(M$)
m_1	Kill Bill: Vol. 1	Tarantino	176
m_2	Inglourious Basterds	Tarantino	322
m_3	Once Upon a Time in Hollywood	Tarantino	377

Cast (endo)

	Name
a_1	Brad Pitt
a_2	Leonardo DiCaprio
a_3	Zoë Bell
a_4	Uma Thurman

MovieAwards (exo)

Movie ID	Award
Inglourious Basterds	Academy
Inglourious Basterds	BAFTA
Once Upon a Time in Hollywood	Academy
Once Upon a Time in Hollywood	BAFTA

Query Q_1

$Q() = \texttt{Movies}(x, \text{Tarantino}, r),$
$\texttt{Cast}(y),$
$\texttt{DirectingAwards}(z),$
$\texttt{MovieCast}(y, x),$
$\texttt{MovieAwards}(x, z)$

Query Q_2

$\langle MAX, r,$
$Q(x, y, r) =$
$\texttt{Movies}(x, \text{Tarantino}, r),$
$\texttt{Cast}(y),$
$\texttt{MovieCast}(y, x)\rangle$

MovieCast (exo)

Movie ID	Actor ID
Kill Bill: Vol. 1	Uma Thurman
Inglourious Basterds	Brad Pitt
Once Upon a Time in Hollywood	Brad Pitt
Inglourious Basterds	Zoë Bell
Once Upon a Time in Hollywood	Zoë Bell
Once Upon a Time in Hollywood	Leonardo DiCaprio

Figure 1: Running example: an IMDB-like database and two queries

compilation approach for Banzhaf- and Shapley-based fact attribution from queries without aggregates to queries with aggregates, so from semiring to semimodule expressions.

Yet, the computation of Banzhaf/Shapley values for semimodule expressions remains expensive for most benchmarks used in the literature [2, 11]. *The second main contribution of this paper is the interplay of two novel techniques that we incorporate in our semimodule compilation algorithm: lifted compilation and gradient-based Banzhaf/Shapley value computation.* These techniques significantly reduce the computation time for fact attribution for *queries with and even without aggregates.*

In more detail, our contributions are as follows. We introduce LExaBan and LExaShap, two novel algorithms to compute Banzhaf and respectively Shapley values for SPJUA (Select-Project-Join-Union-Aggregate) queries. We conduct experiments with these algorithms and the state-of-the-art algorithms ExaBan and ExaShap for fact attribution and SPJU queries using more than a million instances from 3 databases [2]. For queries without aggregates, LExaBan achieves 2-3 orders of magnitude improvement over ExaBan and LExaShap achieves more than 2 orders of magnitude improvement over ExaShap. For queries with aggregates, LExaBan and LExaShap are, to our knowledge, the first practical algorithms for Banzhaf and Shapley value computation. The performance gains of LExaBan and LExaShap over the state-of-the-art are primarily due to two novel techniques.

Lifted compilation. A major challenge faced by state-of-the-art approaches to fact attribution in query answering is the need to compile query lineage into forms that admit tractable Banzhaf/Shapley value computation. A prime example of such tractable form is the decomposition tree (d-trees) [12]: This is a Boolean circuit (1) that has gates expressing independence or mutual exclusiveness of their children and (2) on which the Banzhaf/Shapley values can be computed in one bottom-up pass. The compilation can be time consuming and often yields large d-trees.

The key insight behind lifting is that some variables in the lineage are interchangeable. For instance, in the Boolean formula $(x_1 \wedge y_1 \wedge z_1) \vee (x_2 \wedge y_1 \wedge z_1)$ the pairs of variables $\{x_1, x_2\}$ and $\{y_1, z_1\}$ are interchangeable. We extend the d-trees with a new gate type, which replace, or *lift*, conjunctions or disjunctions of variables, which behave the same in the lineage, with a fresh variable. Such lift gates preserve the equivalence to the original formula. Lifting can drastically reduce the compilation time and lead to much smaller d-trees. In our experiments, it leads to more than 2 OOM (orders-of-magnitude) speedup and more than 1 OOM smaller d-trees.

Gradient-based Banzhaf/Shapley value computation. A further major challenge is that we need to compute the Banzhaf/Shapley values for all input facts. Computing such values separately for each fact is computationally expensive. We exhibit a novel relationship between the Banzhaf/Shapley values and the gradients of a function defined by the query lineage. This allows us to devise a back-propagation algorithm to compute these values efficiently over the d-tree. Our gradient technique allows to compute the Banzhaf/Shapley values for all input facts in the same time needed to compute the value for one fact. In our experiments, this gradient technique speeds up the computation by a factor that grows with the size of the instance and up to more than 2 OOM speedup for large instances. This enables the computation of Banzhaf values for instances larger than previously possible.

We first introduce these two techniques for SPJU queries in Sec. 3 and then extend them to SPJUA queries in Sec. 4. In our experiments (Sec. 5), we show that the interplay of both techniques can lead to 3 OOM speedup.

Related Work. Banzhaf and Shapley values, originally introduced in the context of game theory to quantify the contribution of individual players to the value of a cooperative game [4, 31] were used in query answering to quantify the contribution of individual database facts to query answers [19]. Previous work has analyzed the complexity of computing these measures [19, 28] and proposed efficient algorithms to compute them based on knowledge compilation algorithms [2, 11, 17]. Since the problem is computationally intractable in general, multiple heuristics and approximation schemes were proposed. Our algorithms are far more efficient than state of the art algorithms, pushing the boundaries of tractable instances. Many other notions have been used to quantify fact contribution in query answering, including causality [25], responsibility [23] or counterfactuals [24]. Banzhaf and Shapley values are unique in that they are grounded in game theory where their properties have been extensively studied, and previous work has shown their usefulness for query answering. Banzhaf and Shapley values are prevalent in many other fields, notably interpreting model predictions and feature selection in Machine Learning [13, 32]. For instance, the SHAP score [20] leverages and adapts the definition of Shapley values to explain model predictions.

A full version of the paper, including proofs for all formal results, as well as the implementation of all algorithms, are available in [3].

2 PRELIMINARIES

We use $\mathbb{N}$ to denote the set of natural numbers including 0. For $n \in \mathbb{N}$, we define $[n] = \{1, \ldots, n\}$. In case $n = 0$, we have $[n] = \emptyset$.

Boolean Formulas. A *Boolean formula* (or simply formula) over a set $\boldsymbol{X}$ of Boolean variables is either a constant 0 or 1, a variable $x \in \boldsymbol{X}$, a negation $\neg\varphi$, or a conjunction/disjunction $\varphi_1 \wedge \varphi_2$ or $\varphi_1 \vee \varphi_2$ of two formulas φ_1, φ_2. We denote by $\mathrm{Bool}(\boldsymbol{X})$ the set of Boolean formulas over $\boldsymbol{X}$. A *literal* is a constant, a variable, or its negation. The set of variables in φ is denoted $\mathrm{vars}(\varphi)$, and φ is *read-once* if each variable appears at most once. Substituting $x \in \boldsymbol{X}$ with $b \in \{0, 1\}$ in φ is written as $\varphi[x := b]$. A *valuation* $\theta : \mathrm{vars}(\varphi) \to \{0, 1\}$ for φ maps each variable in φ to 0 or 1, and we identify θ with $\{x \in \mathrm{vars}(\varphi) \mid \theta(x) = 1\}$, where $|\theta|$ is the number of variables mapped to 1. The Boolean value of φ under θ is $\varphi[\theta]$, and θ is a *model* of φ if $\varphi[\theta] = 1$. The *model set* $\mathrm{models}(\varphi)$ consists of all models of φ, with *model count* $\#\varphi = |\mathrm{models}(\varphi)|$ and *k-model count* $\#_k\varphi = |\mathrm{models}(\varphi) \cap \binom{\boldsymbol{X}}{k}|$, where $\binom{\boldsymbol{X}}{k}$ is the set of k-element subsets of $\boldsymbol{X}$. Two formulas φ_1, φ_2 are *independent* if $\mathrm{vars}(\varphi_1) \cap \mathrm{vars}(\varphi_2) = \emptyset$ and *exclusive* if $\mathrm{models}(\varphi_1) \cap \mathrm{models}(\varphi_2) = \emptyset$. The set $\mathrm{PosBool}(\boldsymbol{X})$ of *positive formulas* (formulas without negation) forms a commutative semiring under $\vee, \wedge, 0, 1$ [15].

Example 2.1. Consider the formula $\varphi = x \vee y$. The models of φ are $\{x\}$, $\{y\}$, and $\{x, y\}$. Hence, $\#\varphi = 3$, $\#_0\varphi = 0$, $\#_1\varphi = 2$, and $\#_2\varphi = 1$.

Databases. We follow standard relational database notions [1]. We assume a countably infinite set Const of constants used as database values. For a tuple $\boldsymbol{t} \in \mathrm{Const}^k$, $k \in \mathbb{N}$ and $i \in [k]$, we denote its i-th value by $\boldsymbol{t}[i]$. A database consists of *facts* $R(\boldsymbol{t})$, where R is a relation name and $\boldsymbol{t}$ is a tuple matching R's arity. Following prior work, each database D is partitioned into *endogenous* facts D_n and *exogenous* facts D_x [19].

Conjunctive Queries. A *conjunctive query (CQ)* has the form $Q(\boldsymbol{X}) = R_1(\boldsymbol{X}_1), \ldots, R_m(\boldsymbol{X}_m)$, where each R_i is a relation name; each $\boldsymbol{X}_i$ is a tuple of variables and constants matching the arity of R_i. The *head variables* $\boldsymbol{X}$ form a subset of the variables in $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_m$. A CQ is *k-ary* if $|\boldsymbol{X}| = k$ and *Boolean* if $k = 0$. Each $R_i(\boldsymbol{X}_i)$ is called an *atom.* A *union of conjunctive queries (UCQ)* consists of a set of k-ary CQs for some $k \in \mathbb{N}$.

Example 2.2. Query Q_1 in Figure 1 is a CQ asking whether movies directed by Tarantino were nominated for directing awards.

Aggregate Queries. We define *aggregate queries* following prior work [19]. Given a database D, an aggregate query is a triple $\langle \alpha, \gamma, Q \rangle$, where Q is a k-ary UCQ for some $k \in \mathbb{N}$, $\gamma : \mathrm{Const}^k \to \mathbb{R}$ is a function that maps each k-ary value tuple to a numeric value and $\alpha : \mathbb{R}^* \to \mathbb{R}$ is a function that maps a bag of numeric values to one value. Given a database D, let $Q(D) = \{\boldsymbol{t}_1, \ldots, \boldsymbol{t}_k\}$. The result of $\langle \alpha, \gamma, Q \rangle$ over D is defined as $\langle \alpha, \gamma, Q \rangle(D) := \alpha(\gamma(\boldsymbol{t}_i)_{i \in k})$. If γ returns the i-th value of the tuple $\boldsymbol{t}$, we just write i instead of α. We consider SUM and MAX aggregate queries of the following form:

$$\langle \mathrm{SUM}, \gamma, Q \rangle(D) \overset{def}{=} \sum_{\vec{c} \in Q(D)} \gamma(\vec{c})$$

$$\langle \mathrm{MAX}, \gamma, Q \rangle(D) \overset{def}{=} \begin{cases} \max\{\gamma(\vec{c}) \mid \vec{c} \in Q(D)\} & \text{if } Q(D) \neq \emptyset \\ 0 & \text{if } Q(D) = \emptyset \end{cases}$$

The query $\langle \mathrm{MIN}, \gamma, Q \rangle$ is defined analogously to $\langle \mathrm{MAX}, \gamma, Q \rangle$. We define $\langle \mathrm{COUNT}, Q \rangle \overset{def}{=} \langle \mathrm{SUM}, \mathbf{1}, Q \rangle$, where $\mathbf{1}$ maps each output tuple to 1, yielding the number of output tuples of Q. If the aggregate is clear from the context, we refer to an aggregate query by Q.

Remark 2.3. The choice for the MAX and MIN queries to evaluate to 0 if $Q(D) = \emptyset$ is not arbitrary. The alternatives, using ∞ for MIN and $-\infty$ for MAX, are not useful in the context of attribution values, in which we sum up over different valuations of the query.

Example 2.4. Query Q_2 of Figure 1 is an aggregate query that asks for the maximal revenue of a movie directed by Tarantino.

Monoids and Semimodules. The development in this paper uses the algebraic structures of numeric monoids and semimodules constructed using such monoids and the $\mathrm{PosBool}(\boldsymbol{X})$ semiring. We denote by $\overline{\mathbb{R}}$ the set of real numbers $\mathbb{R}$ including $\{\infty, -\infty\}$.

Definition 2.5. A numeric monoid (or simply, a monoid) $(\overline{\mathbb{R}}, +_M, 0_M)$ consists of a binary operation $+_M : \overline{\mathbb{R}} \times \overline{\mathbb{R}} \to \overline{\mathbb{R}}$, and a neutral element $0_M \in \overline{\mathbb{R}}$ s.t. for all $m_1, m_2, m_3 \in \overline{\mathbb{R}}$, the following holds:

$$(m_1 +_M m_2) +_M m_3 = m_1 +_M (m_2 +_M m_3)$$
$$0 +_M m_1 = m_1 +_M 0 = m_1$$

A monoid is *commutative* if $m_1 + m_2 = m_2 + m_1$ for all $m_1, m_2 \in M$.

Monoids naturally model many aggregate functions. For instance, the SQL aggregate function MAX, which returns the maximum value in a relation column is modeled by the monoid $(\max, -\infty)$.

Next, we introduce PosBool-semimodules, which combine monoid values with positive Boolean formulas to represent numerical values conditioned by Boolean formulas.

Definition 2.6. Given a monoid $M = (\overline{\mathbb{R}}, +_M, 0_M)$ and the semiring $(\mathrm{PosBool}(\boldsymbol{X}), \vee, \wedge, 0, 1)$, the triple $(\mathrm{PosBool}(\boldsymbol{X}), M, \otimes)$ is a PosBool-semimodule where $\otimes : \mathrm{PosBool}(\boldsymbol{X}) \times \overline{\mathbb{R}} \to \overline{\mathbb{R}}$, such that for all $\varphi_1, \varphi_2 \in \mathrm{PosBool}(\boldsymbol{X})$ and $m_1, m_2 \in \overline{\mathbb{R}}$, the following axioms hold:

(1) $\varphi_1 \otimes (m_1 +_M m_2) = \varphi_1 \otimes m_1 +_M \varphi_1 \otimes m_2$
(2) $(\varphi_1 \vee \varphi_2) \otimes m_1 = \varphi_1 \otimes m_1 +_M \varphi_2 \otimes m_1$
(3) $(\varphi_1 \wedge \varphi_2) \otimes m_1 = \varphi_1 \otimes (\varphi_2 \otimes m_1)$
(4) $\varphi_1 \otimes 0_M = 0 \otimes m_1 = 0_M$
(5) $1 \otimes m_1 = m_1$

A *semimodule expression* is either an element from M or of one of the forms $\Phi_1 +_M \Phi_2$ and $\varphi \otimes \Phi_1$ for semimodule expressions Φ_1 and Φ_2 and a Boolean formula $\varphi \in \mathrm{PosBool}(\boldsymbol{X})$. It easily follows from the definition of semimodules that every semimodule expression can be equivalently written as $\varphi_1 \otimes m_1 +_M \cdots +_M \varphi_k \otimes m_k$, where each φ_i is a Boolean formula and each m_i is an element from M. We abbreviate the latter expression by $\sum_M \varphi_i \otimes m_i$.

Consider a semimodule expression $\Phi = \sum_M \varphi_i \otimes m_i$. The *Boolean part* of Φ is $\bigvee \varphi_i$. Given a valuation θ over $\boldsymbol{X}$, we denote $S_\theta = \{i | \varphi_i[\theta] = 1\}$. By $\Phi[\theta]$, we denote the monoid value computed by $\sum_M \left[\varphi_j[\theta] \otimes m_j\right]_{j \in S_\theta}$. Note that this is equal to $\sum_M m_j$ where the sum ranges over all j for which φ_j evaluates to 1 over θ.

Remark 2.7. Our semantics for semimodule expressions considers the summation only over those terms $\varphi_i \otimes m_i$ in Φ for which φ_i is evaluated to 1. This is to avoid Φ taking the default value of the monoid for the valuation that maps all φ_i to 0 (see Remark 2.3).

For a semimodule expression Φ, we denote by $\#^p\Phi$ the number of valuations θ such that $\Phi[\theta] = p$ and by $\#^p_k\Phi$ the number of such valuations of size k. We define the variables $\mathsf{vars}(\Phi)$ of Φ to be the variables $\mathsf{vars}(\varphi)$ of φ and the model count $\#\Phi$ of Φ to be the model count $\#\varphi$ of φ, where φ is the Boolean part of Φ. We say that two semimodule expressions T_1 and T_2 are independent (resp. exclusive) if their Boolean parts are independent (resp. exclusive).

Query Grounding. A *grounding* of a CQ Q w.r.t. a database D is a mapping G of the variables of Q to constants such that by replacing every variable X by $G(X)$, every atom in Q becomes a fact in D. We denote this set of facts by $\mathsf{facts}(Q, D, G)$. A grounding for a UCQ is a grounding for one of its CQs. Each grounding G yields an *output tuple*, which is the restriction of G to the head variables of Q. Multiple groundings may yield the same output tuple. We use $G(Q, D, \boldsymbol{t})$ to denote the set of groundings yielding $\boldsymbol{t}$. We use $Q(D)$ to denote the set of output tuples for Q w.r.t. D.

Query Lineage. Consider a database $D = D_n \cup D_x$. To construct the lineage of a query over D, we first associate each endogenous fact (or tuple) f in D_n with a Boolean variable $v(f)$. Given a UCQ Q and an output tuple $\boldsymbol{t}$, the lineage of $\boldsymbol{t}$ with respect to Q over D, denoted by $\mathsf{lin}(Q, D, \boldsymbol{t})$, is a positive Boolean formula in Disjunctive Normal Form over the variables $v(f)$ of facts f in D_n:

$$\mathsf{lin}(Q, D, \boldsymbol{t}) = \bigvee_{G \in G(Q,D,\boldsymbol{t})} \quad \bigwedge_{f \in facts(Q,D,G) \cap D_n} v(f)$$

That is, the lineage is a disjunction over all groundings yielding $\boldsymbol{t}$. For each such grounding G, the lineage has a conjunctive clause consisting of all endogenous facts in $\mathsf{facts}(Q, D, G)$. If Q is a Boolean query, we abbreviate $\mathsf{lin}(Q, D, ())$ by $\mathsf{lin}(Q, D)$.

For an aggregate query $\langle \alpha, \gamma, Q \rangle$, let $Q(D) = \{\boldsymbol{t}_1, \dots, \boldsymbol{t}_k\}$. The lineage of the query over D is a semimodule expression of the form:

$$\mathsf{lin}(\langle \alpha, \gamma, Q \rangle, D) = \sum\nolimits_M \mathsf{lin}(Q, D, \boldsymbol{t}_i) \otimes \gamma(\boldsymbol{t}_i),$$

where the summation uses the $+_M$ operator of the monoid M corresponding to α. For simplicity, we consider any fact f such that $v(f)$ is not in the lineage to be exogenous.

Example 2.8. Table 1 presents the lineage of the UCQ Q_1 and the aggregate query Q_2 in Figure 1. The lineage of Q_1 is a DNF boolean formula, while the lineage of Q_2 is a semimodule expression with the Max monoid. The boolean variables in the lineages correspond to tuples of the database in Figure 1. For instance, the variable a_1 corresponds to the Cast tuple 'Brad Pitt' annotated with a_1.

Banzhaf and Shapley Values. We use the Banzhaf and Shapley values to measure the contribution of database facts to the answers of Boolean and aggregate queries [2, 19].

Definition 2.9 (Banzhaf Value). Given a Boolean or aggregate query Q and a database $D = D_x \cup D_n$, the Banzhaf value of an endogenous fact $f \in D_n$ is defined as:

$$\mathsf{Banzhaf}(Q, f, D) \overset{def}{=} \sum_{D' \subseteq D_n \setminus \{f\}} Q(D' \cup \{f\} \cup D_x) - Q(D' \cup D_x)$$

Given a Boolean formula or a semimodule expression Ψ over a variable set $\boldsymbol{X}$, the Banzhaf value of $x \in \boldsymbol{X}$ is:

$$\mathsf{Banzhaf}(\Psi, x) = \sum_{\boldsymbol{Y} \subseteq \boldsymbol{X} \setminus \{x\}} \Psi[\boldsymbol{Y} \cup \{x\}] - \Psi[\boldsymbol{Y}] \tag{1}$$

In case of a Boolean formula φ, the Banzhaf value of a variable x can be computed based on the model counts of the formulas obtained from φ by substituting x by the constants 1 and 0 [2]:

$$\mathsf{Banzhaf}(\varphi, x) = \#\varphi[x := 1] - \#\varphi[x := 0] \tag{2}$$

Definition 2.10 (Shapley value). For sets $\boldsymbol{Y}, \boldsymbol{Z}$, let $C^{\boldsymbol{Z}}_{\boldsymbol{Y}} = C_{\boldsymbol{Y}} = \frac{|\boldsymbol{Y}|!(|\boldsymbol{X}|-|\boldsymbol{Y}|-1)!}{|\boldsymbol{X}|!}$. Given a Boolean or aggregate query Q and a database $D = D_x \cup D_n$, the Shapley value of an endogenous fact $f \in D_n$ is defined as:

$$\mathsf{Shapley}(Q, f, D) \overset{def}{=} \sum_{Y \subseteq D_n \setminus \{f\}} C^{D_n}_{Y} \cdot \left(Q(Y \cup \{f\} \cup D_x) - Q(Y \cup D_x)\right)$$

Given a Boolean formula or a semimodule expression Ψ over a variable set $\boldsymbol{X}$, the Shapley value of $x \in \boldsymbol{X}$ is:

$$\mathsf{Shapley}(\Psi, x) = \sum_{\boldsymbol{Y} \subseteq \boldsymbol{X} \setminus \{x\}} C^{\boldsymbol{X}}_{\boldsymbol{Y}} \cdot (\Psi[\boldsymbol{Y} \cup \{x\}] - \Psi[\boldsymbol{Y}]) \tag{3}$$

Prior work connected the Banzhaf and Shapley values for database facts and those of boolean variables [19]. Given a Boolean or aggregate query Q, a database $D = D_x \cup D_n$ and a fact $f \in D_n$:

$$\mathsf{Banzhaf}(Q, D, f) = \mathsf{Banzhaf}(\mathsf{lin}(Q, D), v(f))$$
$$\mathsf{Shapley}(Q, D, f) = \mathsf{Shapley}(\mathsf{lin}(Q, D), v(f))$$

Example 2.11. Consider the lineage of Q_1, φ as presented in figure 1. Calculating the Banzhaf or Shapley value of the variable a_1 representing the *actor* fact of Brad Pitt. Consider the assignment $\theta = \{m_3, m_2\}$. The marginal contribution of a_1 on θ is $\varphi[\theta \cup \{a_1\}] - \varphi[\theta] = 1$ because θ is not a satisfying assignment for φ but $\theta \cup \{a_1\}$ is. The Banzhaf value is achieved by summing up the marginal contributions across all assignments. For Shapley value, the marginal contribution of a_1 to θ would be multiplied by a coefficient of $\frac{2!(7-2-1)!}{7!}$ since $|\theta| = 2$ and $|vars(\varphi)| = 7$. The Shapley value is achieved by summing up the marginal contributions across all assignments multiplied by the corresponding coefficients.

Decomposition Trees. Prior work uses knowledge compilation techniques to compute Banzhaf and Shapley values [2, 11, 17]. The key idea is to compile query lineage into a circuit that enables efficient computation of these values. We introduce decomposition trees for Boolean formulas and semimodule expressions. The definition we provide here differs slightly from prior work, as it incorporates non-binary gates [10].

Definition 2.12. A *decomposition tree*, or d-tree for short, is defined recursively as follows:

- If v is a 0/1 constant or a variable then v is a d-tree.
- Let $T_{\varphi_1}, \dots, T_{\varphi_n}$ be d-trees for pairwise independent formulas $\varphi_1, \dots, \varphi_n$, respectively, let T_φ and T_ψ be d-trees for formulas φ and ψ, respectively. and let v be a variable not in $vars(\varphi) \cup vars(\psi)$, Then,

$$\begin{tikzcd}[column sep=tiny] & \oplus \arrow[dl, dash] \arrow[dr, dash] & \\ T_{\varphi_1} & \cdots & T_{\varphi_n} \end{tikzcd}, \quad \begin{tikzcd}[column sep=tiny] & \odot \arrow[dl, dash] \arrow[dr, dash] & \\ T_{\varphi_1} & \cdots & T_{\varphi_n} \end{tikzcd}, \text{ and } \quad \begin{tikzcd}[column sep=tiny] & \sqcup_v \arrow[dl, dash] \arrow[dr, dash] & \\ T_{\varphi} & & T_{\psi} \end{tikzcd}$$

are d-trees for $\bigvee_{i \in [n]} \varphi_i$, $\bigwedge_{i \in [n]} \varphi_i$, and $(v \wedge \varphi) \vee (\neg v \wedge \psi)$, respectively. The latter is referred to as Shannon expansion, and creates two exclusive formulas. φ and ψ are called the 1 and 0 branches resp. and v is called the condition.

Following prior work, we add the following constructs to deal with semimodule expressions [12]:

Table 1: Lineages and lifted lineages for queries Q_1 and Q_2 from Figure 1. Colored variables are mapped to formulas.

Query	Lineage	Lifted Lineage
Q_1	$(d_1 \wedge a_1 \wedge m_3) \vee (d_1 \wedge a_1 \wedge m_2) \vee (d_1 \wedge a_3 \wedge m_3) \vee (d_1 \wedge a_2 \wedge m_3) \vee (d_1 \wedge a_3 \wedge m_2)$ $\vee (d_2 \wedge a_1 \wedge m_3) \vee (d_2 \wedge a_1 \wedge m_2) \vee (d_2 \wedge a_3 \wedge m_3) \vee (d_2 \wedge a_2 \wedge m_3) \vee (d_2 \wedge a_3 \wedge m_2)$	$(d_{1,2} \wedge a_{1,3} \wedge m_3) \vee (d_{1,2} \wedge a_{1,3} \wedge m_2) \vee (d_{1,2} \wedge a_2 \wedge m_3)$ $\ell = \{d_{1,2} \mapsto (d_1 \vee d_2), a_{1,3} \mapsto (a_1 \vee a_3), a_2 \mapsto a_2, m_2 \mapsto m_2, m_3 \mapsto m_3\}$
Q_2	$(a_1 \wedge m_3) \otimes 377 +_{\max} (a_2 \wedge m_3) \otimes 377 +_{\max} (a_3 \wedge m_3) \otimes 377$ $+_{\max} (a_1 \wedge m_2) \otimes 322 +_{\max} (a_3 \wedge m_2) \otimes 322 +_{\max} (a_4 \wedge m_1) \otimes 176$	$(a_{1,3} \wedge w_3) \vee (a_2 \wedge w_3) \vee (a_{1,3} \wedge w_2) \vee (w_1)$ $\ell = \{a_{1,3} \mapsto (a_1 \vee a_3),$ $w_1 \mapsto ((a_4 \wedge m_1) \otimes 176), w_2 \mapsto (m_2 \otimes 322), w_3 \mapsto (m_3 \otimes 377), a_2 \mapsto a_2\}$

- Every monoid value $m \in \overline{\mathbb{R}}$ is a d-tree.
- Let $T_{\varphi_1}, \dots, T_{\varphi_n}$ be d-trees for pairwise independent boolean formulas $\varphi_1, \dots, \varphi_n$, and T_Φ be a d-tree for a semimodule expression Φ whose Boolean part is independent from each of the formulas $\varphi_1, \dots, \varphi_n$. Let $T_{\Phi_1}, \dots, T_{\Phi_n}$ be d-trees for pairwise independent semimodule expressions $\Phi_1, \dots, \Phi_n$. Then,
$$\begin{tikzcd}[column sep=small] & \otimes \arrow[dl, dash] \arrow[d, dash] \arrow[dr, dash] & \\ T_{\varphi_1} & \cdots T_{\varphi_n} & T_\Phi \end{tikzcd} \quad \text{and} \quad \begin{tikzcd}[column sep=small] & \oplus \arrow[dl, dash] \arrow[dr, dash] & \\ T_{\Phi_1} & \cdots & T_{\Phi_n} \end{tikzcd}$$
are d-trees for the semimodule expression $(\bigwedge_{i \in [n]} \varphi_i) \otimes \Phi$ and $\sum_M \Phi_i$ respectively.

The size of a d-tree is its number of nodes. Knowledge compilation algorithms construct a d-tree by iteratively decomposing a Boolean formula or a semimodule expression into simpler parts.

Example 2.13. Consider the semimodule expression $(a_1 \wedge b_1) \otimes 3 +_M (a_1 \wedge b_2) \otimes 7$. Using Axiom (3) in Definition 2.6, we rewrite it as $(a_1 \otimes (b_1 \otimes 3)) +_M (a_1 \otimes (b_2 \otimes 7))$, and applying Axiom (1) gives $a_1 \otimes ((b_1 \otimes 3) +_M (b_2 \otimes 7))$. This expression corresponds to a d-tree $\otimes(a_1, \oplus(\otimes(b_1, 3), \otimes(b_2, 7)))$.

The notions of variables, valuations and Banzhaf and Shapley values extend naturally to d-trees as they may be interpreted with respect to the boolean formula the d-tree captures. We thus overload notations and use them for formulas or d-trees interchangeably.

3 ATTRIBUTION FOR UNION OF CONJUNCTIVE QUERIES

This section presents our algorithms LExaBan and LExaShap for computing Banzhaf and Shapley values for UCQs. They are extended to aggregate queries in Section 4. The starting point for our algorithms is the state-of-the-art approach [2], which first compiles the query lineage into a d-tree and then computes Banzhaf/Shapley values in one pass over the d-tree. Our algorithms employ two novel optimizations to significantly improve the efficiency of the compilation and the subsequent computation steps. Section 3.1 introduces the lifting optimization, which exploits symmetries in the query lineage for more efficient d-tree compilation. Section 3.2 introduces the computation of Banzhaf and Shapley values by evaluating and differentiating arithmetic circuits based on the d-tree.

3.1 Lifted Compilation of Lineage Into D-tree

A Boolean formula may admit multiple equivalent representations, each balancing the size and the computational tractability of model counting and Banzhaf and Shapley value computation. Knowledge compilation aims to transform succinct yet intractable representations into tractable and reasonably succinct representations. In this work, we consider query lineages as the input representation and d-trees as the target compiled form. Our key insight is that leveraging structural symmetries within the query lineage during compilation reduces both the compilation time and the size of the resulting d-tree. To achieve this, we introduce *lifting*, a technique that identifies conjunctions or disjunctions of variables with identical Banzhaf or Shapley values and replaces them with fresh variables.[1]

Consider two disjoint sets $\boldsymbol{X}$ and $\boldsymbol{Y}$ of Boolean variables. A *lifted formula* over $\boldsymbol{X} \cup \boldsymbol{Y}$ is a pair (φ, ℓ), where $\varphi \in \mathsf{Bool}(\boldsymbol{Y} \cup \boldsymbol{X})$ is a formula over $\boldsymbol{Y} \cup \boldsymbol{X}$ and $\ell : \mathsf{vars}(\varphi) \longrightarrow \mathsf{Bool}(\boldsymbol{X})$ maps each variable in φ to a formula over $\boldsymbol{X}$. The *inlining* $\mathsf{inline}(\varphi, \ell)$ of a lifted formula is obtained by replacing each variable y in φ by the formula $\ell(y)$. A lifted DNF formula is a lifted formula (φ, ℓ), where φ is in DNF. In the rest of this section, we refer only to positive DNF formulas.

Example 3.1. Consider the DNF formulas $\varphi_0 = (x_1 \wedge x_2 \wedge x_3) \vee (x_4 \wedge x_5 \wedge x_3)$, $\varphi_1 = (y_1 \wedge y_2) \vee (y_3 \wedge y_2)$, and $\varphi_2 = (y_4 \wedge y_2)$ and the functions $\ell_0 = \{x \mapsto x | x \in \mathsf{vars}(\varphi_0)\}$, $\ell_1 = \{y_1 \mapsto (x_1 \wedge x_2), y_2 \mapsto x_3, y_3 \mapsto (x_4 \wedge x_5)\}$, and $\ell_2 = \{y_2 \mapsto x_3, y_4 \mapsto \big((x_1 \wedge x_2) \vee (x_4 \wedge x_5)\big)\}$. We have $\mathsf{inline}(\varphi_0, \ell_0) = \varphi_0$, $\mathsf{inline}(\varphi_1, \ell_1) = \big((x_1 \wedge x_2) \wedge x_3\big) \vee \big((x_4 \wedge x_5) \wedge x_3\big)$, and $\mathsf{inline}(\varphi_2, \ell_2) = \big(x_1 \wedge x_2) \vee (x_4 \wedge x_5)\big) \wedge x_3$. The latter two inlinings are equivalent to the formula φ_0.

Consider a DNF formula $\varphi = \bigvee_{i \in n} C_i$, where each clause C_i is a conjunction of variables. For a clause C_i and variable x, we denote by $C_i \setminus \{x\}$ the clause that results from C_i by omitting x. We call the set $\{C_i \setminus \{x\} \mid i \in [n], x \in C_i\}$ the *cofactor* of x. Two variables in φ are called *cofactor-equivalent* if they have the same cofactor. Two variables x and y are called *interchangeable* if for each clause C_i, it holds that x appears in C_i if and only if y appears in C_i. A variable set $\boldsymbol{V}$ is called a *maximal set of cofactor-equivalent (interchangeable)* variables if every pair of variables in $\boldsymbol{V}$ is cofactor-equivalent (interchangeable) and this does not hold for any superset of $\boldsymbol{V}$. A lifted DNF formula (φ, ℓ) is called *saturated* if it contains neither a set of cofactor-equivalent variables nor a set of interchangeable variables of size greater than one.

Example 3.2. Consider again the formulas φ_0, φ_1, and φ_2 from Example 3.1. The sets $\{x_1, x_2\}$ and $\{x_4, x_5\}$ are both maximal sets of interchangeable variables in φ_0. The set $\{y_1, y_3\}$ is a maximal set of cofactor-equivalent variables in φ_1 with cofactor $\{y_2\}$. Whereas the formulas φ_0 and φ_1 are not saturated, the formula φ_2 is.

Lifting DNF Formulas. We describe how a DNF formula can be systematically translated into a saturated lifted DNF formula. We first introduce some further notation. Consider a lifted DNF formula (φ, ℓ) over $\boldsymbol{X} \cup \boldsymbol{Y}$, where $\varphi = \bigwedge_{i \in [n]} C_i$. Let $\boldsymbol{V} = \{y_1, \dots, y_k\}$ be a set of cofactor-equivalent variables in φ, where each variable has cofactor $\{N_1, \dots, N_p\}$. Let $C_{i_1}, \dots, C_{i_m}$ be all clauses in φ containing

[1]Unlike query-based lineage factorization [26], which applies uniformly across all lineages of a query, lifting is a fine-grained factorization of the query lineage specific to the data instance. Whereas achieving minimal-size instance-specific factorization of query lineage is computationally expensive [22], lifting can be computed efficiently.

Algorithm 1 Lift

```
Input: Lifted DNF formula $(\varphi, \ell)$
Output: Saturated lifted DNF formula
1: while $(\varphi, \ell)$ is not saturated do
2:     if $\varphi$ has a maximal set $V$ of cofactor-equivalent variables with $|V| > 1$ then
3:         $(\varphi, \ell) \leftarrow$ lift-or$(\varphi, \ell, V)$
4:     if $\varphi$ has a maximal set $V$ of interchangeable variables with $|V| > 1$ then
5:         $(\varphi, \ell) \leftarrow$ lift-and$(\varphi, \ell, V)$
6: return $(\varphi, \ell)$
```

variables from V. We define lift-or(φ, ℓ, V) to be the lifted DNF formula (φ', ℓ'), where: (1) φ' results from φ by omitting the clauses $C_{i_1}, \ldots, C_{i_m}$ and adding the new clause $y \wedge \bigwedge_{i \in [p]} N_i$ for a fresh variable $z \in Y \setminus \text{vars}(\varphi)$; (2) the function ℓ' is defined by $\ell'(y) = \bigvee_{i \in [k]} \ell(y_i)\}$ and $\ell'(y') = \ell(y')$ for all $\ell' \in \text{vars}(\varphi)$ with $y' \neq y$. Now, assume that $V = \{y_1, \ldots, y_k\}$ consists of interchangeable variables in φ. We denote by lift-and(φ, ℓ, V) the lifted DNF formula (φ', ℓ'), where: (1) $\varphi' = C'_1 \wedge \cdots \wedge C'_n$ such that, given a fresh variable $y \in Y \setminus \text{vars}(\varphi)$, each C'_i results from C_i by replacing the variables $y_1, \ldots, y_k$ by y; (2) ℓ' is defined by $\ell'(y) = \bigwedge_{i \in [k]} \ell(y_i)\}$ and $\ell'(y') = \ell(y')$ for all $y' \in \text{vars}(\varphi)$ with $y' \neq y$.

The function Lift in Algorithm 1 transforms any lifted DNF formula φ into a saturated lifted DNF formula. It repeatedly calls the function lift-or or lift-and as long as φ contains a variable set V with $|V| > 1$, such that all variables in V are either cofactor-equivalent or interchangeable, respectively.

Proposition 3.3. Let φ be a DNF formula, ℓ the identity function on vars(φ), and (φ', ℓ') the output of Lift(φ, ℓ) in Algorithm 1. Then, (φ', ℓ') is a saturated lifted formula such that: (1) inline(φ', ℓ') is equivalent to φ. (2) $\ell'(x)$ and $\ell'(y)$ are independent formulas for each distinct variables $x, y \in \text{vars}(\varphi')$. (3) $\ell'(x)$ is a read-once formula for each $x \in \text{vars}(\varphi')$.

Example 3.4. We explain how the procedure Lift transforms the lineage φ_0 of the query Q_1 depicted in Table 1 (top left). Let ℓ_0 be the identity function that maps every variable in φ_0 to itself. The variables d_1 and d_2 have the same cofactor $\big\{\{a_1, m_3\}, \{a_1, m_2\}, \{a_3, m_3\}, \{a_2, m_3\}, \{a_3, m_2\}\big\}$. The procedure executes lift-or$(\varphi_0, \ell_0, \{d_1, d_2\})$, which returns the lifted formula (φ_1, ℓ_1), where

$$\varphi_1 = (d_{1,2} \wedge a_1 \wedge m_3) \vee (d_{1,2} \wedge a_1 \wedge m_2) \vee (d_{1,2} \wedge a_3 \wedge m_3) \vee \\ (d_{1,2} \wedge a_2 \wedge m_3) \vee (d_{1,2} \wedge a_3 \wedge m_2),$$

and $\ell_1 = \{d_{1,2} \mapsto (d_1 \vee d_2)\} \cup \{x \mapsto x | x \in \text{vars}(\varphi) \text{ and } x \neq d_{1,2}\}$. In φ_1, the variables a_1 and a_3 have the same cofactor $\{\{d_{1,2}, m_3\}, \{d_{1,2}, m_2\}\}$. The procedure executes lift-or$(\varphi_1, \ell_1, \{a_1, a_3\})$ and obtains (φ_2, ℓ_2) shown in Table 1 (top right). The formula φ_2 does not have two or more variables that are cofactor-equivalent or interchangeable. Hence, (φ_2, ℓ_2) is saturated.

D-Tree Compilation Using Lifting. Algorithm 2 gives our lifted compilation procedure for the query lineage (Boolean formula) into a d-tree. The input lineage φ is first translated into a saturated lifted formula (ψ, ℓ) (Line 1). The lifted formula is then passed to the sub-procedure _Compile, which traverses recursively over the

Algorithm 2 LiftedCompile

```
Input: DNF lineage $\varphi$ for query $Q$ on database $D$
Output: d-tree for $\varphi$
1: $(\psi, \ell) \leftarrow$ Lift$(\varphi, \ell')$ where $\ell'$ is identity function on vars$(\varphi)$
2: return _Compile$(\psi, \ell)$

3: procedure _Compile$(\psi, \ell)$
4:     if $\psi$ is a variable $x$ then return $\ell(x)$
       switch $\psi$
5:         case $\psi_1 \vee \cdots \vee \psi_n$ for independent $\psi_1, \ldots, \psi_n$
6:             $T_\psi \leftarrow \bigoplus_{i \in [n]}$ _Compile$(\psi_i, \ell|_{\psi_i})$
7:         case $\psi_1 \wedge \cdots \wedge \psi_n$ for independent $\psi_1, \ldots, \psi_n$
8:             $T_\psi \leftarrow \bigodot_{i \in [n]}$ _Compile$(\psi_i, \ell|_{\psi_i})$
9:         default
10:            Pick a most frequent variable $y$ in $\psi$
11:            $T_\psi \leftarrow (y \odot$ _Compile(Lift$(\psi[y := 1], \ell|_{\psi[y:=1]}))) \oplus$
                   $(\neg y \odot$ _Compile(Lift$(\psi[y := 0], \ell|_{\psi[y:=0]}))$
       return $T_\psi$
```

structure of ψ. If ψ is a single variable x, the procedure returns $\ell(x)$, which is a read-once formula (Line 4). If ψ is a disjunction of independent sub-formulas $\psi_1, \ldots, \psi_n$, it first constructs the d-trees for $(\psi_1, \ell|_{\psi_1}), \ldots, (\psi_n, \ell|_{\psi_n})$, where $\ell|_{\psi_i}$ is the restriction of the domain of ℓ onto the variables of ψ_i. Then, it combines the d-trees using $\oplus$ (Line 5). If ψ is a conjunction of independent sub-formulas $\psi_1, \ldots, \psi_n$, the procedure combines the d-trees of the sub-formulas using $\odot$ (Line 7). Otherwise, it performs Shannon expansion on a variable y as follows. It constructs d-trees T_0 for Lift$(\psi[y := 0], \ell|_{\psi[y:=0]})$ and T_1 for Lift$(\psi[y := 1], \ell|_{\psi[y:=1]})$. Then, in combines these d-trees using the $\sqcup_{\ell(y)}$ gate with 0-branch T_0 and 1-branch T_1. (Lines 10-11). The lifting procedure is applied to $\psi[y := 1]$ and $\psi[y := 0]$, since the substitution of y by a constant can create new equivalences between other variables.

Example 3.5. We apply Algorithm 2 to the lineage of Q_1 in Table 1 (top left). The saturated lifted lineage (φ, ℓ) constructed in Line 1 is given in Table 1 (top right). Since the variable $d_{1,2}$ appears in all clauses of φ, we factor out $d_{1,2}$ from all clauses and decompose the formula using independent-and: $d_{1,2} \odot \big((a_{1,3} \wedge m_3) \vee (a_{1,3} \wedge m_2) \vee (a_2 \wedge m_3)\big)$ and proceeds recursively on both sides of $\odot$. Since ℓ maps $d_{1,2}$ to the read-once formula $d_1 \vee d_2$, we return the latter formula for $d_{1,2}$. The formula on the right side of $\odot$ can only be decomposed using Shannon expansion on one of its variables. We choose $a_{1,3}$, one of its most frequent variables. We then perform lifting on each of the new formulas resulted from the Shannon expansion. This process continues until we obtain the d-tree from Figure 2 (ignore for now the numeric values appearing next to nodes).

Examples 3.4 and 3.5 demonstrate two benefits of our lifting approach over compilation without lifting: faster compilation time and smaller resulting d-tree. The compilation is more efficient since the lifted lineage can be small and Shannon expansion can be applied to the new fresh variables, which can represent large formulas.

3.2 Gradient-Based Computation

We propose a novel approach for Banzhaf and Shaply computation, yielding the following result:

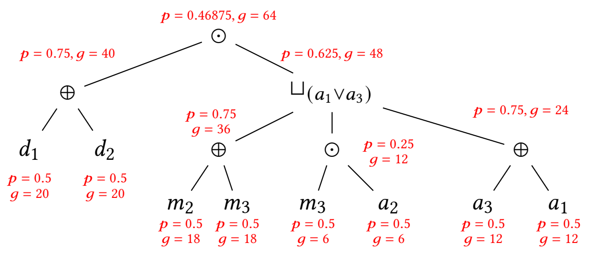


Figure 2: Lifted d-tree for the lineage in Figure 1. Each node is annotated with probability p and partial derivative g computed during the Banzhaf value computation with the gradient approach. Node m_3 is shown once for lack of space.

Theorem 3.6. For a given d-tree T, the Banzhaf and Shapley values of all variables in T can be computed in time $O(|T|)$ and respectively $O(|T| \cdot |\text{vars}(T)|^2)$.

Our approach interprets a d-tree as a function over random variables and reduces the computation of Banzhaf and Shapley values to the evaluation and computation of the gradient of this function.[2] We first introduce our approach for the computation of Banzhaf values and then show how to extend it to Shapley values.

Probabilistic Interpretation of D-Trees. We associate with each Boolean variable x occurring in a d-tree leaf, a variable p_x taking values in $[0, 1]$. A value assigned to p_x is the probability for $x = 1$. $Pr[T]$ is then a function over these variables, defined recursively using the definitions in Table 2. Its value is the probability that the Boolean formula represented by the d-tree T evaluates to 1.

From $Pr[T]$ to Banzhaf values. We next connect the Banzhaf value of a variable x to the partial derivative of $Pr[T]$ w.r.t. x.

Proposition 3.7. Given a d-tree T and a variable $x \in \text{vars}(T)$, the following holds:

$$\text{Banzhaf}(T, x) = 2^{|\text{vars}(T)|-1} \cdot \left(\frac{\partial Pr[T]}{\partial p_x}(\vec{\frac{1}{2}}) \right)$$

where p_x is the Boolean variable corresponding to x, $\frac{\partial Pr[T]}{\partial p_x}$ is the partial derivative of $Pr[T]$ with respect to p_x, and $\vec{\frac{1}{2}}$ is the column vector $(\frac{1}{2}, \ldots, \frac{1}{2})$ of length $|\text{vars}(T)|$.

Proof. We can rewrite $Pr[T]$ as

$$Pr[T] = \sum_{S \subseteq \text{vars}(T)} Pr[S] \cdot T[S] = \sum_{S \subseteq \text{vars}(T)} \left(\prod_{x \in S} p_x \cdot \prod_{y \notin S} (1 - p_y) \right) T[S],$$

where $T[S]$ is the Boolean value of the formula represented by T under the valuation S. Given a variable $x \in \text{vars}(T)$, we separately consider for each valuation S the cases where x appears in S and where it does not, and obtain:

$$Pr[T] = \sum_{S \subseteq \text{vars}(T) \setminus \{x\}} \Bigg(\prod_{z \in S} p_z \cdot \prod_{y \notin S \cup \{x\}} (1 - p_y) \\ \cdot \left(p_x \cdot T[S \cup \{x\}] + (1 - p_x) \cdot T[S] \right) \Bigg)$$

[2]There are two seminal works related to our gradient-based approach: The Baur-Strassen result on the efficient computation of a multivariate function defined by an arithmetic circuit and its gradient vector [5], and the Darwiche result on efficient inference in Bayesian networks [9]. To the best of our knowledge, no prior work uses gradient-based computation for Banzhaf/Shapley values.

Table 2: Equations defining the probability and the partial derivatives for different gates. T is the d-tree rooted at the gate and the T_i's are its child sub-trees.

Gate	Probability Expression $Pr[T]$	Partial Derivative
$\oplus$	$1 - \prod_{i \in [n]} (1 - Pr[T_i])$	$\frac{\partial(Pr[T])}{\partial(Pr[T_i])} = \prod_{j \in [n] \setminus \{i\}} (1 - Pr[T_j]) = \frac{1 - Pr[T]}{1 - Pr[T_i]}$
$\odot$	$\prod_{i \in [n]} Pr[T_i]$	$\frac{\partial(Pr[T])}{\partial(Pr[T_i])} = \prod_{j \in [n] \setminus \{i\}} Pr[T_j] = \frac{Pr[T]}{Pr[T_i]}$
$\sqcup_f$	$Pr[f] \cdot Pr[T_1]$ $+(1 - Pr[f]) \cdot Pr[T_0]$	$\frac{\partial(Pr[T])}{\partial(Pr[T_1])} = Pr[f]$ $\frac{\partial(Pr[T])}{\partial(Pr[T_0])} = 1 - Pr[f]$ $\frac{\partial(Pr[T])}{\partial(Pr[f])} = Pr[T_1] - Pr[T_0]$

By taking the partial derivative of the above function with respect to p_x at the point $\vec{\frac{1}{2}}$, we get:

$$\frac{\partial Pr[T]}{\partial p_x}(\vec{\frac{1}{2}}) = \sum_{S \subseteq \text{vars}(T) \setminus \{x\}} \left(\frac{1}{2}\right)^{|\text{vars}(T)|-1} \cdot (T[S \cup \{x\}] - T[S])$$
$$= \left(\frac{1}{2}\right)^{|\text{vars}(T)|-1} \cdot \text{Banzhaf}(T, x) \qquad \square$$

Efficient Computation. Table 2 gives the computation rules for the probability of a d-tree T based on the probabilities of its sub-trees; this follows standard probability theory. For each sub-tree T_i of T, we can also compute the partial derivative of $Pr[T]$ w.r.t. $Pr[T_i]$, where the latter is considered a variable in the former. Using the chain rule, we can then compute the gradient of $Pr[T]$ w.r.t. the variables at the leaves of T.

The procedure GradientBanzhaf in Algorithm 3 computes probabilities and partial derivatives at $\vec{\frac{1}{2}}$ (normalized by $2^{|\text{vars}(T)|-1}$) as follows. Given a d-tree T_φ, the algorithm gradually computes two quantities for each tree node v: $v.p$ which is the probability of T_v, and $v.g$ which is $\frac{\partial Pr(T_\varphi)}{\partial Pr(T_v)}(\vec{\frac{1}{2}})$, where T_v is the d-tree rooted at v. First, $v.p$ values are initialized to $\frac{1}{2}$ and the $v.g$ value of the root is initialized to $2^{|\text{vars}(T_\varphi)|-1}$ to account for normalization. Probabilities are computed in a bottom-up fashion, using the expressions in the second column of Table 2 (Lines 1 - 3). Partial derivatives are computed in a top-down fashion, using the expressions in the third column of Table 2 and the chain rule (Lines 5-6). For a variable v, the sum of partial derivatives for different leaf nodes representing v is equal to $\frac{\partial Pr[T]}{p_v}(\vec{\frac{1}{2}})$ which equals its Banzhaf value by Prop. 3.7.

Example 3.8. Reconsider the d-tree from Figure 2, where its nodes are now annotated with p and g values. We explain the computation for the $\oplus$ node v that is the left child of the root. Its p value is computed using the equation for $\oplus$ from Table 2 and the probability of its sub-trees d_1 and d_2: $v.p = 1 - (1 - d_1.p) \cdot (1 - d_2.p) = 1 - 0.5 \cdot 0.5 = 0.75$. Its g value is: $v.g = v.parent.g \cdot \frac{\partial v.parent.p}{\partial v.p} = v.parent.g \cdot \frac{v.parent.p}{v.p} = 64 \cdot \frac{0.46875}{0.75} = 40$ where we chose the equation for the derivative according to the gate of v's parent.

From Banzhaf to Shapley values. Our approach can be adapted to Shapley values. We first extend our probability model to address assignment sizes. Every variable x is assigned a probability p_x of having a satisfying assignment of size 1. By definition, each variable's probability of having an unsatisfying assignment of size 0 is $\frac{1}{2}$. Therefore, a variable x with probability p_x has a probability of $\frac{1}{2}$ to have an unsatisfying assignment of size 0, a probability of $\frac{1}{2} - p_x$ of having an unsatisfying assignment of size 1 and a probability of p_x of having a satisfying assignment of size 1.

We now define the k-probability function that is equal to the probability that a d-tree has a satisfying assignment of size k. For a given d-tree T and an integer $k \in \mathbb{N}$

$$Pr_k[T] = \sum_{S \subseteq vars(T), |S|=k} \sum_{S' \subseteq S} T[S'] \cdot \prod_{y \in S'} p_y \prod_{z \in S \setminus S'} (\frac{1}{2} - p_z) \prod_{w \in vars(T) \setminus S} \frac{1}{2}$$

The Shapley values are linked to the $k-probabilities$ and derivatives using the following proposition:

Proposition 3.9. Given a d-tree T and a variable $x \in \text{vars}(T)$, the following holds:

$$\text{Shapley}(T, x) = 2^{|\text{vars}(T)|-1} \cdot \sum_{k \in [|\text{vars}(T)|]} C_{k-1} \cdot \left(\frac{\partial Pr_k[T]}{\partial p_x} (\vec{\frac{1}{2}}) \right)$$

where $C_k = \frac{|k!| \cdot ||\text{vars}(T)| - k - 1|!}{|\text{vars}(T)|!}$ is the shapely coefficient.

Proof. Given a variable $x \in \text{vars}(T)$, we separately consider for each valuation S and subset S' the cases where x appears in S', in $S \setminus S'$ or in $vars(T) \setminus S$, and obtain:

$$Pr_k[T] = \sum_{S \subseteq vars(T), |S|=k, x \in S} \sum_{S' \subseteq S \setminus \{x\}} \left(T[S'](\frac{1}{2} - p_x) + T[S' \cup \{x\}] p_x\right) \cdot \prod_{y \in S'} p_y \prod_{z \in S \setminus S' \cup \{x\}} (\frac{1}{2} - p_z) \prod_{w \in vars(T) \setminus S} \frac{1}{2}$$
$$+ \sum_{S \subseteq vars(T) \setminus \{x\}, |S|=k} \sum_{S' \subseteq S} T[S'] \cdot \prod_{y \in S'} p_y \prod_{z \in S \setminus S'} (\frac{1}{2} - p_z) \prod_{w \in vars(T) \setminus S} \frac{1}{2}$$

By taking the partial derivative of the above function with respect to p_x we get:

$$\sum_{S \subseteq vars(T), |S|=k, x \in S} \sum_{S' \subseteq S \setminus \{x\}} \left(T[S' \cup \{x\}] - T[S']\right) \cdot \prod_{y \in S'} p_y \prod_{z \in S \setminus S' \cup \{x\}} (\frac{1}{2} - p_z) \prod_{w \in vars(T) \setminus S} \frac{1}{2}$$

Evaluating the partial derivative at the point $\vec{\frac{1}{2}}$ we obtain:

$$\sum_{S \subseteq vars(T), |S|=k, x \in S} \left(T[S] - T[S \setminus \{x\}]\right) \cdot \prod_{y \in S} \frac{1}{2} \prod_{w \in vars(T) \setminus S} \frac{1}{2}$$
$$= (\frac{1}{2})^{|vars(T)|-1} \sum_{S \subseteq vars(T) \setminus x, |S|=k-1} \left(T[S \cup \{x\}] - T[S]\right)$$

Where the first equality holds from taking only the subsets S' without multiplications of $1 - p_y$ since those would be equal to zero, and the second equality holds by reindexing the summation. Finally, we can sum up the partial derivatives multiplied by the relevant constants and get:

$$\sum_{k \in [|\text{vars}(T)|]} C_{k-1} \cdot \left(\frac{\partial Pr_k[T]}{\partial p_x} (\vec{\frac{1}{2}}) \right) = (\frac{1}{2})^{|\text{vars}(T)|-1}$$
$$\cdot \sum_{k \in [|\text{vars}(T)|]} C_{k-1} \cdot \sum_{S \subseteq vars(T) \setminus x, |S|=k-1} \left(T[S \cup \{x\}] - T[S]\right)$$
$$= \sum_{S \subseteq vars(T) \setminus x} C_{|S|} \left(T[S \cup \{x\}] - T[S]\right) = (\frac{1}{2})^{|\text{vars}(T)|-1} \text{Shapley}(T, x)$$

□

Algorithm 3 GradientBanzhaf

```
Input: D-tree $T_\varphi$ for formula $\varphi$
Output: Banzhaf values for all variables in $\varphi$
1: Initialize $v.p \leftarrow \frac{1}{2}$ for each variable node $v$
2: for each node $v$ in the tree in bottom-up order do
3:     Compute $v.p$ according to Table 2 and $v.gate$
4: $T.root.g = 2^{|\text{vars}(\varphi)|-1}$
5: for each node $v$ excluding $T.root$ in top-down order do
6:     $v.g \leftarrow v.parent.g \cdot \frac{\partial v.parent.p}{\partial v.p}$ according to Table 2 and $v.parent.gate$
7: for each variable $x \in \text{vars}(\varphi)$ do
8:     $\text{Banzhaf}(T_\varphi, x) \leftarrow \sum_{\text{node } v \text{ in } T \text{ for variable } x} v.g$
9: return $\text{Banzhaf}(T_\varphi, x)$ for all variables $x \in \text{vars}(\varphi)$
```

To compute Shapley values, the equations in Table 2 are adapted to compute $Pr_k[T]$ instead of $Pr[T]$, and $\frac{\partial(\sum_k C_k \cdot \text{Pr}_k[T])}{\partial Pr_j[T_i]}(\vec{\frac{1}{2}})$ for each $j \in [n]$ instead of $\frac{\partial Pr[T]}{\partial Pr[T_i]}(\vec{\frac{1}{2}})$. Similar equations to those shown in Table 2 are obtained for these partial derivatives. Lines 1 - 3 are then changed to compute a vector of all k-probabilities for each node. Line 4 is modified to initialize the root's vector of derivatives to be the Shapley coefficients, and Lines 5 - 6 are modified to compute the partial derivative for each value of j.

Algorithms. The LExaBan (LExaShap) algorithm for computing Banzhaf (Shapley) values uses Algorithm 2 to compile the query lineage to a d-tree T and then Algorithm 3 (its adaptation to Shapley values) to T to compute Banzhaf (Shapley) values for all variables in T, or equivalently for all tuples contributing to the lineage.

Complexity. For each node in the d-tree T, its probability can be computed in time linear in the number of its children, so in time $O(|T|)$ for all nodes in T. For Banzhaf computation, the gradient, i.e., the vector of partial derivatives w.r.t. each leaf node can be computed in $O(1)$ per node using the equations in Table 2, so in $O(|T|)$ time for the entire gradient. For Shapley computation, we need to compute the k-probabilities at each node v and their partial derivatives; the number of the k-probabilities is $|\text{vars}(T_v)|$. Each $k-$probability and partial derivative can be computed in $O(|\text{vars}(T)|)$ time, resulting in both parts of the algorithm performing $O(|\text{vars}(T)|^2)$ computations per node.

4 ATTRIBUTION FOR AGGREGATE QUERIES

In this section, we introduce algorithms for computing Banzhaf and Shapley values for aggregate queries based on d-trees for semimodule expressions (Section 2).

4.1 Basic Algorithms

We separately consider linear aggregates (SUM and COUNT) and non-linear ones (MIN and MAX).

SUM *and* COUNT. Let $\Phi = \sum_i \varphi_i \otimes m_i$ be a semimodule expression where the sum uses the $+_M$-operator of the monoid, and let $\varphi = \bigvee \varphi_i$ be the Boolean part of Φ. In case the aggregate is SUM, $+_M$ is the arithmetic +, and we obtain:

$$\mathsf{Banzhaf}(\Phi, x) = \sum_{S \subseteq \mathsf{vars}(\varphi) \setminus \{x\}} \Phi[S \cup \{x\}] - \Phi[S]$$
$$= \sum_{S \subseteq \mathsf{vars}(\varphi) \setminus \{x\}} \sum_i \left(\varphi_i[S \cup \{x\}] - \varphi_i[S]\right) \cdot m_i = \sum_i \mathsf{Banzhaf}(\varphi_i, x) \cdot m_i$$

Analogously[3], it holds that $\mathsf{Shapley}(\Phi, x) = \sum_i \mathsf{Shapley}(\varphi_i, x) \cdot m_i$. For the COUNT aggregate, we set $m_i = 1$ for all i. Thus, to compute the Banzhaf/Shapley value for a variable x in Φ, we (1) construct a d-tree T_i for each φ_i, (2) compute for each T_i and each $x \in \varphi_i$ the Banzhaf/Shapley value for x as in Section 3, and (3) derive the Banzhaf/Shapley of x in Φ using the above equations.

MIN *and* MAX. For MIN and MAX which are not linear, we adopt a different approach. We first construct a d-tree for the entire semimodule expression and then compute Banzhaf/Shapley values from the d-tree. The former was shown in [12] and we next show how the latter is performed.

Let $\mathsf{PV}_\Phi = \{\Phi(\theta) | \theta \subseteq \mathsf{vars}(\Phi)\}$ be the set of values to which the semimodule expression Φ can evaluate to under possible valuations. If Φ is the lineage of an aggregate query $\langle \text{MIN}, \gamma, Q\rangle$ or $\langle \text{MAX}, \gamma, Q\rangle$, the size of PV_Φ is bounded by the size of the result of Q, which is polynomial in the input database size. We next relate Banzhaf/Shapley values of a variable x in a semimodule expression with the model counts of the expression under substitutions that set x to 0 or 1:

Proposition 4.1. Given a semimodule expression Φ and a variable $x \in \mathsf{vars}(\Phi)$, it holds:

$$\mathsf{Banzhaf}(\Phi, x) = \sum_{p \in \mathsf{PV}_\Phi} (\#^p\Phi[x := 1] - \#^p\Phi[x := 0]) \cdot p$$
$$\mathsf{Shapley}(\Phi, x) = \sum_{p \in \mathsf{PV}_\Phi, k \in [|\mathsf{vars}(\Phi)|]} \left[(\#^p_k\Phi[x := 1] - \#^p_k\Phi[x := 0]) \cdot p\right] \cdot C_k$$

where $C_k = \frac{|k!| \cdot ||\mathsf{vars}(\Phi)| - k - 1|!}{|\mathsf{vars}(\Phi)|!}$.

We can further show, via a dynamic programming algorithm:

Proposition 4.2. Let T be a d-tree representing a semimodule expression $\Phi \in \mathsf{PosBool}(X) \otimes M$ s.t. $M = (\overline{\mathbb{R}}, \max, -\infty)$ or $M = (\overline{\mathbb{R}}, \min, \infty)$. Let $p \in \mathsf{PV}_\Phi$ and $k \in [|\mathsf{vars}(T)|]$. We can compute $\#^p\Phi$ and $\#^p_k\Phi$ in PTIME.

Proof. We show the computation for binary gates and the results may easily be extended to handle non-binary gates as well. In the following, let $M \in \{\min, \max\}$ be the monoid.

Lemma 4.3. Let Φ be a semimodule expression and let ψ be a boolean formula such that ψ and the boolean part of Φ are independent. Let $S = \psi \otimes \Phi$ and $p \in M$, $p \neq 0_M$, then:

$$\#^p S = \#\psi \cdot \#^p\Phi$$
$$\#^p_k S = \sum_{i \in [k]} \#_i\psi \cdot \#^p_{k-i}\Phi$$

Proof. The equations hold because each assignment (of size k) where S evaluates to p is constructed out of a satisfying assignment of ψ and an assignment for Φ that evaluates to p (where the combined sizes are k). □

Lemma 4.4. Let $S = \Phi_1 +_M \Phi_2$ such that Φ_1 and Φ_2 are independent, then:

$$\#^p S = \sum_{r,t \in PV_\Phi, r +_M t = p} (\#^r\Phi_1 \cdot \#^t\Phi_2)$$
$$+ \#^p\Phi_1 \cdot \left(2^{|\mathsf{vars}(\Phi_2)|} - \#\Phi_2\right) + \#^p\Phi_2 \cdot \left(2^{|\mathsf{vars}(\Phi_1)|} - \#\Phi_1\right)$$
$$\#^p_k S = \sum_{i \in [k]} \Bigg[\sum_{r,t \in PV_\Phi, r +_M t = p} (\#^r_i\Phi_1 \cdot \#^t_{k-i}\Phi_2)$$
$$+ \#^p_i\Phi_1 \cdot \left(\binom{|\mathsf{vars}(\Phi_2)|}{k-i} - \#_{k-i}\Phi_2\right) \quad + \#^p_{k-i}\Phi_2 \cdot \left(\binom{|\mathsf{vars}(\Phi_1)|}{i} - \#_i\Phi_1\right) \Bigg]$$

Proof. The equations hold because every satisfying assignment (of size k) where S evaluates to p is constructed out of a assignment for which Φ_1 evaluates to r and an assignment for which Φ_2 evaluates to t (where the sum of assignments sizes is k) such that $r +_M t = p$. This can be seen as the sum of such assignments, by whether Φ_1 alone is satisfied, Φ_2 alone is satisfied, or both. □

Lemma 4.5. Let Φ_1, Φ_2 be exclusive semimodule expressions s.t. Φ_1 and Φ_2 have identical sets of variables. Let $S = \Phi_1 +_M \Phi_2$, then:

$$\#^p S = \#^p\Phi_1 + \#^p\Phi_2$$
$$\#^p_k S = \#^p_k\Phi_1 + \#^p_k\Phi_2$$

Proof. Since the satisfying assignments for the boolean parts of the semimodule expressions are disjoint, any assignment (of size k) of S that evaluates to p is either an assignment (of size k) that evaluates to p of Φ_1 or an assignment that evaluates to p of Φ_2. □

Having shown the computation for each type of gate, a bottom-up algorithm to compute assignment counts for all (polynomialy many) relevant values p and k and all d-tree nodes follows directly, thereby concluding the proof of Proposition 4.2. □

Example 4.6. Figure 3 presents a d-tree for the lineage of Q_2 from Table 1, along with some steps of the model counts computation. Denote the $\oplus$ node that is the right child of the right child of the root by T. Its left subtree has one assignment with value of 322, and its right subtree has one assignment with value of 377. Applying the equation from Lemma 4.4 we get that $\#^{377}T = 1 \cdot 1 + 1 \cdot 1 + 0 \cdot 1 = 2$, as seen in the Figure.

Combined, the two propositions provide a PTIME algorithm for computing Banzhaf and Shapley values given a d-tree representation of the lineage of an aggregate query.

[3]This is analogous to the observation already made in [19].

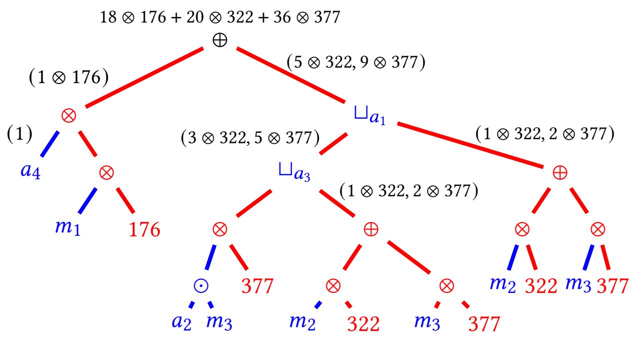


Figure 3: A d-tree for the semimodule expression lineage of Q_2 from Table 1. The blue part represents the boolean component of the expression while the red one represents a semimodule component.

4.2 Lifting for Aggregates

We extend our technique of lifting variables presented in Section 3.1 from UCQs to aggregate queries. First, for linear aggregates (SUM and COUNT), our solution in Section 4.1 constructs a separate d-tree for each formula φ_i in the Boolean part $\bigvee_i \varphi_i$ of the given semimodule expression. Thus, we can apply the procedures Lift and LiftedCompile in Algorithms 1 and 2 to each formula φ_i. For MIN and MAX, new machinery is needed. In the following, we show the construction for MAX. The construction for MIN is analogous.

Extending the Definitions. Given disjoint variable sets $\boldsymbol{X}$ and $\boldsymbol{Y}$, a *lifted semimodule expression* over $\boldsymbol{X} \cup \boldsymbol{Y}$ is a pair (φ, ℓ) where φ is a Boolean formula over $\boldsymbol{X} \cup \boldsymbol{Y}$ and $\ell : \text{vars}(\varphi) \to \text{PosBool}(\boldsymbol{X}) \cup (\text{PosBool}(\boldsymbol{X}) \otimes M)$ is a mapping from variables in φ to either semimodule expressions or Boolean formulas. If φ is in DNF, we refer to (φ, ℓ) as a *lifted DNF semimodule expression*. The *inlining* $\text{inline}(\varphi, \ell)$ is obtained in two steps: First, we replace each variable $x \in \text{vars}(\varphi)$ by $\ell(x)$; then, we replace each $\lor$ between semimodule expressions by $+_M$ and each $\land$ between a formula and a semimodule expression by $\otimes$. We call a lifted semimodule expression *valid* if exactly one variable in each DNF caluse is mapped to a semimodule expression.

Translating Lineage to a Lifted DNF Semimodule Expression. Recall that the lineage of an aggregate query is captured by a semimodule expression $\Phi = \sum \varphi_i \otimes m_i$, where each $\varphi_i = \bigvee_j C_{ij}$ is in DNF. We transform Φ to the equivalent form $\Phi' = \sum_{i,j} C_{i,j} \otimes m_i$. This preserves equivalence for $(\overline{\mathbb{R}}, \max, -\infty)$ and $(\overline{\mathbb{R}}, \min, \infty)$ due to their idempotence. We further transform Φ' into a lifted DNF semimodule expression (φ, ℓ) with $\varphi = \bigvee_{i,j}(C_{i,j} \land w_i)$ and $\ell = \{w_i \mapsto m_i\} \cup \{x \mapsto x \mid x \in \text{vars}(\varphi) \text{ and } x \neq w_i\}$. Note that (φ, ℓ) is valid and its inlining is equivalent to Φ. We add this translation as an initial preprocessing step to Algorithm 2.

Extending the Lifting Algorithm. Given a lifted DNF semimodule expression (φ, ℓ), cofactor-equivalence and interchangeability are defined as in the case of lifted Boolean DNF formulas. The application of lift-and or lift-or to a set of cofactor-equivalent or respectively interchangeable variables maintains the validity of the expression, which can be verified using Definition 2.6. Algorithm 1 works without changes for lifted DNF semimodule expressions.

Example 4.7. Consider the semimodule expression that is the lineage of Query Q_2 in Table 1. We first translate the expression to the lifted DNF semimodule expression $\varphi = (a_1 \land m_3 \land t_1) \lor (a_2 \land m_3 \land t_1) \lor (a_3 \land m_3 \land t_1) \lor (a_1 \land m_2 \land t_2) \lor (a_3 \land m_2 \land t_2) \lor (a_4 \land m_1 \land t_3)$; $\ell = \{t_1 \mapsto 377, t_2 \mapsto 322, t_3 \mapsto 176\} \cup \{x \mapsto x | x \in \text{vars}(\varphi) \setminus \{t_1, t_2, t_3\}\}$. Then, we execute the initial lifting in Algorithm 2. The algorithm searches for cofactor-equivalent or interchangeable sets of variables. It detects that a_1 and a_3 are cofactor-equivalent, thus, it performs lift-or on them. It also detects that each of the sets $\{m_3, t_1\}$, $\{m_2, t_2\}$, and $\{a_4, m_1, t_3\}$ are interchangeable and thus performs lift-and on them. The resulting expression after these operations is given in Table 1. The algorithm continues compilation as described in Section 3.1 to achieve a d-tree for the lineage.

4.3 Gradients for Aggregates

We now adapt the gradient-based computation from Section 3.2 to the MAX aggregate (the case of MIN is similar). In contrast to d-trees with Boolean outcomes, a semimodule expression can yield a numeric value for a given valuation over its Boolean variables. Our probabilistic interpretation thus looks at *expected values*.

Expected Values and Gradients. Given d-tree T for s semimodule expression Φ, we denote by $E[T]$ the expected value of the d-tree given probabilities p_x for the boolean variables. That is: $E[T] = \sum_{S \subseteq \text{vars}(T)} Pr[S] \cdot T[S]$ (where in contrast to Section 3.2, T[S] is now not necessarily in $[0, 1]$). Proposition 3.7 then trivially extends to expected values:

$$\text{Banzhaf}(T, x) = 2^{|\text{vars}(T)|-1} \cdot \left(\frac{\partial E[T]}{\partial p_x}\left(\vec{\frac{1}{2}}\right)\right) \tag{4}$$

We further extend the definition of probabilities and k-probabilities, so that $Pr[T = m_i] = \sum_{S \subseteq \text{vars}(T), T[S]=m_i} Pr[S]$ and $Pr_k[T = m_i] = \sum_{S \subseteq \text{vars}(T), |S|=k, T[S]=m_i} Pr[S]$.

Now, using the linearity of expectation, we can write:

$$\frac{\partial E[T]}{\partial p_x} = \sum_{m_i \in \text{PV}_\Phi} m_i \cdot \frac{\partial Pr[T = m_i]}{\partial p_x}$$

From Gradients to Banzhaf and Shapley. Similarly to Section 3.2, we obtain Banzhaf and Shapley values of all facts via partial derivatives. Concretely, we introduce an order over the variables, and then use variable names as indices (identifying each variable with its location in the order). Let J be the Jacobian matrix, namely $J_x^i = \frac{\partial(v_i \cdot Pr[T=v_i])}{\partial p_x}$. Further let J^k be the matrix defined by $J_x^{k,i} = \frac{\partial(v_i \cdot Pr_k[T=v_i])}{\partial p_x}$ for each variable x. We can show:

$$\text{Banzhaf}(T, x) = 2^{|\text{vars}(T)|-1} \cdot \sum_{i \in [n]} J_x^i\left(\vec{\frac{1}{2}}\right)$$

$$\text{Shapley}(T, x) = 2^{|\text{vars}(T)|-1} \cdot \sum_{i \in [n]} \sum_{k \in [|\text{vars}(T)|]} J_x^{k,i}\left(\vec{\frac{1}{2}}\right) \cdot C_{k-1}$$

where $C_k = \frac{|k!| \cdot |D_n - k - 1|!}{|D_n|!}$.

Banzhaf and Shapley Computation. We adapt Algorithm 3 to compute Jacobians via back-propagation, for a d-tree representing the lineage of an aggregate query. The adaptations are as follows. Lines 1-3 are replaced with the computation of the probability for each different possible value. The root derivative in Line 4 is initialized to a vector of possible values. Lastly, the computation in

Lines 5-6 is replaced with the computation of the Jacobian matrix values. This allows us to compute a vector/matrix of derivatives at each node and obtain the Banzhaf/Shapley values at the leaves according to Equation 4.

5 EXPERIMENTS

This section details our experimental setup and results.

5.1 Experimental Setup and Benchmarks

We implemented all algorithms in Python 3.11 and performed experiments on a Linux Debian 14.04 machine with 1TB of RAM and an Intel(R) Xeon(R) Gold 6252 CPU @ 2.10GHz processor.

Algorithms. We benchmarked variants of our algorithm LExaBan, with and without the two optimization techniques, for SPJU and aggregate queries. We also compared to two state-of-the-art competitors that also use compilation to d-trees, but only work for SPJU queries and do not include the variable lifting and gradient optimizations [2]: ExaBan for exact computation and AdaBan for deterministic approximate computation with error guarantees. AdaBan was shown [2] to be successful for instances where exact algorithms failed and proved to be significantly faster and more accurate than a Monte-Carlo randomized approximation from prior work [11]. For Shapley value computation, we benchmarked our algorithm LExaShap and compared it to the state-of-the-art competitor ExaShap [2] that is the Shapley variant of ExaBan. For aggregate queries with linear aggregates, we also compared to a variant where we use ExaBan as the algorithm of choice for each boolean formula and then use linearity as explained in Section 4.1 to compute Banzhaf values with respect to the aggregate queries.

Datasets. We tested the algorithms using 301 queries evaluated over three datasets: Academic, IMDB and TPC-H (SF1). The workload (see Table 3) is based on prior work on Banzhaf/Shapley values for query answering [2, 11]. The lineage for all output tuples of these queries was constructed using ProvSQL [30]. We created two variants of each benchmark: one including SPJU queries, and one including aggregate queries. The SPJU variant includes all queries from IMDB and Academic as is (they do not include aggregate queries), and all queries without nested subqueries and with aggregates removed from TPC-H. The aggregate queries variant (named $Academic_{agg}$, $IMDB_{agg}$, $TPC-H_{agg}$) was constructed as follows. For TPC-H it includes 7 queries that were preserved as is, namely without removing aggregation. These are the queries for which ProvSQL managed to compute all lineages in an hour or less. To further extend the benchmark, we created multiple variants of each of these queries, by changing the aggregation function to MAX, COUNT and SUM (the solution for MIN is almost identical to that of MAX). For IMDB and Academic, we created a synthetic aggregate variant with each of MAX, COUNT and SUM introduced to each query, and applied over randomly chosen values. These datasets are highly challenging, including very large semimodule lineage expressions. Overall, the resulting dataset consists of nearly 1M Boolean lineage expressions and 429 semimodule lineage expressions for each aggregate.

Measurements. We define an instance as the computation of the Banzhaf or Shapley values for all variables in a lineage of an output tuple of a query over a dataset. We report the success rate of each algorithm over the instances, where failure is declared in case an algorithm did not terminate an instance within one hour. We also report statistics of runtimes across all instances (average, median, maximal runtime, and percentiles). The pX columns in the tables give the execution times for the X-th percentile of the instances.

Table 3: Statistics of the datasets used in the experiments.

Dataset	#Queries	# Lineages	#Vars avg/max	#Clauses avg/max
Academic	92	7,865	79 / 6,027	74 / 6,025
IMDB	197	986,030	25 / 27,993	15 / 13,800
TPC-H	12	165	1,918 / 139,095	863 / 75,983
TPC-H_{agg}	7	140	746 / 6959	233 / 2076

5.2 Summary of Experimental Findings

We next overview our main findings.

(1) For queries without aggregates, LExaBan consistently and significantly outperforms the state-of-the-art algorithm ExaBan for Banzhaf computation and even the AdaBan approximation algorithm. LExaBan was consistently faster than these baselines, achieving a speedup of up to 3 orders of magnitude for the hardest instances. This speedup also leads to LExaBan achieving success on more than 90% of the boolean lineage expressions where ExaBan failed, (and more than 99.98% of the boolean expressions overall). (2) LExaBan was successful on aggregate queries: For SUM and COUNT, LExaBan succeeded in less than 100th of a second on all TPC-H_{agg} dataset and achieved success on all instances of Academic_{agg} and over 95% of the instances of $IMDB_{agg}$, far surpassing the extension of ExaBan to these instances. The non-linear MAX aggregate is far more challenging, and no (variant of) prior work is applicable for MAX queries; yet LExaBan was successful for more than 93% of the instances of TPC-H_{agg}. It was successful for over 78% and 61% of the MAX queries from the highly challenging Academic_{agg} and $IMDB_{agg}$ synthetic datasets respectively.

(3) The lifting technique significantly improved the compilation used by all variants of our approach: compilation time improved by more than 2 orders of magnitude compared to compiling without the lifting technique. In addition, compilation resulted in 1-2 orders of magnitude smaller d-trees.

(4) The gradient technique proved to be highly effective for large instances, pushing the boundary of computable instances: The technique yielded speedups across all instance sizes, with improvements exceeding two orders of magnitude for the largest instances.

(5) The lifting and gradient techniques also improve Shapley value computation: LExaShap shows speed-ups of over two orders of magnitude over ExaShap. LExaShap achieves success on over 75% of the Boolean expression instances ExaShap failed on.

5.3 Banzhaf Computation for SPJU Queries

Success rate. Table 4 shows the success rates of LExaBan, ExaBan, and AdaBan, in the setting for which ExaBan and AdaBan were designed, namely queries without aggregates. LExaBan achieves higher success rates at both lineage and query levels for all datasets. On Academic, LExaBan succeeded on all instances. On IMDB, LExaBan achieves 99.98% success at the lineage level,

Table 4: Query success rate: Percentage of queries for which the algorithms finish for all instances of a query. Lineage success rate: Percentage of instances (over all queries) for which the algorithms finish. The timeout is one hour.

Dataset	Algorithm	Query Success Rate	Lineage Success Rate
Academic	LExaBan	100.00%	100.00%
	ExaBan	98.85%	99.99%
	AdaBan	98.85%	99.99%
IMDB	LExaBan	95.43%	99.98%
	ExaBan	81.73%	99.63%
	AdaBan	87.82%	99.81%
TPC-H	LExaBan	75.00%	93.33%
	ExaBan	58.33%	91.52%
	AdaBan	75.00%	92.73%

Table 5: Runtime of LExaBan, ExaBan and AdaBan. Note that the instance for Academic on which LExaBan is slowest is one on which ExaBan and AdaBan failed on.

Dataset	Algorithm	Success rate	Execution times [sec]					
			Mean	p50	p90	p95	p99	Max
Academic	LExaBan	100.00%	0.44	4E-4	2E-3	5E-3	0.14	3455
	ExaBan	99.99%	2.07	1E-3	0.01	0.20	164.5	563.5
	AdaBan	99.99%	0.76	1E-3	7E-3	0.05	60.05	173.7
IMDB	LExaBan	99.98%	0.18	1E-3	5E-3	0.01	0.09	1219
	ExaBan	99.63%	1.58	2E-3	0.02	0.08	10.37	1793
	AdaBan	99.81%	1.82	1E-3	0.01	0.05	7.81	1802
TPC-H	LExaBan	93.33%	0.12	7E-4	3E-3	0.62	1.95	8.73
	ExaBan	91.52%	4.23	0.89	0.94	51.05	61.98	69.18
	AdaBan	92.73%	2.37	3E-3	0.14	3.15	81.55	166.3

and more than 95% success at query level. LExaBan succeeds on more than 95% of instances on which ExaBan failed, and 90% of instances on which AdaBan failed. An improvement (albeit of a lesser extent) was also observed for TPC-H.

Runtime performance. Table 5 gives the runtime of LExaBan, ExaBan and AdaBan for the Academic, IMDB and TPC-H datasets. LExaBan outperforms both ExaBan and AdaBan significantly. Despite achieving success for more instances, it keeps a smaller mean runtime compared with ExaBan and AdaBan. For Academic, except for the unique instance where it achieves success and the other algorithms fail on, runtime is below 5 seconds, which represents 119X improvement over ExaBan, and 36X improvement over AdaBan. Similar trends are observed for the IMDB and TPC-H datasets, where LExaBan achieves 110X and respectively 106X improvements for the most expensive instance for ExaBan (Table 6). For the instances on which ExaBan failed (Table 7), LExaBan still achieves success and low runtimes. For IMDB, this is an almost 95% success rate with a median runtime of 4.73 seconds, representing at least 761X speedup for more than half of those instances. For TPC-H, LExaBan succeeded on 3 more instances; for one such instance LExaBan needed 1.95 seconds, which means a speedup of at least 1846X over ExaBan which did not terminate in 1 hour.

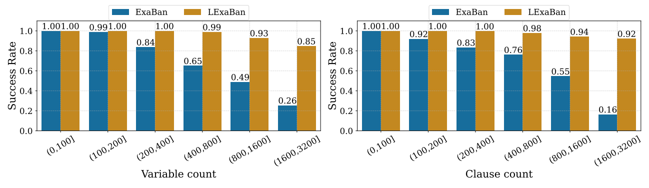
Effect of lineage structure on runtime performance. Fig. 4 shows the effect of the number of variables and clauses in the query lineage on the success rates and runtimes of LExaBan and ExaBan. When the number of variables/clauses increases, LExaBan's success rate remains higher, while its runtime increases slower than for ExaBan. This highlights the advantages of both our techniques: lifting helps LExaBan compile the lineage faster and into a smaller d-tree, while the gradient helps LExaBan share the computation among many variables (see further analysis below).

Table 6: Runtime of LExaBan and ExaBan on instances for which ExaBan succeeded.

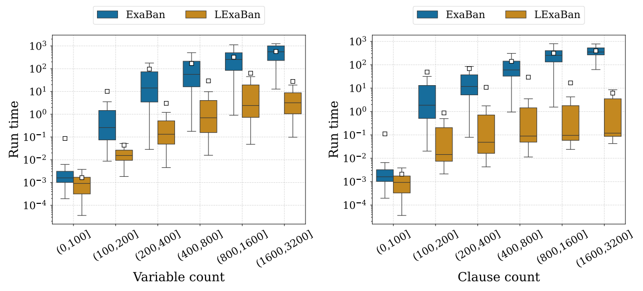
Dataset	algorithm	Execution times [sec]						
		Mean	p50	p75	p90	p95	p99	Max
Academic	LExaBan	4E-3	4E-4	1E-3	2E-3	5E-3	0.14	4.99
	ExaBan	2.07	1E-3	2E-3	0.01	0.20	164.5	563.5
IMDB	LExaBan	7E-3	1E-3	2E-3	4E-3	9E-3	0.05	16.26
	ExaBan	1.58	2E-3	3E-3	0.02	0.08	10.37	1793
TPC-H	LExaBan	0.04	7E-4	8E-4	1E-3	0.53	0.65	0.65
	ExaBan	4.23	0.89	0.93	0.94	51.05	61.98	69.18

Table 7: Runtime of LExaBan on instances for which ExaBan failed. There was one such instance in Academic.

Dataset	Success rate	Execution times [sec]						
		Mean	p50	p75	p90	p95	p99	Max
Academic	100%	3455	-	-	-	-	-	3455
IMDB	94.74%	48.97	4.73	22.88	168.2	375.9	448.2	1219
TPC-H	21.43%	4.21	1.95	5.34	7.38	8.05	8.60	8.73



(a) Success rate (average over all instances in each group)



(b) Runtime (ranges over all instances in each group)

Figure 4: Success rate and runtime of LExaBan and ExaBan for all queries, grouped by the number of variables or clauses in the lineage. An interval $[i, j]$ represents the set of lineages with #vars (# clauses) between i and j.

5.4 Banzhaf Computation for Aggregate Queries

We then tested LExaBan on aggregate queries. Table 8 shows the results for the TPC-H$_{agg}$ dataset. The performance is near-perfect for SUM and COUNT aggregates (incurring less than a millisecond for all instances). LExaBan also terminated quickly for most instances with MAX aggregate; specifically, it significantly outperforms a naive variant of LExaBan that does not include the optimizations.

For the synthetic and highly challenging $IMDB_{agg}$ and $Academic_{agg}$ datasets, we again observe good performance for SUM (and COUNT, for which the results were very similar and are omitted for lack of

Table 8: Success rate and runtime of LExaBan for the $TPC-H_{agg}$ aggregate queries. Max(Naive) refers to running LExaBan without optimizations on queries with MAX.

Aggregate	Success rate	Execution times [sec]						
		Mean	p50	p75	p90	p95	p99	Max
SUM	100%	2E-4	2E-6	2E-6	1E-3	2E-3	2E-3	2E-3
COUNT	100%	2E-4	1E-6	2E-6	1E-3	2E-3	2E-3	2E-3
MAX	93.57%	224.8	2E-3	3E-3	5E-3	2990	3349	3517
Max(Naive)	86.43%	3E-3	1E-3	1E-3	2E-3	3E-3	9E-3	0.26

Table 9: Success rate and runtime of LExaBan and ExaBan for queries with SUM aggregate (similar results for COUNT aggregate are not shown) and MAX aggregates.

Dataset	Algorithm & Aggregate	Success rate	Execution times [sec]					
			Mean	p50	p90	p95	p99	Max
$Academic_{agg}$	LExaBan $_{sum}$	100.00%	40.1	0.08	0.48	0.58	496	3561
	ExaBan$_{sum}$	97.70%	6.20	0.09	1.88	12.6	105	379.4
	LExaBan $_{max}$	78.26%	15.11	3E-3	37.04	104.5	253.4	289.1
$IMDB_{agg}$	LExaBan$_{sum}$	95.43%	37.54	0.06	9.77	71.56	774	2450
	ExaBan$_{sum}$	81.19%	62.7	0.27	65.8	117	1767	3286
	LExaBan $_{max}$	61.93%	52.25	0.09	23.79	242.6	734.9	2407

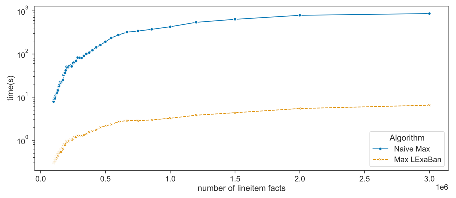


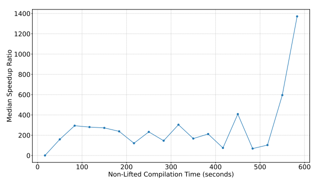
Figure 5: Runtime of LExaBan for a TPC-H query 5 with MAX aggregate, when varying the number of `Lineitem` facts.

space). Specifically, LExaBan has succeeded on all SUM query instances for $Academic_{agg}$ and 95.43% of such instances for $IMDB_{agg}$. When compared to a variant that uses ExaBan for the individual boolean formulas occurring in the semimodule expressions, LExaBan was clearly superior. For MAX, there is no such baseline, and LExaBan was still successful for over 78% for $Academic_{agg}$ and over 61% for $IMDB_{agg}$. Furthermore, for both datasets, it terminated in less than 0.1 of a second for over 50 % of the lineages.

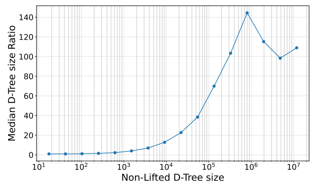
To better understand the performance over hard instances, we designed an experiment where we control the size of semimodule expressions by considering subsets of the tuples in the TPC-H `Lineitem` relation. Fig. 5 shows the average runtime for all lineages of a representative "hard" query (TPC-H Query 5). LExaBan is able to compute the Banzhaf value on average in about 1.6 seconds, when the number of `Lineitem` facts is 3 million, while without the lifting and gradient optimizations (dubbed Naïve in the figure), computation incurred on average over 10 minutes.

5.5 Effect of Lifting and Gradient Techniques

Fig. 6 shows that the variable lifting technique can speed-up the compilation of query lineage by over one order of magnitude on average and reduce the size of the constructed d-trees by more than two orders of magnitude. These benefits increase as we consider increasingly larger lineage expressions. This experiment considers



(a) D-tree compilation time with and without variable lifting.



(b) D-tree size with and without variable lifting.

Figure 6: Effect of variable lifting on LExaBan's compilation of lineage of Boolean queries into d-trees.

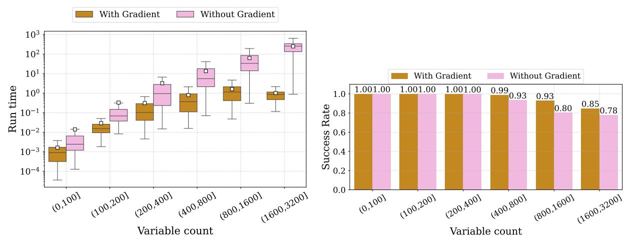


Figure 7: Effect of the gradient technique on the success rate and runtime of LExaBan for lineage of Boolean queries.

the lineage of all Boolean (Academic, IMDB, TPC-H) queries that can be processed within 10 minutes.

Fig. 7 shows the success rate and runtime of LExaBan with and without the gradient technique for the lineage of all Boolean (Academic, IMDB, TPC-H) queries. The technique was beneficial for all considered instances, and its benefit increased with the instance size. For instances with more than 1600 variables, the average speedup is more than 2 orders of magnitude. One might expect a speedup of at most n for instances with n variables, since the gradient technique allows to compute the Banzhaf value simultaneously for all n variables. However, not all branches of the d-tree participate in the Banzhaf computation for each variable, and thus there is potential for significant computation sharing across variables.

5.6 Shapley Computation

We also report on the success rate and runtime of LExaShap, our extension of LExaBan to compute the Shapley value. The main competitor is ExaShap from prior work [2], which also uses d-tree compilation but does not use our lifting and gradient techniques.

Table 10 shows the runtimes of LExaShap and ExaShap for instances on which the latter succeeds. Table 11 shows the success rates and runtimes of LExaShap for instances on which ExaShap fails. On average, LExaShap's speedup over ExaShap is more than one order of magnitude across all datasets. LExaShap also has a much higher success rate: 79% for the IMDB instances on which ExaShap fails, with a median time of 13.04 seconds; and 41% for the TPC-H instances on which ExaShap fails, with a median runtime of 23.94 seconds. This represents a speedup of at least 276X and respectively 150X.

Table 10: Runtimes for LExaShap and ExaShap on instances for which ExaShap succeeds.

Dataset	Algorithm	Execution times [sec]						
		Mean	p50	p75	p90	p95	p99	Max
Academic	LExaShap	0.02	2E-3	4E-3	9E-3	0.01	0.05	48.40
	ExaShap	0.45	1E-3	3E-3	0.04	0.22	0.37	1397
IMDB	LExaShap	0.07	6E-3	0.02	0.05	0.10	0.57	520.3
	ExaShap	3.35	3E-3	0.01	0.12	0.79	44.00	3574
TPCH	LExaShap	2E-3	2E-3	4E-3	5E-3	5E-3	5E-3	5E-3
	ExaShap	7E-3	3E-3	3E-3	5E-3	9E-3	0.02	0.56

Table 11: Runtimes for LExaShap on instances for which ExaShap fails.

Dataset	Success rate	Execution times [sec]						
		Mean	p50	p75	p90	p95	p99	Max
IMDB	79.13%	61.54	13.04	59.41	171.8	295.7	689.3	2835
TPCH	41.68%	22.90	23.94	26.82	27.89	29.02	29.92	30.14

6 CONCLUSION

In this paper, we have introduced a novel practical approach for computing Banzhaf and Shapley values in query answering. Such values can measure the contributions of the input facts to the query answer and can be also used to explain the query answer.

Our approach is both more general and more efficient than the state-of-the-art: its generality is manifested in its support for aggregate queries; its efficiency is due to the incorporation of two novel techniques, namely lifted compilation of query results lineage to decomposition trees, and gradient-based Banzhaf/Shapley values computation over decomposition trees. Our experimental evaluation shows that the approach is faster than the state-of-the-art by several orders of magnitude.

Future work includes extending our approach to broader query classes (e.g. queries with negation and recursion) and developing efficient approximation schemes for Banzhaf/Shapley values. Another line of work is exploring the applicability of our algorithms to other attribution measures such as the SHAP score.

ACKNOWLEDGMENTS

Ahmet Kara and Dan Olteanu would like to acknowledge the UZH Global Strategy and Partnerships Funding Scheme. The work of Omer Abramovich and Daniel Deutch was partially funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 804302).

REFERENCES

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

[2] Omer Abramovich, Daniel Deutch, Nave Frost, Ahmet Kara, and Dan Olteanu. 2024. Banzhaf Values for Facts in Query Answering. *Proc. ACM Manag. Data* 2, 3, Article 123 (2024), 26 pages. https://doi.org/10.1145/3654926

[3] Omer Abramovich, Daniel Deutch, Nave Frost, Ahmet Kara, and Dan Olteanu. 2025. LExaBan-LExaShap. https://github.com/Omer-Abramovich/LExaBan-LExaShap. Accessed: 2025-03-01.

[4] J.F. Banzhaf. 1965. Weighted voting doesn't work: A mathematical analysis. *Rutgers Law Review* 19, 2 (1965), 317–343.

[5] Walter Baur and Volker Strassen. 1983. The complexity of partial derivatives. *Theoretical Computer Science* 22, 3 (1983), 317–330. https://doi.org/10.1016/0304-3975(83)90110-X

[6] Leopoldo Bertossi, Benny Kimelfeld, Ester Livshits, and Mikaël Monet. 2023. The Shapley Value in Database Management. *SIGMOD Rec.* 52, 2 (Aug. 2023), 6–17. https://doi.org/10.1145/3615952.3615954

[7] Meghyn Bienvenu, Diego Figueira, and Pierre Lafourcade. 2024. Shapley value computation in ontology-mediated query answering. In *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning* (Hanoi, Vietnam) *(KR '24)*. Article 15, 11 pages. https://doi.org/10.24963/kr.2024/15

[8] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends Databases* 1, 4 (April 2009), 379–474. https://doi.org/10.1561/1900000006

[9] Adnan Darwiche. 2003. A differential approach to inference in Bayesian networks. *J. ACM* 50, 3 (2003), 280–305. https://doi.org/10.1145/765568.765570

[10] Adnan Darwiche and Pierre Marquis. 2002. A knowledge compilation map. *J. Artif. Int. Res.* 17, 1 (Sept. 2002), 229–264.

[11] Daniel Deutch, Nave Frost, Benny Kimelfeld, and Mikaël Monet. 2022. Computing the Shapley Value of Facts in Query Answering. In *SIGMOD*. 1570–1583. https://doi.org/10.1145/3514221.3517912

[12] Robert Fink, Larisa Han, and Dan Olteanu. 2012. Aggregation in probabilistic databases via knowledge compilation. *Proc. VLDB Endow.* 5, 5 (Jan. 2012), 490–501. https://doi.org/10.14778/2140436.2140445

[13] Daniel Fryer, Inga Strümke, and Hien Nguyen. 2021. Shapley values for feature selection: The good, the bad, and the axioms. *Ieee Access* 9 (2021), 144352–144360.

[14] Boris Glavic, Alexandra Meliou, and Sudeepa Roy. 2021. Trends in Explanations: Understanding and Debugging Data-driven Systems. *Found. Trends Databases* 11, 3 (Aug. 2021), 226–318. https://doi.org/10.1561/1900000074

[15] Jonathan S. Golan. 1999. Semirings and their applications. https://api.semanticscholar.org/CorpusID:122727231

[16] Ahmet Kara, Dan Olteanu, and Dan Suciu. 2024. From Shapley Value to Model Counting and Back. *Proc. ACM Manag. Data* 2, 2, Article 79 (May 2024), 23 pages. https://doi.org/10.1145/3651142

[17] Pratik Karmakar, Mikaël Monet, Pierre Senellart, and Stephane Bressan. 2024. Expected Shapley-Like Scores of Boolean functions: Complexity and Applications to Probabilistic Databases. *Proc. ACM Manag. Data* 2, 2, Article 92 (2024), 26 pages. https://doi.org/10.1145/3651593

[18] Majd Khalil and Benny Kimelfeld. 2022. The complexity of the Shapley value for regular path queries. *arXiv preprint arXiv:2212.07720* (2022).

[19] Ester Livshits, Leopoldo Bertossi, Benny Kimelfeld, and Moshe Sebag. 2021. The Shapley Value of Tuples in Query Answering. *Logical Methods in Computer Science* Volume 17, Issue 3, Article 22 (Sep 2021). https://doi.org/10.46298/lmcs-17(3:22)2021

[20] Scott M Lundberg and Su-In Lee. 2017. A unified approach to interpreting model predictions. In *NeurIPS*. 4765–4774. http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions.pdf

[21] Xuan Luo, Jian Pei, Cheng Xu, Wenjie Zhang, and Jianliang Xu. 2024. Fast Shapley Value Computation in Data Assemblage Tasks as Cooperative Simple Games. *Proc. ACM Manag. Data* 2, 1, Article 56 (March 2024), 28 pages. https://doi.org/10.1145/3639311

[22] Neha Makhija and Wolfgang Gatterbauer. 2024. Minimally Factorizing the Provenance of Self-join Free Conjunctive Queries. *Proc. ACM Manag. Data* 2, 2 (2024), 104. https://doi.org/10.1145/3651605

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

[24] Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. 2011. Bringing Provenance to Its Full Potential Using Causal Reasoning. In *TaPP*, Peter Buneman and Juliana Freire (Eds.). USENIX Association.

[25] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. 2014. Causality and explanations in databases. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1715–1716. https://doi.org/10.14778/2733004.2733070

[26] Dan Olteanu and Jakub Zavodny. 2012. Factorised representations of query results: size bounds and readability. In *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, Alin Deutsch (Ed.). ACM, 285–298. https://doi.org/10.1145/2274576.2274607

[27] L. S. Penrose. 1946. The Elementary Statistics of Majority Voting. *Journal of the Royal Statistical Society* 109, 1 (1946), 53–57. http://www.jstor.org/stable/2981392

[28] Alon Reshef, Benny Kimelfeld, and Ester Livshits. 2020. The impact of negation on the complexity of the Shapley value in conjunctive queries. In *PODS*. 285–297.

[29] Benedek Rozemberczki, Lauren Watson, Péter Bayer, Hao-Tsung Yang, Olivér Kiss, Sebastian Nilsson, and Rik Sarkar. 2022. The shapley value in machine learning. *arXiv preprint arXiv:2202.05594* (2022).

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

[31] 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

[32] Jianyuan Sun, Guoqiang Zhong, Kaizhu Huang, and Junyu Dong. 2018. Banzhaf random forests: Cooperative game theory based random forests with consistency. *Neural Networks* 106 (2018), 20–29.

[33] Jiachen T Wang and Ruoxi Jia. 2023. Data banzhaf: A robust data valuation framework for machine learning. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 6388–6421.