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

OMER ABRAMOVICH, Tel Aviv University, Israel
DANIEL DEUTCH, Tel Aviv University, Israel
NAVE FROST, eBay Research, Israel
AHMET KARA, OTH Regensburg, Germany
DAN OLTEANU, University of Zurich, Switzerland

In this paper, we introduce a novel approach to computing the contribution of input tuples to the result of the query, quantified by the Banzhaf and Shapley values. In contrast to prior algorithmic work that focuses on Select-Project-Join-Union queries, ours is the first practical approach for queries with aggregates. It relies on two novel optimizations that are essential for its practicality and significantly improve the runtime performance already for queries without aggregates. The first optimization exploits the observation that many input tuples have the same contribution to the query result, 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 with the same complexity as for one of them. Experiments with a million instances over 3 databases 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 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 [6, 26] and Shapley [30] 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 to the game outcome by summing up the marginal contributions of the player over the subsets (or permutations, for Shapley) of players. Prior work used Banzhaf and Shapley values for attribution in query answering: the contribution of a database tuple to the query result is quantified by defining a game where the players are database tuples and the objective function is given by each tuple in the query result [2, 12, 20]. Banzhaf and Shapley values are different, yet closely related measurements. Previous works have demonstrated the usefulness of both values for explanations in query answering [2, 8, 9, 17–20, 22, 27] and for feature importance, data valuation, feature selection, and explanations in machine learning [14, 28, 31, 32]. The main difference is that the sum of Shapley values over all input tuples is equal to the query result (this is 1 in case of a Boolean query and the numeric result for an aggregate query), whereas this is not the case for Banzhaf. This may be considered an advantage of Shapley values, in that they better capture relative contributions. Nevertheless, prior work [2] showed experimentally that they tend to agree on the relative order of fact contribution in query answering.

Yet computing Banzhaf values is typically more efficient than computing Shapley values [2]. Our work observes this for aggregate queries, too.

Example 1.1. Figure 1 depicts a fragment of an IMDB-like database about movies directed by Tarantino, award nominations, and cast. We consider two queries. Query Q_1 asks whether Tarantino was nominated for a "best director" award for movies with actors from the Cast relation. The answer is true: there are two such movies in the fragment (and three in the full database). We would like to understand which actor contributed most to this answer, using

Authors' addresses: 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.

`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	Award
Inglourious Basterds	Academy
Inglourious Basterds	BAFTA
Once Upon a Time in Hollywood	Academy
Once Upon a Time in Hollywood	BAFTA

Query Q_1

$$Q() = \texttt{Movies}(X, \text{Tarantino}, R),\ \texttt{Cast}(Y),\ \texttt{DirectingAwards}(Z),\ \texttt{MovieCast}(X,Y),\ \texttt{MovieAwards}(X,Z)$$

Query Q_2

$$\langle MAX, R, Q(X,Y,R) = \texttt{Movies}(X, \text{Tarantino}, R),\ \texttt{Cast}(Y),\ \texttt{MovieCast}(X,Y)\rangle$$

`MovieCast` (exo)

Movie	Actor
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

Fig. 1. Running example: a fragment of an IMDB-like database and two queries.

Banzhaf/Shapley values as the measure. In our fragment, both Brad Pitt and Zoë Bell contributed most as they appear in the two movies. If we consider the full database but restrict the MovieCast relation to only include the top-5 actors (first 5 listed in IMDB) for each movie, the most influential actor is Brad Pitt, who had a leading role in two of the three movies. Query Q_2 retrieves the maximum revenue of a movie directed by Tarantino. Again, we seek to identify the actor who contributed most to the query answer. In our fragment, Brad Pitt and Zoë Bell are the top contributors according to the Banzhaf/Shapley values. For the full database (again with only the top-5 actors in MovieCast), the top contributor is Samuel L. Jackson.

In this paper, we study the problem of *computing the contribution of database tuples to the results of aggregate queries, quantified by the Banzhaf and Shapley values.* The computational problem is highly intractable in general: already for Boolean non-hierarchical queries, exact computation of Shapley and Banzhaf values is #P-hard [20]. *All prior practical approaches were designed for Select-Project-Join-Union (SPJU) queries and do not work for queries with aggregates. Their runtime is far from negligible, especially for computing the Banzhaf/Shapley values for all input-output tuple pairs.*

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

The computation of Shapley and Banzhaf values resorts to the compilation of the query lineage, which is a polynomial over variables representing the input tuples [10, 15, 25] into tractable circuits, such as d-trees [13]. Yet aggregate queries require lineage formalisms beyond the Boolean semiring for queries without aggregates. *The first main contribution of this paper is an extension of this compilation approach to the lineage of queries with aggregates.*

Computing Banzhaf/Shapley values for aggregate queries remains expensive for most benchmarks used in the literature [2, 12]. *The second main contribution of this paper is the interplay of two novel techniques that we incorporate in our 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 SPJU queries using over a million instances [2]. For queries without aggregates, LExaBan achieves 2-3 orders of magnitude (OOM) improvement over ExaBan, while LExaShap achieves more than 2 OOM 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. Lifting exploits the observation that some variables (input tuples) in Boolean functions (query lineage) are interchangeable in the sense that they behave the same; in particular, the Banzhaf/Shapley values of interchangeable variables are the same. For instance, the pairs of variables $\{x_1, x_2\}$ and $\{y_1, z_1\}$ are interchangeable in $(x_1 \land y_1 \land z_1) \lor (x_2 \land y_1 \land z_1)$. We accommodate lifting into lineage compilation by extending the d-trees with a new gate type, which replaces conjunctions and disjunctions of interchangeable variables with a fresh variable. Lifting preserves equi-satisfiability. In our experiments, lifting leads to more than 2 OOM speedup and more than 1 OOM smaller d-trees.

Gradient computation. Computing Banzhaf/Shapley values separately for each input fact, as done in all prior work, is expensive. We show how to compute them for all the facts in the same complexity as for one fact. For this, we expose a fundamental connection between the vector of Banzhaf/Shapley values, one per input fact, and the gradient of a function that computes the probability of the query lineage with respect to each of its variables. In particular, we show that the Banzhaf value vector is precisely this gradient where the probability of each variable is 1/2. The vector of Shapley values can be recovered similarly. We devise a back-propagation algorithm to compute these values efficiently over any d-tree. In our experiments, this gradient technique speeds up the computation by a factor that grows with the size of the instance and up to 2 OOM speedup for large instances. This enables the computation 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 Sec. 5 we show that the interplay of both techniques can lead to 3 OOM speedup.

An extended version of this paper is available online [3].

2 PRELIMINARIES

Table 1 summarizes the notation used in the paper. We use $\mathbb{N}$ to denote the set of natural numbers including 0.

Boolean Formulas. A *(Boolean) formula* φ over a set $\boldsymbol{X}$ of Boolean variables is either a constant 0 or 1, a variable $x \in \boldsymbol{X}$, a negation $\neg\varphi_1$, a conjunction $\varphi_1 \land \varphi_2$, or a disjunction $\varphi_1 \lor \varphi_2$ of formulas φ_1 and φ_2. $\mathsf{Bool}(\boldsymbol{X})$ ($\mathsf{PosBool}(\boldsymbol{X})$) is the set of all (positive) Boolean formulas over $\boldsymbol{X}$. A *literal* is a constant, a variable, or its negation. The set of variables in φ is denoted $\mathsf{vars}(\varphi)$. A formula is *read-once* if each variable appears at most once. $\varphi[x := b]$ denotes the substitution of $x \in \boldsymbol{X}$ with $b \in \{0, 1\}$ in φ. A *valuation* $\theta : \mathsf{vars}(\varphi) \to \{0, 1\}$ maps each variable in φ to 0 or 1; we identify θ with $\{x \in \mathsf{vars}(\varphi) \mid \theta(x) = 1\}$, 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* $\mathsf{models}(\varphi)$ consists of all models of φ, with *model count* $\#\varphi = |\mathsf{models}(\varphi)|$ and k*-model count* $\#_k\varphi = |\mathsf{models}(\varphi) \cap \binom{\boldsymbol{X}}{k}|$, where $\binom{\boldsymbol{X}}{k}$ is the set of k-element subsets of $\boldsymbol{X}$. Two formulas φ_1 and φ_2 are *independent* if $\mathsf{vars}(\varphi_1) \cap \mathsf{vars}(\varphi_2) = \emptyset$ and *exclusive* if $\mathsf{models}(\varphi_1) \cap \mathsf{models}(\varphi_2) = \emptyset$.

Example 2.1. Consider the formula $\varphi = x \lor y$. 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 assume a countably infinite set Const of constants used as database values. A k*-tuple* $\boldsymbol{t}$ for $k \in \mathbb{N}$ is an element from Const^k. A database consists of *facts* $R(\boldsymbol{t})$, where R is a relation name with some arity $k \in \mathbb{N}$ and $\boldsymbol{t}$ is a k-tuple. As in [20], each database D is partitioned into *endogenous* facts D_n and *exogenous* facts D_x. We focus on the

Table 1. Overview of notation introduced in Sec. 2. φ is a Boolean formula; Φ is a Boolean⊗Number Pairs expression.

$[n]$	$\{1,\ldots,n\}$; $[n]=\emptyset$ if $n=0$
$\mathrm{Bool}(\boldsymbol{X})$ ($\mathrm{PosBool}(\boldsymbol{X})$)	Set of all (positive) Boolean formulas over $\boldsymbol{X}$
$\varphi[x := b]$	Substitution of variable x in φ by $b \in \{0,1\}$
$\varphi[\theta]$	Boolean value of φ under valuation θ
$\#\varphi$ ($\#_k\varphi$)	Number of models of φ (of size k)
$\sum_i^M \varphi_i \otimes m_i$	Boolean⊗Number Pairs (BNP) using monoid M
$\#^p\Phi$ ($\#_k^p\Phi$)	Number of valuations θ (of size k) s.t. $\Phi[\theta]=p$
$D_x, D_n \subseteq D$	Exogenous/endogenous facts in database D
$\mathrm{lin}(Q, D, \boldsymbol{t})$	Lineage of $\boldsymbol{t}$ with respect to Q over D
$\mathrm{lin}(\langle\alpha,\gamma,Q\rangle, D)$	Lineage of aggregate query $\langle\alpha,\gamma,Q\rangle$ over D

contribution of endogenous facts to the query result. Intuitively, we separate the facts in the database that are under analysis/control from those that are fixed/externally imposed.

Example 2.2. In Fig. 1, all facts labeled “endo" (“exo") are endogenous (respectively exogenous), meaning that we (do not) wish to analyze their contributions, e.g., Cast is labeled “endo" and we want to find out which actors contributed the most to the query result.

Conjunctive Queries. A *conjunctive query (CQ)* [1] has the form $Q(\boldsymbol{X}) = R_1(\boldsymbol{X}_1),\ldots,R_m(\boldsymbol{X}_m)$, where each $R_i(\boldsymbol{X}_i)$ is an *atom*, R_i is a relation symbol, and $\boldsymbol{X}_i$ is a tuple of variables and constants. The set $\boldsymbol{X} \subseteq \boldsymbol{X}_1 \cup \cdots \cup \boldsymbol{X}_m$ consists of the free (head) variables. A CQ is k*-ary* if $|\boldsymbol{X}| = k$ and *Boolean* if $k = 0$. A *union of conjunctive queries (UCQ)* consists of a set of k-ary CQs for some $k \in \mathbb{N}$.

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 replacing every variable X by $G(X)$ turns every atom in Q to a fact in D. We denote this set of facts by $\mathrm{facts}(Q, D, G)$. A grounding for a UCQ is a grounding for one of its CQs. Each grounding G yields an *output tuple*, as the restriction of G to the head variables of Q. Multiple groundings may yield the same output tuple. We denote the set of groundings yielding $\boldsymbol{t}$ by $G(Q, D, \boldsymbol{t})$ and the set of all output tuples, which defines the result of Q over D, by $Q(D)$. For an output tuple $\boldsymbol{t} \in Q(D)$, a head variable $X \in \boldsymbol{X}$, and a set $\boldsymbol{Y} \subseteq \boldsymbol{X}$ of head variables, we denote by $\boldsymbol{t}[X]$ the X-value in $\boldsymbol{t}$ and by $\boldsymbol{t}[Y]$ the projection of $\boldsymbol{t}$ onto $\boldsymbol{Y}$. A Boolean query is satisfied if it has a grounding.

Example 2.3. For Q_1 in Fig. 1 and $X =$ ’Inglourious Basterds’, $R = 322$, $Y =$ ’Brad Pitt’, and $Z =$ ’Academy’, we obtain a grounding and thus the query is satisfied.

Aggregate Queries. An aggregate query is a triple $\langle\alpha,\gamma,Q\rangle$, where Q is a k-ary UCQ for some $k \in \mathbb{N}$, $\gamma : \mathrm{Const}^k \rightarrow \mathbb{R}$ maps each k-tuple over Const to a numeric value and $\alpha : \mathbb{R}^* \rightarrow \mathbb{R}$ aggregates tuples of numbers to single numbers [20]. When evaluating an aggregate query Q over a database D, γ is applied to each tuple in $Q(D)$ to yield a number, and α is applied on the set of these numbers. Let $Q(D) = \{\boldsymbol{t}_1,\ldots,\boldsymbol{t}_n\}$. The result of $\langle\alpha,\gamma,Q\rangle$ is defined as $\langle\alpha,\gamma,Q\rangle(D) := \alpha\big((\gamma(\boldsymbol{t}_i))_{i\in[n]}\big)$. If $\gamma(\boldsymbol{t})$ returns the X-value of $\boldsymbol{t}$ for a head variable X, we just write X instead of $\gamma(\boldsymbol{t})$. We focus on the aggregations MAX, MIN, SUM, and COUNT defined as follows:

$$\langle \text{SUM}, \gamma, Q\rangle(D) \overset{def}{=} \sum_{\boldsymbol{t}\in Q(D)} \gamma(\boldsymbol{t})$$

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

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

Remark 2.4. The choice for the MAX/MIN queries to evaluate to 0 if $Q(D) = \emptyset$, as in [20], is not arbitrary. In the definition of Banzhaf and Shapley values below, we sum over marginal contributions w.r.t. the query result of individual tuples for different sub-databases, potentially including ones for which the query result is empty. The alternative of defining the result of aggregation over the empty set to be $\pm\infty$ would lead to unintuitive Banzhaf/Shapley values of $\pm\infty$.

Example 2.5. Query Q_2 of Fig. 1 is an aggregate query that asks for the maximal revenue of a movie directed by Tarantino. By setting $X =$ 'Once Upon a Time in Hollywood', $R = 377$, and $Y =$ 'Brad Pitt', we obtain the maximal revenue 377.

Monoids. We denote by $\overline{\mathbb{R}}$ the real numbers $\mathbb{R}$ including $\{\infty, -\infty\}$.

Definition 2.6. A (numeric) monoid $M = (\overline{\mathbb{R}}, +_M, 0_M)$ 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:

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

$$0_M +_M m_1 = m_1 +_M 0_M = m_1$$

A monoid is *commutative* if $m_1 + m_2 = m_2 + m_1$ for all $m_1, m_2 \in \overline{\mathbb{R}}$. It is idempotent if $m + m = m$ for all $m \in \overline{\mathbb{R}}$.

Monoids naturally model aggregate functions, as illustrated next:

Example 2.7. To model the aggregate functions MAX and MIN, we use the monoids $(\overline{\mathbb{R}}, \max, -\infty)$ and respectively $(\overline{\mathbb{R}}, \min, \infty)$. We use $\pm\infty$ as neutral values as usual for MIN and MAX despite of Remark 2.4, because 0 is not a neutral value for MIN (MAX) in presence of positive (resp. negative) numbers; adjustments in light of the remark will be made in our notion of valuations below. For SUM and COUNT, we use $(\overline{\mathbb{R}}, +, 0)$. The monoids for MIN and MAX are idempotent and the monoid for SUM and COUNT is not.

Boolean⊗Number Pairs. A Boolean⊗Number Pair (BNP) represents a collection of numeric values conditioned on Boolean formulas. Consider a variable set $\boldsymbol{X}$ and a bag $\mathcal{B} = \{\!\{(\varphi_i, m_i) \mid i \in [n]\}\!\}$, where each φ_i is a positive Boolean formula from PosBool$(\boldsymbol{X})$ and each m_i is an element from $\overline{\mathbb{R}}$ for a commutative numeric monoid $M = (\overline{\mathbb{R}}, +_M, 0_M)$. We call $\Phi = \sum_i^M \varphi_i \otimes m_i$ a BNP over $\boldsymbol{X}$ and M. The *Boolean part* of Φ is defined as $\varphi = \bigvee_i \varphi_i$ and its variables are $\text{vars}(\Phi) = \text{vars}(\varphi) = \bigcup_i \text{vars}(\varphi_i)$. Given a valuation θ of φ, we define VB as the bag of values m_i whose formulas φ_i evaluate to 1 under θ: $VB = \{\!\{m_i \mid (\varphi_i, m_i) \in \mathcal{B} \text{ and } \varphi_i[\theta] = 1\}\!\}$. We define the value of Φ under θ as $\Phi[\theta] = \sum_{m\in VB}^M m$ if $VB \neq \emptyset$ and $\Phi[\theta] = 0_{\mathbb{R}}$ otherwise, where $0_{\mathbb{R}}$ is the real number 0 and the summation $\sum^M$ uses the $+_M$ operator of the monoid. The explicit account for the case of valuations that yield an empty set is due to Remark 2.4. A valuation θ is a *model* of Φ if it is a model of φ. The *model count* $\#\Phi$ of Φ is defined as the model count $\#\varphi$ of φ. For a value $p \in \overline{\mathbb{R}}$, 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 say that two BNP expressions Φ_1 and Φ_2 are *independent* (resp. *exclusive*) if their Boolean parts are independent (resp. exclusive).

Example 2.8. Consider the BNP $\Phi = (x \vee y) \otimes 3 + (x \vee y) \otimes 3 + x \otimes 2 + y \otimes 4$ using the monoid $(\overline{\mathbb{R}}, +, 0)$. Its Boolean part is equivalent to $\varphi = x \vee y$ and its set of variables is $\mathrm{vars}(\Phi) = \{x, y\}$. The valuation $\theta = \{x \mapsto 1, y \mapsto 0\}$ only satisfies the first three formulas in Φ. Hence, $\Phi[\theta] = 3 + 3 + 2 = 8$. We have $\#\Phi = \#\varphi = 3$ and $\#^8\Phi = 1$. It holds that $\#^8_2\Phi = 0$, since the only valuation of size 2 is $\theta' = \{x \mapsto 1, y \mapsto 1\}$ and $\Phi[\theta'] = 12$.

Semimodules. If the monoid $M = (\overline{\mathbb{R}}, +_M, 0_M)$ used by a BNP is idempotent, then the BNP can be interpreted as a semimodule expression[1], where the operator $\otimes : \mathsf{PosBool}(\boldsymbol{X}) \times \overline{\mathbb{R}} \to \overline{\mathbb{R}}$ obeys the following axioms:

(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) $1 \otimes m_1 = m_1$

Our notion of valuation for BNPs also extends to semimodules.

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:

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

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 from $\mathrm{facts}(Q, D, G)$. If Q is a Boolean query, we abbreviate $\mathsf{lin}(Q, D, ())$ by $\mathsf{lin}(Q, D)$.

Table 2. Lineages and lifted lineages for queries Q_1 and Q_2 from Fig. 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	$(a_1 \wedge m_3) \otimes 377 +_{\max} (a_2 \wedge m_3) \otimes 377 +_{\max} (a_3 \wedge m_3) \otimes 377$ $+_{\max} (a_1 \wedge m_2) \otimes 322 +_{\max} (a_3 \wedge m_2) \otimes 322 +_{\max} (a_4 \wedge m_1) \otimes 176$	$(a_{1,3} \wedge w_3) \vee (a_2 \wedge w_3) \vee (a_{1,3} \wedge w_2) \vee (w_1)$ $\ell = \{a_{1,3} \mapsto (a_1 \vee a_3),$ $w_1 \mapsto ((a_4 \wedge m_1) \otimes 176), w_2 \mapsto (m_2 \otimes 322), w_3 \mapsto (m_3 \otimes 377), a_2 \mapsto a_2\}$

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

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

[1] Previous work [4], which proposed using semimodules to capture lineage, also noted that PosBool[X] is only "compatible" in the context of this construction with idempotent monoids. This is why for the non-idempotent case we use BNP instead.

where the summation uses the $+_M$ operator of the monoid M associated with α. For MIN and MAX, this lineage is a semimodule expression since their corresponding monoids are idempotent.

Example 2.9. Table 2 shows the lineage of the queries from Fig. 1: This is a positive DNF formula for Q_1 and a semimodule expression with the Max monoid for Q_2. The Boolean variables correspond to database facts, e.g., a_1 corresponds to Cast('Brad Pitt').

Banzhaf and Shapley Values. We use such values to measure the contribution of database facts to query results [2, 20].

Definition 2.10 (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:

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

Definition 2.11 (Shapley Value)**.** Let $C^{\boldsymbol{X}}_{\boldsymbol{Y}} = \frac{|Y|!(|X|-|Y|-1)!}{|X|!}$ for variable sets $\boldsymbol{Y}, \boldsymbol{X}$ (when $\boldsymbol{X}$ is clear from context, we just use $C_{\boldsymbol{Y}}$). 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:

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

Banzhaf and Shapley values sum up marginal contributions of facts to query results over sub-databases. They differ in that the former weigh this contribution based on sub-database sizes. These weights guarantee that the sum of Shapley values over all facts equals the query result. This is not the case for Banzhaf.

Given a Boolean formula or a BNP Ψ over a variable set $\boldsymbol{X}$, the Banzhaf and Shapley values of $x \in \boldsymbol{X}$ are:

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

$$\mathsf{Shapley}(\Psi, x) = \sum_{Y \subseteq \boldsymbol{X} \setminus \{x\}} C^{\boldsymbol{X}}_{\boldsymbol{Y}} \cdot (\Psi[Y \cup \{x\}] - \Psi[Y]) \tag{2}$$

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]:

$$\mathsf{Banzhaf}(\varphi, x) = \#\varphi[x := 1] - \#\varphi[x := 0] \tag{3}$$

Prior work connected the Banzhaf and Shapley values for database facts and those of Boolean variables [20]. Given a Boolean or aggregate query Q, a database $D = D_x \cup D_n$ and a fact $f \in D_n$:

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

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

For non-Boolean queries, the Banzhaf and Shapley values are defined with respect to each output tuple.

Example 2.12. Consider the lineage φ of Q_1 in Table 2. We sketch the computation of the Banzhaf and Shapley values of the variable a_1 representing the fact Cast('Brad Pitt'). Consider the valuation $\theta = \{d_1, m_3\}$. The marginal contribution of a_1 on θ is $\varphi[\theta \cup \{a_1\}] - \varphi[\theta] = 1$ because θ is not a model for φ but $\theta \cup \{a_1\}$ is. The Banzhaf value is achieved by summing up the marginal contributions across all valuations. For Shapley value, the marginal contributions are multiplied by the corresponding coefficients; e.g., the contribution of a_1 to θ is multiplied by $\frac{2!(7-2-1)!}{7!}$, since $|\theta| = 2$ and $|vars(\varphi)| = 7$. Consider now the lineage Φ of Q_2 in Table 2 and the valuation $\theta' = \{m_3\}$. We have $\Phi[\theta'] = 0$, since θ' is not a model of any Boolean formula in the BNP Φ. This is an example of the effect of our special treatment of aggregation over an empty set of values. Further, $\Phi[\theta' \cup \{a_1\}] = 377$, so the marginal contribution of a_1 to the sub-database consisting of only m_3 is 377.

Decomposition Trees. We next introduce decomposition trees for Boolean and semimodule expressions.

Definition 2.13 ([13])**.** A *decomposition tree* (d-tree), is defined recursively as follows:

- If v is a 0/1 constant or a variable then v is a d-tree.
- Let $T_{\varphi_1}, \ldots, T_{\varphi_n}$ be d-trees for pairwise independent formulas $\varphi_1, \ldots, \varphi_n$, 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,

$$\begin{array}{c} \oplus \\ \diagup \quad \diagdown \\ T_{\varphi_1} \;\cdots\; T_{\varphi_n} \end{array} \quad , \quad \begin{array}{c} \odot \\ \diagup \quad \diagdown \\ T_{\varphi_1} \;\cdots\; T_{\varphi_n} \end{array} \quad , \text{ and } \quad \begin{array}{c} \sqcup_v \\ \diagup \quad \diagdown \\ T_{\varphi} \quad T_{\psi} \end{array}$$

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

We next extend d-trees to represent semimodule expressions [13]:

- Every monoid value $m \in \overline{\mathbb{R}}$ is a d-tree.
- Let $T_{\varphi_1}, \ldots, T_{\varphi_n}$ be d-trees for pairwise independent Boolean formulas $\varphi_1, \ldots, \varphi_n$, and T_Φ be a d-tree for a semimodule expression Φ whose Boolean part is independent from each of $\varphi_1, \ldots, \varphi_n$. Let $T_{\Phi_1}, \ldots, T_{\Phi_n}$ be d-trees for pairwise independent semimodule expressions $\Phi_1, \ldots, \Phi_n$. Then,

$$\begin{array}{c} \otimes \\ \diagup \quad \diagdown \diagdown \\ T_{\varphi_1} \;\cdots\; T_{\varphi_n} \;\; T_\Phi \end{array} \qquad \text{and} \qquad \begin{array}{c} \oplus \\ \diagup \quad \diagdown \\ T_{\Phi_1} \;\cdots\; T_{\Phi_n} \end{array}$$

  are d-trees for $(\bigwedge_{i\in[n]} \varphi_i) \otimes \Phi$ and $\sum_M \Phi_i$ respectively.

The size of a d-tree is its number of nodes. D-trees can be constructed by iteratively decomposing a Boolean formula or a semimodule expression into simpler parts.

Example 2.14. Consider the semimodule expression $(a_1 \wedge b_1) \otimes 3 +_M (a_1 \wedge b_2) \otimes 7$. Using Axiom (3) of semimodules, 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 represented by the d-tree. We thus use them for formulas and d-trees interchangeably.

3 ATTRIBUTION FOR UNION OF CONJUNCTIVE QUERIES

We present our algorithms LExaBan and LExaShap for computing Banzhaf and Shapley values for UCQs. Starting with the query lineage in DNF (computed by tools such as ProvSQL [29] or GProM [5]), we first compile it into a d-tree using a novel lifting optimization (Sec. 3.1), then introduce gradient-based algorithms (Sec. 3.2) to compute Banzhaf/Shapley based on the lifted d-tree.

Algorithm 1 Lift

Input: Lifted DNF formula (φ, ℓ)
Output: Saturated lifted DNF formula

```
while $(\varphi, \ell)$ is not saturated do
    if $\varphi$ has a maximal set $V$ of cofactor-equivalent variables with $|V| > 1$ then
        $(\varphi, \ell) \leftarrow \mathsf{lift\text{-}or}(\varphi, \ell, V)$
    if $\varphi$ has a maximal set $V$ of interchangeable variables with $|V| > 1$ then
        $(\varphi, \ell) \leftarrow \mathsf{lift\text{-}and}(\varphi, \ell, V)$
return $(\varphi, \ell)$
```

3.1 Lifted Compilation of Lineage Into D-tree

Consider two disjoint sets X and Y of Boolean variables. A *lifted formula* over $X \cup Y$ is a pair (φ, ℓ), where $\varphi \in \mathsf{Bool}(Y \cup X)$ is a formula over $Y \cup X$ and $\ell : \mathsf{vars}(\varphi) \to \mathsf{Bool}(X)$ maps each variable in φ to a formula over 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 V is called a *maximal set of cofactor-equivalent (interchangeable)* variables if every pair of variables in V is cofactor-equivalent (interchangeable) and this does not hold for any superset of 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 translated into a saturated lifted DNF formula. We first introduce some further notation. Consider a lifted DNF formula (φ, ℓ) over $X \cup Y$, where $\varphi = \bigwedge_{i \in [n]} C_i$. Let $V = \{y_1, \ldots, y_k\}$ be a set of cofactor-equivalent variables in φ, where each variable has cofactor $\{N_1, \ldots, N_p\}$. Let $C_{i_1}, \ldots, C_{i_m}$ be all clauses in φ containing variables from V. We define $\mathsf{lift\text{-}or}(\varphi, \ell, V)$ 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 clauses $y \wedge N_1, \ldots, y \wedge N_p$ for a fresh variable $y \in Y \setminus \mathsf{vars}(\varphi)$; (2) the function ℓ' is defined by $\ell'(y) = \bigvee_{i \in [k]} \ell(y_i)\}$ and $\ell'(y') = \ell(y')$ for all $y' \in \mathsf{vars}(\varphi')$ with $y' \neq y$. Now, assume that $V = \{y_1, \ldots, y_k\}$ consists of interchangeable variables in φ. We denote by $\mathsf{lift\text{-}and}(\varphi, \ell, V)$ the lifted DNF formula (φ', ℓ'), where: (1) $\varphi' = C'_1 \wedge \cdots \wedge C'_n$ such that, given a fresh variable $y \in Y \setminus \mathsf{vars}(\varphi)$, 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 $y' \in \mathsf{vars}(\varphi')$ with $y' \neq y$.

The function Lift in Algorithm 1 transforms a lifted DNF formula φ into a saturated one. It repeatedly invokes lift-or or lift-and while φ contains a variable set $\boldsymbol{V}$ with $|\boldsymbol{V}| > 1$, such that all variables in $\boldsymbol{V}$ are either cofactor-equivalent or interchangeable, respectively. Intuitively, both cofactor-equivalent and interchangeable variables exhibit a symmetry in the structure of the formula. Consequently, we can simplify the formula without loss of information.

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

Proof. Let φ be a DNF formula, ℓ the identity function on $\mathsf{vars}(\varphi)$, and (φ', ℓ') the output of $\mathsf{Lift}(\varphi, \ell)$ in Algorithm 1. Consider the sequence $(\varphi, \ell) = (\varphi_1, \ell_1), \ldots, (\varphi_n, \ell_n) = (\varphi', \ell')$, where $(\varphi_2, \ell_2), \ldots, (\varphi_{n-1}, \ell_{n-1})$ are the intermediate lifted formulas constructed by the algorithm. It follows from the stopping condition at Line 1 of the algorithm that (φ', ℓ') is saturated.

In the following, we show by induction on i that Properties (1)–(3) from Proposition **??** hold for each (φ_i, ℓ_i) with $i \in [n]$, i.e., (1) $\mathsf{inline}(\varphi_i, \ell_i)$ is equivalent to φ, (2) $\ell_i(x)$ and $\ell_i(y)$ are independent formulas for each distinct variables $x, y \in \mathsf{vars}(\varphi_i)$, and (3) $\ell_i(x)$ is a read-once formula for each $x \in \mathsf{vars}(\varphi_i)$.

Base Case: Since ℓ_1 is the identity function on $\mathsf{vars}(\varphi)$ and $\varphi_1 = \varphi$, Properties (1)–(3) obviously hold for (φ_1, ℓ_1).

Induction Hypothesis: We assume that Properties (1)–(3) hold for (φ_i, ℓ_i) for some $i \in [n-1]$.

Induction Step: We show that Properties (1)–(3) hold for $(\varphi_{i+1}, \ell_{i+1})$. Without loss of generality, we assume that $(\varphi_{i+1}, \ell_{i+1})$ results from (φ_i, ℓ_i) by the application of the lift-or operator. The case of the lift-and operator is handled analogously. Let $\boldsymbol{V} = \{y_1, \ldots, y_k\}$ be a maximal set of cofactor-equivalent variables in φ_i such that each variable y_i has cofactor $\{N_1, \ldots, N_p\}$. Let $C_{i_1}, \ldots, C_{i_m}$ be all clauses in φ_i containing a variable from $\boldsymbol{V}$. By assumption, φ_{i+1} results from φ_i by omitting the clauses $C_{i_1}, \ldots, C_{i_m}$ and adding the clauses $y \wedge N_1, \ldots, y \wedge N_p$ for a fresh variable y; the function ℓ_{i+1} is defined by $\ell_{i+1}(y) = \bigvee_{j \in [k]} \ell_i(y_j)$ and $\ell_{i+1}(y') = \ell_i(y')$ for all $y' \in \mathsf{vars}(\varphi_{i+1})$ with $y' \neq y$.

We show that Property (1) holds for $(\varphi_{i+1}, \ell_{i+1})$, i.e., $\mathsf{inline}(\varphi_{i+1}, \ell_{i+1})$ is equivalent to φ. Let φ_i' be the formula obtained from φ_i by replacing y with $\bigvee_{j \in [k]} y_j$. It follows from the distributivity of conjunction over disjunction that φ_i' is equivalent to φ_i. Hence, $\mathsf{inline}(\varphi_i', \ell_i)$ is equivalent to $\mathsf{inline}(\varphi_i, \ell_i)$. By induction hypothesis, $\mathsf{inline}(\varphi_i, \ell_i)$, hence $\mathsf{inline}(\varphi_i', \ell_i)$, is equivalent to φ. By construction, $\mathsf{inline}(\varphi_i', \ell_i) = \mathsf{inline}(\varphi_{i+1}, \ell_{i+1})$. It follows that $\mathsf{inline}(\varphi_{i+1}, \ell_{i+1})$ is equivalent to φ.

We prove that Property (2) holds for $(\varphi_{i+1}, \ell_{i+1})$, i.e., $\ell_{i+1}(x)$ and $\ell_{i+1}(y)$ are independent formulas for each distinct variables $x, y \in \mathsf{vars}(\varphi_{i+1})$. For the sake of contradiction, assume that Property (2) does not hold for $(\varphi_{i+1}, \ell_{i+1})$. Since Property (2) holds for (φ_i, ℓ_i) (induction hypothesis), there must be a variable $x \in \mathsf{vars}(\varphi_{i+1})$ such that z appears in $\ell_{i+1}(x)$ and $\ell_{i+1}(y)$, where y is the fresh variable in φ_{i+1}. From the construction of ℓ_{i+1}, it follows that there must be a variables $z' \in \mathsf{vars}(\varphi_i)$ such that z appears in $\ell_i(x)$ and $\ell_i(z')$. This, however, contradicts our assumption that Property (2) holds for (φ_i, ℓ_i).

Now, we prove that Property (3) holds for $(\varphi_{i+1}, \ell_{i+1})$, i.e., $\ell_{i+1}(x)$ is a read-once formula for each $x \in \mathsf{vars}(\varphi_{i+1})$. Consider a variable $x \in \mathsf{vars}(\varphi_{i+1})$. If x is not the fresh variable y, then $x \in \mathsf{vars}(\varphi_i)$ and $\ell_i(x) = \ell_{i+1}(x)$. By induction hypothesis, $\ell_i(x)$ is a read-once formula. If x is equal to the fresh variable y, then $\ell_{i+1}(x) = \bigvee_{j \in [k]} \ell_i(y_j)$. By induction hypothesis, the formulas $\ell_i(y_1), \ldots, \ell_i(y_k)$ are independent and read-once. Hence, $\ell_{i+1}(x)$ is a read-once formula. □

Example 3.4. We explain how the procedure Lift transforms the lineage φ_0 of the query Q_1 depicted in Table 2 (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 2 (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.

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 $T_\psi$
```

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 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). For conjunctions of independent subformulas, the procedure similarly combines the compiled trees using $\odot$ (Line 7). Otherwise, it performs Shannon expansion on a variable y: it compiles Lift$(\psi[y := 0], \ell|\psi[y := 0])$ and Lift$(\psi[y := 1], \ell|\psi[y := 1])$ into d-trees T_0 and T_1, then combines them with a $\sqcup_{\ell(y)}$ gate, where T_0 and T_1 become the 0- and 1-branches, respectively (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 symmetries.

Example 3.5. We apply Algorithm 2 to the lineage of Q_1 in Table 2 (top left). The saturated lifted lineage (φ, ℓ) constructed in Line 1 is given in Table 2 (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

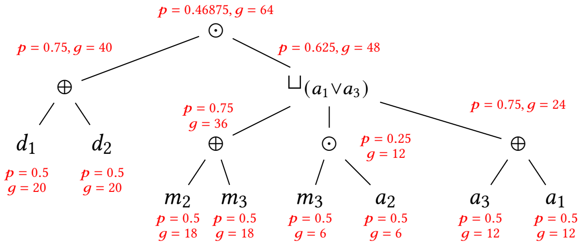


Fig. 2. Lifted d-tree for the lineage in Figure 1. Each node is annotated with probability p and partial derivative g computed during Banzhaf value computation using gradients.

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:

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

We will interpret a d-tree as a function over random variables and reduces the computation of Banzhaf and Shapley values to the computation of the gradient of this function.[2] We first show the computation for Banzhaf and then 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 3. 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 $x \in \text{vars}(T)$:

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

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 $|\text{vars}(T)|$.

[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 [7], and the Darwiche result on efficient inference in Bayesian networks [11]. To the best of our knowledge, no prior work uses gradient-based computation for Banzhaf/Shapley values.

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

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

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

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

For the cases where x does not appear in, we obtain:

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

And over all cases, we obtain:

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

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

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

Intuitively, evaluating the derivative at $\vec{\frac{1}{2}}$ gives each subset equal likelihood. Thus, the partial derivative increases the weight of subsets containing the variable and decreases it for those that do not.

Efficient Computation. Using Prop. 3.7, we can express the probability of a d-tree as a function of the probabilities of its leaf variables and derive its partial derivatives with respect to each variable by recursively applying the chain rule. Prop. 3.7 shows that these derivatives coincide with the Banzhaf values up to normalization. Table 3 gives rules for computing the probability of a d-tree T from its sub-trees, following standard probability theory, as well as the partial derivatives $\partial Pr[T]/\partial Pr[T_i]$ for each sub-tree T_i, which together enable gradient computation with respect to the leaf variables.

The procedure GradientBanzhaf in Algorithm 3 implements this 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 $\frac{1}{2}$, as our goal is to compute derivatives as in Table 3, and evaluate them at $\vec{\frac{1}{2}}$, as in Prop. 3.7; the $v.g$ value of the root is initialized to $2^{|\mathsf{vars}(T_\varphi)|-1}$ to account for normalization. Probabilities are computed bottom-up using the expressions in the second column of Table 3 (Lines 1 - 3). Partial derivatives are computed in a top-down fashion, using the expressions in the third column of Table 3 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.

Table 3. 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
$\oplus$	$1 - \prod_{i\in[n]} (1 - Pr[T_i])$	$\frac{\partial(Pr[T])}{\partial(Pr[T_i])} = \prod_{j\in[n]\setminus\{i\}} (1 - Pr[T_j]) = \frac{1-Pr[T]}{1-Pr[T_i]}$
$\odot$	$\prod_{i\in[n]} Pr[T_i]$	$\frac{\partial(Pr[T])}{\partial(Pr[T_i])} = \prod_{j\in[n]\setminus\{i\}} Pr[T_j] = \frac{Pr[T]}{Pr[T_i]}$
$\sqcup_f$	$Pr[f] \cdot Pr[T_1]$ $+(1 - Pr[f]) \cdot Pr[T_0]$	$\frac{\partial(Pr[T])}{\partial(Pr[T_1])} = Pr[f]$ $\frac{\partial(Pr[T])}{\partial(Pr[T_0])} = 1 - Pr[f]$ $\frac{\partial(Pr[T])}{\partial(Pr[f])} = Pr[T_1] - Pr[T_0]$

Example 3.8. Reconsider the d-tree from Figure 2, where its nodes are now annotated with p and g values. Consider the $\oplus$ node v that is the root's left child. Its p value is computed using the equation for $\oplus$ from Table 3 and the probability of its sub-trees d_1 and d_2: $v.p = 1-(1-d_1.p)\cdot(1-d_2.p) = 1-0.5\cdot 0.5 = 0.75$. We then have $v.g = v.parent.g \cdot \frac{\partial v.parent.p}{\partial v.p} = v.parent.g \cdot \frac{v.parent.p}{v.p} = 64 \cdot \frac{0.46875}{0.75} = 40$.

Shapley values. We next link Shapley values to gradients:

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

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

To compute Shapley values, the equations in Table 3 are adapted to compute $Pr_k[T]$ instead of $Pr[T]$, and $\frac{\partial(\sum_k C_k \cdot \mathrm{Pr}_k[T])}{\partial Pr_j[T_i]}(\vec{\frac{1}{2}})$ 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 3 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 v, $Pr[v]$ is computable in time linear in its number of children, so $O(|T|)$ for all nodes. For Banzhaf, the gradient w.r.t. each leaf node can be computed in $O(1)$ following Table 3, so in $O(|T|)$ total time. For Shapley, we need to compute $|\mathsf{vars}(T_v)|$ expressions and partial derivatives at each node v and each may be computed in $O(|\mathsf{vars}(T)|)$ time, resulting in $O(|\mathsf{vars}(T)|^2)$ per node and $O(|T| \cdot |\mathsf{vars}(T)|^2)$ in total.

4 ATTRIBUTION FOR AGGREGATE QUERIES

4.1 Basic Algorithms

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

Algorithm 3 GradientBanzhaf

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

SUM *and* COUNT. Our solution for linear aggregates uses linearity of Banzhaf and Shapley to reduce the problem to the case of SPJU queries. Let $\Phi = \sum_i^M \varphi_i \otimes m_i$ be a BNP where the sum uses the $+_M$-operator of the monoid M, and let $\varphi = \bigvee \varphi_i$ be the Boolean part of Φ. For SUM, $+_M$ is the standard +, 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, $\mathsf{Shapley}(\Phi, x) = \sum_i \mathsf{Shapley}(\varphi_i, x) \cdot m_i$ (as observed already in [20]). For COUNT, 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 Sec. 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. As explained in Sec. 2, the lineage for queries with these aggregates is a semimodule expression. 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 [13] and we next show how the latter is performed.

Let $\mathrm{PV}_\Phi = \{\Phi[\theta] | \theta \subseteq \mathsf{vars}(\Phi)\}$ 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 $\langle \mathrm{MIN}, \gamma, Q\rangle$ or $\langle \mathrm{MAX}, \gamma, Q\rangle$, the size of PV_Φ is bounded by the size of the result of Q, which is polynomial in the input database size. We 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:

$$\mathsf{Banzhaf}(\Phi, x) = \sum_{p \in \mathrm{PV}_\Phi} (\#^p\Phi[x := 1] - \#^p\Phi[x := 0]) \cdot p$$

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

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

Proof. Proposition 4.1 is proved by partitioning Equations 1,2 based on the monoid values.

For the Banzhaf value:

$$
\begin{aligned}
\mathsf{Banzhaf}(\Phi, x) &= \sum_{Y \subseteq \mathsf{vars}(\Phi) \setminus \{x\}} \Phi(Y \cup \{x\}) - \Phi(Y) \\
&= \sum_{Y \subseteq \mathsf{vars}(\Phi) \setminus \{x\}} \Phi(Y \cup \{x\}) - \sum_{Y \subseteq \mathsf{vars}(\Phi) \setminus \{x\}} \Phi(Y) \\
&= \sum_{p \in PV_\Phi} \sum_{\substack{Y \subseteq \mathsf{vars}(\Phi) \setminus \{x\}, \\ \Phi(Y \cup \{x\}) = p}} \Phi(Y \cup \{x\}) - \sum_{p \in PV_\Phi} \sum_{\substack{Y \subseteq \mathsf{vars}(\Phi) \setminus \{x\}, \\ \Phi(Y) = p}} \Phi(Y) \\
&= \sum_{p \in PV_\Phi} \sum_{\substack{Y \subseteq \mathsf{vars}(\Phi) \setminus \{x\}, \\ \Phi(Y \cup \{x\}) = p}} p - \sum_{p \in PV_\Phi} \sum_{\substack{Y \subseteq \mathsf{vars}(\Phi) \setminus \{x\}, \\ \Phi(Y) = p}} p \\
&= \sum_{p \in \mathsf{PV}_\Phi} (\#^p \Phi[x := 1] - \sum_{p \in \mathsf{PV}_\Phi} \#^p \Phi[x := 0]) \cdot p \\
&= \sum_{p \in \mathsf{PV}_\Phi} (\#^p \Phi[x := 1] - \#^p \Phi[x := 0]) \cdot p
\end{aligned}
$$

The proof for Shapley values is similar.

□

We can further show, via a dynamic programming algorithm:

Proposition 4.2. Let T be a d-tree representing a semimodule expression Φ over a variable set $\boldsymbol{X}$ and the monoid $M = (\overline{\mathbb{R}}, \max, -\infty)$ or the monoid $M = (\overline{\mathbb{R}}, \min, \infty)$. For $p \in \mathsf{PV}_\Phi$ 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:

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

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:

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

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

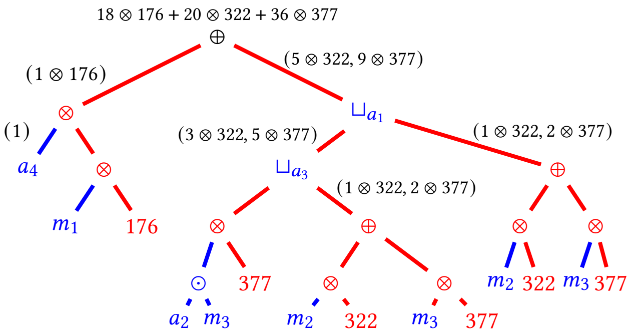


Fig. 3. A d-tree for the semimodule expression lineage of Q_2 from Table 2. The blue part represents the boolean component of the expression while the red one represents a semimodule component.

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 2, 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.

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.

4.2 Lifting for Aggregates

We extend our lifting technique (Sec. 3.1) from UCQs to aggregate queries. First, for linear aggregates (SUM and COUNT), our solution in Sec. 4.1 constructs a separate d-tree for each formula φ_i in the Boolean part $\bigvee_i \varphi_i$ of the given BNP expression. Thus, we can apply Lift and LiftedCompile in Algorithms 1 and 2 to each formula φ_i. For MIN and MAX, new machinery is needed. 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 $\ell : \mathsf{vars}(\varphi) \rightarrow \mathsf{PosBool}(\boldsymbol{X}) \cup (\mathsf{PosBool}(\boldsymbol{X}) \otimes M)$ is a mapping from variables in φ to either positive Boolean formulas over $\boldsymbol{X}$ or semimodule expressions over $\boldsymbol{X}$ and semimodule M (denoted by $\mathsf{PosBool}(\boldsymbol{X}) \cup (\mathsf{PosBool}(\boldsymbol{X}) \otimes M)$). If φ is in DNF, we refer to (φ, ℓ) as a *lifted DNF semimodule expression.* The *inlining* $\mathsf{inline}(\varphi, \ell)$ is obtained in two steps: First, we replace each variable $x \in \mathsf{vars}(\varphi)$ by $\ell(x)$; then, we replace each $\vee$ between semimodule expressions by $+_M$ and each $\wedge$ between a formula and a semimodule expression by $\otimes$. We call a lifted semimodule expression *valid* if exactly one variable in each DNF clause 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 $\Phi = \sum_i^M \varphi_i \otimes m_i$, where each $\varphi_i = \bigvee_j C_{ij}$ is in DNF. We transform Φ to the equivalent form $\Phi' = \sum_{i,j}^M C_{i,j} \otimes m_i$. 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 $\varphi = \bigvee_{i,j}(C_{i,j} \wedge w_i)$ and $\ell = \{w_i \mapsto m_i\} \cup \{x \mapsto x \mid x \in \mathsf{vars}(\varphi) \text{ and } x \neq w_i\}$. 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 for lifted Boolean DNF formulas. The application of lift-and or lift-or to a set of cofactor-equivalent or interchangeable variables maintains the validity of the expression. Algorithm 1 may then be applied to lifted DNF semimodule expressions.

Example 4.7. Consider the lineage of Query Q_2 in Table 2. We first translate it into the lifted DNF semimodule expression $\varphi = (a_1 \wedge m_3 \wedge t_1) \vee (a_2 \wedge m_3 \wedge t_1) \vee (a_3 \wedge m_3 \wedge t_1) \vee (a_1 \wedge m_2 \wedge t_2) \vee (a_3 \wedge m_2 \wedge t_2) \vee (a_4 \wedge m_1 \wedge t_3)$; $\ell = \{t_1 \mapsto 377, t_2 \mapsto 322, t_3 \mapsto 176\} \cup \{x \mapsto x | x \in \mathsf{vars}(\varphi) \setminus \{t_1, t_2, t_3\}\}$. 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 and 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. Table 2 gives the resulting expression. The algorithm continues he compilation as described in Sec. 3.1 to obtain a d-tree for the lineage.

4.3 Gradients for Aggregates

We now adapt the gradient-based computation from Sec. 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 looks at *expected values.*

Expected Values and Gradients. Given a d-tree T for a semimodule expression Φ, $E[T]$ denotes the expected value of T given probabilities p_x for the Boolean variables: $E[T] = \sum_{S \subseteq \mathsf{vars}(T)} Pr[S] \cdot T[S]$. Prop. 3.7 extends to expected values:

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

We define the probability and k-probability of a d-tree T as follows: $Pr[T = m_i] = \sum_{S \subseteq \mathsf{vars}(T), T[S] = m_i} Pr[S]$ and $Pr_k[T = m_i] = \left(\frac{1}{2}\right)^{|\mathsf{vars}(T)|-k} \sum_{\substack{S \subseteq \mathsf{vars}(T) \\ |S|=k, T[S]=m_i}} \sum_{S' \subseteq S} T[S'] \cdot \prod_{y \in S'} p_y \prod_{z \in S \setminus S'} \left(\frac{1}{2} - p_z\right)$.

Using the linearity of expectation, we can write:

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

From Gradients to Banzhaf and Shapley. Similarly to Sec. 3.2, we obtain the Banzhaf and Shapley values of all facts via partial derivatives. Assume an order over the variables and then use variable names as indices (identifying each variable with its position in the order). Let J be the Jacobian matrix, namely $J^i_x = \frac{\partial(v_i \cdot Pr[T=v_i])}{\partial p_x}$. Further let J^k be the matrix defined by $J^{k,i}_x = \frac{\partial(v_i \cdot Pr_k[T=v_i])}{\partial p_x}$ for each variable x. Note that the entries J^i_x and $J^{k,i}_x$ in the Jacobian matrices are functions of all variables in $\mathrm{vars}(T)$. We can show:

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

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

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

Banzhaf and Shapley Computation. Similarly to Sec. 3.2, we can compute the Banzhaf or Shapley value by using the d-trees to compute the expectation, and then compute the Jacobians via back propagation. We adapt Algorithm 3 to incorporate these changes for a d-tree representing the lineage of an aggregate query. 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. 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.

GROUP-BY and Additional Aggregates. We have focused so far on aggregate queries without grouping and with MIN, MAX, SUM and COUNT. Our solution directly extends to queries with GROUP BY: as in prior work [4, 20], lineage is defined for each group as a BNP/semimodule expression. Shapley and Banzhaf values are then defined and computed with respect to each group, exactly as for aggregates without GROUP BY. Linear aggregates can be supported like SUM and COUNT, by computing the Banzhaf/Shapley value for each term in the BNP and then combining these values using the aggregate function. In contrast, non-linear aggregates such as AVG require non-trivial development, in particular to define the aggregate-specific gradients as in Table 3. This is left as future work.

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 algorithms LExaBan and LExaShap, with and without the two optimization techniques, for SPJU and aggregate queries. We compare them with the state-of-the-art solutions for Banzhaf

Table 4. 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

and Shapley computation [2] that only work for SPJU queries and do not use the variable lifting and gradient optimizations: ExaBan and ExaShap for exact computation of Banzhaf and Shapley values and AdaBan for approximation of Banzhaf values with error guarantees (such approximation is not available for Shapley). For aggregate queries with SUM/COUNT, we also compared to a variant where we use ExaBan for each boolean formula and then use linearity as explained in Section 4.1 to compute Banzhaf values.

Datasets. The workload (see Table 4) of 301 queries over three datasets: Academic, IMDB and TPC-H (SF1) is based on prior work on Banzhaf/Shapley values for query answering [2, 12]. Lineage for all output tuples of all queries was constructed using ProvSQL [29]. 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 computed all lineages in an hour or less. All 7 TPC-H aggregate queries have GROUP BY clauses, resulting in 140 lineage instances. To further extend the benchmark, we created multiple variants of each of these queries: for TPC-H we changed the aggregation to MAX, COUNT, 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, applied over randomly chosen values. These datasets include very large and highly challenging lineage expressions. In total, the workload consists of nearly 1M Boolean lineage expressions and 429 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. Percentiles are computed with respect to the set of instances each algorithm succeeded on. Since the algorithms success rate is different, these percentiles generally correspond to different instances.

5.2 Banzhaf Computation for SPJU Queries

Success rate. Table 5 reports the success rates of LExaBan, ExaBan, and AdaBan for queries without aggregates. On Academic, LExaBan succeeded on all instances. On IMDB, LExaBan topped ExaBan and even the approximate solution AdaBan in the lineage-level and achieved significantly greater query-level success (>95%). LExaBan succeeds on more than 95% (90%) of instances on which ExaBan (AdaBan) failed. For TPC-H, many queries exhibit hard lineage

Table 5. 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%
	LExaShap	97.82%	98.84%
	ExaShap	97.82%	98.84%
IMDB	LExaBan	95.43%	99.98%
	ExaBan	81.73%	99.63%
	AdaBan	87.82%	99.81%
	LExaShap	87.31%	99.89%
	ExaShap	65.48%	99.53%
TPC-H	LExaBan	75.00%	93.33%
	ExaBan	58.33%	91.52%
	AdaBan	75.00%	92.73%
	LExaShap	58.33%	91.52%
	ExaShap	50.00%	85.46%

instances and so the query success rates are lower for all algorithms. LExaBan again tops both ExaBan and AdaBan in lineage success rate and is tied with AdaBan in query success rate.

Runtime performance. Table 6 gives the runtime of LExaBan, ExaBan and AdaBan for the Academic, IMDB and TPC-H datasets. LExaBan outperforms both competitors significantly. Despite achieving success for more instances, it keeps a smaller mean runtime than ExaBan and AdaBan. For Academic, except for the unique instance where it succeeds and the other algorithms fail on, runtime is below 5 seconds, yielding 119X and 36X speedups over ExaBan and AdaBan, respectively. 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 7). For the instances on which ExaBan failed (Table 8), 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.

5.3 Banzhaf Computation for Aggregate Queries

Table 9 reports the results for LExaBan on aggregate queries on TPC-H$_{agg}$. The performance is excellent for SUM and COUNT aggregates (under 1 msec for each instance). LExaBan is also fast for most instances with the MAX aggregate. LExaBan without optimizations succeeds in only 86.43% of the instances, compared to 93.57% success rate when using the optimizations. When the unoptimized version succeeds, it is still slightly slower than LExaBan, though both algorithms perform excellently (under 0.26 sec).

Table 10 shows the results for the synthetic and challenging IMDB$_{agg}$ and Academic$_{agg}$ datasets. We again observe good performance for SUM (and COUNT, for which the results are very similar and omitted). Specifically, LExaBan succeeds on all SUM query instances for Academic$_{agg}$ and 95.43% of query instances for IMDB$_{agg}$. It is clearly superior to the variant that uses ExaBan for the individual Boolean formulas in the semimodule expressions. For MAX, there is no baseline to compare against and LExaBan is successful in 78% (61%) of the instances for Academic$_{agg}$ (IMDB$_{agg}$).

Table 6. 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

Table 7. 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 8. 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

Table 9. Success rate and runtime of LExaBan for the TPC-H$_{agg}$ aggregate queries.

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

For both datasets, it finishes in less than 0.1 sec for over 50% of the lineage instances. Fig. 4 shows the average runtime for all lineage instances of a representative hard query (TPC-H Query 5) when the size of the Lineitem relation is varied. LExaBan needs about 1.6 sec on average even for the 3M facts in Lineitem, compared to over 10 min for a “naive" version without optimizations.

Table 10. 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	Success	Execution times [sec]					
	& Aggregate	rate	Mean	p50	p90	p95	p99	Max
`Academic` *agg*	LExaBan$_{sum}$	100%	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$_{\mathrm{max}}$	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$_{\mathrm{max}}$	61.93%	52.25	0.09	23.79	242.6	734.9	2407

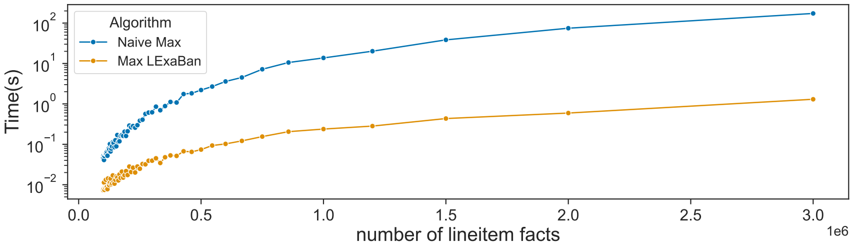
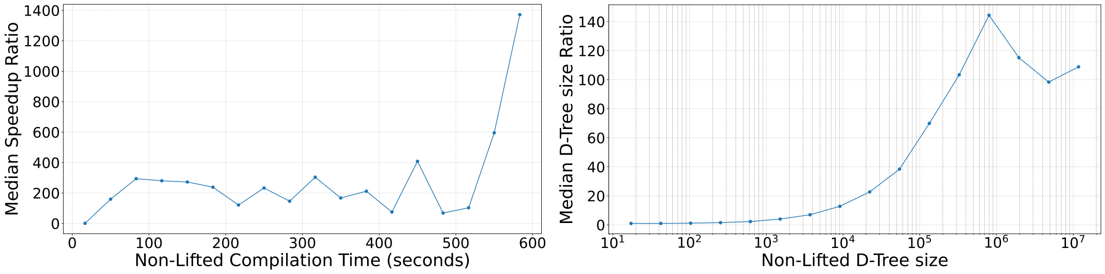


Fig. 4. Runtime of LExaBan for a TPC-H query 5 with MAX aggregate, when varying the number of `Lineitem` facts.



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

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

Fig. 5. Effect of variable lifting on LExaBan's compilation of lineage of Boolean queries into d-trees.

5.4 Effect of Lifting and Gradient Techniques

Fig. 5 shows that variable lifting can speed-up lineage compilation by over one order of magnitude and reduce the d-tree size by over two orders of magnitude. These benefits increase as the lineage size grows. This experiment includes all Boolean queries that can be processed within 10 min. Fig. 6 shows the success rate and runtime of LExaBan with and without the gradient technique for Boolean queries lineages. The technique is beneficial for all instances; the benefit increases with the instance size. For instances with over 1600 variables, the average speedup is more than 2 orders of magnitude.

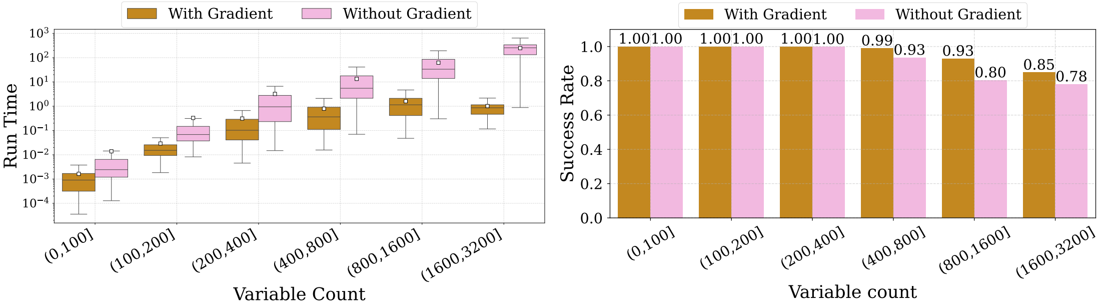


Fig. 6. Effect of the gradient technique on the success rate and runtime of LExaBan for lineage of Boolean queries.

Table 11. 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 12. 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

5.5 Shapley Computation

Table 5 shows that our extension of LExaBan to compute the Shapley value achieves a higher query and lineage success rates than the state-of-the-art ExaShap [2]. Table 11 shows the runtimes of LExaShap and ExaShap on instances where ExaShap succeeds. Table 12 shows LExaShap's success rates and runtimes on instances where ExaShap fails. LExaShap's speedup over ExaShap is over an order of magnitude on average. LExaShap also has a much higher success rate: 79% for the IMDB instances on which ExaShap fails, with a median time of 13.04 sec; and 41% for the TPC-H instances on which ExaShap fails, with a median runtime of 23.94 sec, yielding speedups of at least 276X and 150X, respectively.

5.6 Memory Consumption

Table 13 shows the memory consumption[3] of LExaBan, ExaBan, LExaShap, and ExaShap on representative instances for which ExaBan succeeds, with d-tree sizes spanning different percentiles of the overall distribution. For 99% of these instances, LExaBan uses less than 15MB and up to 37% less memory than ExaBan. For the largest instance solved by

[3] This is measured as peak resident memory, sampled throughout execution using a background monitoring thread that uses the psutil Python library.

Table 13. Memory usage for various percentiles of d-tree sizes on instances (all datasets) for which ExaBan succeeds.

Algorithm	**Memory consumption (MB)**					
	p25	**p50**	**p75**	**p90**	**p99**	**Max**
LExaBan	11.65	11.71	11.77	11.90	13.31	109.67
ExaBan	18.54	18.71	18.82	19.06	21.13	7516.16
LExaShap	74.85	75.05	75.25	75.70	77.20	268.10
ExaShap	18.52	18.62	18.87	18.94	21.45	-

ExaBan, ExaBan uses over 7.3GB, while LExaBan uses only 109MB, 69x less. LExaShap incurs an overhead that is independent of the problem size and due to the `scipy.signal` library that handles the gradient (not needed by the other algorithms). For the largest instance solved by ExaBan, LExaShap uses 268MB, while ExaShap fails.

5.7 Effect of Lineage Structure

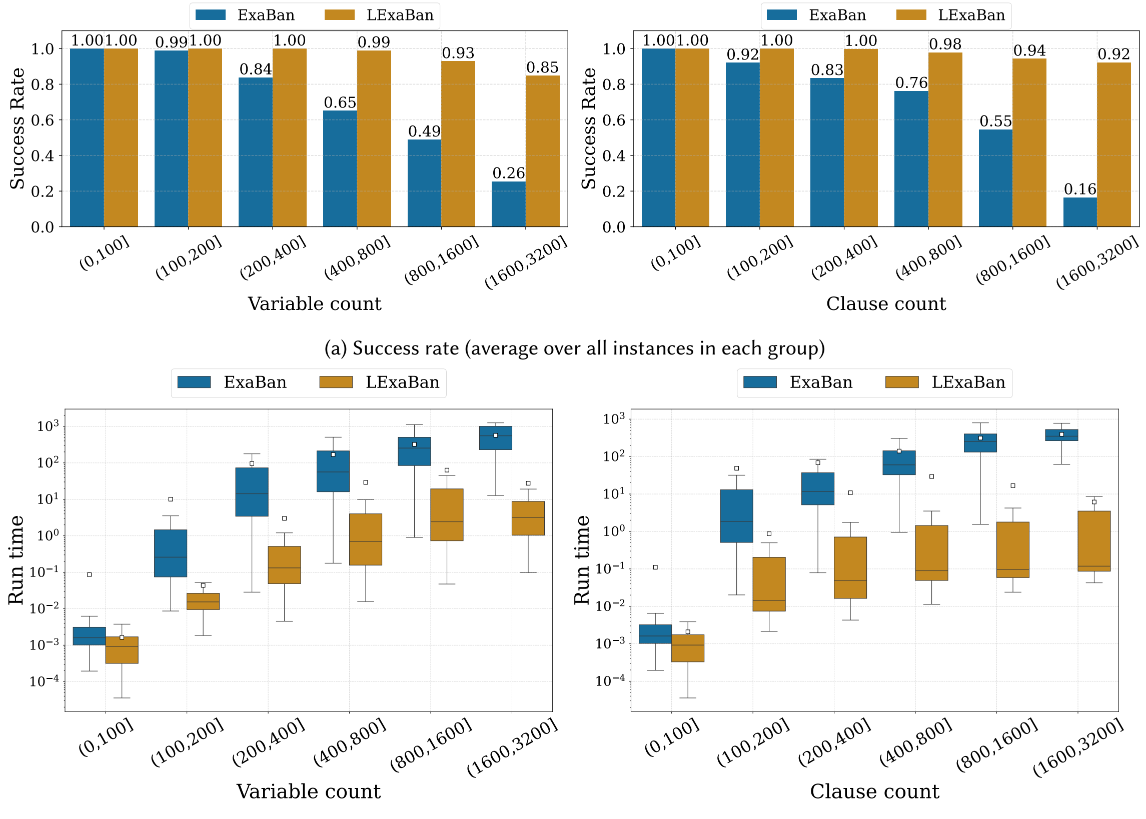
Fig. 7 shows the effect of the number of variables and clauses in the lineage on the performance of LExaBan and ExaBan. LExaBan's success rate is consistently higher and its runtime increases slower than for ExaBan. Fig. 8 presents the effect of two measurements of lineage connectivity: (1) h-index [16], namely the maximal k so that at least k variables occur at least k times in the lineage; (2) maximal connected component size of a graph whose nodes are variables and edges reflect variable co-occurrence in a clause. All algorithms slow down as connectivity grows (fluctuations are due to small sample sizes for some values of these measurements), yet LExaBan and LExaShap handle highly interconnected lineages much better than ExaBan and ExaShap. ExaBan and ExaShap do not terminate for lineages with h-index greater than 53, whereas LExaBan and LExaShap can handle lineages with h-index of 120.

5.8 Breakdown of Total Runtime

We conclude with a breakdown of the runtime, including the time to compute the output tuples (query execution), the lineage via ProvSQL, and the Banzhaf values for all datasets (Fig. 9). As queries may produce many output tuples and ProvSQL outputs lineage for each of them, we break down LExaBan runtime by percentiles of the execution time w.r.t. output tuples. LExaBan usually incurs a small overhead: for 81% (85%) of queries, attribution for all input tuples wrt any output tuple is computed faster than query execution (resp. query execution and lineage computation, combined). For 7% of queries, attribution for at least one output tuple is 2 OOM slower than query execution. Overall, the total explanation time (lineage + Banzhaf) exceeds the query execution time in only 27% of queries, and exceeds it by more than 10x in 13% of queries.

6 RELATED WORK

Recent work laid the theoretical foundation of attribution via Banzhaf and Shapley values in query answering [8, 12, 17, 18, 20, 27]. All prior practical approaches to computing such values rely on compiling the lineage into tractable circuits such as d-DNNFs [12] or d-trees [2]. In this paper, we follow the latter approach and non-trivially extend it along two dimensions. First, we accommodate aggregate queries. This is challenging for two reasons: (1) the lineage of aggregate queries has a more complex form than that of SPJU queries studied in all prior work, and (2) attribution computation requires new machinery, accounting for the interaction between aggregate functions and Shapley/Banzhaf values. Ours is the first practical approach for aggregate queries: [20] includes a theoretical investigation of aggregate queries but no practically efficient algorithm, whereas [2, 12] do not support aggregates.



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

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

Fig. 7. Performance of LExaBan and ExaBan for all lineages, grouped by number of variables/clauses. $[i, j]$ represents lineages with #vars (# clauses) between i and j.

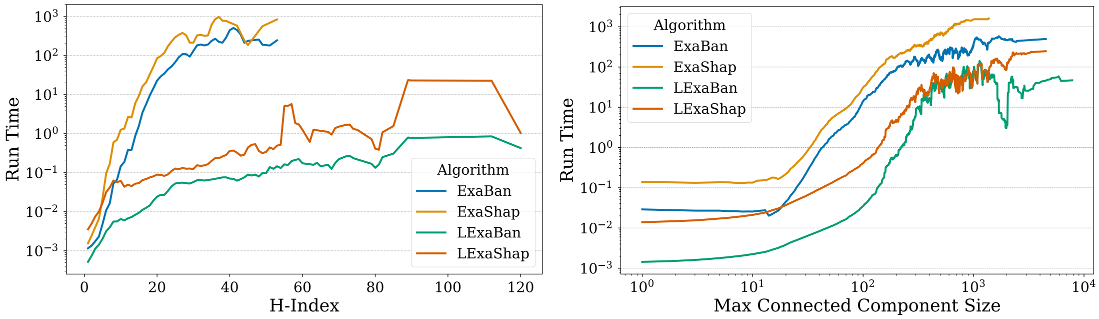


Fig. 8. Effect of the H-index and maximum component size of the lineage on the algorithms' execution time. Both metrics were smoothed using a moving average, with window sizes 3 (H-index) and 25 (component size).

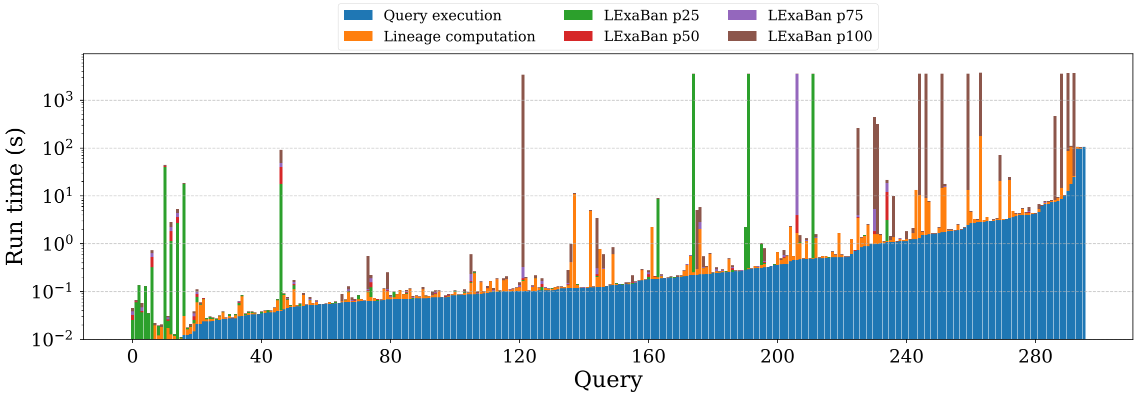


Fig. 9. Breakdown of runtime for all queries sorted by the time to compute the output tuples (query execution time).

Second, our approach consistently and significantly outperforms the state-of-the-art, by up to 3 (2) orders of magnitude for Banzhaf (Shapley). This speedup enables the computation of attribution for instances that prior work cannot handle (over 90% of these difficult instances for Banzhaf and over 75% for Shapley). The novel techniques of *lifting* and *gradient-based computation* introduced in this paper are the reason for this speedup and enable the processing of expensive SPJU queries well beyond the reach of prior work, and of aggregate queries. In particular, lifting improves compilation time by over 2 orders of magnitude; gradient-based computation yields speedups across all instance sizes, with improvements exceeding 2 orders of magnitude for the largest instances.

Beyond Shapley and Banzhaf, further notions have been used to quantify fact contribution in query answering: causality [25], responsibility [23], and counterfactuals [24]. The SHAP score [21] leverages and adapts the definition of Shapley values to explain model predictions. Extending our techniques to further forms of attribution is an intriguing challenge for future work.

7 CONCLUSIONS

We have introduced a novel approach for computing attribution for query answering. Our approach is more general, as it supports aggregate queries, and faster by several orders of magnitude than the state of the art. Future work includes extending our approach to broader query classes, e.g., queries with AVG aggregation and negation, and further attribution measures, e.g., the SHAP score.

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. *Advancing Fact Attribution: Aggregate Queries and Novel Algorithms (Extended Version).* Technical Report. Extended version of the paper, available at https://github.com/Omer-Abramovich/LExaBan-LExaShap.

[4] Yael Amsterdamer, Daniel Deutch, and Val Tannen. 2011. Provenance for Aggregate Queries. In *PODS.* 153–164. https://doi.org/10.1145/1989284.1989302

[5] Bahareh Sadat Arab, Su Feng, Boris Glavic, Seokki Lee, Xing Niu, and Qitian Zeng. 2018. GProM - A Swiss Army Knife for Your Provenance Needs. *IEEE Data Eng. Bull.* 41, 1 (2018), 51–62. http://sites.computer.org/debull/A18mar/p51.pdf

[6] John F Banzhaf III. 1965. Weighted Voting Doesn't Work: A Mathematical Analysis. *Rutgers Law Review* 19, 2 (1965), 317–343. https://heinonline.org/HOL/LandingPage?handle=hein.journals/rutlr19&div=19&id=&page=

[7] 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

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

[9] Meghyn Bienvenu, Diego Figueira, and Pierre Lafourcade. 2024. Shapley Value Computation in Ontology-Mediated Query Answering. In *KR.* Article 15, 11 pages. https://doi.org/10.24963/kr.2024/15

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

[11] 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

[12] 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

[13] Robert Fink, Larisa Han, and Dan Olteanu. 2012. Aggregation in Probabilistic Databases via Knowledge Compilation. *PVLDB* 5, 5 (Jan. 2012), 490–501. https://doi.org/10.14778/2140436.2140445

[14] Daniel Vidali Fryer, Inga Strümke, and Hien D. Nguyen. 2021. Shapley Values for Feature Selection: The Good, the Bad, and the Axioms. *IEEE Access* 9 (2021), 144352–144360. https://doi.org/10.1109/ACCESS.2021.3119110

[15] Boris Glavic, Alexandra Meliou, and Sudeepa Roy. 2021. Trends in Explanations: Understanding and Debugging Data-Driven Systems. *Proc. VLDB Endow.* 11, 3 (2021), 226–318. https://doi.org/10.1561/1900000074

[16] J. E. Hirsch. 2005. An Index to Quantify an Individual's Scientific Research Output. *PNAS* 102, 46 (2005), 16569–16572. https://doi.org/10.1073/pnas.0507655102

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

[18] 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

[19] Majd Khalil and Benny Kimelfeld. 2023. The Complexity of the Shapley Value for Regular Path Queries. In *ICDT*, Vol. 255. 11:1–11:19. https://doi.org/10.4230/LIPICS.ICDT.2023.11

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

[21] 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

[22] 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 (2024), 28 pages. https://doi.org/10.1145/3639311

[23] Alexandra Meliou, Wolfgang Gatterbauer, Katherine F. Moore, and Dan Suciu. 2010. The Complexity of Causality and Responsibility for Query Answers and Non-Answers. *PVLDB* 4, 1 (2010), 34–45. https://www.vldb.org/pvldb/vol4/p34-meliou.pdf

[24] Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. 2011. Bringing Provenance to Its Full Potential Using Causal Reasoning. In *TaPP.* https://www.usenix.org/conference/tapp11/bringing-provenance-its-full-potential-using-causal-reasoning

[25] Alexandra Meliou, Sudeepa Roy, and Dan Suciu. 2014. Causality and Explanations in Databases. *PVLDB* 7, 13 (Aug. 2014), 1715–1716. https://doi.org/10.14778/2733004.2733070

[26] L. S. Penrose. 1946. The Elementary Statistics of Majority Voting. *J. Royal Stats. Soc.* 109, 1 (1946), 53–57. http://www.jstor.org/stable/2981392

[27] 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. https://doi.org/10.1145/3375395.3387664

[28] Benedek Rozemberczki, Lauren Watson, Péter Bayer, Hao-Tsung Yang, Oliver Kiss, Sebastian Nilsson, and Rik Sarkar. 2022. The Shapley Value in Machine Learning. In *IJCAI*. 5572–5579. https://doi.org/10.24963/IJCAI.2022/778

[29] Pierre Senellart, Louis Jachiet, Silviu Maniu, and Yann Ramusat. 2018. ProvSQL: Provenance and Probability Management in PostgreSQL. *PVLDB* 11, 12 (2018), 2034–2037. https://hal.inria.fr/hal-01851538/file/p976-senellart.pdf

[30] 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

[31] 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. https://doi.org/10.1016/J.NEUNET.2018.06.006

[32] Jiachen T. Wang and Ruoxi Jia. 2023. Data Banzhaf: A Robust Data Valuation Framework for Machine Learning. In *AISTATS*, Vol. 206. 6388–6421. https://proceedings.mlr.press/v206/wang23e.html

A MISSING DETAILS IN SECTION 3

We provide a full proposition for the Shapley analogue for Proposition 3.7.

Recall the definition of the k-probability function. For a given d-tree T and an integer $k \in \mathbb{N}$

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

Proposition A.1. Given a d-tree T and a variable $x \in \mathrm{vars}(T)$, the following holds:

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

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

Proof. Given a variable $x \in \mathrm{vars}(T)$, 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:

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

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

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

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

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

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:

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

$$\cdot \sum_{k\in[|\mathsf{vars}(T)|]} C_{k-1}\cdot \sum_{S\subseteq vars(T)\setminus x, |S|=k-1} \big(T[S\cup\{x\}] - T[S]\big)$$

$$= \sum_{S\subseteq vars(T)\setminus x} C_{|S|}\big(T[S\cup\{x\}] - T[S]\big) = (\frac{1}{2})^{|\mathsf{vars}(T)|-1}\mathsf{Shapley}(T,x)$$

□

We provide here the analogous equations to Table 3 that are necessary for the computations of Shapley values.

Table 14. 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. Here, T_{-T_i} represents the d-tree T when excluding the sub-tree T_i, and $A_n^m = \frac{1}{2^n}\cdot\binom{n}{m}$ represents the portion of assignments of n variables of size m.

Gate	Probability Expression $Pr_k[T]$	Partial Derivative
$\oplus$	$A^k_{\lvert\mathsf{vars}(T)\rvert} - \sum_{j_1,\ldots,j_n,\sum j_1,\ldots j_n=k}\prod_{i\in[n]} A^{j_i}_{\lvert\mathsf{vars}(T_i)\rvert} - Pr_{j_i}[T_i]$	$A^{k-j}_{\lvert\mathsf{vars}(T_{-T_i})\rvert} - Pr_{k-j}[T_{-T_i}]$
$\odot$	$\sum_{j_1\ldots,j_n,\sum j_1\ldots,j_n=k,\forall_i.j_i>0}\prod_{i\in[n]} Pr_{j_i}[T_i]$	$Pr_{k-j}[T_{-T_i}]$
$\sqcup_f$	$\sum_{j\in[k]} Pr_j[T_1]\cdot Pr_{k-j}[f] + Pr_j[T_0]\cdot(A^{k-j} - Pr_{k-j}[f])$	$\frac{\partial Pr_k[T]}{\partial Pr_j[T_1]} = Pr_{k-j}[f]$ $\frac{\partial Pr_k[T]}{\partial Pr_j[T_0]} = (A^{k-j} - Pr_{k-k}[f])$ $\frac{\partial Pr_k[T]}{\partial Pr_j[f]} = Pr_{k-j}[T_1] - Pr_{k-j}[T_0]$