Tractable Circuits in Database Theory

Antoine Amarilli
LTCI, Télécom Paris
Institut Polytechnique de Paris
antoine.amarilli@telecom-paris.fr

Florent Capelli
Université d'Artois, CNRS, UMR 8188
Centre de Recherche en Informatique de Lens
capelli@cril.fr

ABSTRACT

This work reviews how database theory uses tractable circuit classes from knowledge compilation. We present relevant query evaluation tasks, and notions of tractable circuits. We then show how these tractable circuits can be used to address database tasks. We first focus on Boolean provenance and its applications for aggregation tasks, in particular probabilistic query evaluation. We study these for Monadic Second Order (MSO) queries on trees, and for safe Conjunctive Queries (CQs) and Union of Conjunctive Queries (UCQs). We also study circuit representations of query answers, and their applications to enumeration tasks: both in the Boolean setting (for MSO) and the multivalued setting (for CQs and UCQs).

1 Introduction

The field of *knowledge compilation* [62] studies how to efficiently reason about propositional knowledge bases, and how to represent logic formulas as data structures that ensure the tractability of tasks such as Boolean satisfiability. These data structures are often based on *decision diagrams* such as OBDDs [43] or on restricted versions of Boolean *circuits*; they often naturally correspond to the trace of an algorithm [77]. Such data structures can then be used to reason on the knowledge base, using different tools depending on the task (satisfiability, model counting, etc.).

These tasks are naturally connected to database problems: e.g., satisfiability may be seen as testing whether a query has an answer, and model counting can be seen as finding its number of answers. The main difference is that knowledge compilation focuses on propositional logic, which can be seen as the minimal setting where the techniques can be applied. It is hence only natural to use these techniques and tools from knowledge compilation and adapt them to the setting of databases. In this paper, we review this line of work which designs efficient algorithms for database tasks using tractable circuit representations of Boolean functions or relations.

We identify two main ways in which circuit classes are used in database theory. The first is via Boolean functions that naturally arise in databases, such as *Boolean provenance*, or *answer functions* that represent the output of queries. Knowledge compilation proposes efficient formalisms in which we can represent such Boolean functions and ensure that some tasks are tractable over them. Thus, if we can efficiently compute the Boolean provenance or answer functions and represent them in a tractable circuit class, then we can show some theoretical tractability results, and/or use practical tools. This approach has been used across different domains, e.g., query evaluation on probabilistic databases [80], enumeration for monadic second-order queries [8], and Shapley value computation [63]; we will review these works and more. Conversely, if we can show that the output to some problems *cannot* be tractably represented as a circuit in a given class, then we show that these problems cannot efficiently be solved by algorithms whose trace falls in that class: this approach is followed by [28] among others.

The second family of applications of circuit classes in databases is under the guise of *factorized databases* [114], in which relations are succinctly represented as circuits with restricted operators from relational algebra. This notion was introduced to understand various tasks on the answer set of restricted kinds of Conjunctive Queries (CQs) [26, 115, 135]. As we will explain, factorized databases can be seen as natural generalizations of the circuits used in knowledge compilation, going from the Boolean domain to a multivalued domain [110]. We will then explain how such circuits can be used to recover known results.

In both these settings, we see the strengths of circuits: they give a unifying view on scattered results, and they provide a modular and generic way to solve complex tasks. Namely, they make it possible to design query algorithms whose only task is to produce a circuit that falls in a given class. The circuit can then be fed to existing algorithms and software implementations which can solve various problems on the circuit independently from how it was built.

The paper is organized as follows. Section 2 first reviews how *Boolean functions* naturally occur in database tasks. We focus on *aggregation tasks* that we relate to *Boolean provenance*, and *enumeration tasks* that we relate to *answer functions*. We then study in Section 3 how to efficiently represent such functions: we review known circuit classes from knowledge compilation and the associated tractability results and software implementations. We then explain how the tractable circuit classes of Section 3 allow us to address the tasks of Section 2: we start in Section 4 with Monadic Second-Order (MSO) queries on trees, then study aggregation tasks for CQs and Union of CQs (UCQs) in Section 5. We then move in Section 6 to the multivalued perspective: we see how tractable circuit classes can be seen as factorized relations to succinctly represent the answer set of CQs. We close with an overview of other directions and questions for future research in Section 7.

2 Database Tasks and Boolean Functions

We first show in this section how various database tasks can be expressed in terms of *Boolean functions*, to be later represented by circuits. We focus on two main kinds of tasks. First, we present tasks expressed in terms of the *Boolean provenance* of a query on the input data. We focus on *aggregation tasks* on the provenance, which intuitively involve some form of counting. Second, we will study how tasks can be expressed in terms of Boolean functions that capture the answer of queries, e.g., in MSO. We focus there on *enumeration tasks*, which ask for the computation of witnesses. We last sketch *other tasks* related to circuits that we do not investigate in detail.

2.1 Boolean Provenance

We first recall some standard terminology. Provenance is defined over *(relational) instances*, which are sets of facts over a signature. Formally, a *signature* consists of a set of relation names with an associated arity, a *fact* for a relation name R of arity n is an expression of the form $R(a_1,\dots,a_n)$ with $a_1,\dots,a_n$ being values, and an *instance* is a set of facts. A *Boolean query* Q is then a function that maps instances $\mathbb{D}$ to a Boolean value indicating whether the query is satisfied by $\mathbb{D}$. The *Boolean provenance* of Q over an instance $\mathbb{D}$ then simply describes the truth status of Q on all *subinstances* of $\mathbb{D}$, i.e., all subsets of its facts. Formally:

DEFINITION 2.1. *Let $\mathbb{D}$ be an instance and Q be a Boolean query. The* provenance *of Q on $\mathbb{D}$ is the Boolean function from $2^{\mathbb{D}}$ to $\{0,1\}$ that maps each subinstance $\mathbb{D}' \in 2^{\mathbb{D}}$ of $\mathbb{D}$ to 1 if Q is true on $\mathbb{D}'$ and to 0 otherwise.*

As a Boolean function, provenance can be represented in multiple ways, e.g., as *Boolean formulas* like in the example below, or as *Boolean circuits* (see Section 3).

EXAMPLE 2.2. *Let $\mathbb{D} = \{R(a), R(a'), S(b)\}$ be an instance and Q be the Boolean CQ $\exists x y\, R(x), S(y)$ asking for the presence of an R-fact and of an S-fact. The provenance of Q on $\mathbb{D}$ is the function denoted by the Boolean formula $(R(a) \vee R(a')) \wedge S(b)$.*

Semiring provenance and more general semirings. Boolean provenance as defined here is the special case of *semiring provenance* [73] for the semiring $\mathrm{Bool}[X]$ of Boolean functions. One advantage of Boolean provenance is that it is purely semantic, i.e., it considers the query as a black-box. By contrast, provenance for more general semirings like $\mathbb{N}[X]$ often depends on how the query is executed. We note that circuit notions have also been defined for such general provenance semirings [12, 64], in connection with arithmetic circuits; but we only discuss Boolean provenance from now on.

Now that we have defined Boolean provenance for queries, we explain how database tasks can be rephrased in terms of Boolean provenance. We will focus here on *aggregation tasks*, where we intuitively want to perform a kind of counting over subsets of the input instance.

Uniform reliability. The simplest aggregation task is *uniform reliability* (UR) for a Boolean query Q: given an instance $\mathbb{D}$, count the subinstances of $\mathbb{D}$ that satisfy Q. This can be solved via *model counting*. Formally:

DEFINITION 2.3. *Let ϕ be a Boolean function over variables X. The* model counting *problem (#SAT) for ϕ asks how many valuations of X satisfy ϕ. Formally, for $Y \in 2^X$, we write ν_Y for the Boolean valuation over X that maps $x \in X$ to 1 if $x \in Y$ and to 0 otherwise. #SAT is the problem of computing $\#\phi := |\{Y \in 2^X \mid \nu_Y \text{ satisfies } \phi\}|$.*

Hence, if ϕ if the provenance of Q on an instance $\mathbb{D}$, then the answer to UR for Q is the answer to #SAT for ϕ.

Probabilistic query evaluation. A generalization of UR is *probabilistic query evaluation* (PQE) for Q. In this setting, we are given a so-called *tuple-independent database* (TID): it consists of an instance $\mathbb{D}$ with a function $\pi\colon \mathbb{D} \to [0,1]$ mapping each fact of $\mathbb{D}$ to a rational probability value. We assume independence across facts, and consider the product probability distribution on subinstances of $\mathbb{D}$, where the probability of $\mathbb{D}' \subseteq \mathbb{D}$ is $\pi(\mathbb{D}') := \prod_{F \in \mathbb{D}'} \pi(F) \times \prod_{F \in \mathbb{D} \setminus \mathbb{D}'} (1 - \pi(F))$. We want to know the total probability of the subinstances of $\mathbb{D}$ that satisfy Q, namely, $\sum_{\mathbb{D}' \subseteq \mathbb{D}, \mathbb{D}' \text{ satisfies } Q} \pi(\mathbb{D}')$. This problem can be solved via *weighted model counting*:

DEFINITION 2.4. *Let $(K, \otimes, +)$ be a semiring, and let ϕ be a Boolean function over X. We write $Lit(X)$ the set of literals over X, i.e., $X \cup \{\neg x \mid x \in X\}$. Let $w\colon Lit(X) \to K$ be a* weight function *giving a weight in K to each literal over X. The* weight *of a Boolean valuation $\nu\colon X \to \{0,1\}$ is then $w(\nu) := \bigotimes_{x \in X, \nu(x)=1} w(x) \times$*

$\bigotimes_{x\in X, \nu(x)=0} w(\neg x)$. *Then, the* weighted model counting *problem (WMC) for ϕ is to compute the total weight of satisfying valuations, i.e.,* $\sum_{\nu \text{ satisfies } \phi} w(\nu)$.

Note that model counting for ϕ reduces to WMC with a weight of 1 for each literal. We then have that PQE for Q on a TID $(\mathbb{D}, \pi)$ amounts to WMC for the provenance ϕ of Q on $\mathbb{D}$ in the semiring of rationals $(\mathbb{Q}, \times, +)$ with weights given by π.

As the tasks UR and PQE are often intractable, we may prefer to study *approximate model counting* (ApproxMC) and its weighted variant (ApproxWMC). In these variants, instead of solving the problem exactly, we wish to compute an approximation of the model count or of the probability. We focus on *multiplicative approximations*: given an error $\varepsilon > 0$, we must compute an approximation $\bar{x}$ of the true value x that ensures $(1-\varepsilon)x \leq \bar{x} \leq (1+\varepsilon)x$. We often allow approximation algorithms to be *randomized*: then, the output of the algorithm must be a correct approximation with probability at least $2/3$. We say that we have a *fully polynomial-time approximation scheme* (FPRAS) for Approx(W)MC when we have a randomized algorithm to compute a multiplicative approximation of the count with running time polynomial in the input instance and in the inverse of the desired error ε.

Shapley values. Another aggregation task is the computation of *Shapley values*, which can be used to quantify the contribution of a fact to making the query true [35]. In this setting, we fix a query Q and we are given as input an instance $\mathbb{D}$, which is partitioned between so-called *exogenous* facts $\mathbb{D}_{\mathrm{x}}$, which are always present, and *endogenous* facts $\mathbb{D}_{\mathrm{n}}$, which may be present or absent. The Shapley value is then an aggregate over all subinstances of $\mathbb{D}$ that contain all the exogenous facts of $\mathbb{D}_{\mathrm{x}}$. We omit the formal definition of the Shapley value; see [35]. Note that the Shapley value reduces in particular to counting how many subsets of endogenous facts of a given cardinality satisfy Q together with the exogenous facts [100].

The computation of Shapley values can be posed on the Boolean provenance ϕ of the query Q, but imposing that all exogenous facts are kept. This amounts to partial evaluation of ϕ, namely, setting the variables of $\mathbb{D}_{\mathrm{x}}$ to 1.

2.2 Boolean Answer Functions

We now move on from Boolean provenance to a different kind of Boolean functions, this time defined to represent the results of queries. It will be easier to define these functions for queries with one free second-order variable, i.e., queries that return subset of domain elements as answers, in particular queries expressed in MSO. We will revisit this perspective in Section 6 to work on more conventional query results that consist of relations.

Formally, the semantics of a query $Q(X)$ with a free second-order variable is that X stands for a set of elements taken from the active domain: given an instance $\mathbb{D}$, letting D be the *domain* of $\mathbb{D}$ (the set of elements occurring in facts of $\mathbb{D}$), the *answers* of Q on $\mathbb{D}$ is the set of subsets $A \subseteq D$ such that $Q(X := A)$ is true.

EXAMPLE 2.5. *On a signature with one binary relation R and one unary relation V, consider the query $Q(X) := \neg(\exists y z\, (R(y,z) \vee R(z,y)) \wedge V(y) \wedge V(z) \wedge X(y) \wedge X(z))$. Then, given an instance $\mathbb{D}$ with domain D, we have that $Q(X := A)$ holds precisely when $A \subseteq D$ is an independent set of the graph represented by $\mathbb{D}$ with vertices coded by V and edges coded by R.*

In this setting, for a query Q, given an instance $\mathbb{D}$ with domain D, we can naturally define a Boolean function ϕ describing the answers of Q on $\mathbb{D}$, called the *answer function* of ϕ on $\mathbb{D}$. Formally, the variables of the answer function are the domain elements in D, and it is satisfied by a Boolean valuation $\nu\colon D \to \{0,1\}$ precisely when the set $A_\nu := \{a \in D \mid \nu(a) = 1\}$ is an answer to Q.

For a query $Q(X_1, \ldots, X_k)$ with multiple second-order variables, we can also define the answer function of Q on an instance $\mathbb{D}$ with domain D: it is a Boolean function with variable set $D \times [k]$, satisfied by the valuations $\nu\colon D \times [k] \to \{0,1\}$ such that the tuple of the sets $A_i := \{a \in D \mid \nu((a,i)) = 1\}$ for $i \in [k]$ is an answer to Q.

We now study how problems over the set of answers of a query Q can be posed over their Boolean answer function. One example is *answer counting*: the number of answers of Q over an instance $\mathbb{D}$ is the number of satisfying assignments of the answer function of Q on $\mathbb{D}$. However, in this section, we focus on *enumeration tasks*: given $\mathbb{D}$, we must compute a set of solutions of Q over $\mathbb{D}$.

Finding and enumerating satisfying valuations. The simplest task on an answer function ϕ is simply to decide if the query has an answer, i.e., if ϕ has a satisfying valuation (called *satisfiability* or SAT); and to compute one if it exists. A variant is *uniform sampling*, i.e., a randomized algorithm that must produce one satisfying valuation uniformly at random. We note that the task can be made approximate by allowing a *failure probability* or doing *near-uniform sampling*.

However, it may also be important to *compute all satisfying valuations*. One challenge to formalize the task is that the number of answers to produce can be large, which makes it difficult to measure efficiency.

EXAMPLE 2.6. *Continuing Example 2.5, consider the task of enumerating all query answers, i.e., all independent sets of the input graph. A naive algorithm to compute the answers is to list each possible subset A and test if it is an independent set. The naive algorithm takes time $\Omega(2^n)$, but it is hard to improve on this, because the worst-case complexity of the task is also $\Omega(2^n)$. Indeed,* *the running time cannot be less than the number of answers to produce, and given an instance $\mathbb{D}$ with n facts, there may be up to 2^n answers (e.g., for n isolated vertices). What we intuitively want is to beat $\Omega(2^n)$ on the instances where the answer set is small; or to produce the first few solutions faster than $\Omega(2^n)$.*

This example illustrates why we study algorithms in terms of the output size. One way is *enumeration algorithms*, which have been studied in many contexts (see, e.g., [133]). In data management, enumeration algorithms distinguish two phases: a *preprocessing phase* to perform some precomputations, and an *enumeration phase* to produce all solutions with small *delay* between consecutive solutions. We omit the formal definition of enumeration algorithms; see, e.g., [120].

DEFINITION 2.7. *Given a Boolean function ϕ on variables X, the task of* enumerating satisfying valuations *(Enum) of ϕ asks us to efficiently produce the list of all satisfying valuations, with small preprocessing time and small delay between any two successive valuations. Note that, in the enumeration, we may write each valuation $\nu\colon X \to \{0,1\}$ as the set $\{x \in X \mid \nu(x) = 1\}$: this may be more concise when the Hamming weight of ν is small.*

Thus, for the database tasks presented above, we can devise enumeration algorithms for queries via answer functions: compute a representation of the answer function, then enumerate its satisfying valuations.

Ranked enumeration and ranked access. Beyond the mere listing of satisfying valuations, it is often relevant to list them *in a specific order*, or to find the *top-k satisfying valuations* in this order. We also want to do *ranked access*: quickly access the i-th solution in this order, which generalizes tasks such as quantile computation [49].

DEFINITION 2.8. *Given a Boolean function ϕ on variables X, given a total order $<$ defined on the set 2^X of valuations of X, the task of* ranked enumeration of satisfying valuations *of ϕ under $<$ is to enumerate the satisfying valuations of ϕ in the order given by $<$. The task of computing the* top-k satisfying valuations*, given $k \in \mathbb{N}$, is to compute the k satisfying valuations that are first according to $<$. The task of* ranked access *is to return, given $i \in \mathbb{N}$, the i-th satisfying valuation in the order, or fail if the number of satisfying valuations is less than i.*

There are two main ways in which we can represent an order on valuations without materializing its comparability graph explicitly. One first method is *lexicographic orders*: we are given a total order $<'$ on the set X of variables, and extend it lexicographically to valuations. Formally, given two valuations $\nu, \nu' \in 2^X$, we have $\nu < \nu'$ if, letting x be the smallest variable for $<'$ such that $\nu(x) \neq \nu'(x)$, we have $\nu(x) = 0$ and $\nu'(x) = 1$. The second method is *weights*: we fix a semigroup $(K, \otimes)$, and define the weight $w(\nu)$ of a valuation ν for a weight function w like in Definition 2.4. Now, if we pick some total order $<'$ on semigroup values, we can sort valuations according to their weight in K: we may typically assume that $<'$ is compatible in some sense with the semigroup law (e.g., *subset-monotonicity* [7,128]), and often impose some tie-breaking rule to make the order total. In particular, when $K = \mathbb{Q}$ and weights are probabilities (like for PQE), we can compute the *most probable satisfying valuation* or the *most probable falsifying valuation*.

2.3 Other Tasks via Boolean Functions

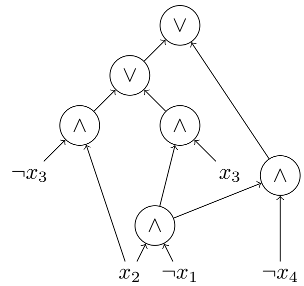
Boolean functions can be used for many data management tasks beyond the ones presented so far. We close the section by briefly alluding to other such tasks.

First, the notion of *database repairs*, specifically in the case of *maximal subset repairs*, can be studied via Boolean provenance by looking at the maximal satisfying valuations of the provenance. Thus, the problem of *subset repair counting* [44, 45, 93] can be seen as the aggregation task of counting the maximal satisfying valuations of the provenance of a certain Boolean query; and the task of *enumerating subset repairs* [93] amounts to the enumeration task of listing the maximal satisfying valuations. A connection could also be phrased for *consistent query answering* [33,34].

Second, more generally, the setting of *reverse data management* [103] provides other examples of problems that can be phrased in terms of Boolean provenance. These problems can consider ways to delete facts in an input instance in order to satisfy a property, for instance *minimum witness* [76,104] (delete as many facts as possible while making the query true) or *resilience* [71,102] (delete as few facts as possible to make the query false). These problems can be posed on the Boolean provenance: find satisfying valuations of small Hamming weight, or falsifying valuations of large Hamming weight.

Third, in the setting of *uncertain databases* [1], we may consider instances where some facts are *uncertain*, i.e., they may be present or absent; and other facts are *certain* and are necessarily present. These correspond to endogenous and exogenous facts in the case of Shapley value computation. We can then study tasks such as *certain query answers* [79], i.e., is the query always true on all possible worlds of an uncertain instance, i.e., is the Boolean provenance tautological. A related task is *query-guided uncertainty resolution* [65] which interactively determines the truth value of the query, by probing individual facts while optimizing, e.g., the worst-case decision tree height. This amounts to *stochastic Boolean function evaluation* [5] of the Boolean provenance.

Fourth, many the tasks defined so far can be extended to *incremental maintenance*: compute the result (or its representation, e.g., as an enumeration index), and maintain this result efficiently under *updates* to the data. In general, circuit representations are amenable to incremental maintenance: we can reevaluate the circuit on a different valuation whenever the data changes. However, this idea assumes that the initial set of facts never grows, i.e., facts can only be removed or added back.



x_1	x_2	x_3	x_4
0	1	0	0
0	1	0	1
0	1	1	0
0	1	1	1
1	1	0	0
1	1	0	1

Figure 1: A DNNF and its satisfying valuations

3 Knowledge Compilation

We have seen in Section 2 that many database tasks can be rephrased to problems on Boolean functions. However, these reductions do not directly give tractable algorithms, because many of these problems are hard. For example, SAT is NP-complete already for Conjunctive Normal Form formulas (CNFs) ϕ. Computing $\#\phi$ is even harder: it is #P-complete [129], and NP-hard to approximate to a $2^{n-\varepsilon}$-factor for every $\varepsilon > 0$, even if ϕ has no negation and clauses of size at most 2 [118]. However, tractability can hold for other representations of Boolean functions. For instance, for a Disjunctive Normal Form (DNF) formula ϕ, finding a satisfying valuation is easy. While computing $\#\phi$ is also #P-complete for DNFs, we can approximate it to a $(1 \pm \varepsilon)$ factor with high probability via a Monte-Carlo-based FPRAS, namely, the Karp-Luby algorithm [86].

Thus, one approach to make the tasks of Section 2 tractable is to build representations of the Boolean functions that are tractable for the task at hand. This modular approach often makes it possible to show several tractability results at once, and allows us to use efficient practical implementations (see Section 3.3).

To achieve this, we turn to the field of *knowledge compilation* [62], which studies tractable representations for Boolean functions and their properties. We review definitions, results, and tools from this field in this section, which we will use in Sections 4 to 6 for various database tasks featuring aggregation and enumeration.

3.1 DNNF Circuits

Most representations of Boolean functions studied in the literature are restricted forms of *Boolean circuits*. The use of circuits allows for efficient sharing and factorization, while the restrictions ensure tractability. One of the most general such representation is Decomposable Negation Normal Form (DNNF) circuits [60]. We start by defining Boolean circuits before introducing DNNFs.

Circuits and NNF circuits. Formally, a *Boolean circuit C* on variables X is a Directed Acyclic Graph together with a distinguished node called the *output*. The nodes with indegree 0, called the *inputs of the circuit*, are each labeled either with a constant 1 or 0 or with a variable $x \in X$. The internal nodes of C, called *gates*, are labeled by $\wedge$, $\vee$ or $\neg$. Given a gate v, every gate w that has an edge to v is called an *input of v*. Every $\neg$-gate has exactly one input. Given a gate v of C, we let $\mathsf{var}(v)$ be the set of variables $x \in X$ that appear in the subcircuit rooted in C, that is, we can reach v by a directed path from an input of C labeled by x. The size $|C|$ of a Boolean circuit is the number of edges in the DAG.

Every gate g of C computes a Boolean function f_g on $\mathsf{var}(g)$: given a valuation $\nu\colon \mathsf{var}(g) \to \{0,1\}$, the value of f_g is defined by setting the inputs of the circuit with ν and evaluating internal gates inductively up to g. The Boolean function on X computed by C is then the function f_o on $\mathsf{var}(o)$ for o the output of C, extended to X by allowing the variables of $X \setminus \mathsf{var}(o)$ to take any value.

We focus on Boolean circuits in *Negation Normal Form* (NNF) where the input of every negation gate is an input of the circuit; alternatively, NNF circuits are circuits built on literals using $\wedge$ and $\vee$ on internal gates. See Figure 1 for an example of an NNF circuit. Any Boolean circuit can be put in NNF in linear time using De Morgan's laws iteratively to push negations down.

DNNF circuits. We now move to restrictions on NNF circuits aimed at enforcing tractability of problems (e.g., SAT). Imposing NNF is not sufficient for this: as we explained, it is essentially without loss of generality. Intuitively, what makes SAT hard is that finding satisfying valuations ν_1 for f_1 and ν_2 for f_2 does not help to find one for $f_1 \wedge f_2$, because ν_1 and ν_2 may be inconsistent on their shared variables. To avoid this, we use *decomposable* $\wedge$-gates. A $\wedge$-gate g is *decomposable* if for every pair of distinct inputs g_1, g_2 of g, we have $\mathsf{var}(g_1) \cap \mathsf{var}(g_2) = \emptyset$. A *Decomposable Negation Normal Form (DNNF)* circuit is an NNF circuit C where every $\wedge$-gate is decomposable. Observe that decomposability makes SAT tractable: we can propagate satisfying valuations upwards in C, building satisfying valuations for decomposable $\wedge$-gates by concatenating satisfying valuations of their inputs. The circuit depicted on Figure 1 is a DNNF.

Note that, in database terms, a decomposable $\wedge$-gate g is a Cartesian product: if we see the Boolean function computed by each input g_i of g as a relation R_i over attributes $\mathsf{var}(g_i)$ and domain $\{0,1\}$, then the relation computed by g is $R_1 \times R_2$. We revisit this in Section 6.

DNNF circuits clearly generalize DNF formulas, while retaining some of their tractability. Indeed, we can solve SAT on a DNNF circuit C in time $O(|C|)$, and enumerate the satisfying assignments with a delay of $O(n|C|)$ where $n = |X|$. DNNFs are also closed under partial evaluation, called *conditioning* in knowledge compilation [62]: given a DNNF circuit C on variables X computing f and a partial valuation ν on variables $Y \subseteq X$, we can build in time $O(|C|)$ a DNNF computing $f|_\nu$. Indeed, simply replace each node labeled with a variable $x \in Y$ by the constant $\nu(x)$: this does not affect decomposability.

3.2 Restrictions on DNNF Circuits

We now introduce additional restrictions to make more tasks tractable. Figure 2 and Table 1 summarize the circuit classes studied and the complexity of tasks.

Determinism. Unlike enumeration, aggregation tasks such as #SAT are not tractable on DNNFs, as they are already hard on DNFs. Intuitively, #SAT is hard because $\#f_1$ and $\#f_2$ does not give $\#(f_1 \vee f_2)$: indeed, f_1 and f_2 may be sharing some of their satisfying valuations. *Determinism* [59] is a way to make such tasks tractable. In a Boolean circuit, a $\vee$-gate g is said to be *deterministic* if for every pair of distinct inputs g_1, g_2 of g, the function $f_{g_1} \wedge f_{g_2}$ is not satisfiable. In other words, if ν is a satisfying valuation of f_g, then there is exactly one input g_i of g such that $\nu|_{\mathsf{var}(g_i)}$ is a satisfying valuation of g_i. A *deterministic Decomposable Negation Normal Form* (d-DNNF) circuit is a DNNF where every $\vee$-gate is deterministic. The DNNF in Figure 1 is *not* a d-DNNF.

Determinism makes it possible to count satisfying valuations bottom-up in time $O(|C|)$ (if arithmetic operations take constant time). Indeed, we can compute bottom-up in C the number of satisfying valuations of each gate: we have $\#(f_1 \wedge f_2) = \#f_1 \times \#f_2$ thanks to decomposability, and $\#(f_1 \vee f_2) = \#f_1 + \#f_2$ thanks to determinism. Determinism also ensures the tractability of WMC for weights in any semiring $(K, \oplus, \otimes)$ [94], using only $O(|C|)$ semiring operations, provided that the circuit is *smooth*, i.e., we have $\mathsf{var}(g') = \mathsf{var}(g)$ for every input g' of a $\vee$-gate g. Otherwise, an $|\mathsf{var}(C)|$ factor is usually needed to make the circuit smooth, which can sometimes be improved [124]. Further, when solving WMC with weights defined in a ring, it suffices to impose determinism and decomposability without NNF. This leads to *d-Ds* [107].

Determinism can also help for enumeration tasks: on a d-DNNF C, we can efficiently enumerate satisfying valuations in increasing order of a semiring weighting of the literals [7, 36], or uniformly sample satisfying valuations in $O(\mathrm{depth}(C) \times |\mathsf{var}(C)|)$ after a preprocessing in $O(|C|)$ [123].

Decision. One limitation of determinism is that it is a semantic property that may be hard to check on arbitrary circuits. Indeed, given a DNNF C and a $\vee$-gate g of C, it is coNP-hard to check whether g is deterministic. This is not always a problem as circuits are sometimes deterministic by construction; but sometimes we prefer to enforce a sufficient condition for determinism, called *decision gates*, which is easy to check syntactically. A decision gate intuitively tests the value of one variable $x \in X$ before proceeding further. More precisely, a $\vee$-gate g is a *decision gate* on x if the inputs of g are exactly two $\wedge$-gates g_0, g_1, where g_0 has one input labeled by $\neg x$ and g_1 has one input labeled by x. In other words, g is of the form $(\neg x \wedge g_0) \vee (x \wedge g_1)$; note that the literals $\neg x$ and x ensure that g is deterministic. For instance, the leftmost $\vee$-gate of Figure 1 is a decision gate on x_3. A *decision DNNF (dec-DNNF)* is a DNNF where every $\vee$-gate is a decision gate. All dec-DNNFs are d-DNNFs, and their depth can always be reduced to $O(|X|)$ in linear time.

A circuit that only has decision gates is often called a *binary decision diagram* or a *branching program*: this includes in particular *Ordered Binary Decision Diagrams* (OBDDs) [43]. OBDDs can be seen as DNNFs having only decision gates and having an order $<$ on X such that decision gates testing a variable x_i must come before gates testing variables x_j whenever $x_i < x_j$. OBDDs without this order requirement but with the analogue of decomposability are called *Free Binary Decision Diagrams* (FBDDs). See [6] for a recent survey on the relationships between circuit classes and diagram classes.

Structuredness. Structuredness [116] is a generalization of orders in OBDD tailored for DNNFs. For a variable set X, a *variable tree* or *v-tree* for X is a full binary tree whose leaves are in one-to-one correspondence with X. Given a DNNF circuit C on variables X and a v-tree T for X, we say that a $\wedge$-gate g in C *respects* T if g has exactly two inputs g_1, g_2 and if there is a node t of T with inputs t_1, t_2 such that $\mathsf{var}(g_1) \subseteq \mathsf{var}(t_1)$ and $\mathsf{var}(g_2) \subseteq \mathsf{var}(t_2)$ where $\mathsf{var}(t_i)$ is the set of variables in the leaves of the subtree of T rooted at t_i. A DNNF C *respects* T if every $\wedge$-gate of C does. We call C *structured* (denoted as SDNNF) if it respects some v-tree.

In essence, structuredness restricts how $\wedge$-gates are allowed to split the variables. It naturally appears in many algorithms building DNNFs from other Boolean function representations, e.g., building a DNNF from a bounded-treewidth circuit [14, 38]. Structuredness unlocks two new results: first, there is an FPRAS to approximately count the satisfying valuations of (non-deterministic) SDNNFs [23]. Second, enumerating the satisfying valuations of a d-SDNNF C can be done with preprocessing $O(|C|)$ and output-linear delay $O(|X|)$ [8].

Table 1: Tractable tasks for circuit classes. The input circuit C is of size s, depth d, and n variables; k is the number of solutions to output (for Enum and Sampling).

Task	**Circuit class**	**Complexity**	**Ref.**
WMC	d-DNNF	$O(ns)$	[94]
WMC in ring	d-D	$O(ns)$	[107]
ApproxWMC	DNF	FPRAS	[86, 127]
ApproxWMC	SDNNF	FPRAS	[23]
Enum	d-SDNNF	$O(s+nk)$	[8]
Sampling	d-DNNF	$O(s+dnk)$	[123]

3.3 Tools

An attractive feature of knowledge compilation is that many of its algorithms have implementations (at least experimental ones). We now survey these practical tools.

There are two main families of knowledge compilation tools. First, *top-down* tools are based on a generalization of the DPLL algorithm, known as exhaustive DPLL [24]. It is based on a recursive procedure originally devised for solving #SAT but which implicitly compiles into dec-DNNF formulas [77]. It compiles a CNF formula ϕ as follows: pick some variable x, recursively compile a circuit with gates g_0 computing $\phi[x := 0]$ and g_1 computing $\phi[x := 1]$, and add a decision variable g on x connected to g_0 and g_1. When $\phi = \phi_1 \wedge \phi_2$ with ϕ_1, ϕ_2 on disjoint variables, the algorithm compiles them independently and connects the two circuits with a decomposable $\wedge$-gate. Efficient algorithms rely on two main ingredients: (1) a *caching mechanism* that remembers previously-computed formulas to reuse them elsewhere in the circuit; and (2) a *heuristic* to choose variables to branch on, e.g., to break down the formula into smaller connected components. The knowledge compiler d4 [97, 98] implements this algorithm, trying, e.g., to find balanced cutsets in the formula and performing oracle calls to SAT solvers to cut unsatisfiable branches. The #SAT solver SharpSat-TD [95, 96] uses a heuristic that is guided by a tree decomposition computed via the FlowCutter algorithm [74, 126]. It has recently been modified into a knowledge compiler [91, 92].

Second, *bottom-up knowledge compilers* take a CNF formula ϕ and build circuits computing each clause of ϕ, before combining them into a bigger circuit computing ϕ. This works if the target circuit class efficiently supports conjunction: combine two circuits C_1, C_2 into a circuit computing $C_1 \wedge C_2$. Libraries such as CuDD [125] use this approach to compile CNFs to OBDDs. Further, the knowledge compiler SDD [58, 61] compiles into a subclass of structured d-DNNFs: these, unlike OBDDs [39, 117], can efficiently handle any bounded-treewidth CNF formula.

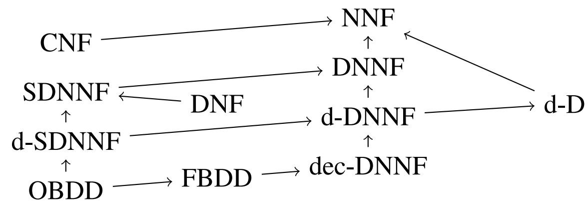


Figure 2: Inclusions between circuit classes. Arrows denote inclusion (i.e., linear-time transformations); they are all known to be strict in the sense that reverse arrows do not exist, except inclusions involving d-Ds which are not separated from d-DNNF [106] and only conditionally separated from NNF. Most results are in [6, 14].

4 Circuits for MSO Queries over Trees

We have seen in Section 2 how database tasks can be expressed in terms of Boolean functions, and seen in Section 3 how such functions can be represented as tractable circuits. We now start surveying how circuit-based methods can be used to solve database tasks. We start in this section by queries in *monadic second-order logic* (MSO) over tree-shaped data. This covers in particular the evaluation of word automata over textual documents, including the so-called *document spanners* [68]; and the evaluation of MSO queries over bounded-treewidth data via Courcelle's theorem [55]. We first give brief definitions of this setting, then study PQE for Boolean MSO queries and enumeration tasks for MSO queries with free variables. Throughout this section we adopt the *data complexity* perspective, i.e., the MSO query is always fixed, and the complexity is always a function of the input instance. We will study aggregation tasks (specifically, PQE) and enumeration tasks for MSO queries over trees. In both settings, the results will proceed by constructing tractable circuits to represent the Boolean provenance (for PQE) or the answer function (for enumeration).

Preliminaries. We consider queries over Σ-*trees*, or simply *trees*, which consist of nodes labeled with a fixed alphabet Σ. We assume trees to be rooted, ordered, binary, and full. The queries that we run over trees are expressed in MSO: this language extends first-order logic with quantification over sets, on a signature where we can test the label of tree nodes and the child and parent relationship between tree nodes. For example, we can express in MSO that there are two incomparable nodes with a certain label, or that nodes with a certain label are totally ordered by the descendant relation. An MSO query Q may be Boolean, in which case it can equivalently be expressed as a *tree automaton*; or it may have free variables. We can always assume without loss of generality that each free variable X is second-order, because Q can assert if necessary that X must be a singleton.

Aggregation tasks. We start with aggregation tasks for Boolean MSO queries Q, specifically probabilistic query evaluation (PQE). The PQE problem for Q asks for the probability that Q is satisfied on an input *probabilistic tree*, as we will define shortly. From there, by [12], using Courcelle's theorem [55], these results generalize to PQE over tuple-independent databases of *bounded treewidth*; and bounded treewidth is in some sense the most general condition that ensures tractability [13, 18].

Formally, to define probabilistic trees, we distinguish a *default label* $e \in \Sigma$; a probabilistic tree then consists of a Σ-tree T with a function π giving a probability $\pi(n)$ to each tree node n of T. The semantics is that $\mathscr{T} = (T, \pi)$ represents a probability distribution on possible worlds which are Σ-trees with same skeleton as T: each node n either keeps its label in T with probability $\pi(n)$, or "reverts" to the default label e with probability $1 - \pi(n)$, all these choices being independent. PQE for Q on $\mathscr{T}$ then asks for the total probability of the possible worlds of $\mathscr{T}$ that satisfy Q.

Following earlier results on probabilistic XML [54], it is then known [12] that PQE for Q can be computed in PTIME in the input $\mathscr{T}$. Specifically, we can define as in Section 2 the *provenance* of Q on T as the Boolean function defined on the nodes of T that maps each valuation to 0 or 1 depending on whether the corresponding possible world satisfies Q. One can then show, representing Q as a tree automaton A, that the provenance can be computed in time $O(|A| \times |T|)$ as an SDNNF whose v-tree follows T. There is also a correspondence between properties of A and of C, e.g., if A is unambiguous then C is a d-SDNNF [6]. The tractability of WMC for d-DNNF then implies that PQE for any fixed MSO query on probabilistic trees can be solved in PTIME.

Enumeration tasks. We now study MSO queries with free second-order variables $Q(X_1, \ldots, X_k)$, again running over trees. We first consider the task of enumerating query answers, again in data complexity. It is known that, in this setting, the answers to Q on an input tree T can be enumerated with preprocessing in $O(|T|)$, and with delay which is output-linear, i.e., which depends only on the size of each produced answer. This was shown first by Bagan [25], then by Kazana and Segoufin (only for free first-order variables) [88]. This result can be recaptured via knowledge compilation: a variant of provenance computation allows us to obtain d-SDNNF representations of the answer functions of MSO queries [8]. We can then enumerate the results of the query with linear preprocessing and output-linear delay with an algorithm to enumerate the satisfying valuations for this circuit class [8].

Similar results have also been shown for *document spanners*, which essentially amounts to evaluating MSO queries specified by automata over words, in particular via algorithms that also ensure tractability in the input automaton [10, 11]. In more expressive settings, the work of Muñoz and Riveros [108, 109] uses so-called *enumerable compact sets*, which resemble d-DNNF circuits, to achieve efficient enumeration on *nested documents* and *SLP-compressed documents*; and these are also used in [15] to enumerate the results of *annotation grammars*.

Enumeration algorithms for MSO over words and trees have been further extended to *ranked enumeration*: first by Bourhis et al. [37] on words, with weights defined by MSO cost functions; then on trees [7], with weights defined on partial assignments by so-called subset-monotone ranking functions. The latter work explicitly uses the circuit approach via smooth multivalued d-DNNF circuits.

We last mention the *incremental maintenance* of enumeration structures for MSO queries on trees. The point of such structures is to enumerate the answers of MSO queries while supporting updates to the underlying data; we want to handle updates (and restart the enumeration) without re-running the preprocessing phase from scratch. Note that this generalizes the task of incrementally maintaining Boolean MSO queries [27]. In this setting, for *relabeling* updates to the underlying tree, the best known bounds are obtained via d-DNNF representations of answer functions [9], improving on the work by Losemann and Martens [101]. Specifically, [9] shows that enumeration with linear preprocessing and output-linear delay can be extended to support relabeling updates in logarithmic time; tractability in the automaton is also possible [10].

5 Aggregative Tasks for CQs and UCQs

Having shown the uses of circuits for MSO queries on trees, we now move to Boolean CQs and UCQs over arbitrary data. We focus on aggregation tasks and study enumeration tasks in Section 6. We first focus on *conjunctive queries* (CQs), which are existentially quantified conjunctions of atoms; and on *unions of conjunctive queries* (UCQs). We show how circuits can be used for *probabilistic query evaluation* (PQE) on tuple-independent databases (TIDs), and its special case *uniform reliability* (UR). We first concentrate on *exact* (i.e., non-approximate) PQE, and we study it in data complexity, i.e., for fixed queries Q: first for CQs under a *self-join-freeness* assumption, then for UCQs. We then move to approximate PQE, and to *combined complexity* where both Q and the TID are given as input. Last, we cover Shapley values as another aggregation task.

Exact PQE for self-join-free CQs. To study PQE, one first class of queries to consider are the so-called *self-join-free CQs* (SJFCQs). A CQ is *self-join-free* if each relation occurs only once, i.e., there are no two atoms with the same symbol. For such queries, a dichotomy on PQE was shown by Dalvi and Suciu [56]: a SJFCQ is either *hierarchical*, in which case PQE is in polynomial-time data complexity; or it is *non-hierarchical*, in which case PQE is #P-hard in data complexity (and so is UR [17]). The tractability of hierarchical SJFCQs can be explained via tractable circuit representations of the Boolean provenance, which is called the *intensional* approach to PQE. More specifically, for hierarchical SJFCQs, the provenance can be computed in polynomial-time as an OBDD, in fact even as a read-once formula [111].

Exact PQE for UCQs. Following this dichotomy on PQE for SJFCQs, Dalvi and Suciu have shown a far more general dichotomy on UCQs: the PQE problem enjoys PTIME data complexity for some UCQs (called *safe*), and for all others (the *unsafe* UCQs) there is a #P-hardness result for PQE [57], indeed even for UR for most unsafe UCQs [89]. Ten years later, understanding this dichotomy in terms of tractable circuits is still an open research problem. Indeed, the algorithm of [57] follows the so-called *extensional* approach for PQE and directly computes the probability of the query. It does not follow the *intensional approach* of going via provenance. The *intensional vs. extensional conjecture* [106] thus asks whether we can solve PQE for any safe UCQ by computing a provenance representation in a tractable circuit class and invoking WMC on that class.

The intensional vs. extensional conjecture was investigated first by Jha and Suciu [80]: they characterize the strict subset of safe UCQs whose provenance can be expressed as read-once formulas (like for hierarchical SJFCQs), and also the larger strict subset, called *inversion-free UCQs*, for which we can build OBDDs in PTIME. It was then shown in [40] that safe UCQs that are not inversion-free do not admit polynomial-size provenance representations even as d-SDNNFs. Jha and Suciu [80] also give sufficient conditions on safe UCQs to admit polynomial-size FBDD provenance representations, but without a characterization. It was later shown in [28] that some safe UCQs admit no polynomial-size provenance representation as so-called DLDDs, implying the same for dec-DNNFs and FBDDs.

However, the intensional vs. extensional conjecture is still open for more expressive circuit classes with tractable WMC. It remains open whether the class of d-DNNFs can tractably represent the provenance of all safe UCQs (with [80] conjecturing that it does not). The question is also open for the more general class of d-Ds, which is in fact not yet separated from d-DNNFs [106]. The ability to compute polynomial d-Ds to represent the provenance of all safe UCQs currently hinges on the unproven *non-cancelling intersections conjecture* [20].

Approximate PQE for UCQs and combined complexity. Faced with the general intractability of PQE for unsafe UCQs, it is natural to settle for approximate PQE. Additive approximations can be obtained simply via Monte Carlo sampling [127], and multiplicative approximations can always be obtained through the intensional approach. Namely, for any fixed UCQ, we can represent its provenance as a monotone DNF in PTIME data complexity, and we can then solve approximate PQE by solving approximate weighted model counting (ApproxWMC) on the DNF via the Karp-Luby algorithm [86] (see Section 3), giving an FPRAS for the task.

This tractability of approximate PQE leads to the question of finding efficient algorithms in combined complexity, i.e., when the query is also given as input. In this light, Van Bremen and Meel [130] have studied the combined complexity of SJFCQs of *bounded hypertreewidth*, which are tractable for non-probabilistic query evaluation [72], and extended this tractability result to approximate PQE. We will explain in Section 6 how their algorithm can be understood via SDNNF circuits and via the FPRAS of [22] mentioned in Section 3. Note that [130] generally does not extend to CQs with self-joins [21]. We also note that provenance-based approaches have also been used for tractable combined algorithms for *exact PQE*, e.g., via β-acyclic positive DNFs [19, 41].

Shapley values. We can use circuits for other aggregation tasks than PQE: one important example is computing the *Shapley value* of facts in relational instances. This task has been shown to be tractable whenever we can compute a representation of the query provenance as a d-D [63], and the same was shown for the related notion of Banzhaf values [4]. As another example of an aggregation task, the computation of the Shapley value (and variants) has also been recently extended to the setting of probabilistic instances [85], also using d-D circuits.

Implementation. We last mention *ProvSQL* [122], as it is a concrete instantiation of the circuit-based approach to provenance [64]. ProvSQL is a module of the PostgreSQL relational database management system, which adds the possibility to track the provenance of query results as a circuit throughout query evaluation. These general-purpose circuits can then be used for various applications. In particular, they can be used for PQE via existing knowledge compilation tools for WMC (see Section 3.3), or for Shapley value computation [85].

6 Circuits to Represent Query Answers

We now move to a different perspective on circuits, which we will use in particular for enumeration tasks with CQs. We do not use answer functions for CQs, because such queries cannot easily express that a variable is assigned to *exactly one value*. For this reason, unlike previous sections, we will go beyond Boolean functions and Boolean circuits, and adopt a *multivalued* perspective: we will show how circuits can succinctly represent *relations*, in particular query answers.

Representing Relations as Circuits. Let $f\colon \{0,1\}^X \to \{0,1\}$ be a Boolean function. We can easily see $f^{-1}(1) \subseteq \{0,1\}^X$ as a relational table R, whose domain is $\{0,1\}$ and whose attributes (in the named perspective) is X; see, e.g., the table right of Figure 1. What is more, we can see circuit representations of f as a factorized representation of R. More specifically, a DNNF C on variables X can be seen as a factorized way of building up a relation from elementary relations of the form $x=0$ or $x=1$. Then, $\wedge$-gates are a natural join of their input relations (denoted by $\bowtie$), while $\vee$-gates are unions.

Unfortunately, the semantics of $\vee$-gates does not precisely correspond to the union operator of relational algebra. Indeed, the union operator applies to tables with the same attributes, whereas two Boolean functions f and g on different variable sets $X \neq Y$ can be disjoined as $f \vee g$, giving a function on variables $X \cup Y$. This issue does not arise with *smooth* circuits (see Section 3), but to interpret $\vee$ for general circuits we must extend the union operator. Formally, given two relations $R \subseteq D^X$ and $S \subseteq D^Y$, the *extended union* $R \mathbin{\bar\cup} S$ of R and S is the relation on attributes $X \cup Y$ defined as $(R \times D^{Y \setminus X}) \cup (S \times D^{X \setminus Y})$[1]. If $X = Y$ then we clearly have $R \mathbin{\bar\cup} S = R \cup S$.

We can now directly generalize NNFs and their variations to *relational circuits* on any finite domain: see Table 2 for a summary. Namely, a $\{\bar\cup, \bowtie\}$-circuit C on attributes X and domain D is a circuit whose internal gates are labeled by either $\bar\cup$ or $\bowtie$ and whose input gates are labeled by relations of the form x/d where $x \in X$ and $d \in D$. The *attributes* $\mathsf{attr}(g) \subseteq X$ of a gate g of C are the attributes labeling the inputs of C that have a directed path to g. Further, g computes a relation $\mathsf{rel}(g) \subseteq D^{\mathsf{attr}(g)}$ inductively defined from the relations computed by its inputs and from the label of g. The circuit C computes $\mathsf{rel}(C)$ which is $\mathsf{rel}(o)$ for o the output of C, again extended by allowing arbitrary values in D for the attributes of $X \setminus \mathsf{attr}(o)$. One can easily check that if C is a $\{\bar\cup, \bowtie\}$-circuit on domain $\{0,1\}$, then we can get an NNF computing $\mathsf{rel}(C)$ by renaming every input of the form $x/1$ by x and $x/0$ by $\neg x$ and replacing every $\bowtie$-gate by a $\wedge$-gate and every $\bar\cup$-gate by a $\vee$-gate.

Now, if a $\bowtie$-gate has inputs whose attributes are pairwise disjoint, it actually computes the Cartesian product of these inputs, and we can denote it by $\times$ and call it *decomposable*. Hence, a $\{\bar\cup, \times\}$-circuit is a $\{\bar\cup, \bowtie\}$-circuit where every $\bowtie$-gate is in fact computing a Cartesian product. Such *decomposable* circuits correspond to DNNFs

[1]Note that, unlike union, the extended union depends on the domain D. An alternative choice is to fix a default value d and define the union as $R \mathbin{\bar\cup'} S = (R \times \{d\}^{Y \setminus X}) \cup (S \times \{d\}^{X \setminus Y})$. When $D = \{0,1\}$, the choice of $d = 0$ corresponds to the *zero-suppressed semantics* studied for decision diagrams [105, 134].

Table 2: From Boolean circuits to relational circuits.

Boolean Circuits	**Relational Circuits**	
NNF	$\{\bar\cup, \bowtie\}$	
DNNF	$\{\bar\cup, \times\}$	or $\{\cup, \times\}$ if smooth
d-DNNF	$\{\bar\uplus, \times\}$	or $\{\uplus, \times\}$ if smooth
dec-DNNF	$\{\mathsf{dec}, \times\}$	

when restricted to the Boolean domain. Similarly, if a $\bar\cup$-gate g has inputs $g_1, \ldots, g_k$ such that $\mathsf{attr}(g_1) = \cdots = \mathsf{attr}(g_k)$, it actually computes a union and we denote it by $\cup$. This gives $\{\cup, \times\}$-circuits, which correspond to smooth DNNFs. We can also generalize structuredness: a $\times$-gate g with two inputs g_1, g_2 *respects* a vtree T on X if there is a node t in T such that $\mathsf{attr}(g_1) \subseteq \mathsf{attr}(t_1)$ and $\mathsf{attr}(g_2) \subseteq \mathsf{attr}(t_2)$ where t_1, t_2 are the children of t in T and each $\mathsf{attr}(t_i)$ is the set of attributes labeling the leaves of the subtree of T rooted at t_i.

We can also generalize determinism: denoting disjoint union by $\uplus$, we have $\{\uplus, \times\}$-circuits which correspond to smooth d-DNNFs. Again, the disjointness of $\cup$-gates is a semantic property that may be intractable to verify. Like in Section 3, we can enforce it by a sufficient syntactic condition: a $\uplus$-gate is a *decision gate* if it is of the form $\biguplus_{d \in D'} [x/d] \times g_d$ for a subset $D' \subseteq D$, where the $[x/d]$ are input gates. A $\{\mathsf{dec}, \times\}$*-circuit* is a $\{\cup, \times\}$-circuit where every $\cup$-gate is a decision gate.

Factorized Databases. The notion of relational circuits that we present here is related to *factorized databases* as introduced by Olteanu and Závodný [113, 114]. Their *d-representations* correspond to $\{\cup, \times\}$-circuits, and their *f-representations* further require that the circuit is actually a tree, i.e., there is no sharing. Further, many factorized database algorithms follow a tree over the database attributes, so that they build structured $\{\mathsf{dec}, \times\}$-circuits. The link between factorized databases and knowledge compilation is also explored in [110].

Building and Using Circuits. We now turn to the use of relational circuits to solve enumeration tasks for CQs. Deciding whether a CQ has at least one answer is NP-hard in combined complexity [52] and the same reduction shows that counting query answers is #P-hard. Thus, a fruitful line of research studies how to design algorithms that use the query structure to enumerate or count CQ answers efficiently. For example, Yannakakis observed in [135] that one can test with linear data complexity whether an *acyclic* CQ has at least one answer: further, his algorithm is also tractable in the query size. This result has been generalized in many ways: to constant-delay enumeration for the so-called *free-connex acyclic CQs* by Bagan, Durand and Grandjean [26]; to counting for projection-free CQs in [115] and for arbitrary CQs in [67]; to direct access in [42, 51] for some lexicographical orders; or to semiring-based aggregation queries in [3, 81]. Relational circuits offer a unifying view of all these results and more, thanks to a claim initially shown in the setting of factorized databases:

THEOREM 6.1 ([114]). *Given a free-connex acyclic conjunctive query Q and a database $\mathbb{D}$, we can build a structured $\{\mathsf{dec}, \times\}$-circuit in time $O(\text{poly}\,|Q| \cdot |\mathbb{D}|)$ computing $Q(\mathbb{D})$. The circuit is of size $O(\text{poly}\,|Q| \cdot |\mathbb{D}|)$.*

This result is actually also shown for non-acyclic CQs in [114]. In this case, the transformation will not be linear wrt $|\mathbb{D}|$ anymore but of the form $O(|\mathbb{D}|^k)$ for k a constant that depends only on the structure of Q. When Q is in the class of free-connex acyclic CQs, we have $k = 1$. Otherwise, k intuitively measures how far Q is from being free-connex acyclic [69]; if Q is projection-free it is at most, e.g., the hypertreewidth of Q [72].

There are two main techniques to prove Theorem 6.1, that is, to build circuits of linear size for free-connex acyclic conjunctive queries. The first one can be seen as the trace of Yannakakis's algorithm [135] and is intuitively the approach taken in [114]. This result decides whether $Q(\mathbb{D})$ is empty by alternatively joining and projecting the atoms of Q in an order that can be obtained from the fact that Q is free-connex acyclic (see [99, Chapter 6]). This ensures that all intermediate results have size $O(|\mathbb{D}|)$. The trace of the joins and projections done during the evaluation corresponds to a $\{\mathsf{dec}, \times\}$-circuit of size $O(\text{poly}\,|Q| \cdot |\mathbb{D}|)$ computing $Q(\mathbb{D})$.

Another approach to build a circuit for $Q(\mathbb{D})$ is inspired by the DPLL algorithm from top-down knowledge compilers [24]. The circuit is built by following a static order on the attributes and recursively compiling $Q[x_1 = d]$ for every value $d \in D$, which corresponds to building a decision gate on x_1. If the query can be written as $Q_1 \wedge Q_2$ at some recursive call with $\mathsf{attr}(Q_1) \cap \mathsf{attr}(Q_2) = \emptyset$, then Q_1 and Q_2 are compiled independently and their subcircuits are joined by a $\times$-gate. By choosing an order that witnesses the free-connex acyclicity of Q, and adding a caching mechanism to remember previously computed queries, one can show that this algorithm also builds a $\{\mathsf{dec}, \times\}$-circuit of size $O(\text{poly}\,|Q| \cdot |\mathbb{D}|)$ computing $Q(\mathbb{D})$. If another order is chosen, the output of the algorithm is still a circuit but its size may not be linear in $|\mathbb{D}|$. The detailed algorithm is given in [47] with order-dependent complexity bounds ; it is also exemplified in [46].

The tractable tasks from Table 1 can straightforwardly be generalized to $\{\mathsf{dec}, \times\}$-circuits. In particular, if n is the number of attributes of C, $|\mathsf{rel}(C)|$ can be computed in time $O(n|C|)$ and $\mathsf{rel}(C)$ can be enumerated with delay $O(n)$: this fact was observed in [114] and used to motivate factorized databases. In particular, in data complexity, the value n is a constant, hence we recover the result from [26] that $Q(\mathbb{D})$ can be enumerated with constant-delay data complexity (more precisely $O(\text{poly}\,|Q|)$) after a $O(\text{poly}\,|Q| \cdot |\mathbb{D}|)$ preprocessing when Q is free-connex acyclic, the preprocessing being here the construction of the circuit from Theorem 6.1. We note that, conditionally, for self-join-free conjunctive queries, it is known that only free-connex acyclic queries enjoy this tractability guarantee [26]. This conditional lower bound does not extend to CQs with self-joins [50], nor to UCQs [42].

More interestingly, the DPLL-based construction naturally produces $\{\mathsf{dec}, \times\}$-circuits of a very particular form: the circuit is built in a way where there exists an order $\{x_1, \ldots, x_n\}$ on the attributes of Q such that, for every decision gate g on x_i with input $g_1, \ldots, g_k$, we have $\mathsf{attr}(g_p) \subseteq \{x_{i+1}, \ldots, x_n\}$. This makes it possible, after a preprocessing time of $O(|C|)$, to do direct access for the lexicographical order induced by $x_1, \ldots, x_n$ in time $O(n \cdot \log |\mathbb{D}|)$ [47]. Combined with Theorem 6.1, it is an alternative proof of the results of [51]. The approach also generalizes to non-acyclic queries, matching results from [42], and to so-called *signed conjunctive queries*.

From Relational Circuits to Provenance. Representing answers of CQ with relational circuits is close to computing provenance, and has actually been one of the original motivations [112]. We make this intuition formal by explaining how relational circuits can be used to compute the Boolean provenance of queries.

Consider a signature σ and a projection-free CQ Q (that is, every variable of Q is free). We first define a signature σ^*: for each relation R in σ we add a relation R^* in σ^* with arity increased by 1. We then denote by Q^* a query on σ^* obtained as follows: for each atom $A = R(\mathbf{x})$ of Q, add an atom $R^*(\mathbf{x}, y_A)$ to Q', where y_A is a fresh attribute for the atom A. Given an instance $\mathbb{D}$ on σ, we also let $\mathbb{D}^*$ be the instance on σ^* where each symbol R^* is interpreted as R in $\mathbb{D}$ but each fact F is augmented with a fresh value a_F. It is easy to check that $Q(\mathbb{D})$ and $Q^*(\mathbb{D}^*)$ are isomorphic up to the added identifiers. Moreover, if Q is acyclic, then Q^* is also acyclic (they even have the same hypertreewidth).

Now, let C be a $\{\cup, \times\}$-circuit computing $Q^*(\mathbb{D}^*)$. Its inputs are of two kinds: relations of the form x/a for x an attribute of Q and a a value of $\mathbb{D}$, and relations of the form y_A/a_F where a_F is a fresh value added for the tuple F. We modify C in two steps. First, let C' be the circuit obtained from C by replacing each input y_A/a_F of the second kind by a variable X_F on domain $\{0,1\}$. Note that C' is generally not decomposable, but decomposability is preserved if Q is self-join-free: for each fact F of $\mathbb{D}$, the variable X_F of C' then corresponds to only one input label of C, namely inputs of the form y_A/a_F where A is the one atom of Q for the relation used by F. In any case, a Boolean valuation ν of the variables $\{X_F \mid F \in \mathbb{D}\}$ can be naturally identified as a subinstance $\mathbb{D}_\nu \subseteq \mathbb{D}$ containing the facts F of $\mathbb{D}$ such that $\nu(X_F) = 1$, and we can then easily see that C' computes $\{\tau \times \nu \mid \nu \in 2^{\mathbb{D}}, \tau \in Q(\mathbb{D}_\nu)\}$. Second, let C'' be the circuit obtained from C' by existentially projecting away every variable of the first kind, i.e., the variables that are not of the form X_F. If C' is decomposable then C'' also is; however C'' is generally not deterministic. Hence, C'' is a $\{\cup, \times\}$-circuit computing $\{\nu \mid \nu \in 2^{\mathbb{D}}$ s.t. $Q(\mathbb{D}_\nu) \neq \emptyset\}$. That is, it computes the Boolean provenance of the Boolean query obtained by existentially quantifying Q. Further, C'' has size less than $|C|$. Last, if Q is self-join-free then C'' is a DNNF, and C'' is structured if C was.

Among other things, this result allows us to recover the result of Van Bremen and Meel [130] mentioned in Section 5. Recall that they give a combined FPRAS for approximate PQE with self-join-free CQs Q of bounded hypertreewidth. This is straightforward in the data complexity sense (see Section 5), so the interesting point is the tractability in combined complexity. Applying Theorem 6.1 (extended for queries of hypertreewidth k) to the query Q^* defined as above, we get a structured $\{\mathsf{dec}, \times\}$-circuit that computes $Q^*(\mathbb{D}^*)$ and which has size $O(\text{poly}\,|Q| \cdot |\mathbb{D}|^k)$. As Q is self-join-free, our transformation above gives a SDNNF C computing the Boolean provenance of Q of size $O(\text{poly}\,|Q| \cdot |\mathbb{D}|^k)$. We can then use the FPRAS from [23] on C to solve approximate PQE for Q on $\mathbb{D}$, which concludes.

7 Perspectives

We have seen how results from database theory can often be obtained via tractable circuit classes, or how some existing proofs can be rephrased in this vocabulary. We have focused on two main kinds of tasks: *aggregation tasks*, including PQE and Shapley value computation; and *enumeration tasks*, including ranked enumeration and direct access. We have focused on the settings of MSO queries and of CQs and UCQs. We believe that this survey illustrates the versatility of circuit techniques, which can sometimes offer a unified view on the tractability of multiple tasks in different areas. Circuits also serve as a convenient intermediate language between database algorithms (which deal with the query and data) and task-specific algorithms (e.g., satisfiability, counting, etc.). This view makes it possible to give modular algorithms and to leverage existing solver implementations.

We have focused on circuit classes from knowledge compilation, but we point out that circuits have been used in other interesting ways in database theory, for example, arithmetic circuits for semiring provenance [64], circuits for secure aggregation on data shared by different parties [131], or circuits for efficient parallel evaluation of queries [90, 132]. Similarly, we have chosen to focus on query evaluation applications in databases, but tractable circuits are also used in closely related topics such as CSPs [29, 48], homomorphism representations [32], or the computation of SHAP-scores [22].

We believe that tractable circuits still have a lot to bring to database theory, and vice-versa; we conclude by highlighting some directions for future work. We see two main research axes: studying how far we can bring circuit methods in established contexts, and introducing them in new contexts. For the first axis, the *intensional-extensional conjecture* for PQE of UCQs is a clearly identified setting where we do not know how far circuits can be taken to recapture existing results [20, 106]. A similar research direction would be the *incremental maintenance of enumeration structures* for MSO on trees: while circuit methods handle substitution updates in logarithmic time [9], better complexities are possible, at least in the case of Boolean queries on words [16]: it is unclear for now whether such results can be explained in terms of circuits. It is also unclear which incremental results can be recaptured by circuits in the setting of incremental PQE [30], incremental enumeration for CQs [31, 83, 84] or CQs with aggregates [78, 82], or more general results in algorithms on dynamic data [75]. Moreover, there remain tractability results for aggregate tasks in the database and CSP literature that have not been directly explained from a circuit perspective but which use similar counting techniques, for example results on counting the number of answers of UCQs [53, 70] or complex aggregation over semiring-annotated data [3, 81].

For the second axis, we believe that circuits could be applied to entirely new areas. One possibility is query enumeration for First-Order logic (FO), e.g., over bounded-degree structures [66, 87]. It is not known if such results, and their subsequent extensions [119, 121], can be captured in terms of circuits. A second setting in which circuits could be relevant is *database repairs*, in particular counting subset repairs [44, 45, 93], enumerating them [93], and more generally representing them in a factorized way. One last question is to connect circuits to the study of efficient algorithms for *Datalog evaluation* including provenance computation in various semirings [2]. Can tractable algorithms for these tasks be connected to algorithms producing tractable circuit representations? Can fine-grained complexity lower bounds be connected to circuit lower bounds?

Acknowledgements. This work was supported by project ANR KCODA, ANR-20-CE48-0004, and by project ANR CQFD, ANR-18-CE23-0003-02. We are grateful to Tim van Bremen and Benny Kimelfeld for their insightful feedback.

8 References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases*, volume 8. 1995. Chapter 19.

[2] M. Abo Khamis, H. Q. Ngo, R. Pichler, D. Suciu, and Y. R. Wang. Convergence of Datalog over (pre-) semirings. *JACM*, 71(2), 2024.

[3] M. Abo Khamis, H. Q. Ngo, and A. Rudra. FAQ: Questions asked frequently. In *PODS*, 2016.

[4] O. Abramovich, D. Deutch, N. Frost, A. Kara, and D. Olteanu. Banzhaf values for facts in query answering. *CoRR*, abs/2308.05588, 2023.

[5] S. R. Allen, L. Hellerstein, D. Kletenik, and T. Ünlüyurt. Evaluation of monotone DNF formulas. *Algorithmica*, 77, 2017.

[6] A. Amarilli, M. Arenas, Y. Choi, M. Monet, G. V. den Broeck, and B. Wang. A circus of circuits: Connections between decision diagrams, circuits, and automata. *CoRR*, abs/2404.09674, 2024.

[7] A. Amarilli, P. Bourhis, F. Capelli, and M. Monet. Ranked enumeration for MSO on trees via knowledge compilation. In *ICDT*, 2024.

[8] A. Amarilli, P. Bourhis, L. Jachiet, and S. Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, 2017.

[9] A. Amarilli, P. Bourhis, and S. Mengel. Enumeration on trees under relabelings. In *ICDT*, 2018.

[10] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, 2019.

[11] A. Amarilli, P. Bourhis, S. Mengel, and M. Niewerth. Enumeration on trees with tractable combined complexity and efficient updates. In *PODS*, 2019.

[12] A. Amarilli, P. Bourhis, and P. Senellart. Provenance circuits for trees and treelike instances. In *ICALP*, 2015.

[13] A. Amarilli, P. Bourhis, and P. Senellart. Tractable lineages on treelike instances: Limits and extensions. In *PODS*, 2016.

[14] A. Amarilli, F. Capelli, M. Monet, and P. Senellart. Connecting knowledge compilation classes and width parameters. *ToCS*, (2020), 2020.

[15] A. Amarilli, L. Jachiet, M. Muñoz, and C. Riveros. Efficient enumeration for annotated grammars. In *PODS*, 2022.

[16] A. Amarilli, L. Jachiet, and C. Paperman. Dynamic membership for regular languages. In *ICALP*, 2021.

[17] A. Amarilli and B. Kimelfeld. Uniform reliability of self-join-free conjunctive queries. *LMCS*, 2022.

[18] A. Amarilli and M. Monet. Weighted counting of matchings in unbounded-treewidth graph families. In *MFCS*, 2022.

[19] A. Amarilli, M. Monet, and P. Senellart. Conjunctive queries on probabilistic graphs: Combined complexity. In *PODS*, 2017.

[20] A. Amarilli, M. Monet, and D. Suciu. The non-cancelling intersections conjecture. *CoRR*, abs/2401.16210, 2024.

[21] A. Amarilli, T. van Bremen, and K. S. Meel. Conjunctive queries on probabilistic graphs: The limits of approximability. In *ICDT*, 2024.

[22] M. Arenas, P. Barceló, L. Bertossi, and M. Monet. On the complexity of SHAP-score-based explanations: Tractability via knowledge compilation and non-approximability results. *Journal of Machine Learning Research*, 24(63), 2023.

[23] M. Arenas, L. A. Croquevielle, R. Jayaram, and C. Riveros. When is approximate counting for conjunctive queries tractable? In *STOC*, 2021.

[24] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *FOCS*, 2003.

[25] G. Bagan. MSO queries on tree decomposable structures are computable with linear delay. In *CSL*, 2006.

[26] G. Bagan, A. Durand, and É. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, 2007.

[27] A. Balmin, Y. Papakonstantinou, and V. Vianu. Incremental validation of XML documents. *TODS*, 29(4), 2004.

[28] P. Beame, J. Li, S. Roy, and D. Suciu. Exact model counting of query expressions: Limitations of propositional methods. *TODS*, 42(1), 2017.

[29] C. Berkholz, S. Mengel, and H. Wilhelm. A characterization of efficiently compilable constraint languages. In *STACS*, 2024.

[30] C. Berkholz and M. Merz. Probabilistic databases under updates: Boolean query evaluation and ranked enumeration. In *PODS*, 2021.

[31] C. Berkholz and N. Schweikardt. Constant delay enumeration with FPT-preprocessing for conjunctive queries of bounded submodular width. In *MFCS*, 2019.

[32] C. Berkholz and H. Vinall-Smeeth. A dichotomy for succinct representations of homomorphisms. In *ICALP*, 2023.

[33] L. Bertossi. Consistent query answering in databases. *ACM SIGMOD Record*, 35(2), 2006.

[34] L. Bertossi. Database repairs and consistent query answering: Origins and further developments. In *PODS*, 2019.

[35] L. Bertossi, B. Kimelfeld, E. Livshits, and M. Monet. The Shapley value in database management. *ACM SIGMOD Record*, 52(2), 2023.

[36] P. Bourhis, L. Duchien, J. Dusart, E. Lonca, P. Marquis, and C. Quinton. Pseudo polynomial-time top-k algorithms for d-DNNF circuits. *CoRR*, abs/2202.05938, 2022.

[37] P. Bourhis, A. Grez, L. Jachiet, and C. Riveros. Ranked enumeration of MSO logic on words. In *ICDT*, 2021.

[38] S. Bova, F. Capelli, S. Mengel, and F. Slivovsky. On compiling CNFs into structured deterministic DNNFs. In *SAT*, 2015.

[39] S. Bova and F. Slivovsky. On compiling structured CNFs to OBDDs. *TCS*, 61, 2017.

[40] S. Bova and S. Szeider. Circuit treewidth, sentential decision, and query compilation. In *PODS*, 2017.

[41] J. Brault-Baron, F. Capelli, and S. Mengel. Understanding model counting for β-acyclic CNF-formulas. In *STACS*, 2015.

[42] K. Bringmann, N. Carmeli, and S. Mengel. Tight fine-grained bounds for direct access on join queries. In *PODS*, 2022.

[43] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, 100(8), 1986.

[44] M. Calautti, M. Console, and A. Pieris. Counting database repairs under primary keys revisited. In *PODS*, 2019.

[45] M. Calautti, E. Livshits, A. Pieris, and M. Schneider. Counting database repairs entailing a query: The case of functional dependencies. In *PODS*, 2022.

[46] F. Capelli. An example of DPLL algorithm for factorized databases. `https://florent.capelli.me/cytoscape/dpll.html`. Accessed: 2024-05-06.

[47] F. Capelli and O. Irwin. Direct access for conjunctive queries with negations. In *ICDT*, 2024.

[48] C. Carbonnel, M. Romero, and S. Živný. Point-width and Max-CSPs. *TALG*, 16(4), 2020.

[49] N. Carmeli. Accessing answers to conjunctive queries with ideal time guarantees. In *DL*, 2023.

[50] N. Carmeli and L. Segoufin. Conjunctive queries with self-joins, towards a fine-grained complexity analysis. In *PODS*, 2023.

[51] N. Carmeli, S. Zeevi, C. Berkholz, A. Conte, B. Kimelfeld, and N. Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. *TODS*, 47(3), 2022.

[52] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, 1977.

[53] H. Chen and S. Mengel. Counting answers to existential positive queries: A complexity classification. In *SIGMOD*, 2016.

[54] S. Cohen, B. Kimelfeld, and Y. Sagiv. Running tree automata on probabilistic XML. In *PODS*, 2009.

[55] B. Courcelle. The monadic second-order logic of graphs. I. Recognizable sets of finite graphs. *Inf. Comput.*, 85(1), 1990.

[56] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *VLDBJ*, 16(4), 2007.

[57] N. Dalvi and D. Suciu. The dichotomy of probabilistic inference for unions of conjunctive queries. *JACM*, 59(6), 2013.

[58] A. Darwiche. SDD source. `http://reasoning.cs.ucla.edu/sdd/`. Accessed: 2024-05-06.
[59] A. Darwiche. On the tractable counting of theory models and its application to truth maintenance and belief revision. *Journal of Applied Non-Classical Logics*, 11(1-2), 2001.
[60] A. Darwiche. Decomposable negation normal form. *JACM*, 48(4), 2001.
[61] A. Darwiche. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, 2011.
[62] A. Darwiche and P. Marquis. A knowledge compilation map. *JAIR*, 17, 2002.
[63] D. Deutch, N. Frost, B. Kimelfeld, and M. Monet. Computing the Shapley value of facts in query answering. In *SIGMOD*, 2022.
[64] D. Deutch, T. Milo, S. Roy, and V. Tannen. Circuits for Datalog provenance. In *ICDT*, volume 3, 2014.
[65] O. Drien, M. Freiman, A. Amarilli, and Y. Amsterdamer. Query-guided resolution in uncertain databases. *SIGMOD*, 1(2), 2023.
[66] A. Durand and É. Grandjean. First-order queries on structures of bounded degree are computable with constant delay. *TOCL*, 8(4), 2007.
[67] A. Durand and S. Mengel. The complexity of weighted counting for acyclic conjunctive queries. *Journal of Computer and System Sciences*, 80(1), 2014.
[68] R. Fagin, B. Kimelfeld, F. Reiss, and S. Vansummeren. Document spanners: A formal approach to information extraction. *JACM*, 62(2), 2015.
[69] W. Fischl, G. Gottlob, and R. Pichler. General and fractional hypertree decompositions: Hard and easy cases. In *PODS*, 2018.
[70] J. Focke, L. A. Goldberg, M. Roth, and S. Živnỳ. Counting answers to unions of conjunctive queries: Natural tractability criteria and meta-complexity. *PACMMOD*, 2024.
[71] C. Freire, W. Gatterbauer, N. Immerman, and A. Meliou. The complexity of resilience and responsibility for self-join-free conjunctive queries. *PVLDB*, 9(3), 2015.
[72] G. Gottlob, N. Leone, F. Scarcello, et al. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64, 2002.
[73] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *PODS*, 2007.
[74] M. Hamann and B. Strasser. Graph bisection with Pareto optimization. *Journal of Experimental Algorithmics (JEA)*, 23, 2018.
[75] K. Hanauer, M. Henzinger, and C. Schulz. Recent advances in fully dynamic graph algorithms–A quick reference guide. *JEA*, 2022.
[76] X. Hu and S. Sintos. Finding smallest witnesses for conjunctive queries. In *ICDT*, 2024.
[77] J. Huang and A. Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *IJCAI*, 2005.
[78] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *SIGMOD*, 2017.
[79] T. Imieliński and W. Lipski Jr. Incomplete information in relational databases. *JACM*, 31(4), 1984.
[80] A. Jha and D. Suciu. Knowledge compilation meets database theory: Compiling queries to decision diagrams. *Theory of Computing Systems*, 52(3), 2013.
[81] M. R. Joglekar, R. Puttagunta, and C. Ré. AJAR: Aggregations and joins over annotated relations. In *PODS*, 2016.
[82] A. Kara, H. Q. Ngo, M. Nikolic, D. Olteanu, and H. Zhang. Maintaining triangle queries under updates. *TODS*, 45(3), 2020.
[83] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Trade-offs in static and dynamic evaluation of hierarchical queries. In *PODS*, 2020.
[84] A. Kara, M. Nikolic, D. Olteanu, and H. Zhang. Conjunctive queries with free access patterns under updates. In *ICDT*, 2023.
[85] P. Karmakar, M. Monet, P. Senellart, and S. Bressan. Expected Shapley-like scores of boolean functions: Complexity and applications to probabilistic databases. *PACMMOD*, 2024.
[86] R. M. Karp, M. Luby, and N. Madras. Monte-Carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3), 1989.
[87] W. Kazana and L. Segoufin. First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science*, 7, 2011.
[88] W. Kazana and L. Segoufin. Enumeration of monadic second-order queries on trees. *TOCL*, 14(4), 2013.
[89] B. Kenig and D. Suciu. A dichotomy for the generalized model counting problem for unions of conjunctive queries. In *PODS*, 2021.
[90] J. Keppeler, T. Schwentick, and C. Spinrath. Work-efficient query evaluation with PRAMs. *CoRR*, abs/2301.08178, 2023.
[91] R. Kiesel and T. Eiter. SharpSat-TD Github repository. `https://github.com/raki123/sharpsat-td`. Accessed: 2024-05-06.
[92] R. Kiesel and T. Eiter. Knowledge compilation and more with SharpSAT-TD. In *KR*, 2023.
[93] B. Kimelfeld, E. Livshits, and L. Peterfreund. Counting and enumerating preferred database repairs. *Theoretical Computer Science*, 837, 2020.
[94] A. Kimmig, G. Van den Broeck, and L. De Raedt. Algebraic model counting. *Journal of Applied Logic*, 22, 2017.
[95] T. Korhonen and M. Järvisalo. SharpSat-TD Github repository. `https://github.com/Laakeri/sharpsat-td`. Accessed: 2024-05-06.
[96] T. Korhonen and M. Järvisalo. Integrating tree decompositions into decision heuristics of propositional model counters. In *CP*, 2021.
[97] J. Lagniez and P. Marquis. An improved decision-DNNF compiler. In *IJCAI*, 2017.
[98] J.-M. Lagniez and P. Marquis. D4 Github repository. `https://github.com/crillab/d4v2`. Accessed: 2024-05-06.
[99] L. Libkin. *Elements of finite model theory*. 2004. ISBN: `978-3540212027`.
[100] E. Livshits, L. Bertossi, B. Kimelfeld, and M. Sebag. The Shapley value of tuples in query answering. *LMCS*, 17, 2021.
[101] K. Losemann and W. Martens. MSO queries on trees: Enumerating answers under updates. In *CSL–LICS*, 2014.
[102] N. Makhija and W. Gatterbauer. A unified approach for resilience and causal responsibility with integer linear programming (ILP) and LP relaxations. *SIGMOD*, 1(4), 2023.
[103] A. Meliou, W. Gatterbauer, and D. Suciu. Reverse data management. *PLVDB*, 4(12), 2011.
[104] Z. Miao, S. Roy, and J. Yang. Explaining wrong queries using small examples. In *SIGMOD*, 2019.
[105] S. Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *IDAC*, 1993.
[106] M. Monet. Solving a special case of the intensional vs extensional conjecture in probabilistic databases. In *PODS*, 2020.
[107] M. Monet and D. Olteanu. Towards deterministic decomposable circuits for safe queries. In *AMW*, 2018.
[108] M. Muñoz and C. Riveros. Streaming enumeration on nested documents. In *ICDT*, 2022.
[109] M. Muñoz and C. Riveros. Constant-delay enumeration for SLP-compressed documents. In *ICDT*, 2023.
[110] D. Olteanu. Factorized databases: A knowledge compilation perspective. In *BeyondNP (AAAI Workshops)*, 2016.
[111] D. Olteanu and J. Huang. Using OBDDs for efficient query evaluation on probabilistic databases. In *SUM*, volume 5291, 2008.
[112] D. Olteanu and J. Závodnỳ. On factorisation of provenance polynomials. In *TaPP*, 2011.
[113] D. Olteanu and J. Závodnỳ. Factorised representations of query results: size bounds and readability. In *ICDT*, 2012.
[114] D. Olteanu and J. Závodnỳ. Size bounds for factorised

representations of query results. *TODS*, 40(1), 2015.
[115] R. Pichler and S. Skritek. Tractable counting of the answers to conjunctive queries. *Journal of Computer and System Sciences*, 79(6), 2013.
[116] K. Pipatsrisawat and A. Darwiche. New compilation languages based on structured decomposability. In *AAAI*, 2008.
[117] I. Razgon. On OBDDs for CNFs of bounded treewidth. In *KR*, 2014.
[118] D. Roth. On the hardness of approximate reasoning. *Artificial Intelligence*, 82(1-2), 1996.
[119] N. Schweikardt, L. Segoufin, and A. Vigny. Enumeration for FO queries over nowhere dense graphs. *JACM*, 69(3), 2022.
[120] L. Segoufin. A glimpse on constant delay enumeration. In *STACS*, 2014.
[121] L. Segoufin and A. Vigny. Constant delay enumeration for FO queries over databases with local bounded expansion. In *ICDT*, 2017.
[122] P. Senellart, L. Jachiet, S. Maniu, and Y. Ramusat. ProvSQL: Provenance and probability management in PostgreSQL. *PVLDB*, 11(12), 2018.
[123] S. Sharma, R. Gupta, S. Roy, and K. S. Meel. Knowledge compilation meets uniform sampling. In *LPAR*, 2018.
[124] A. Shih, G. V. den Broeck, P. Beame, and A. Amarilli. Smoothing structured decomposable circuits. In *NeurIPS*, 2019.
[125] F. Somenzi. CUDD: CU decision diagram package release 3.0.0 (2015). `https://web.archive.org/web/20171208230728/http://vlsi.colorado.edu/~fabio/CUDD/cudd.pdf`. Accessed: 2024-05-06.
[126] B. Strasser. Computing tree decompositions with FlowCutter: PACE 2017 submission. *CoRR*, abs/1709.08949, 2017.
[127] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic databases*, volume 3. 2011. ISBN: 978-1608456802.
[128] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Any-k algorithms for enumerating ranked answers to conjunctive queries. *CoRR*, abs/2205.05649, 2022.
[129] L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3), 1979.
[130] T. van Bremen and K. S. Meel. Probabilistic query evaluation: The combined FPRAS landscape. In *PODS*, 2023.
[131] Y. Wang and K. Yi. Secure Yannakakis: Join-aggregate queries over private data. In *SIGMOD*, 2021.
[132] Y. Wang and K. Yi. Query evaluation by circuits. In *PODS*, 2022.
[133] K. Wasa. Enumeration of enumeration algorithms, 2016.
[134] I. Wegener. *Branching programs and binary decision diagrams: Theory and applications*. SIAM, 2000. Section 8.1.
[135] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, volume 81, 1981.