

Building Relational Circuits

Florent Capelli  

Univ. Artois, CNRS, UMR 8188, Centre de Recherche en Informatique de Lens (CRIL),
F-62300 Lens, France

Abstract

We review two algorithms which allow to build a factorized representation of the answers set of join queries. In a nutshell, the representation builds a circuit representing the answers set of a join query by starting from atomic relations and iteratively combine them by either constructing the Cartesian product or the disjoint union of previously computed relations. The first one can be seen as the trace of the celebrated Yannakakis algorithm, building the answer set from the inputs to the output of the circuit while the second adopts a top-down approach which can be seen as a generalization of the exhaustive DPLL algorithm, originally designed to solve the #SAT problem.

2012 ACM Subject Classification Information systems → Relational database model

Keywords and phrases Conjunctive queries, factorized databases, knowledge compilation

Digital Object Identifier 10.4230/LIPIcs.ICDT.2026.3

Category Invited Talk

Supplementary Material *InteractiveResource*: <https://florent.capelli.me/algorithms/dpll/>

Funding This work was supported by KCODA project, ANR-20-CE48-0004.

1 Introduction

From the seminal paper by Codd [19], it has been quite clear that organizing data as relations is a powerful abstraction, leading to a clear and intuitive separation of the data itself, its structure and how it can be queried. In a way, a query can be seen as a succinct way of specifying a relation. Yet, this succinctness comes at the price of making the relation harder to manipulate and most of the time, one ends up materializing it, which can be costly [18], before using the results. A notorious result by Yannakakis identified a class of queries for which this materialization was not necessary to decide whether the query has at least one answer, namely, the class of acyclic queries [50] (called acyclic scheme in the original paper), a large body of work has focused on finding other tractable tasks that can be efficiently solved on acyclic join queries, beyond simply deciding whether it has at least one answer. For example, it has been observed that one can efficiently enumerate the answers of an acyclic join query [7], compute their number [44], aggregates the answer over different semirings [1, 31] etc. Following the line of research on factorized databases [42, 43, 40] having its root in the field of knowledge compilation [23, 40] which focuses on efficient representations of knowledge bases, we argue in this paper that this tractability can be seen as the fact that the answer sets of acyclic queries can be efficiently represented in a factorized, yet tractable data structure, namely, a *relational circuit*.

This paper contains notes related to my invited ICDT lecture. It aims at giving an introduction to relational circuits (see Section 3.1), how to use them (see Section 3.2) and how to construct them (see Section 4), either by adapting Yannakakis algorithm (see Section 4.1) or by using an algorithm originally designed for solving #SAT (see Section 4.2). We then explore the consequences of these constructions regarding several known tractability results from the literature (see Section 5). It does not aim at giving full formalization and voluntarily stays high level. We also tried to give many pointers to related literature but many other relevant and interesting results and research are still unfortunately missing.



© Florent Capelli;
licensed under Creative Commons License CC-BY 4.0
29th International Conference on Database Theory (ICDT 2026).

Editors: Balder ten Cate and Maurice Funk; Article No. 3; pp. 3:1–3:20



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

2 Preliminaries

Tuples and relations. We let D^X to be the set of mapping from a set X of *attributes* to a set D called the *domain*. An element $\tau \in D^X$ is called a *tuple over attributes X and domain D* . For $\tau \in D^X$, we let $\text{attr}(\tau) =_{\text{def}} X$ to be the attributes of τ . Given a set of attributes Y , we let $\tau|_Y$ to be the tuple defined over attributes $Y \cap \text{attr}(\tau)$ and defined as $\tau|_Y(y) = \tau(y)$ for every $y \in Y \cap \text{attr}(\tau)$. If τ_1, τ_2 are two tuples such that $\tau_1|_Z = \tau_2|_Z$ where $Z = \text{attr}(\tau_1) \cap \text{attr}(\tau_2)$, we say that τ_1 and τ_2 are consistent and write $\tau_1 \simeq \tau_2$. In this case, we let $\tau_1 \bowtie \tau_2$ to be the tuple defined over $\text{attr}(\tau_1) \cup \text{attr}(\tau_2)$ as $(\tau_1 \bowtie \tau_2)(x) = \tau_1(x)$ if $x \in \text{attr}(\tau_1)$ and $(\tau_1 \bowtie \tau_2)(x) = \tau_2(x)$ otherwise. We write $\tau_1 \times \tau_2$ when $\text{attr}(\tau_1) \cap \text{attr}(\tau_2) = \emptyset$ to insist on the fact that they have disjoint attributes. We write tuple between angle brackets and with a list of couples of the form x/d where x is an attribute and d a domain value, meaning that the tuple assigns value d to x . For example $\tau =_{\text{def}} \langle x/1, y/2, z/1 \rangle$ is the tuple over attributes $\{x, y, z\}$ such that $\tau(x) = \tau(z) = 1$ and $\tau(y) = 2$. We denote by $\langle \rangle$ the empty tuple (that is, the only tuple whose attributes are \emptyset , and the identity of \times).

A *relation* $R \subseteq D^X$ on attributes X and domain D is a set of tuples. We extend notations of tuples to relations: we define $\text{attr}(R) =_{\text{def}} X$ to be the attributes of R , the projection of R over attributes Y to be the relation over $X \cap Y$ defined as $R|_Y := \{\tau|_Y \mid \tau \in R\}$. The *natural join* of two relations R and S is defined as $R \bowtie S := \{\tau \bowtie \sigma \mid \tau \in R, \sigma \in S, \tau \simeq \sigma\}$. Again, if $\text{attr}(R) \cap \text{attr}(S) = \emptyset$, we write $R \times S$ instead of $R \bowtie S$. In this case, we say that $R \times S$ is the *Cartesian product* of R and S . Observe in this case that $|R \times S| = |R| \times |S|$. Given two relations $R, S \subseteq D^X$ over attributes X , we write $R \cup S$ for the union of relations R and S , seen as sets of tuples. Now when $R \subseteq D^X$ and $S \subseteq D^Y$ are not defined over the same attributes, we define the *extended union of R and S over domain D* , denoted by $R \bar{\cup}_D S$, to be $R \times D^{Y \setminus X} \cup S \times D^{X \setminus Y}$. We simply write $R \bar{\cup} S$ when D is clear from context. Given a relation R over attributes X and τ a tuple over attributes Y , we denote by R/τ the relation over attributes $X \setminus Y$ containing the tuple σ such that $\sigma \times \tau|_{Y \cap X} \in R$. Despite the notation being non standard, we think it is natural: the “/” symbol here is to be interpreted as the inverse of the Cartesian product \times .

Join and conjunctive queries. In this paper, we focus on join queries (also known as full conjunctive queries or quantifier-free conjunctive queries). A join query is traditionally defined only over relation names, and the content of each relation, the *data*, is defined outside of the query itself, in a database. To ease notations and concepts, we slightly deviate from this point of view in this paper and we define a *join query* $Q = R_1(X_1), \dots, R_m(X_m)$ as a finite list of relations, called *the atoms of Q* , $R_1 \subseteq D^{X_1}, \dots, R_m \subseteq D^{X_m}$ over a finite domain D and finite sets of attributes X_1, \dots, X_m .

The *variables* of a join query Q are defined as $\text{var}(Q) =_{\text{def}} \bigcup_{i=1}^m X_i$. A join query Q implicitly represents a relation over attributes $\text{var}(Q)$ and domain D which is defined as performing the natural join of each relation it contains. Formally, we define *the answers set* $\text{ans}(Q)$ of Q as $\text{ans}(Q) =_{\text{def}} \{\tau \in D^{\text{var}(Q)} \mid \forall i \in [m], \tau|_{X_i} \in R_i\} \subseteq D^{\text{var}(Q)}$.

We consider two notions of sizes for join queries, inherited from the original separation between the logical layer and the data layer. The *query size of Q* , denoted by $|Q|$ is defined as $\sum_{R \in Q} |\text{var}(R)|$. The query size does not take into account the content of the relation, but only their structure. The *data size of Q* , denoted by $\|Q\|$, is defined as $\sum_{R \in Q} |\text{var}(R)| \cdot |R|$. The data size is sensible to the number of tuples in each relation, while the query size only depends on the number of relations and their arities.

A *conjunctive query* is defined as a pair (Q, Z) where Q is a join query and $Z \subseteq \text{var}(Q)$ is a subset of attributes of Q called *the free variables of Q* . We often simply write it as $Q(Z) = R_1(X_1), \dots, R_m(X_m)$ to make the free variables explicit. The answers set $\text{ans}(Q(Z))$ is defined as $\text{ans}(Q)|_Z$, that is, $\{\tau|_Z \mid \tau \in \text{ans}(Q)\}$. We extend the query size and data size to conjunctive queries, where $|Q(Z)|$ is defined as $|Q|$ and $\|Q(Z)\|$ as $\|Q\|$.

Model of computation. We use the word-RAM model of computation where registers contain $O(\log n)$ bits, where n is the size of the input and arithmetic operations over two registers can be done in constant time. Observe that in this model, we can perform arithmetic operations and comparisons over integers bounded by n^c in polynomial time in c (actually, in time $O(c \log c)$) [29]. In the case of join queries over variable X and domain D , we will often use numbers of size up to $|D|^{|X|}$ (for example, when counting tuples), which means that this number can be manipulated in time $O(|X| \log |X|)$. Using radix sort, we can also sort m such values or tuples in time $O(|X| \cdot (|D| + m))$ [20, Section 6.3]. In other words, we can sort the tuples of a relation R in time linear in $|R| \cdot |\text{var}(R)|$.

3 Relational Circuits

In this section, we introduce a succinct way of describing relations using circuits whose syntactic properties make them suitable for further analysis.

3.1 Main definitions

From a general point of view, a relational circuit is a circuit whose inputs are labeled by atomic relations and gates are functions building new relations from its input relations. In this setting, we can see a join query $Q = R_1(X_1), \dots, R_m(X_m)$ as a flat relational circuit, with relations R_i being inputs of a single \bowtie -labeled gate. We show how one can use factorization and syntactic restrictions to design representations that are both succinct and tractable.

Join circuits. We start with a very general definition that serves as a starting point before restricting it into more interesting classes of circuits. A $\{\bowtie, \bar{\cup}\}$ -circuit C on attributes X and domain D is defined as a labeled direct acyclic multi-graph (that is, there may be more than one edge between two vertices of the graph) with one distinguished vertex called the *output of C* and denoted by $\text{out}(C)$. The vertices of the underlying DAG of C are called *gates*. If two gates g, g' are connected by a directed edge $g \rightarrow g'$, we say that g is an *input of g'* and that g' is an *output of g* . The gates of C are labeled as follows:

- Every gate g without input (that is, without any incoming edge) is called *an input of the circuit* and is labeled by either \perp , \top or x/d for some attribute $x \in X$ and domain value $d \in D$. If g is labeled by \top or \perp , we say that g is a constant input.
- Every other gate (that is, every gate with at least one input) is labeled by either $\bar{\cup}$ or \bowtie .

The attributes $\text{attr}(g)$ of a gate g of C are defined to be the set of attributes appearing in at least one input of the subcircuit rooted at g . In other words, $\text{attr}(g) = \emptyset$ if g is labeled by \top or \perp , $\text{attr}(g) = \{x\}$ if g is labeled by x/d for some $d \in D$, and $\text{attr}(g) = \bigcup_{i=1}^k \text{attr}(g_i)$ if g_1, \dots, g_k are the inputs of g .

Each gate g computes a relation $\text{rel}(g) \subseteq D^{\text{attr}(g)}$ defined inductively as:

- If g is an input then: either it is labeled by \perp and $\text{rel}(g) = \emptyset$, or by \top and $\text{rel}(g) = \{\langle \rangle\}$, or by x/d then $\text{rel}(g) = \{\langle x/d \rangle\}$.
- If g has input g_1, \dots, g_k and is labeled by \bowtie then $\text{rel}(g) = \text{rel}(g_1) \bowtie \dots \bowtie \text{rel}(g_k)$.
- If g has input g_1, \dots, g_k and is labeled by $\bar{\cup}$ then $\text{rel}(g) = \text{rel}(g_1) \bar{\cup} \dots \bar{\cup} \text{rel}(g_k)$.

3:4 Building Relational Circuits

The *relation* $\text{rel}(C)$ computed by C is defined as $\text{rel}(C) := \text{rel}(\text{out}(C)) \times D^{X \setminus Z}$ where $Z = \text{attr}(\text{out}(C))$. If X is not made explicit, we simply let $\text{rel}(C) = \text{rel}(\text{out}(C))$.

The *size* $|C|$ of a $\{\bowtie, \bar{\cup}\}$ -circuit is defined as the number of edges of its underlying graph. Sometimes, the size of circuits is defined as the number of gates, but this definition is often less natural. Indeed, an algorithm visiting every edge of the circuit is linear in the size of the input but not necessarily linear in the number of gates in the circuit since there may be $O(N^2)$ edges in a circuit with N nodes.

The family of $\{\bowtie, \bar{\cup}\}$ -circuits is not interesting from a complexity point of view as they are more general than join queries. We now restrict $\{\bowtie, \bar{\cup}\}$ -circuits in order to find representations of relations that are more tractable than join queries.

Cartesian products. The main syntactic restriction that we impose on relational circuits is the following: in a $\{\bowtie, \bar{\cup}\}$ -circuit, we say that a \bowtie -gate g is a Cartesian product gate, or \times -gate for short, if $\text{attr}(g_1) \cap \text{attr}(g_2) = \emptyset$ for every distinct input g_1, g_2 of g . Observe that in this case, if g_1, \dots, g_k are the inputs of g , then $\text{rel}(g) = \times_{i=1}^k \text{rel}(g_i)$. A $\{\times, \bar{\cup}\}$ -circuit is a $\{\bowtie, \bar{\cup}\}$ -circuit where every \bowtie -gate is a \times -gate. Obviously, given a $\{\bowtie, \bar{\cup}\}$ -circuit C , we can check whether it is a $\{\times, \bar{\cup}\}$ -circuit in time $O(|C| \cdot |X|)$ by computing $\text{attr}(g)$ for every g .

Union gates. Another useful restriction regarding relational circuits is to restrict how $\bar{\cup}$ -gates are operating. Indeed, $\bar{\cup}$ -gates are cumbersome to use because they force us to keep track of the attributes of each gate and to be explicit about the domain. We hence define a $\{\times, \cup\}$ -circuit as a $\{\times, \bar{\cup}\}$ -circuit such that for every $\bar{\cup}$ -gate g with input g_1, \dots, g_k , we have $\text{attr}(g_1) = \dots = \text{attr}(g_k)$. In this case, one can easily check that $\text{rel}(g) = \bigcup_{i=1}^k \text{rel}(g_i)$.

Disjoint unions. While $\{\times, \cup\}$ -circuits allow for enumeration, one can observe that they do not allow to efficiently compute statistics on $\text{rel}(C)$. Indeed, one can easily see that it is $\#\text{P}$ -hard to compute $|\text{rel}(C)|$ when C is a $\{\times, \cup\}$ -circuit given on the input, by a direct reduction to the problem $\#\text{DNF}$, though, if one is interested in approximation, this problem admits an FPRAS [36]. The hardness of counting mainly stems from the fact that unions of relations in the circuit may overlap, which makes the cardinality of the union hard to estimate.

To avoid it, we need an extra property: given a $\{\times, \cup\}$ -circuit C , we say that a \cup -gate g with inputs g_1, \dots, g_k is *deterministic*¹ if for every $i < j \leq k$, we have $\text{rel}(g_i) \cap \text{rel}(g_j) = \emptyset$. In this case, observe that $\bigcup_{i=1}^k \text{rel}(g_i)$ is a disjoint union, denoted by $\uplus_{i=1}^k \text{rel}(g_i)$. A deterministic \cup -gate will be denoted by \uplus . As for union, we also have \uplus_D to denote the union of relations on different attributes but having no common tuple on their shared attributes. A $\{\times, \uplus\}$ -circuit is then a $\{\times, \cup\}$ -circuit where every \cup -gate is deterministic.

Given a $\{\times, \uplus\}$ -circuit, we can now compute $|\text{rel}(C)|$ with $O(|C|)$ arithmetic operations using a straightforward dynamic programming, see Section 3.2 for details.

Decision-gates. One can see that checking whether a given \cup -gate is deterministic is not easy. Indeed, it is a semantic property on the relation computed by each gate and not an easy-to-check syntactic property. In practice, we know that a \cup -gate is deterministic because the algorithm which built the circuit formally guarantees that this is the case. While in some

¹ We keep here the usual terminology from knowledge compilation [22] and from seminal work on factorized databases [43], though the naming is slightly misleading, as it is not deterministic in the usual sense. “Unambiguous” would be a better term.

cases, determinism may depend on complex reasons, in every algorithm we will describe in this document, determinism is ensured because some attribute x has a different value in each subcircuit. To model this very specific case, we introduce the notion of decision-gate: a decision-gate g is a gate labeled by a variable x and each incoming edge e of g is labeled by a value $d_e \in D$. Moreover, if $e_1 = g_1 \rightarrow g, \dots, e_k = g_k \rightarrow g$ are the incoming edges of g respectively labeled by d_1, \dots, d_k , then for every $i < j$, $d_i \neq d_j$ and $x \notin \text{attr}(g_i)$. We define $\text{rel}(g) = \biguplus_{i=1}^k \{\langle x/d_i \rangle\} \times \text{rel}(g_i)$. Clearly, by definition, we could rewrite every decision-gate with k incoming edges using only \uplus -gates and \times -gates² and at most $3k$ edges, hence decision-gates is only a useful syntactic sugar. A $\{\times, \text{dec}\}$ -circuit C is a circuit whose every gate are either Cartesian products or decision-gates.

Ordered $\{\times, \cup\}$ -circuitdec. Different paths of decision-gates in a $\{\times, \text{dec}\}$ -circuit may use different order. For example, we could have a path testing variable x then y then z and another path testing x then z then y . That said, the algorithm we give in Section 4 will always test variables in the same order, and we will actually use this fact to recover some results on direct access. Let $\pi = (x_1, \dots, x_n)$ be an order on X . We say that a $\{\times, \text{dec}\}$ -circuit on attributes X *respects order* π if for every decision-gate g labeled by x_i and g' an input of g , we have $\text{attr}(g') \subseteq \{x_{i+1}, \dots, x_n\}$.

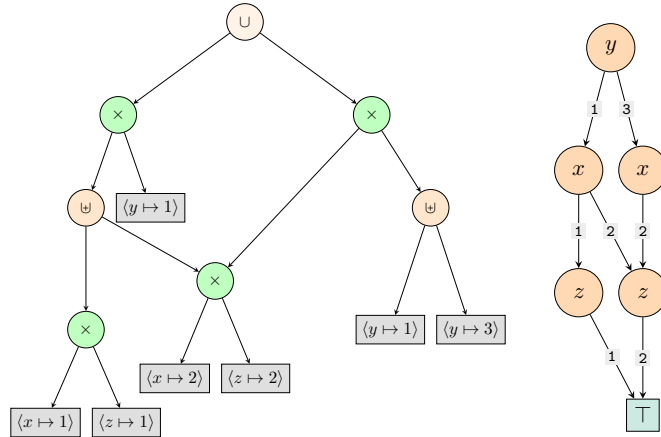
We observe here that some \uplus -gates are not decision-gates. For example, let g_0, g_1 be gates such that $\text{rel}(g_0) = \{\langle x/0, y/0 \rangle, \langle x/1, y/1 \rangle\}$ and $\text{rel}(g_1) = \{\langle x/0, y/1 \rangle, \langle x/1, y/0 \rangle\}$, then a \cup -gate with input g_0, g_1 is deterministic because $\text{rel}(g_0)$ only contains tuples with an even number of ones and $\text{rel}(g_1)$ with an odd number of ones but it cannot be seen as a decision-gate since both x and y can take values 0 and 1 in $\text{rel}(g_0)$ and in $\text{rel}(g_1)$. In this case, determinism is witnessed by a parity argument. Actually, this idea forms the base of the argument to show that $\{\times, \uplus\}$ -circuit are exponentially more succinct than $\{\times, \text{dec}\}$ -circuit [9].

► **Example 1.** We illustrate the previous definitions on Figure 1. We give two circuits computing the same relation $R(x, y, z) = \{\langle x/1, y/1, z/1 \rangle, \langle x/2, y/1, z/2 \rangle, \langle x/2, y/3, z/2 \rangle\}$. Observe that the output of the first circuit is a \cup -gate which is not a \uplus -gate because the relation computed by its two input gates both contain the tuple $\langle x \mapsto 2, y \mapsto 1, z \mapsto 2 \rangle$. The second circuit only has decision-gates.

Constants elimination and smoothing. While we can be flexible on the syntax of relational circuits, useless gates may induce complexity. In particular, constant gates (\perp -gates and \top -gates) may induce hidden costs. We observe that we can remove them in the circuit with a linear time procedure as follows: we remove every input of \times -gates labeled by \top and every input of \cup -gate labeled by \perp without changing the relation they compute. We also replace a \times -gates with a \perp -labeled input by a \perp -gate and a \cup -gate with a \top -labeled input by a \top -gate. Each change reduces the size of the circuit by at least one, hence applying these transformations iteratively finishes and returns a circuit without constant gates.

We also observe here that $\{\times, \bar{\cup}\}$ -circuits may also be transformed into $\{\times, \cup\}$ -circuit in time $O(|X| \cdot |C|)$: we can indeed precompute $\text{attr}(g)$ for every gate of C and, for each $\bar{\cup}$ -gate g of C with input g' , we can add a \times -gate g'' between g' and a circuit computing D^{Δ_i} where $\Delta = \text{attr}(g) \setminus \text{attr}(g')$ and plug g'' into g instead of g' . This operation is known as smoothing in knowledge compilation [23] and can sometimes be costly if the number of variables is high. One can avoid it in some cases [46, 4]. In this work, the algorithm we will present naturally build $\{\times, \text{dec}\}$ -circuit so we will not focus much on $\{\times, \bar{\cup}\}$ -circuit nor $\{\times, \uplus\}$ -circuit.

² Observe that for this, we need the condition that $x \notin \text{attr}(g_i)$.



■ **Figure 1** A $\{\times, \cup\}$ -circuit and a $\{\times, \text{dec}\}$ -circuit computing the same relation $R(x, y, z) = \{\langle x/1, y/1, z/1 \rangle, \langle x/2, y/1, z/2 \rangle, \langle x/2, y/3, z/2 \rangle\}$.

3.2 Tractable tasks

In this section, we show how we can exploit properties of relational circuits to get insights on the relation they compute. This approach is akin to the one used in knowledge compilation [23]: for each class of circuits, we try to understand the tractable tasks and their complexity.

Enumeration. Given a $\{\times, \cup\}$ -circuit C on attributes X and domain D , one of the first tasks one can do is to list every tuple of $\text{rel}(C)$. To measure the complexity of such algorithm, it is customary to analyze the time spent between two outputs of the algorithm, which we call the *delay*. Sometimes, we need to precompute some values on the circuit before starting. This time will be accounted for as preprocessing time.

It is easy to see that we can find a tuple in $\text{rel}(C)$ in time $O(|C|)$ by inductively constructing for each g , a tuple τ_g in $\text{rel}(g)$ for every gate g in the circuit, or report that $\text{rel}(g) = \emptyset$. But we can even output every tuple in $\text{rel}(C)$ with delay $O(|X| \cdot |C|)$ as follows. The main idea is to observe that giving a tuple τ over $Y \subseteq X$ and $x \in X \setminus Y$, we can find in time $O(|C|)$ the subset $D' \subseteq D$ such that D' is the set of domain values for which $\tau \times \langle x/d \rangle$ can be extended to a full tuple of $\text{rel}(C)$. This can be done by plugging every variable in Y to their value (ie, replace input y/d by \top if $d = \tau(y)$ and by \perp otherwise) and every input in $z \in X \setminus (Y \cup \{x\})$ by \top . It gives a $\{\times, \cup\}$ -circuit on variable $\{x\}$ whose computed relation is exactly D' . By propagating constants in the circuit in time $O(|C|)$, it simplifies to a circuit where D' can be read directly as the remaining input of the form x/d . This is enough to implement a flashlight search to enumerate $\text{rel}(C)$ by constructing each tuple one variable at a time, without never exploring partial tuples that cannot be extended to full solutions: we start by computing every possible values D' that x_1 can take in $\text{rel}(C)$, then pick $d \in D'$ and find every value of x_2 can take when x_1 is set to d etc. This method ensures that every partial tuple we built can be extended to a full tuple in $\text{rel}(C)$. Once such a tuple is found, we backtrack to the last variable for which it remains at least one unexplored value. Between two outputs, we do at most $|X|$ oracle calls to the previously describe procedure, hence a delay of $O(|C| \cdot |X|)$. This method can even be extended to the case where one wants to output the tuples by increasing weights, see [3] for details.

The delay of this algorithm may be too high to recover the existing enumeration bounds from the database literature, as we would like to have a complexity that only depends on the number of variables of C . We show that in the case of $\{\times, \uplus\}$ -circuit, we get better bounds. Indeed, to enumerate $\text{rel}(g)$ when g is a \uplus -gate with inputs g_1, \dots, g_k , one only needs to successively enumerate $\text{rel}(g_i)$ for every $1 \leq i \leq k$. The delay is thus $\max d_i$ where d_i is the delay to enumerate $\text{rel}(g_i)$. To enumerate $\text{rel}(g)$ where g is a \times -gate with input g_1, \dots, g_k , we enumerate one tuple from $\text{rel}(g_1)$, then every tuple from $\text{rel}(g_2) \times \dots \times \text{rel}(g_k)$. The delay between two outputs is $\sum_i d_i$ since it is the time needed to find the first tuple. If we again ensure that each \times -gate has no trivial input, we can show that the delay is $O(nh)$ where h is the depth of the circuit and n the number of variables. Observe that in the worst-case, the depth h could be of the order of $|C|$. We can however always rewrite into a circuit with depth at most n by ensuring the every \uplus -gate has no other \uplus -gate as input. Doing it naively may however increase the size of the circuit quadratically. Amarilli, Bourhis, Jachiet and Mengel proposed a work-around in [4] to quickly find the gates that could be reached from a \uplus -gate in linear time allowing for enumeration with delay $O(n)$ after linear preprocessing.

Observe that in the particular case of $\{\times, \text{dec}\}$ -circuit, the previous algorithm directly gives $O(n)$ delay as long as for every \times -gate g with input g_1, \dots, g_k , we have $\text{attr}(g_i) \neq \emptyset$, which can be ensured with linear time preprocessing by simply removing constants in the circuit.

Counting. Given a $\{\times, \uplus\}$ -circuit, we can now compute $|\text{rel}(C)|$ with $O(|C|)$ arithmetic operations using a straightforward dynamic programming. For each gate g , we compute $|\text{rel}(g)|$ inductively: if g is an input, then $|\text{rel}(g)| = 0$ if g is a \perp -gate and $|\text{rel}(g)| = 1$ otherwise. If g is a \times -gate with input g_1, \dots, g_k , then $|\text{rel}(g)| = \prod_{i=1}^k |\text{rel}(g_i)|$. If g is a \uplus -gate with input g_1, \dots, g_k then $|\text{rel}(g)| = \sum_{i=1}^k |\text{rel}(g_i)|$. Hence, we can compute $|\text{rel}(C)|$ with $O(|C|)$ arithmetic operations. For every g , $|\text{rel}(g)|$ does not exceed $|D|^{|X|}$, where D is the domain of C and X its attributes. Hence, each number can be stored in at most $|X|$ registers in a RAM machine, and each arithmetic operation can be performed in time $O(|X| \log |X|)$, resulting in a final complexity of $O(|X| \log |X| \cdot |C|)$. For $\{\times, \uplus\}$ -circuit, we have a similar result but we need to precompute $\text{attr}(g)$ for every gate g . If g is a \cup -gate, we have $|\text{rel}(g)| = \sum_{i=1}^k |\text{rel}(g_i)| \cdot |D|^{\delta_i}$ where $\delta_i = |\text{attr}(g) \setminus \text{attr}(g_i)|$.

Sampling. Interestingly the previous algorithm can be leveraged into a uniform sampling algorithm. Assume $|\text{rel}(g)|$ has been precomputed for every gate g of C . If one wants to draw a tuple $\tau \in \text{rel}(g)$ uniformly, we can do this simple procedure:

- If g is a \perp -gate, then fail (this case should not happen unless $\text{rel}(C) = \emptyset$).
- If g is a \top -gate or a gate labeled by x/d , then return the only tuple in $\text{rel}(g)$.
- If g is a \uplus -gate with input g_1, \dots, g_k , then draw $1 \leq i \leq k$ with probability $\frac{|\text{rel}(g_i)|}{|\text{rel}(g)|}$ and uniformly sample in $\text{rel}(g_i)$.
- If g is a \times -gate with input g_1, \dots, g_k then return $\tau_1 \times \dots \times \tau_k$ where τ_i is sampled uniformly from $\text{rel}(g_i)$.

In the case of $\{\times, \uplus\}$ -circuit, without further assumptions, we can show that this procedure runs in time $O(|X| \cdot h)$ where h is the depth of the circuit, that is, the longest path from the output to an input of C . If C is a $\{\times, \text{dec}\}$ -circuit and if we have removed trivial inputs from \times -gate (that is, for every \times -gate g , $\text{attr}(g) \neq \emptyset$), then we can show that this procedure runs in time $O(|X|)$.

Direct Access. We can go one step further and compute a numbering on $\text{rel}(C)$ so that we can efficiently access $\text{rel}(C)[i]$, as if it was a table. This kind of algorithm have been called “direct access algorithm” [8, 16, 25, 17, 14]. We define an order on $\text{rel}(C)$ by inductively defining, for every gate g of C , a one-to-one mapping $I_g : \text{rel}(g) \rightarrow [0, N_g - 1]$ where $N_g = |\text{rel}(g)|$. We call I_g an index on $\text{rel}(g)$. We first assume for every gate g , its input g_1, \dots, g_k are listed in some arbitrary order. We now define the index functions inductively.

- If g is an input, not labeled by \perp , we define $I_g(\tau) = 0$, for τ the only assignment in $\text{rel}(g)$.
- If g is a \uplus -gate with input g_1, \dots, g_k , let $\tau \in \text{rel}(g)$. We define $I_g(\tau) = \sum_{i=1}^{j-1} N_i + I_{g_j}(\tau)$ where $N_i = |\text{rel}(g_i)|$ for every $i \leq k$ and j is the only value in $[k]$ such that $\tau \in \text{rel}(g_j)$. In other words, we order $\text{rel}(g)$ by first taking every tuple from g_1 , then g_2 etc.
- If g is a \times -gate with input g_1, \dots, g_k , let $\tau \in \text{rel}(g)$. Let $\tau_i = \tau|_{\text{attr}(g_i)}$ and $N_i = |\text{rel}(g_i)|$. We by $\tau_i \in \text{rel}(g_i)$ for every $i \leq k$. We define

$$I_g(\tau) =_{\text{def}} \sum_{j=1}^k I_{g_j}(\tau_j) \prod_{i=j+1}^k N_i,$$

that is $I_g(\tau)$ is the rank of $(I_{g_1}(\tau_1), \dots, I_{g_k}(\tau_k))$ in $[N_1] \times \dots \times [N_k]$ ordered lexicographically.

Interestingly, we can revert this indexing structure. Indeed, given $I \in \{0, \dots, |\text{rel}(g)| - 1\}$, we can efficiently compute $I_g^{-1}(I)$, that is, the tuple $\tau \in \text{rel}(g)$ such that $I_g(\tau) = I$. Indeed, assume we have precomputed, for every \uplus -gate g with input g_1, \dots, g_k and $j \leq k$, the value $l_j = \sum_{i=1}^j |\text{rel}(g_i)|$ and stored them in an ordered table $L_g = [l_1, \dots, l_k]$. We also precompute $|\text{rel}(g)|$ for every \times -gate g .

First assume g is a \uplus -gate and let $I \leq |\text{rel}(g)|$. To find the I^{th} tuple of $\text{rel}(g)$, we need look for the smallest j such that $l_j \leq I$ which can be done via a binary search with $O(\log k) = O(\log |C|)$ comparisons, each comparison being on integers smaller than $|D|^{|X|}$. Then we inductively output the $I_{g_i}^{-1}(I - l_j)$ tuple of g_i .

Now assume g is a \times -gate with input g_1, \dots, g_k and let $I \leq |\text{rel}(g)|$. We are looking for $\tau = \tau_1 \times \dots \times \tau_k \in \text{rel}(g_1) \times \dots \times \text{rel}(g_k)$ such that

$$I = \sum_{j=1}^k I_{g_j}(\tau_j) \prod_{i=j+1}^k N_i,$$

where $N_i = |\text{rel}(g_i)|$ has been precomputed already. Observe that $I_{g_k}(\tau_k) = I \bmod N_k$ since every other term of the sum can be divided by N_k . Similarly, $I_{g_{k-1}}(\tau_{k-1}) = (I/N_k) \bmod N_{k-1}$ where I/N_k is the quotient of the division of I by N_k . We similarly have $I_{g_j}(\tau_j) = (I / \prod_{i=j+1}^k N_i) \bmod N_j$. Hence we can compute inductively τ_j as $I_{g_j}^{-1}((I / \prod_{i=j+1}^k N_i) \bmod N_j)$.

Hence, after a precomputation step of $O(|X| \cdot |C|)$, we can, on input I , fail if $I > |\text{rel}(C)|$, and otherwise, output $\tau = I_{\text{out}(C)}^{-1}(I)$ with $O(h|X| \log |C|)$ comparisons or arithmetic operations on integers smaller than $|D|^{|X|}$, where h is the depth of C . Hence, in total time $O(h|X|^2 \cdot (\log |X|) \cdot (\log |C|) \cdot (\log |D|))$.

This direct access algorithm has interesting consequences. It allows to enumerate $\text{rel}(C)$ in a uniformly chosen order, or, equivalently, perform sampling without replacement: we sample or randomly enumerate values in $\{1, \dots, |\text{rel}(C)|\}$, and output the corresponding tuple in $\text{rel}(C)$. This can be done efficiently by adapting the Fisher-Yates shuffle algorithm in a lazy way, as in [17].

We again get better complexity bounds for $\{\times, \text{dec}\}$ -circuits. Indeed, in this case, we can bound the arity of \uplus -gates (in the form of decision-gates) by the size of the domain D and we can show that we can recover the I^{th} element of $\text{rel}(C)$ with $O(|X| \log |D|)$ arithmetic

operations because each recursive call is done on gates having strictly less attributes, hence a total time of $O(|X|^2 \cdot (\log |X|) \cdot (\log |D|))$. More interestingly, if the circuit respects order (x_1, \dots, x_n) , then we can actually solve the direct access problem for the lexicographical order induced by (x_1, \dots, x_n) on D^X with a slightly more involved access technique that can still be executed with $O(|X| \log |D|)$ arithmetic operations and comparisons. Indeed, in this case, given a tuple τ over attributes $\{x_1, \dots, x_k\}$ with $k \leq n$, we can find how many tuple from $\text{rel}(C)$ are smaller than τ in the lexicographical order. We can use it to build the I^{th} element of $\text{rel}(C)$ by discovering it one variable at a time, using counting queries to find the value of the next variable with a binary search, see [14] for details.

We conclude this section by summarizing the results we will use the most in the context of databases:

► **Theorem 2.** *Given an ordered $\{\times, \text{dec}\}$ -circuit C on attributes X and domain D respecting order π , we can:*

- Enumerate $\text{rel}(C)$ with preprocessing $O(|C|)$ and delay $O(|X|)$,
- Compute $|\text{rel}(C)|$ in time $O(|X| \cdot |C|)$,
- Uniformly sample $\tau \in \text{rel}(C)$ after preprocessing $O(|X| \cdot |C|)$,
- Find the I^{th} element of $\text{rel}(C)$ according to the lexicographical order induced by π in time $O(|X|^2 \log |X| \log |D|)$ after $O(|X| \cdot |C|)$ preprocessing.

3.3 Previous work

Factorized databases. Relational circuits originated under the name *factorized databases* in the seminal paper by Olteanu and Závodný [42]. In this document, we decided to use the name relational circuits to describe the object used to represent the relation, in contrast to the term factorized databases which, in our opinion, describe a set of techniques covering, in particular, the notion of relational circuit. In this work, the class of $\{\times, \cup\}$ -circuits appears under the name *factorized representation with definitions*, or d-representations, while $\{\times, \boxplus\}$ -circuit have been introduced as deterministic d-representations. In [43], the definition explicitly enforces unions to be defined over gates having the same attributes. We decided to deviate from the original terminology because we think that $\{\times, \cup\}$ -circuit is more natural than d-representation and it allows to be more precise on the syntactic restriction we want to enforce on the circuit, where the term d-representation is too general.

We want to clearly distinguished in this paper between the syntactic properties of relational circuits and the algorithm constructing the circuits. This is why, in this work, we decided to separate the notion of *relational circuits* that simply represent relations, sometimes with specific semantics or syntactic restrictions, and how such relational circuits are produced.

NNFs and Relational circuits. Syntactically restricted Boolean circuits have been used for representing Boolean functions since the 80s, at first through the lens of *ordered binary decision diagrams* [13] (which can be seen as $\{\text{dec}\}$ -circuits) and then later generalized to restricted Boolean circuits under the name Decomposable Negation Normal Form (DNNF) [23, 21, 22] with *decomposable \wedge -gates*, corresponding to \times -gates, and *deterministic \vee -gates*, corresponding to \boxplus -gates on domain $\{0, 1\}$. The similarity of knowledge compilation and factorized databases has been observed in [40] but the connections between classes of circuit is not covered. We provide a comparison in Table 1 using the circuit notations from the previous sections and the main classes studied in knowledge compilation. Previous work have been using DNNF in many areas of computer for enumeration by encoding database queries

■ **Table 1** Rosetta Stone of Tractable Circuits.

KC	Relational circuits
NNF circuit	$\{\bowtie, \cup\}$ -circuit if smooth, $\{\bowtie, \bar{\cup}\}$ -circuit otherwise.
DNNF circuit	$\{\times, \cup\}$ -circuit if smooth, $\{\times, \bar{\cup}\}$ -circuit otherwise.
d-DNNF circuit	$\{\times, \uplus\}$ -circuit if smooth, $\{\times, \bar{\uplus}\}$ -circuit otherwise.
dec-DNNF circuit	$\{\times, \text{dec}\}$ -circuit if smooth, $\{\times, \bar{\text{dec}}\}$ -circuit otherwise.
FBDD	$\{\text{dec}\}$ -circuit if smooth, $\{\bar{\text{dec}}\}$ -circuit otherwise.

answers into DNNF [4, 3], probabilistic databases by representing the Boolean provenance a query with DNNFs [41, 10, 9, 48], for computing Shapley scores [6, 11], see [5] for a more detailed introduction on the applications of circuits in databases.

We do not include definitions of the Boolean counterpart of relational circuits and refer to [23]. that one can also recover them by simply taking the corresponding class of relational circuit and specializing them over domain $\{0, 1\}$: indeed each line illustrates the correspondence between a Boolean circuit and relational circuits as follows: when considering circuits on the right column on domain $D = \{0, 1\}$, it corresponds exactly to the Boolean circuit on the left.

The correspondence actually works both ways: for non-binary domain $D = \{d_0, \dots, d_{k-1}\}$, we can encode D with bitstrings over $b = \lceil \log k \rceil$ bits by representing d_i with the binary representation of i over b bits. Hence, given a $\{\bowtie, \cup\}$ -circuit C over attributes $X = \{x_1, \dots, x_n\}$ and domain $D = \{d_0, \dots, d_{k-1}\}$, we can convert it into an NNF over variables $\tilde{X} = \{x_1^0, \dots, x_1^{b-1}, \dots, x_n^0, \dots, x_n^{b-1}\}$ by replacing every input $\langle x_i/d_j \rangle$ by a circuit over variables x_i^0, \dots, x_i^{b-1} accepting only the assignment encoding j in binary. This can easily be done by $\{\text{dec}\}$ -circuit of size $2b$. It can be encoded in any NNF subclass from Table 1, showing that every relation circuit on attributes X and domain D from the right column can be naturally casted into an NNF from the left column with an additional increase of at most $|X| \log |D|$ increase.

We conclude this section by observing that most relational circuits, with relevant syntactic restrictions, can be understood as automaton on finite language. For example, ordered $\{\text{dec}\}$ -circuits correspond to finite state automaton without loop, hence on a finite language, while Context-Free grammar could be associated to restricted $\{\times, \cup\}$ -circuit where \times -gates are only allowed to split variables according to some total order. Other syntactic restriction of $\{\times, \cup\}$ -circuit not presented in this paper can also be shown to correspond to tree automaton. These connections have been studied in [33, 2].

4 Algorithms for Building Circuits

Now that we have introduce relational circuits as a handy way of representing relations, we turn our attention on two main algorithms to build them. The first one can be seen as revisiting the celebrated Yannakakis Algorithm [50] while the second one is a generalization, to the database setting, of the exhaustive DPLL algorithm [45] which has originally been devised for solving $\#\text{SAT}$, but which is known to implicitly build decision-DNNF, the Boolean counterpart of $\{\times, \text{dec}\}$ -circuit [30] and which is used in practice by many knowledge compilers such as d4 [35] or (a modification of) SharpSAT-TD [34, 32].

4.1 Bottom-up compilation and Yannakakis Algorithm

Yannakakis algorithm has originally been devised to check for consistency of acyclic database schemes in time linear in the size of the data [50], which is the same as checking whether an acyclic join query has at least one answer. It has quickly been observed that the same idea allows to also efficiently find every answer of the acyclic queries [7] or count them [44]. In this section, we explain how it could also be used as a way of constructing an ordered $\{\times, \text{dec}\}$ -circuit computing $\text{ans}(Q)$ of size linear in $\|Q\|$ for any acyclic join query Q .

A join query $Q = R_1(X_1), \dots, R_m(X_m)$ is *acyclic* if there exists a tree \mathcal{T} , called a *join tree*, such that:

- The vertices of \mathcal{T} are in one to one correspondence with R_1, \dots, R_m , and for a node t of \mathcal{T} , we let R_t be the relation labeling it.
- For every variable $x \in \text{var}(Q)$, $\{t \mid x \in \text{var}(R_t)\}$ is connected in \mathcal{T} (we call this property the *connectedness of \mathcal{T}*).

It is known that we can decide whether Q is acyclic and if so, construct a join tree of Q in linear time in $|Q|$ [47]. Yannakakis algorithm exploits the properties of join tree to propagate just the right information along it, in order to decide whether $\text{ans}(Q)$ is empty or not. In the rest of this section, we generalize this algorithm into a circuit construction establishing:

► **Theorem 3.** *Let Q be an acyclic join query. We can construct a $\{\times, \text{dec}\}$ -circuit C computing $\text{ans}(Q)$ of size $O(\|Q\|)$ in time $O(\text{poly}(|Q|) \cdot \|Q\|)$. Moreover, C respects an order π on $\text{var}(Q)$.*

Circuit construction. Let Q be a join query and \mathcal{T} a join tree for Q . Assume \mathcal{T} is rooted at an arbitrary node, and let t be a node of \mathcal{T} . We let \mathcal{T}_t be the subtree of \mathcal{T} rooted in t and $Q_{\leq t}$ to be the join query whose atoms are exactly the atoms appearing in \mathcal{T}_t . The crux of Yannakakis algorithm is the following observation: if t_1, t_2 are two children of t , then $\text{var}(Q_{\leq t_1}) \cap \text{var}(Q_{\leq t_2}) \subseteq \text{var}(R_t)$. Indeed, every path from \mathcal{T}_{t_1} to \mathcal{T}_{t_2} must go through t . Hence if x appears on both side, by connectedness, it must be in $\text{var}(R_t)$ too.

Let t_1, \dots, t_k be the children of t in \mathcal{T} . An answer σ of $Q_{\leq t}$ have the following shape:

- Projected over $\text{var}(R_t)$, they must be equal to some tuple τ in R_t .
- Projected over $\text{var}(Q_{\leq t_i})$, they must be an answer of $Q_{\leq t_i}$ that extends τ .

In other words, an answer of $Q_{\leq t}$ is of the form $\tau \times \tau_1 \times \dots \times \tau_k$ where $(\tau \times \tau_i)|_{V_i} \in \text{ans}(Q_{\leq t_i})$ where $V_i = \text{var}(Q_{\leq t_i})$. In other words, we have established:

$$\text{ans}(Q_{\leq t})/\tau = \text{ans}(Q_{\leq t_1})/\tau \times \dots \times \text{ans}(Q_{\leq t_k})/\tau. \quad (1)$$

If we further develop each term of the previous Cartesian product and if we let $\Delta_i = \text{var}(R_{t_i}) \setminus \text{var}(R_t)$, we have: μ

$$\text{ans}(Q_{\leq t_i})/\tau = \bigsqcup_{\sigma \in R_{t_i}, \sigma \simeq \tau} (\text{ans}(Q_{\leq t_i})/\sigma) \times \sigma|_{\Delta_i}. \quad (2)$$

In the original Yannakakis algorithm, these relations are used to dynamically maintain the following information: does $Q_{\leq t}$ have an answer extending $\tau \in R_t$? Indeed, Equation (1) tells us that $Q_{\leq t}$ has an answer extending τ if and only if for every $i \leq k$, $Q_{\leq t_i}$ has an answer extending τ , which by Equation (2) is equivalent to the fact that $Q_{\leq t_i}$ has an answer extending σ for some $\sigma \in R_{t_i}$ with $\sigma \simeq \tau$. In Yannakakis algorithm, the latter has been precomputed and hence, we can quickly decide in time $O(k \cdot |\text{var}(R_t)|)$ whether $Q_{\leq t}$ has an answer extending τ for every τ . When the algorithm reaches the root r of \mathcal{T} , $Q_{\leq r} = Q$ and we can decide whether Q has an answer.

But Equations (1) and (2) can also be seen as a way to build $\text{ans}(Q_{\leq t})/\tau$ for every $\tau \in R_t$ from the relations $\text{ans}(Q_{\leq t_i})/\sigma$ for every $\sigma \in R_{t_i}$, by using decision-gates and one Cartesian product gate. The circuit is build in a bottom-up fashion from the leaves of \mathcal{T} to its root. We build a circuit having the following property: for each node t of \mathcal{T} and $\tau \in R_t$, there is a gate v_t^τ computing $\text{ans}(Q_{\leq t})/\tau$. If t is a leaf of \mathcal{T} , $Q_{\leq t}$ only has one atom R_t . Hence, for every $\tau \in R_t$, $\text{ans}(Q_t)/\tau = \{\langle \rangle\}$ is the relation containing only the empty tuple. Hence, we define v_t^τ as a \top -gate.

Now if t has children t_1, \dots, t_k , we let $\tau \in R_t$. We first create gates to compute $\text{ans}(Q_{\leq t_i})/\tau$. By Equation (2), we can build a tree of decision-gates on attributes Δ_i , rooted in a gate w_i^τ whose leaves correspond to assignment σ_{Δ_i} for every $\sigma \in R_{t_i}$ such that $\sigma \simeq \tau$. Each leaf is connected to the gate $v_{t_i}^\sigma$ that has been inductively constructed and computes, by induction, $\text{ans}(Q_{\leq t_i})/\sigma$. By Equation (2), w_i^τ computes $\text{ans}(Q_{\leq t_i})/\tau$. Now, we simply define v_t^τ as a Cartesian product gate $\times_{i=1}^k w_i^\tau$, and by Equation (2), it computes $\text{ans}(Q_{\leq t})/\tau$.

To be efficient, observe that if $\tau_1, \tau_2 \in R_t$ and $\tau_1|_{\text{var}(R_{t_i})} = \tau_2|_{\text{var}(R_{t_i})}$, then $\text{ans}(Q_{\leq t_i})/\tau_1 = \text{ans}(Q_{\leq t_i})/\tau_2$, hence we can use $w_i^{\tau_1}$ for both, without duplicating the gates.

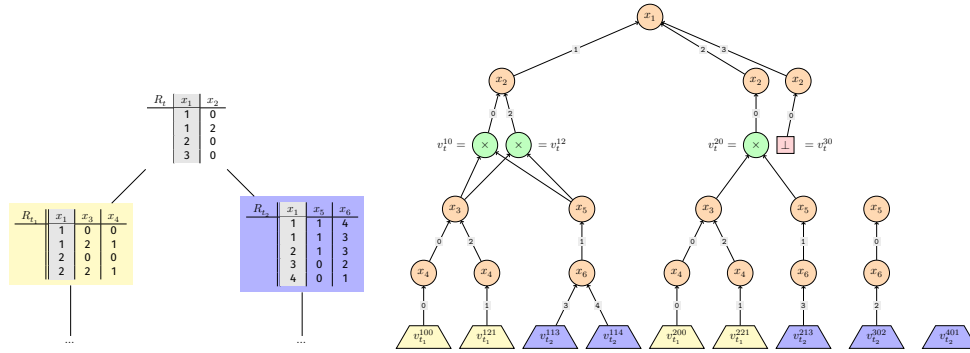
At the root r of \mathcal{T} , we have for every $\tau \in R_r$, a gate v_r^τ computing $\text{ans}(Q)/\tau$. We hence add a tree of decision-gates rooted in a gate v_r whose leaves correspond to τ for every $\tau \in R_r$ and plug it to v_r^τ directly. Gate v_r computes $\text{ans}(Q)$ and we choose it to be the output of C .

► **Example 4.** We illustrate the previous algorithm on the example given on Figure 2. In this example, we assume we have performed the construction in the subtrees rooted in t_1 and t_2 respectively. Hence, we have constructed gates $v_{t_1}^{100}, v_{t_1}^{121}, v_{t_1}^{200}, v_{t_1}^{221}$ and $v_{t_2}^{114}, v_{t_2}^{113}, v_{t_2}^{213}, v_{t_2}^{302}, v_{t_2}^{401}$ corresponding to every tuple of R_{t_1} and R_{t_2} respectively. The only common attribute between R_t and R_{t_1} is x_1 which has been grayed in the figure. Hence, the first step of the construction starts by grouping every tuple in R_{t_1} having the same projection on x_1 and build a decision tree for them. In the example, it means that we group $\langle x_1/1, x_3/0, x_4/0 \rangle$ and $\langle x_1/1, x_3/2, x_4/1 \rangle$ together and hence built a decision tree for the relation $\{\langle x_3/0, x_4/0 \rangle, \langle x_3/2, x_4/1 \rangle\}$. This decision tree is depicted on the bottom left corner of Figure 2. Observe its leaves are connected to $v_{t_1}^{100}$ and $v_{t_1}^{121}$ respectively. The same is done for $\langle x_1/2, x_3/0, x_4/0 \rangle$ and $\langle x_1/2, x_3/2, x_4/1 \rangle$ but we connect the leaves to $v_{t_1}^{200}$ and $v_{t_1}^{221}$. Observe that we built twice the same decision tree because the tuples of R_{t_1} where $x_1 = 1$ projected on x_3, x_4 are the same as the tuples of R_{t_1} where $x_1 = 2$ projected on x_3, x_4 . Yet, we do not identify the subtree because below they may have used x_1 in different ways.

We proceed similarly with R_{t_2} . Observe that in this case, despite having we have 4 distinct values of x_1 in R_{t_2} , we only construct decision-trees for three of them because the case $x_1 = 4$ has no possible extension in R_t .

Now, for each $\tau \in R_t$, we introduce a Cartesian product gate and connect it to the corresponding decision-gates below. For example, for $\tau = \langle x_1/1, x_2/0 \rangle$, we introduce the \times -gate v_t^{10} whose inputs correspond to the roots of the previously constructed decision-tree for tuples of R_{t_1} and R_{t_2} where $x_1 = 1$. Observe in this case that $v_{t_1}^{10}$ and $v_{t_2}^{12}$ have the same inputs, this is where the factorization happens.

Observe that the case $x_1 = 3$ is interesting because it both has corresponding tuples in R_t and R_{t_2} but not in R_{t_1} . In this case, we know that $\text{ans}(Q_{\leq t})/\langle x_1 \mapsto 1 \rangle = \emptyset$ because $R_{t_1}/\langle x_1 \mapsto 1 \rangle = \emptyset$. Hence, we set v_t^{30} to be a \perp -gate and the root of the join tree corresponding to $\langle x_5 \mapsto 0, x_6 \mapsto 2 \rangle$ is dangling in the circuit and will never be used. This is a typical case of the over computation performed by many dynamic programming algorithm: we precompute some values that may have relevance locally but not later in the circuit. The same happens to the tuple where $x_1 = 4$. Neither R_t nor R_{t_1} has such tuples, hence, we have a dangling



■ **Figure 2** Local construction of Yannakakis style compilation algorithm. See Example 4.

tuple again (observe that in this case, we do not even construct the decision-tree). Observe that we can always postprocess the circuit C in $O(|C|) = O(\|Q\|)$ to remove dangling gates and \perp -input propagating constants and removing gates without outputs.

Finally, if t is also the root of the join tree, then it remains to project out the attribute x_1 and x_2 in the circuit, which is done in the last layer of the circuit.

Properties of the circuit. A careful analysis reveals that the number of gates in this circuit is bounded by $2 \cdot \|Q\| + 1$ and can be built in time $O(\text{poly}(\|Q\|) \cdot \|Q\|)$.

Moreover, if we are consistent in the ordering used to built every tree of decision-gates, we can see that the resulting circuit is ordered. It can actually be made to respect any completion to a total order of the following partial order: given $x \in \text{var}(Q)$, let t_x be the last node of \mathcal{T} such that $x \in \text{var}(R_{t_x})$. That is, no ancestor of t_x contain x . Then for $x, y \in \text{var}(Q)$, we write $x \preceq_{\mathcal{T}} y$ if and only if t_y is below t_x in \mathcal{T} . It defines a partial order on $\text{var}(Q)$ that intuitively corresponds to ordering the variables from their order of appearance from the root of \mathcal{T} to its leaves. Let (x_1, \dots, x_n) be an order on $\text{var}(Q)$ completing $\preceq_{\mathcal{T}}$ into a full order. Then, in the construction previously described, we can force the circuit to respect order (x_1, \dots, x_n) . For example, in Example 4, the circuit respects order (x_1, \dots, x_6) , but also $(x_1, x_2, x_5, x_6, x_3, x_4)$.

Beyond acyclic join queries. If a join query Q is not acyclic, we can still use Yannakakis algorithm but with worse guarantees. We first have to generalize the notion of join tree. A *tree decomposition* \mathcal{T} of Q is a tree such that every node of \mathcal{T} is labeled with $B_t \subseteq \text{var}(Q)$ and such that for every atom R of Q , there exists t such that $\text{var}(R) \subseteq B_t$ and such that for every $x \in \text{var}(Q)$, the set $\{t \mid x \in B_t\}$ is connected in \mathcal{T} . Now let Q_t be the join query whose atoms are $R|_{B_t}$ for every atom R of Q such that $\text{var}(R) \cap B_t \neq \emptyset$ and let $\mathcal{R}_t = \text{ans}(Q_t)$. We define Q^* to be the join query whose atom are $\{\mathcal{R}_t \mid t \in \mathcal{T}\}$. By definition, \mathcal{T} is a join tree of Q^* , hence Q^* is acyclic and has the same answers as Q and we can use Yannakakis algorithm to build a $\{\times, \text{dec}\}$ -circuit computing $\text{ans}(Q)$ and of size $O(\|Q^*\|)$. We can always normalize \mathcal{T} to have at most $O(n)$ nodes where $n = |\text{var}(Q)|$, hence $\|Q^*\| = O(n \cdot \max_t |\mathcal{R}_t|)$. Hence, if we know that $|\mathcal{R}_t| \leq U$ and if we can compute \mathcal{R}_t in time $O(U)$, then we can construct a $\{\times, \text{dec}\}$ -circuit computing $\text{ans}(Q)$ in time $\text{poly}(\|Q\|) \cdot |U|$.

The challenge is now to find a relevant upper bound U on $|\mathcal{R}_t| = |\text{ans}(Q_t)|$. The first idea is to bound the number of atoms covering Q_t . Indeed, assume Q_t contains k atoms whose union of variables is equal to $\text{var}(Q_t) = B_t$, then $|\text{ans}(Q_t)| \leq \|Q\|^k$ and we can compute the

join of these k atoms and filter the tuples not in $\text{ans}(Q_t)$ in time $O(\|Q\|^k)$. The number of atoms of Q needed to cover B_t is called the *cover number* $\rho(B_t)$ and the *hypertree width* $\text{htw}(Q, \mathcal{T})$ of Q with respect to \mathcal{T} [27] is defined as $\max_t \rho(B_t)$. From what precedes, we can hence construct a $\{\times, \text{dec}\}$ -circuit computing $\text{ans}(Q)$ in time $\text{poly}(\|Q\|) \cdot \|Q\|^{\text{htw}(Q, \mathcal{T})}$.

We can actually relax the covering notion into a fractional version of it, offering better size guarantees and, using appropriate join algorithms [38, 49, 37, 39, 15], the corresponding time complexity, leading to the more general notion of *fractional hypertree width* $\text{fhtw}(Q, \mathcal{T})$ of Q with respect to \mathcal{T} [28] allowing to construct a $\{\times, \text{dec}\}$ -circuit computing $\text{ans}(Q)$ in time $\text{poly}(\|Q\|) \cdot \|Q\|^{\text{fhtw}(Q, \mathcal{T})}$.

We now explain how to handle a conjunctive query $Q(Z)$. After having compiled a $\{\times, \text{dec}\}$ -circuit for Q , we can project every attributes that are not in Z . This is possible in any $\{\times, \cup\}$ -circuit by replacing, for every $y \notin Z$, every input labeled by y/d with \top . However, doing so comes at the price of loosing determinism, that is, the resulting circuit is now a $\{\times, \cup\}$ -circuit, which does not support model counting nor would it give constant delay enumeration. This is not surprising, as it is known that enumerating or finding the number of answers of acyclic conjunctive queries may not be doable with linear preprocessing [7, 24]. The increased complexity from join to conjunctive queries mostly comes from how the free variables interact with the other. We have a similar phenomenon with ordered $\{\times, \text{dec}\}$ -circuit: indeed, assume that $\text{var}(Q) \setminus Z$ appears at the end of the order, then projecting them preserves determinism, which leads to a smaller ordered $\{\times, \text{dec}\}$ -circuit representing $\text{ans}(Q(Z))$, hence the tractability results of Q transfer to $Q(Z)$.

From the point of view of tree decompositions, this assumption on the order respected by the circuit translates informally into the fact that, in the tree decomposition, the “free variables should be above the projected variables”. This can be formalized as the notion of *free-connex tree decomposition* [7], where there is a connected subset of nodes of \mathcal{T} whose variables are exactly the free-variables. Given a free-connex tree decomposition of fractional hypertree width k of a conjunctive query $Q(Z)$, the previous paragraph shows that we can construct a $\{\times, \text{dec}\}$ -circuit computing $\text{ans}(Q(Z))$ in time $\text{poly}(\|Q\|) \cdot \|Q\|^k$. Another way of seeing this is that from a free-connex tree decomposition, we can construct an acyclic join query Q^* with $\text{ans}(Q^*) = \text{ans}(Q(Z))$, by proceeding similarly as before. The free-connexity ensures that we are able to project out variables in $\text{var}(Q) \setminus Z$.

4.2 Top-down compilation and Exhaustive DPLL

We now explain another algorithm that can be used to construct an ordered $\{\times, \text{dec}\}$ -circuit computing $\text{ans}(Q)$. It can be applied to any query but if Q has fractional hypertree width k , then using the right variable ordering will construct the circuit in time $\text{poly}(\|Q\|) \cdot \|Q\|^k$.

The algorithm is based on a recursive procedure which picks a variable x , add a decision-gate on x , and for every value $d \in D$, recursively computes $\text{ans}(Q)/\langle x/d \rangle$. To avoid blow-up, we add two ingredients: (i) caching, where at each recursive call with parameters (Q, τ) , we detect whether there already exists a gate in the circuit computing $\text{ans}(Q)/\tau$, (ii) connected component analysis, where we detect if $\text{ans}(Q) = \text{ans}(Q_1) \times \dots \times \text{ans}(Q_k)$, in which case, we create a Cartesian product gate and recursively construct circuits for Q_1, \dots, Q_k .

To be efficient, we need to perform caching in a syntactic way only. We formalize it as follows: let τ be a tuple of $Y \subseteq \text{var}(Q)$. If there exists an atom R of Q such that $R/\tau = \emptyset$, then no answer of Q extends τ . In this case, we say that τ is *inconsistent with* Q . Otherwise, if $\text{var}(R) \subseteq Y$, then $\text{ans}(Q)/\tau$ is the same as $\text{ans}(Q')/\tau$ where R has been removed from Q' . We hence denote by $Q/\tau = (Q', \tau|_{Y'})$ where Q' is the join query obtained by removing every atom R in Q such that $\text{var}(R) \subseteq Y$ and $Y' = Y \cap \text{var}(Q')$.

For example, if $Q = R(x_1, x_2)S(x_2, x_3)T(x_3, x_4)$ and $\tau = \langle x_1/1, x_2/2 \rangle \in R$, then $Q/\tau = (S(x_2, x_3)T(x_3, x_4), \langle x_2/2 \rangle)$ because R has been removed from Q and x_1 is no longer a variable in the remaining query. Because of this, instead of writing Q/τ as a couple, we simply write $S(x_2, x_3)T(x_3, x_4)$. Observe that if $\tau' = \langle x_1/2, x_2/2 \rangle \in R$, then $Q/\tau = Q/\tau'$ and $\text{ans}(Q)/\tau = \text{ans}(Q)/\tau'$. This kind of equivalence can be checked purely syntactically, hence in time $\text{poly}(|Q|)$. For this reason, once we have built a gate v in the circuit computing $\text{ans}(Q)/\tau$, we cache v with key Q/τ . Hence, if at some point, another recursive call on (Q', τ') happens and $Q/\tau = Q'/\tau'$, then we can directly pull v from the cache instead of redoing the same computation.

Connected components analysis is also a purely syntactical analysis of the query. Given a join query Q and a tuple τ over variable Y , we consider the graph whose vertices are the atoms of Q/τ and there is an edge between two atoms if they share a variable not in Y (that is, not yet assigned by τ). If this graph has more than one connected component C_1, \dots, C_k , it splits Q into Q_1, \dots, Q_k and τ into τ_1, \dots, τ_k such that $Q = Q_1, \dots, Q_k$, $\tau = \tau_1 \times \dots \times \tau_k$ and $\text{ans}(Q)/\tau = \text{ans}(Q_1)/\tau_1 \times \dots \times \text{ans}(Q_k)/\tau_k$.

We now describe the Exhaustive DPLL algorithm. It takes as input a join query Q' , a tuple τ and an order $\pi = (x_1, \dots, x_n)$ on $\text{var}(Q')$ and returns a gate v in an ordered $\{\times, \text{dec}\}$ -circuit respecting π such that v computes $\text{ans}(Q')/\tau$.

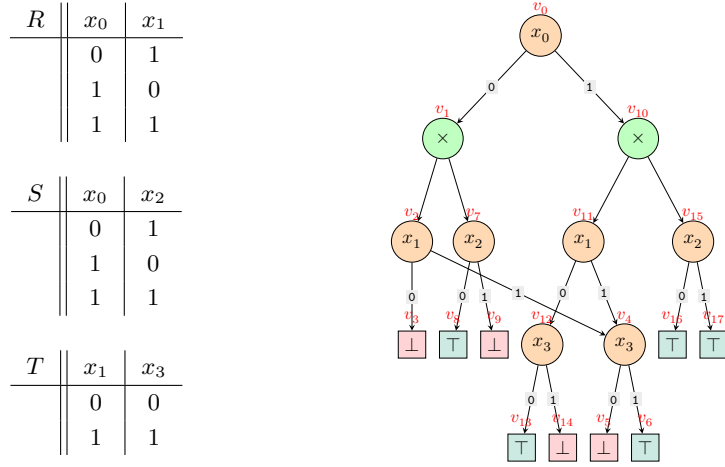
1. If Q' is inconsistent with τ , then return a \perp -gate. It is correct since $\text{ans}(Q')/\tau = \emptyset$.
2. If τ assigns every variable of Q' , then return a \top -gate. It is correct since $\text{ans}(Q')/\tau = \{\langle \rangle\}$.
3. If Q'/τ is mapped in the cache to a gate v , then return v . It is correct since by induction, the cache is correct and hence v computes $\text{ans}(Q')/\tau$.
4. Otherwise, split Q'/τ into connected components $\{Q_1/\tau_1, \dots, Q_k/\tau_k\}$:
 - a. If $k > 1$: create a \times -gate v and for $i = 1, \dots, k$, let $v_i \leftarrow \text{DPLL}(Q_i/\tau_i, \pi)$ which recursively constructs a gate v_i computing $\text{ans}(Q_i)/\tau_i$. Add v_i as an input of v . Now v computes $\times_{i=1}^k \text{ans}(Q_i)/\tau_i = \text{ans}(Q')/\tau$ which is what desired. The algorithm then updates the cache to map Q'/τ with v and returns v .
 - b. If $k = 1$: let x_i be the smallest variable in Q' according to π that is not set by τ . We add a new decision-gate v on variable x_i . For every $d \in D$, we let $v_d \leftarrow \text{DPLL}(Q'/\tau_d, \pi)$ where $\tau_d = \tau \times \langle x_i/d \rangle$ to recursively construct a gate v_d computing $\text{ans}(Q)/\tau_d$ and add an edge labeled by d between v and v_d . By definition, v computes $\bigcup_{d \in D} \text{ans}(Q')/(\tau \times \langle x_i/d \rangle) = \text{ans}(Q')/\tau$. The algorithm updates the cache to map Q'/τ with v and returns v .

On input $(Q, \langle \rangle, \pi)$, Exhaustive DPLL clearly outputs an ordered $\{\times, \text{dec}\}$ -circuit respecting π and computing $\text{ans}(Q)$.

► **Example 5.** We now give an example³. We consider $Q = R(x_0, x_1), S(x_0, x_2), T(x_1, x_3)$ with R, S, T given on Figure 3. With $\pi = (x_0, x_1, x_2, x_3)$, $\text{DPLL}(Q/\langle \rangle, \pi)$ produces the circuit given on Figure 3. Each gate is labeled by the number of recursive call which produced it. For example, v_1 is produced by the first recursive call on input $(Q/\langle x_0/1 \rangle, \pi)$. Observe that $Q/\langle x_0/0 \rangle = R(0, x_1), S(0, x_2), T(x_1, x_3)$ has two connected components: the first one being $R(0, x_1), T(x_1, x_3)$ and the second one being $S(0, x_2)$. Hence a \times -gate is created and linked to the result of the first recursive call producing circuits rooted in v_2 and v_7 .

The edge between v_{11} and v_4 is also interesting because it corresponds to a cache hit. Indeed, the recursive call producing v_{11} is done with parameters $Q_1/\langle x_0/1 \rangle = R(1, x_1), T(x_1, x_3)$. Since there is exactly one connected component, a decision-gate on x_1 is created and a recursive call with input $Q_1/\langle x_0/1, x_1/0 \rangle = T(0, x_3)$ is made, leading to the creation of the circuit

³ An interactive version of this example can be found at <https://florent.capelli.me/algorithms/dpll/> with others.



■ **Figure 3** The circuit produced from Example 5.

rooted at v_{12} . Then, another recursive call is made with input $Q_1/\langle x_0/1, x_1/1 \rangle = T(1, x_3)$ which is the recursive call which produced v_4 since it had parameters $Q_1/\langle x_0/0, x_1/1 \rangle = T(1, x_3)$. Hence, v_4 is directly upon reading the cache after call with parameter Q_2 .

The size of the circuit produced by $\text{DPLL}(Q/\langle \cdot \rangle, \pi)$ mostly depends on π . Interestingly, if Q is acyclic with a join tree \mathcal{T} and π is chosen as for Yannakakis, that is, by ordering the variables from the root of \mathcal{T} to its leaves, then $\text{DPLL}(Q/\langle \cdot \rangle, \pi)$ produces a circuit of size $\text{poly}(|Q|) \cdot \|Q\|$, matching the guarantee of the top-down construction of Yannakakis. This is because once every variable in the root r of \mathcal{T} have been set, it splits Q into as many connected component as r has children in \mathcal{T} . Moreover, if t is a child of \mathcal{T} in r and if two assignments of the variables in r have the same value over the variables in node t , then a cache hit will occur. In a way, we can see Exhaustive DPLL as a mean of constructing a similar circuit as Yannakakis but from the output to the inputs of the circuit, while Yannakakis approach goes from the inputs to the output.

If π is extracted from a tree decomposition \mathcal{T} of Q whose fractional hypertree width is k , then $\text{DPLL}(Q/\tau, \pi)$ produces a circuit of size $\text{poly}(Q) \cdot \|Q\|^k$, see [14] for details, as the analysis is a bit convoluted. Observe that for running $\text{DPLL}(Q/\langle \cdot \rangle, \pi)$, we do not need to know the tree decomposition, this comes handy only in the complexity analysis. In a way, π is the decomposition. We can hence use π to measure how good it is with respect to Q . This actually corresponds to characterization of fractional hypertree width in terms of elimination orders, see [26, 1, 12] for details.

The version of Exhaustive DPLL described above has however an annoying limitation: when branching on a variable x , we brute force over every possible value $d \in D$. Some of these calls will directly return \perp . This leads to a factor of $|D|$ in the time needed to construct the circuit, that is, we construct the circuit in time $|D| \cdot \text{poly}(Q) \cdot \|Q\|^k$ for an order π induced by a decomposition of width k . There are two ways of shaving this factor. We can either use a more involved algorithm to explore only relevant values for variable x , adapting ideas from worst-case optimal join, such that leapfrog join [49]. Another simpler way is to use the binarization technique described in Section 3.3 to reduce the domain to a domain of size 2 while having $|X| \log |D|$ variables, hence leading to a construction in time $\log |D| \cdot \text{poly}(Q) \cdot \|Q\|^k$ as in [14]. We observe however that the binarization technique is also an implicit way of reducing the number of candidates which can also be used in the design of worst-case optimal join algorithm [15].

Finally, we observe that Exhaustive DPLL is flexible: we can handle more than just join queries, as long as we have a recursive way of representing a query whose answer are $\text{ans}(Q)/\tau$ and a caching mechanism. In particular, we have used it in [14] to compute ordered $\{\times, \text{dec}\}$ -circuit representing the answer set of join queries with negated atoms. In this setting, we want to solve a join query but remove answers that satisfy some relations, that we see as negated atoms. It is straightforward to adapt exhaustive DPLL to this kind of queries: we can simply define Q/τ as before, but also removing negated atoms $\neg R$ from Q for which τ cannot be extended to a tuple in R .

5 Fine-grained consequences

We conclude this paper by spelling out the consequences of the tractability of ordered $\{\times, \text{dec}\}$ -circuit from Theorem 2 and the circuit construction from Section 4. In this section, we let $Q(Z)$ be a conjunctive query and \mathcal{T} a free-connex tree decomposition of Q of fractional hypertree width k . We recover the constant delay enumeration of [7] after $\text{poly}(|Q|) \cdot \|Q\|^k$ preprocessing by constructing the circuit for $Q(Z)$ during the preprocessing phase and then enumerating the tuple with delay $O(|Z|)$, which is constant in data complexity. We similarly recover (an extension of) the result from [44] by simply returning $|\text{rel}(C)|$ after having constructed C : the construction of C takes time $\text{poly}(|Q|) \cdot \|Q\|^k$ while computing $|\text{rel}(C)|$ takes $O(|X| \cdot |C|) = \text{poly}(|Q|) \cdot \|Q\|^k$. The direct access algorithm from [16, 12] is also recovered directly since after building the circuit, the access time is $O(\text{poly}(|X|) \log |D|)$, hence, in data complexity, it corresponds to $O(\log |D|)$ from earlier result. DPLL allows to generalize to join queries with negations, see [14].

Finally, we observe that we could have mentioned that the counting algorithm outline in Section 3.2 could be generalized where inputs are labeled with semiring elements and then aggregated over it, allowing to solve FAQ or AJAR types of queries over the circuit with complexity similar to [1, 31], as long as we remain compatible with its underlying order.

References

- 1 Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. Faq: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 13–28, 2016. doi:10.1145/2902251.2902280.
- 2 Antoine Amarilli, Marcelo Arenas, YooJung Choi, Mikaël Monet, Guy Van den Broeck, and Benjie Wang. A circus of circuits: Connections between decision diagrams, circuits, and automata. *CoRR*, abs/2404.09674, 2024. doi:10.48550/arXiv.2404.09674.
- 3 Antoine Amarilli, Pierre Bourhis, Florent Capelli, and Mikaël Monet. Ranked enumeration for MSO on trees via knowledge compilation. In Graham Cormode and Michael Shekelyan, editors, *27th International Conference on Database Theory, ICDT 2024, Paestum, Italy, March 25-28, 2024*, volume 290 of *LIPICs*, pages 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICs.ICDT.2024.25.
- 4 Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, Warsaw, Poland, July 10-14, 2017*, volume 80 of *LIPICs*, pages 111:1–111:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPICs.ICALP.2017.111.
- 5 Antoine Amarilli and Florent Capelli. Tractable circuits in database theory. *SIGMOD Rec.*, 53(2):6–20, 2024. doi:10.1145/3685980.3685982.

- 6 Marcelo Arenas, Pablo Barceló, Leopoldo E. Bertossi, and Mikaël Monet. On the complexity of shap-score-based explanations: Tractability via knowledge compilation and non-approximability results. *J. Mach. Learn. Res.*, 24:63:1–63:58, 2023. URL: <https://jmlr.org/papers/v24/21-0389.html>.
- 7 Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007. doi:10.1007/978-3-540-74915-8_18.
- 8 Guillaume Bagan, Arnaud Durand, Etienne Grandjean, and Frédéric Olive. Computing the jth solution of a first-order query. *RAIRO-Theoretical Informatics and Applications*, 42(1):147–164, 2008. doi:10.1051/ITA:2007046.
- 9 Paul Beame, Jerry Li, Sudeepa Roy, and Dan Suciu. Lower bounds for exact model counting and applications in probabilistic databases. In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*, 2013.
- 10 Paul Beame, Jerry Li, Sudeepa Roy, and Dan Suciu. Counting of query expressions: Limitations of propositional methods. In *Proc. 17th International Conference on Database Theory (ICDT)*, pages 177–188, 2014. doi:10.5441/002/ICDT.2014.20.
- 11 Leopoldo E. Bertossi, Benny Kimelfeld, Ester Livshits, and Mikaël Monet. The shapley value in database management. *SIGMOD Rec.*, 52(2):6–17, 2023. doi:10.1145/3615952.3615954.
- 12 Karl Bringmann, Nofar Carmeli, and Stefan Mengel. Tight fine-grained bounds for direct access on join queries. In *Proceedings of the 41st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 427–436, 2022. doi:10.1145/3517804.3526234.
- 13 Randal E Bryant. Graph-based algorithms for boolean function manipulation. *Computers, IEEE Transactions on*, 100(8):677–691, 1986. doi:10.1109/TC.1986.1676819.
- 14 Florent Capelli and Oliver Irwin. Direct access for conjunctive queries with negations. In Graham Cormode and Michael Shekelyan, editors, *27th International Conference on Database Theory, ICDT 2024, Paestum, Italy, March 25-28, 2024*, volume 290 of *LIPICs*, pages 13:1–13:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPICs.ICDT.2024.13.
- 15 Florent Capelli, Oliver Irwin, and Sylvain Salvati. A simple algorithm for worst case optimal join and sampling. In Sudeepa Roy and Ahmet Kara, editors, *28th International Conference on Database Theory, ICDT 2025, Barcelona, Spain, March 25-28, 2025*, volume 328 of *LIPICs*, pages 23:1–23:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICs.ICDT.2025.23.
- 16 Nofar Carmeli, Nikolaos Tziavelis, Wolfgang Gatterbauer, Benny Kimelfeld, and Mirek Riedewald. Tractable orders for direct access to ranked answers of conjunctive queries. *ACM Transactions on Database Systems*, January 2023. doi:10.1145/3578517.
- 17 Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 393–409, 2020. doi:10.1145/3375395.3387662.
- 18 Ashok K Chandra and Philip M Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, 1977. doi:10.1145/800105.803397.
- 19 Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970. doi:10.1145/362384.362685.
- 20 Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.
- 21 A. Darwiche. Decomposable negation normal form. *J. ACM*, 48(4):608–647, 2001. doi:10.1145/502090.502091.

- 22 Adnan 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):11–34, 2001. doi:10.3166/JANCL.11.11–34.
- 23 Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002. doi:10.1613/JAIR.989.
- 24 Arnaud Durand and Stefan Mengel. Structural tractability of counting of solutions to conjunctive queries. *Theory Comput. Syst.*, 57(4):1202–1249, 2015. doi:10.1007/S00224-014-9543-Y.
- 25 Idan Eldar, Nofar Carmeli, and Benny Kimelfeld. Direct access for answers to conjunctive queries with aggregation. *arXiv preprint arXiv:2303.05327*, 2023. doi:10.48550/arXiv.2303.05327.
- 26 Johannes K Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An smt approach to fractional hypertree width. In *Principles and Practice of Constraint Programming: 24th International Conference, Lille, France, August 27-31, 2018, Proceedings 24*, pages 109–127. Springer, 2018.
- 27 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 21–32. ACM, 1999. doi:10.1145/303976.303979.
- 28 Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms (TALG)*, 11(1):4, 2014.
- 29 David Harvey and Joris Van Der Hoeven. Integer multiplication in time $o(n \log n)$. *Annals of Mathematics*, 193(2):563–617, 2021.
- 30 Jinbo Huang and Adnan Darwiche. DPLL with a trace: From SAT to knowledge compilation. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 156–162, 2005. URL: <http://ijcai.org/Proceedings/05/Papers/0876.pdf>.
- 31 Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 91–106, 2016. doi:10.1145/2902251.2902293.
- 32 Rafael Kiesel and Thomas Eiter. Knowledge compilation and more with sharpsat-td. In Pierre Marquis, Tran Cao Son, and Gabriele Kern-Isberner, editors, *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023, Rhodes, Greece, September 2-8, 2023*, pages 406–416, 2023. doi:10.24963/KR.2023/40.
- 33 Benny Kimelfeld, Wim Martens, and Matthias Niewerth. A formal language perspective on factorized representations. In Sudeepa Roy and Ahmet Kara, editors, *28th International Conference on Database Theory, ICDT 2025, Barcelona, Spain, March 25-28, 2025*, volume 328 of *LIPICs*, pages 20:1–20:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2025. doi:10.4230/LIPICs.ICDT.2025.20.
- 34 Tuukka Korhonen and Matti Järvisalo. Integrating Tree Decompositions into Decision Heuristics of Propositional Model Counters. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, volume 210 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 8:1–8:11, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPICs.CP.2021.8.
- 35 Jean-Marie Lagniez and Pierre Marquis. An improved decision-dnnf compiler. In Carles Sierra, editor, *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, pages 667–673. ijcai.org, 2017. doi:10.24963/IJCAI.2017/93.
- 36 Kuldeep S. Meel and Alexis de Colnet. #cfg and #dnnf admit FPRAS. *CoRR*, abs/2406.18224, 2024. doi:10.48550/arXiv.2406.18224.
- 37 Hung Q. Ngo. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 111–124. ACM, 2018. doi:10.1145/3196959.3196990.

- 38 Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48. ACM, 2012. doi:10.1145/2213556.2213565.
- 39 Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):1–40, 2018. doi:10.1145/3180143.
- 40 Dan Olteanu. Factorized databases: A knowledge compilation perspective. In Adnan Darwiche, editor, *Beyond NP, Papers from the 2016 AAAI Workshop, Phoenix, Arizona, USA, February 12, 2016*, volume WS-16-05 of *AAAI Technical Report*. AAAI Press, 2016. URL: <http://www.aaai.org/ocs/index.php/WS/AAAIW16/paper/view/12638>.
- 41 Dan Olteanu and Jiewen Huang. Using obdds for efficient query evaluation on probabilistic databases. In *International Conference on Scalable Uncertainty Management*, pages 326–340. Springer, 2008. doi:10.1007/978-3-540-87993-0_26.
- 42 Dan Olteanu and Jakub Zavodny. Factorised representations of query results: size bounds and readability. In Alin Deutsch, editor, *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, pages 285–298. ACM, 2012. doi:10.1145/2274576.2274607.
- 43 Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *ACM Transactions on Database Systems*, 40(1):1–44, March 2015. doi:10.1145/2656335.
- 44 Reinhard Pichler and Sebastian Skritek. Tractable counting of the answers to conjunctive queries. *Journal of Computer and System Sciences*, 79(6):984–1001, September 2013. doi:10.1016/J.JCSS.2013.01.012.
- 45 Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. *Theory and Applications of Satisfiability Testing*, 4:7th, 2004.
- 46 Andy Shih, Guy Van den Broeck, Paul Beame, and Antoine Amarilli. Smoothing structured decomposable circuits. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d’Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 11412–11422, 2019. URL: <https://proceedings.neurips.cc/paper/2019/hash/940392f5f32a7ade1cc201767cf83e31-Abstract.html>.
- 47 Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, July 1984. doi:10.1137/0213035.
- 48 Timothy van Bremen and Kuldeep S Meel. Probabilistic query evaluation: The combined fpras landscape. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 339–347, 2023. doi:10.1145/3584372.3588677.
- 49 Todd Veldhuizen. Triejoin: A Simple, Worst-Case Optimal Join Algorithm. *Proceedings of the 17th International Conference on Database Theory (ICDT), Athens, Greece, 2014*, 17(13):96–106, 2014. doi:10.5441/002/ICDT.2014.13.
- 50 Mihalis Yannakakis. Algorithms for Acyclic Database Schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, pages 82–94. VLDB Endowment, 1981.