Direct Access for Conjunctive Queries with Negations

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

Oliver Irwin
Université de Lille, CNRS, Inria, UMR 9189 - CRIStAL, F-59000 Lille, France

Abstract

Given a conjunctive query Q and a database $\mathbf{D}$, a direct access to the answers of Q over $\mathbf{D}$ is the operation of returning, given an index j, the jth answer for some order on its answers. While this problem is #P-hard in general with respect to combined complexity, many conjunctive queries have an underlying structure that allows for a direct access to their answers for some lexicographical ordering that takes polylogarithmic time in the size of the database after a polynomial time precomputation. Previous work has precisely characterised the tractable classes and given fine-grained lower bounds on the precomputation time needed depending on the structure of the query. In this paper, we generalise these tractability results to the case of signed conjunctive queries, that is, conjunctive queries that may contain negative atoms. Our technique is based on a class of circuits that can represent relational data. We first show that this class supports tractable direct access after a polynomial time preprocessing. We then give bounds on the size of the circuit needed to represent the answer set of signed conjunctive queries depending on their structure. Both results combined together allow us to prove the tractability of direct access for a large class of conjunctive queries. On the one hand, we recover the known tractable classes from the literature in the case of positive conjunctive queries. On the other hand, we generalise and unify known tractability results about negative conjunctive queries – that is, queries having only negated atoms. In particular, we show that the class of β-acyclic negative conjunctive queries and the class of bounded nest set width negative conjunctive queries admit tractable direct access.

2012 ACM Subject Classification Information systems → Relational database model

Keywords and phrases Conjunctive queries, factorised databases, direct access, hypertree decomposition

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

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

Acknowledgements I want to thank ...

1 Introduction

The *direct access task*, given a database query Q and a database $\mathbf{D}$, is the problem of outputing on input k, the k-th answer of Q over $\mathbf{D}$ or an error when k is greater than the number of answers of Q, where some order on $[\![Q]\!]^{\mathbf{D}}$, the answers of Q over $\mathbf{D}$, is assumed. This task has been introduced by Bagan, Durand, Grandjean and Olive in [2] and is very natural in the context of databases. It can be used as a building block for many other interesting tasks such as counting, enumerating [2] or sampling without repetition [13, 22] the answers of Q. Of course, if one has access to an ordered array containing $[\![Q]\!]^{\mathbf{D}}$, answering direct access tasks simply consists in reading the right entry of the array. However, building such an array is often expensive, especially when the number of answers of Q is large. Hence, a natural approach for solving this problem is to simulate this method by using a data structure to represent $[\![Q]\!]^{\mathbf{D}}$ that still allows for efficient direct access tasks to be solved but

© Florent Capelli and Oliver Irwin;
licensed under Creative Commons License CC-BY 4.0

that is cheaper to compute than the complete answer set. This approach is thus separated in two phases: a *preprocessing phase* where the datastructure is constructed followed by a phase where direct access tasks are solved. To measure the quality of an algorithm for solving direct access tasks, we hence separate the *preprocessing time* – that is the time needed for the preprocessing phase – and the *access time*, that is, the time needed to answer one direct access query after the preprocessing. For example, the approach consisting in building an indexed array for $[\![Q]\!]^{\mathbf{D}}$ has a preprocessing time in at least the size of $[\![Q]\!]^{\mathbf{D}}$ (and much higher in practice) and constant access time. While the access time is optimal in this case, the cost of preprocessing is often too high to pay in practice.

Previous work has consequently focused on devising methods with better preprocessing time while offering reasonable access time. In their seminal work [2], Bagan, Durand, Grandjean and Olive give an algorithm for solving direct access tasks with linear precomputation time and constant access time on a class of first order logic formulas and bounded degree databases. Bagan [1] later studied the problem for monadic second order formulas over bounded treewidth databases. Another line of research has been to study classes of conjunctive queries that support efficient direct access. In [13], Carmeli, Zeevi, Berkholz, Kimelfeld, and Schweikardt prove that direct access tasks can be solved on acyclic conjunctive queries with linear preprocessing time and polylogarithmic access time for a well-chosen lexicographical order. The results are also generalised to the case of bounded fractional hypertree width queries, a number measuring how far a conjunctive query is from being acyclic. It generalises many results from the seminal paper by Yannakakis establishing the tractability of model checking on acyclic conjunctive queries [35] to the tractability of counting the number of answers of conjunctive queries [31] having bounded hypertree width. This result was later improved by precisely characterising the lexicographical ordering allowing for this kind of complexity guarantees. Fine-grained characterisation of the complexity of answering direct access tasks on conjunctive queries, whose answers are assumed to be ordered using some lexographical order, has been given by Carmeli, Tziavelis, Gatterbauer, Kimelfeld and Riedewald in [12] for the special case of acyclic queries and by Bringmann, Carmeli and Mengel in [8] for the general case. More recently, Eldar, Carmeli and Kimelfeld [15] studied the complexity of solving direct access tasks for conjunctive queries with aggregation.

In this paper, we devise new methods for solving direct access tasks on the answer set of *signed conjunctive queries*, that is, conjunctive queries that may contain negated atoms. This is particularly challenging because only a few tractability results are known on signed conjunctive queries. The model checking problem for signed conjunctive queries being NP-hard on acyclic conjunctive queries with respect to combined complexity, it is not possible to directly build on the work cited in the last paragraph. Two classes of negative conjunctive queries (that is, conjunctive queries where every atom is negated) have been shown so far to support efficient model checking: the class of β-acyclic queries [30, 5] and the class of bounded nested-set width queries [25]. The former has been shown to also support efficient (weighted) counting [7, 11]. Our main contribution is a generalisation of these results to direct access tasks. More precisely, we give an algorithm that efficiently solves direct access tasks on a large class of signed conjunctive queries, which contains in particular β-acyclic negative conjunctive queries, bounded nest-width negative conjunctive queries and bounded fractional hypertree width positive conjunctive query. For the latter case, the complexity we obtain is similar to the one presented in [8] and we also get complexity guarantees depending on a lexicographical ordering that can be specified by the user. Hence our result both improves the understanding of the tractability of signed conjunctive queries and unify the existing results with the positive case. In a nutshell, we prove that the complexity of solving direct

access tasks for a lexicographical order of a signed conjunctive query Q roughly matches the complexity proven in [8] for the worst positive query we could construct by removing some negative atoms of Q and turning the others to positive atoms. It is not surprising since one could simulate such a query by choosing a database where some negated atoms are associated with empty relations and therefore making them virtually useless in the query. However, this result is not trivial to obtain and necessitates the introduction of new tools to handle negated atoms.

As a byproduct, we introduce a new notion of hypergraph width based on elimination order, the β-hyperorder width, that is hereditary – in the sense that the width of every subhypergraph does not exceed the width of the original hypergraph – which makes it particularly well tailored for the study of the tractability of negative conjunctive queries. We show that this notion sits between nest-set width and β-hypertree width [18], but does not suffer from the main drawback of working with β-hypertree width: our width notion is based on a decomposition that works for every subhypergraph.

Our method is based on a two-step preprocessing. Given a signed conjunctive query Q, a database $\mathbf{D}$ and an order $\prec$ on its variables, we start by constructing a circuit which represents $[\![Q]\!]^{\mathbf{D}}$ in a factorised way, enjoying interesting syntactical properties. The size of this circuit depends on the complexity of the order $\prec$ chosen on the variables of Q. We then show that with a second light preprocessing on the circuit itself, we can answer direct access tasks on the circuit in time $\mathsf{poly}(n)\mathsf{polylog}(D)$ where n is the number of variables of Q and D is the domain of $\mathbf{D}$. This approach is akin to the approach used in *factorised databases*, introduced by Olteanu and Závodný [27], a fruitful approach allowing efficient management of the answer sets of a query by working directly on a factorised representation of the answer set instead of working on the query itself [26, 33, 3, 28]. However, the restrictions that we are considering in this paper are different from the ones used in previous work since we need to somehow account for the variable ordering in the circuit itself. The syntactic restrictions we use have already been considered in [11] where they are useful to deal with β-acyclic CNF formulas.

Organisation of the paper. The paper is organised as follows: Section 2 introduces the notations and concepts necessary to understand the paper. We then present the family of circuits we use to represent database relations and the direct access algorithm in Section 3. Section 4 presents the algorithm used to construct a circuit representing $[\![Q]\!]^{\mathbf{D}}$ from a join query Q (that is a conjunctive query without existential quantifiers) and a database $\mathbf{D}$. Upper bounds on the size of the circuits produced are given in Section 4.3 using hypergraph decompositions defined in Section 4.2. Finally Section 5 explicitly states the results we obtain by combining both techniques together, explain how one can go from join query to conjunctive query by existentially projecting variables directly in the circuit and makes connections with the existing literature.

2 Preliminaries

General mathematical notations. Given $n \in \mathbb{N}$, we denote by $[n]$ the set $\{0, \dots, n\}$. When writing down complexity, we use the notation $\mathsf{poly}(n)$ to denote that the complexity is polynomial in n, $\mathsf{poly}_k(n)$ to denote that the complexity is polynomial in n when k is considered a constant (in other words, the coefficients and the degree of the polynomial may depend on k) and $\mathsf{polylog}(n)$ to denote that the complexity is polynomial in $\mathsf{log}(n)$. Moreover, we use the shortcut $\tilde{O}(N)$ to indicate that polylogarithmic factors are ignored, that is, the

complexity is $O(N\mathsf{polylog}(N))$.

Tuples and relations. Let D and X be finite sets. A (named) *tuple* on domain D and variables X is a mapping from X to D. We denote by D^X the set of all tuples on domain D and variables X. A *relation R on domain D and variables X* is a subset of tuples, that is, $R \subseteq D^X$. Given an tuple $\tau \in D^X$ and $Y \subseteq X$, we denote by $\tau|_Y$ the tuple on domain D and variable Y such that $\tau|_Y(y) = \tau(y)$ for every $y \in Y$. Given a variable $x \in X$ and $d \in D$, we denote by $[x \leftarrow d]$ the tuple on variables $\{x\}$ that assigns the value $d \in D$ to x. We denote by $\langle\rangle$ the empty tuple, that is, the only element of $D^\emptyset$. Given two tuples $\tau_1 \in D^{X_1}$ and $\tau_2 \in D^{X_2}$, we say that τ_1 and τ_2 are *compatible*, denoted by $\tau_1 \simeq \tau_2$, if $\tau_1|_{X_1\cap X_2} = \tau_2|_{X_1\cap X_2}$. In this case, we write $\tau_1 \bowtie \tau_2$ the tuple on domain D and variables $X_1 \cup X_2$ defined as

$$(\tau_1 \bowtie \tau_2)(x) = \begin{cases} \tau_1(x) \text{ if } x \in X_1 \\ \tau_2(x) \text{ if } x \in X_2 \end{cases}$$

If $X_1 \cap X_2 = \emptyset$, we write $\tau_1 \times \tau_2$. The *join $R_1 \bowtie R_2$ of R_1 and R_2*, for two relations R_1, R_2 on domain D and variables X_1, X_2 respectively, is defined as $\{\tau_1 \bowtie \tau_2 \mid \tau_1 \in R_1, \tau_2 \in R_2, \tau_1 \simeq \tau_2\}$. Observe that if $X_1 \cap X_2 = \emptyset$, $R_1 \bowtie R_2$ is simply the *cartesian product of R_1 and R_2*. In this case, we denote it by $R_1 \times R_2$. The *extended union of R_1 and R_2*, denoted by $R_1 \mathbin{\overline{\cup}} R_2$, is the relation on domain D and variables $X_1 \cup X_2$ defined as $(R_1 \times D^{X_2\setminus X_1}) \cup (R_2 \times D^{X_1 \setminus X_2})$. When $X_1 = X_2$, the extended union of R_1 and R_2 is simply $R_1 \cup R_2$, that is, the set of tuples over X_1 that are either in R_1 or in R_2.

Let $R \subseteq D^X$ be a relation from a set of variables X to a domain D. We denote $\sigma_F(R)$ as the subset of R where the formula F is true. Throughout the paper, we will assume that both the domain D and the variable set X are ordered. The order on D will be denoted as $<$ and the order on X as $\prec$ and we will often write $D = \{d_1, \dots, d_p\}$ with $d_1 < \dots < d_p$ and $X = \{x_1, \dots, x_n\}$ with $x_1 \prec \dots \prec x_n$. Given $d \in D$, we denote by $\mathsf{rank}(d)$ the number of elements of D that are smaller or equal to d. Both $<$ and $\prec$ induce a lexicographical order $\prec_{\mathsf{lex}}$ on D^X defined as $\tau \prec_{\mathsf{lex}} \tau'$ if there exists $x \in X$ such that for every $y \prec x$, $\tau(y) = \tau'(y)$ and $\tau(x) < \tau'(x)$. Given a integer $k \leqslant \#R$, we denote by $R[k]$ the kth tuple in R for the $\prec_{\mathsf{lex}}$-order.

We will often use the following observation:

▶ **Lemma 1.** *Let $\tau = R[k]$ and $x = \mathsf{min}(\mathsf{var}(R))$. Then $\tau(x) = \mathsf{min}\{d \mid \#\sigma_{x\leqslant d}(R) \geqslant k\}$. Moreover, $\tau = R'[k']$, where $R' = \sigma_{x=d}(R)$ is the subset of R where x is equal to d and $k' = k - \#\sigma_{x<d}(R)$.*

Proof. Let $A = \{d \mid \#\sigma_{x\leqslant d}(R) \geqslant k\}$.

We start by showing that $\tau(x) \in A$, meaning $\#\sigma_{x\leqslant\tau(x)} \geqslant k$. Let $\alpha \preceq_{\mathsf{lex}} \tau$. Since x is the smallest variable, it follows that $\alpha \in \sigma_{x\leqslant\tau(x)}(R)$ as $\alpha(x) \leqslant \tau(x)$. Since there exists exactly k such assignments α (by definition of τ which is the kth tuple of R), we have $\#\sigma_{x\leqslant\tau(x)}(R) \geqslant k$.

We now show that, given a value $d' < \tau(x)$, $d' \notin A$ and as such that $\tau(x)$ is indeed the smallest value in A. Let $\alpha \in \sigma_{x\leqslant d'}$. It follows that $\alpha(x) \leqslant \tau(x)$, and therefore that $\alpha < \tau$. We therefore have that $\sigma_{x\leqslant d'}(R) \subset \{\alpha \mid \alpha \prec_{\mathsf{lex}} \tau\}$. By definition of τ as the kth tuple, the latter set has less than $k-1$ elements. Hence $d' \notin A$. This implies that for any $d \in A, \tau(x) \leqslant d$.

This shows that $\tau(x)$ is indeed the smallest value d such that there exists at least k tuples α where $\alpha(x) \leqslant d$.

The second part of the lemma follows from the following observation: when assigning a value d to the variable x, one actually *eliminates* a certain number of tuples from the initial set. Specifically, the tuples that assign a different value to x.

By definition, k is the cardinal of the set $\{\tau' \mid \tau' \preceq_{\mathsf{lex}} \tau\}$. This set can be written as the disjoint union of the set of tuples where $\tau'(x) < d$ (which are all smaller than τ) and the set of tuples smaller than τ where $\tau'(x) = d$. We therefore have $k = \#\{\tau \mid \tau(x) < d\} + \#\{\tau' \mid \tau' \prec_{\mathsf{lex}} \tau, \tau'(x) = d\}$. By definition, the first set is $\sigma_{x<d}(R)$. The second part of the sum is exactly the index of the tuple in the subset of R where $\tau(x) = d$. We can rewrite the sum as $k = \#\sigma_{x<d}(R) + k'$, implying $k' = k - \#\sigma_{x<d}(R)$. A visual representation of this index transformation can be found in Figure 1. ◀

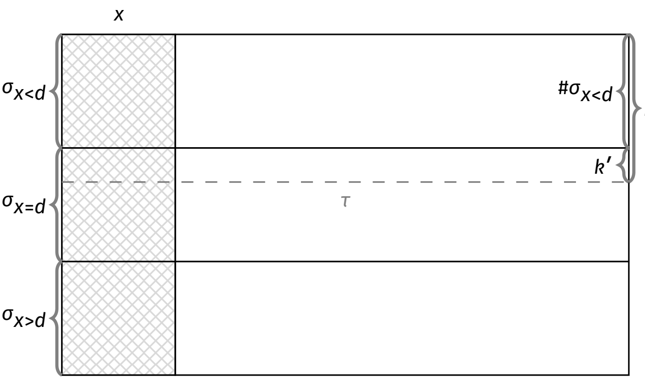


Figure 1 Representation of the link between k and k'

Queries. A *(signed) join query* Q is an expression of the form

$$Q \coloneqq R_1(\mathbf{x_1}), \dots, R_m(\mathbf{x_\ell}), \neg S_{\ell+1}(\mathbf{x_{\ell+1}}), \dots, \neg S_m(\mathbf{x_m})$$

where each R_i and S_j are relation symbols and $\mathbf{x_i}$ are tuples of variables in X. In this paper, we consider *self-join free* queries, that is, we assume that any relation symbol appears at most once in each query. Elements of the form $R_i(\mathbf{x_i})$ are called positive atoms and elements of the form $S_j(\mathbf{x_j})$ are called negative atoms. The set of variables of Q is denoted by $\mathsf{var}(Q)$, the set of positive (resp. negative) atoms of Q is denoted by $\mathsf{atoms}^+(Q)$ (resp. $\mathsf{atoms}^-(Q)$). A *positive join query* is a signed join query without negative atoms. A *negative join query* is a join query without positive atoms. The size $|Q|$ of Q is defined as $\sum_{i=1}^m |\mathbf{x_i}|$, where $|\mathbf{x}|$ denotes the number of variables in $\mathbf{x}$. A *database* $\mathbf{D}$ for Q is an ordered finite set D called the *domain* together with a set of relations $R_i^{\mathbf{D}} \subseteq D^{a_i}$, $S_j^{\mathbf{D}} \subseteq D^{a_j}$ such that $a_i = |\mathbf{x_i}|$. The *answers of Q over* $\mathbf{D}$ is the relation $[\![Q]\!]^{\mathbf{D}} \subseteq D^{\mathsf{var}(Q)}$ defined as the set of $\sigma \in D^X$ such that for every $i \leqslant m$, $\sigma(\mathbf{x_i}) \in R_i^{\mathbf{D}}$ and $\sigma(\mathbf{x_i}) \notin S_i^{\mathbf{D}}$. The size $|\mathbf{D}|$ of the database $\mathbf{D}$ is defined to be the total number of tuples in it plus the size of its domain[1], that is, $|D| + \sum_{i=1}^{\ell} |R_i^{\mathbf{D}}| + \sum_{i=\ell+1}^{m} |S_j^{\mathbf{D}}|$.

A *signed conjunctive query* $Q(Y)$ is a join query Q together with $Y \subseteq \mathsf{var}(Q)$, called the *free variables of* Q and denoted by $\mathsf{free}(Q)$. The answers $[\![Q(Y)]\!]^{\mathbf{D}}$ of a conjunctive query Q over a database D are defined as $[\![Q]\!]^{\mathbf{D}}|_Y$, that is, they are the projection over Y of answers of Q.

[1] We follow the definition of [25] concerning the size of the database. Adding the size of the domain here is essential since we are dealing with negative atoms. Hence a query may have answers even when the database is empty, for example the query $Q = \neg R(x)$ with $R^{\mathbf{D}} = \emptyset$ has $|D|$ answers.

Direct Access tasks. Given a query Q, a database instance $\mathbf{D}$ on ordered domain D and a total order $\prec$ on the variables of Q, a *Direct Access task* [12] is the problem of returning, on input k, the k-th tuple $[\![Q]\!]^{\mathbf{D}}[k]$ for the order $\prec_{\mathsf{lex}}$ if $k < \#[\![Q]\!]^{\mathbf{D}}$ and fails otherwise. We are interested in answering Direct Access tasks using the same setting as [12]: we allow a *precomputation* phase during which a data structure is constructed, followed by an *access* phase. Our goal is to obtain – with a precomputation time that is polynomial in the size of $\mathbf{D}$ – a data structure that can be used to answer any access query in polylogarithmic time in the size of $\mathbf{D}$.

Hypergraphs and Signed Hypergraphs. A *hypergraph* $H = (V, E)$ is defined as a set of *vertices* V and *hyperedges* $E \subseteq 2^V$, that is, a hyperedge $e \in E$ is a subset of V. A *signed hypergraph* $H = (V, E_+, E_-)$ is defined as a set of *vertices* V, *positive edges* $E_+ \subseteq 2^V$ and *negative edges* $E_- \subseteq 2^V$. The *signed hypergraph* $H(Q) = (\mathsf{var}(Q), E_+, E_-)$ *of a signed conjunctive query* $Q(Y)$ is defined as the signed hypergraph whose vertex set is the variables of Q and such that $E_+ = \{\mathsf{var}(a) \mid a \text{ is a positive atom of } Q\}$ and $E_- = \{\mathsf{var}(a) \mid a \text{ is a negative atom of } Q\}$. We observe that when Q is a positive query, $H(Q)$ corresponds to the usual definition of the hypergraph of a conjunctive query since $E_- = \emptyset$.

Let $H = (V, E)$ be a hypergraph. A *subhypergraph* H' of H, denoted by $H' \subseteq H$ is a hypergraph of the form (V, E') with $E' \subseteq E$. In other word, a subhypergraph of H is a hypergraph obtained by removing edges in H. For $S \subseteq V$, we denote by $H \setminus S$ the hypergraph $(V \setminus S, E')$ where $E' = \{e \setminus S \mid e \in E\}$. Given $v \in V$, we denote by $E(v) = \{e \in E \mid v \in e\}$ the set of edges containing v, by $N_H(v) = \bigcup_{e \in E(v)} e$ the *neighborhood of* v *in* H and by $N^*_H(v) = N_H(v) \setminus \{v\}$ the *open neighborhood of* v. We will be interested in the following vertex removal operation on H: given a vertex v of H, we denote by $H/v = (V \setminus \{v\}, E/v)$ where E/v is defined as $\{e \setminus \{v\} \mid e \in E\} \setminus \{\emptyset\} \cup \{N^*_H(v)\}$, that is, H/v is obtained from H by removing v from every edges of H and by adding a new edge that contains the open neighborhood of v.

Given $S \subseteq V$ and $E \subseteq 2^V$, a *covering of* S *with* E is a subset $F \subseteq E$ such that $S \subseteq \bigcup_{e \in F} e$. The *cover number* $cn(S, E)$ *of* S *wrt* E is defined as the minimal size of a covering of S with E, that is, $cn(S, E) = \min\{|F| \mid F \text{ is a covering of } S \text{ with } E\}$. A *fractional covering of* S *with* E is a function $c : E \to \mathbb{R}_+$ such that for every $s \in S$, $\sum_{e \in E(s)} c(e) \geqslant 1$. Observe that a covering is a fractional covering where c has values in $\{0, 1\}$. The *fractional cover number* $fcn(S, E)$ *of* S *wrt* E is defined as the minimal size of a fractional covering of S with E, that is, $fcn(S, E) = \min\{\sum_{e \in E} c(e) \mid c \text{ is a fractional covering of } S \text{ with } E\}$. Fractional covers are particularly interesting because of the following theorem by Grohe and Marx:

▶ **Theorem 2** ([19]). *Let Q be a join query and λ be the fractional cover number of* $\mathsf{var}(Q)$*. Then for every database* $\mathbf{D}$*,* $[\![Q]\!]^{\mathbf{D}}$ *has size at most* $|\mathbf{D}|^{\lambda}$*.*

3 Ordered relational circuits

In this section, we introduce a data structure that can be used to succinctly represent relations. This data structure is an example of factorised representation, such as d-representations [29], but does not need to be structured along a tree, which will allow us to handle more queries, and especially queries with negative atoms – for example β-acyclic signed conjunctive queries, a class of queries that cannot be represented by polynomial size d-representations [11, Theorem 9].

251 ## 3.1 Definitions

252 **Relational circuits.** A $\{\bowtie, \mathsf{dec}\}$-circuit C on variables $X = \{x_1, \dots, x_n\}$ and domain D is a
253 multi-directed acyclic graph[2] with one distinguished gate $\mathsf{out}(C)$ called the *output* of C. The
254 circuit is labelled as follows:

255 - every gate of C with no ingoing edge, called an *input of* C, is labelled by either 0 or 1;
256 - a gate v labelled by a variable $x \in X$ is called a *decision gate*. Each ingoing edge e of v
257 is labelled by a value $d \in D$ and for each $d \in D$, there is at most one ingoing edge of v
258 labelled by d. This implies that a decision gate has at most $|D|$ outgoing edges; and
259 - every other gate is labelled by $\bowtie$.

260 The set of all the decision gates in a circuit C is denoted by $\mathsf{decision}(C)$. Given a gate v of
261 C, we denote by C_v the *subcircuit of* C *rooted in* v to be the circuit whose gates are the gates
262 reachable from v by following a directed path in C. We define the *variable set of* v, denoted
263 by $\mathsf{var}(v) \subseteq X$, to be the set of variables x labelling a decision gate in C_v. The variable
264 evaluated by a decision gate v is denoted by $\mathsf{decvar}(v)$. The *size* $|C|$ of a $\{\bowtie, \mathsf{dec}\}$-circuit is
265 defined to be the number of edges of its underlying DAG.

266 We define the *relation* $\mathsf{rel}(v) \subseteq D^{\mathsf{var}(v)}$ *computed at gate* v inductively as follows: if v is an
267 input labelled by 0, then $\mathsf{rel}(v) = \emptyset$. If v is an input labelled by 1, then $\mathsf{rel}(v) = D^{\emptyset}$, that is,
268 $\mathsf{rel}(v)$ is the relation containing only the empty tuple. Otherwise, let $v_1, \dots, v_k$ be the inputs
269 of v. If v is a $\bowtie$-gate, then $\mathsf{rel}(v)$ is defined to be $\mathsf{rel}(v_1) \bowtie \dots \bowtie \mathsf{rel}(v_k)$. If v is a decision
270 gate labelled by a variable x, $\mathsf{rel}(v) = \big([x \leftarrow \ell(e_1)] \bowtie \mathsf{rel}(v_1)\big) \bar{\cup} \dots \bar{\cup} \big([x \leftarrow \ell(e_k)] \bowtie \mathsf{rel}(v_k)\big)$
271 where e_i is the incoming edge (v_i, v). It is readily verified that $\mathsf{rel}(v)$ is a relation on domain
272 D and variables $\mathsf{var}(v)$. The *relation computed by* C *over a set of variables* X (assuming
273 $\mathsf{var}(C) \subseteq X$), denoted by $\mathsf{rel}_X(C)$, is defined to be $\mathsf{rel}(\mathsf{out}(C)) \times D^{X \setminus \mathsf{var}(\mathsf{out}(C))}$.

274 To ease notation, we use the following convention: if v is a decision-gate and $d \in D$, we
275 denote by v_d the gate of C that is connected to v by an edge labeled by d.

276 Deciding whether the relation computed by a $\{\bowtie, \mathsf{dec}\}$-circuit is non-empty is NP-complete
277 by a straightforward reduction to model checking of conjunctive queries [14]. Such circuits are
278 hence of little use to get tractability results. We are therefore more interested in the following
279 restriction of $\{\bowtie, \mathsf{dec}\}$-circuits: a $\{\times, \mathsf{dec}\}$-circuit C is a $\{\bowtie, \mathsf{dec}\}$-circuit such that: (i) for
280 every $\bowtie$-gate v of C with inputs $v_1, \dots, v_k$ and $i < j \leqslant k$, it holds that $\mathsf{var}(v_i) \cap \mathsf{var}(v_j) = \emptyset$,
281 (ii) for every decision gate v of C labelled by x with inputs $v_1, \dots, v_k$ and $i \leqslant k$, it holds that
282 $x \notin \mathsf{var}(v_i)$. Checking whether the relation computed by a $\{\times, \mathsf{dec}\}$-circuit C is non-empty
283 can be done in time $O(|C|)$ by a dynamic programming algorithm propagating in a bottom-up
284 fashion whether $\mathsf{rel}(v)$ is empty. Similarly, given a $\{\times, \mathsf{dec}\}$-circuit C, one can compute the
285 size of $\mathsf{rel}(C)$ in polynomial time in $|C|$ by a dynamic programming algorithm propagating
286 in a bottom-up fashion $|\mathsf{rel}(v)|$.

287 **Ordered Relational Circuits.** Let X be a set of variables and $\prec$ an order on X. We say
288 that a $\{\times, \mathsf{dec}\}$-circuit C on domain D and variables X is a $\prec$*-ordered* $\{\times, \mathsf{dec}\}$*-circuit* if
289 for every decision gate v of C labelled with $x \in X$, it holds that for every $y \in \mathsf{var}(v) \setminus \{x\}$,
290 $x \prec y$. We simply say that a circuit C is an ordered $\{\times, \mathsf{dec}\}$-circuit if there exists some
291 order $\prec$ on X such that C is a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit.

[2] That is, there may be more than one edge between two nodes u and v.

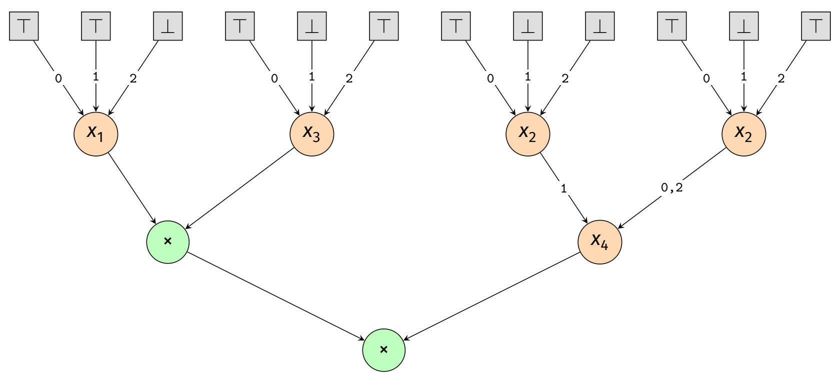


Figure 2 Example of a simple ordered $\{\times, \mathsf{dec}\}$-circuit. The domain used is $\{0, 1, 2\}$ and the variable set is $\{x_1, x_2, x_3, x_4\}$. Notice how the variables on both sides of the $\times$-gates are interweaved.

Frontiers. A *prefix assignment* of size p is an assignment of variables $\tau \in D^{\{x_1,\dots,x_p\}}$ with $p \leqslant n$. When answering direct access tasks, we will need to be able to build the subcircuit associated with a given prefix assignment. When dealing with $\{\times, \mathsf{dec}\}$-circuits, multiple gates can be reached at the same time while following a prefix assignment, due to the $\times$-gates. To handle these cases, we introduce several new notions.

Let v be a decision gate in a $\{\times, \mathsf{dec}\}$-circuit C. We define the set $\mathsf{sink}(v)$ as:

$$\mathsf{sink}(v) = \begin{cases} \bigcup_{w \in \mathsf{inputs}(v)} \mathsf{sink}(w) & \text{if } v \text{ is a } \times\text{-gate} \\ \{v\} & \text{otherwise (that is, } v \text{ is an input or a decision gate)} \end{cases}$$

From this definition, we can infer the following property:

▶ **Lemma 3.** *For any gate v, we have that the set of tuples $\mathsf{rel}(v) = \bigtimes_{w \in \mathsf{sink}(v)} \mathsf{rel}(w)$.*

Proof. We prove this by induction on the circuit. If v is a decision gate or an input, then, as $\mathsf{sink}(v) = \{v\}$, the property is trivial. If v is a $\times$-gate, then by definition, $\mathsf{rel}(v) = \bigtimes_{w \in \mathsf{inputs}(v)} \mathsf{rel}(w)$. By our induction hypothesis, $\mathsf{rel}(w) = \bigtimes_{g \in \mathsf{sink}(w)} \mathsf{rel}(g)$. Therefore, by associativity and commutativity of the Cartesian product, $\mathsf{rel}(v) = \bigtimes_{w \in \mathsf{inputs}(v)} \bigtimes_{g \in \mathsf{sink}(w)} \mathsf{rel}(g) = \bigtimes_{g \in \bigcup_{w \in \mathsf{inputs}(v)} \mathsf{sink}(w)} \mathsf{rel}(g) = \bigtimes_{g \in \mathsf{sink}(v)} \mathsf{rel}(v)$. ◀

Given τ a prefix assignment, the *frontier of τ f_τ* in C is defined algorithmically as follows:

1. instantiate a set F with $\mathsf{out}(C)$, the root of the circuit
2. as long as F is not stable, do:
   - if $v \in F$ is a $\times$-gate, $F := (F \setminus \{v\}) \cup \mathsf{sink}(v)$
   - if $v \in F$ is a decision gate and the variable labelling v is assigned in the prefix ($\mathsf{decvar}(v) \in \{x_1, \dots, x_p\}$), $F := (F \setminus \{v\}) \cup \{v_{\tau(x)}\}$
3. if F contains a $\bot$-gate, then $f_\tau = \emptyset$, otherwise $f_\tau = F$.

If, for a given gate v, the set $\mathsf{sink}(v)$ contains a $\bot$-gate, then the circuit is no longer satisfiable, which is why we return $\emptyset$ in this case. Note that this should not happen while building the k-th solution for C.

Frontiers are particularly useful because they can be efficiently computed and the relation they represent is essentially the tuples of the relation represented by C that agree with τ. The

318 set $\mathsf{var}(f_\tau)$ representing the set of variables of the frontier is defined as $\mathsf{var}(f_\tau) = \bigcup_{v \in f_\tau} \mathsf{var}(v)$.
319 We denote by $\mathsf{rel}(f_\tau)$ the relation on variables $\mathsf{var}(f_\tau)$ defined as $\bigtimes_{v \in f_\tau} \mathsf{rel}(v)$.

320 ▶ Remark 4. For an empty prefix, we have that $f_{\langle\rangle} = \mathsf{sink}(\mathsf{out})$. For a given prefix τ of
321 length p, $f_{\tau \cup \{x_{p+1} \leftarrow d\}}$ can be built from the frontier of τ. Two cases can arise: either the
322 variable x_{p+1} is evaluated by the frontier, meaning that there exists a decision gate $v \in f_\tau$
323 such that $\mathsf{decvar}(v) = x_{p+1}$, or not. In the former case, the frontier associated with the prefix
324 $\tau' = \tau \cup \{x_{p+1}\} \leftarrow d$ is obtained by the following operation: $f_{\tau'} = (f_\tau \setminus \{v\}) \cup \mathsf{sink}(v_d)$. In
325 the latter case, there is no gate labelled by x_{p+1} in the frontier, so it remains untouched,
326 $f_{\tau'} = f_\tau$.

327 For a prefix τ on variables $\{x_1, \dots, x_p\}$, we denote $\sigma_\tau(R)$ the relation $\sigma_{x_1=\tau(x_1),\dots,x_p=\tau(x_p)}(R)$.

328 ▶ **Lemma 5.** *Let τ be a prefix assignment on variables $\{x_1, \dots, x_p\}$. Then we have that*
329 $\sigma_\tau(\mathsf{rel}_X(C)) = \{\tau\} \times \mathsf{rel}(f_\tau) \times D^{\{x_{p+1},\dots,x_n\} \setminus \mathsf{var}(f_\tau)}$.

330 **Proof.** We prove the lemma by induction on the size of the prefix. For an empty prefix $\tau = \langle\rangle$,
331 we have $f_\tau = \mathsf{sink}(\mathsf{out}(C))$. Indeed, if $\mathsf{out}(C)$ is a decision gate or input, then it is trivial,
332 otherwise we simply sink through the $\times$-gate since no variable is assigned. We have that
333 $\sigma_{\langle\rangle}(\mathsf{rel}_X(C)) = \mathsf{rel}_X(C)$, which is itself by definition equal to $\mathsf{rel}(\mathsf{out}(C)) \times D^{X \setminus \mathsf{var}(\mathsf{out}(C))}$. From
334 Lemma 3, we know that $\mathsf{rel}(\mathsf{out}(C)) = \bigtimes_{w \in \mathsf{sink}(\mathsf{out}(C))} \mathsf{rel}(w)$. We know that $\mathsf{var}(\mathsf{out}(C)) =$
335 $\mathsf{var}(f_{\langle\rangle})$. Thus, we have that $\sigma_{\langle\rangle}(\mathsf{rel}_X(C)) = \bigtimes_{w \in \mathsf{sink}(\mathsf{out}(C))} \mathsf{rel}(w) \times D^{\{x_1,\dots,x_n\} \setminus \mathsf{var}(f_{\langle\rangle})}$.
336 Now suppose the property holds for any prefix τ of size p. We now show that it also
337 holds for a prefix $\tau' = \tau \times [x_{p+1} \leftarrow d]$.
338 We can rewrite $\sigma_{\tau'}(\mathsf{rel}_X(C))$ as $\sigma_{x_{p+1}=d}(\sigma_\tau(\mathsf{rel}_X(C)))$. From the induction hypothesis,
339 we have:

340 $$\sigma_{\tau'}(\mathsf{rel}_X(C)) = \sigma_{x_{p+1}=d}\left(\{\tau\} \times \mathsf{rel}(f_\tau) \times D^{\{x_{p+1},\dots,x_n\} \setminus \mathsf{var}(f_\tau)}\right)$$

341 From here, we have two possibilities: either there exists a decision gate $v \in f_\tau$ such that
342 $\mathsf{decvar}(v) = x_{p+1}$ or not. In the first case, we have by definition that $f_{\tau'} = f_\tau \setminus \{v\} \cup \mathsf{sink}(v_d)$.
343 We start by pointing out that for a decision gate v with $x_{p+1} = \mathsf{decvar}(v)$ and $d \in D$,
344 we have $\sigma_{x_{p+1}=d}(\mathsf{rel}(v)) = \{[x_{p+1} \leftarrow d]\} \times \mathsf{rel}(v_d) \times D^{\mathsf{var}(v) \setminus (\{x_{p+1}\} \cup \mathsf{var}(v_d))}$, that is that the
345 relation computed by v when assigning the variable x_{p+1} labelling v a value d is equal to the
346 relation computed by its input v_d extended by the set of tuples representing the different
347 valuations for the variables not evaluated by the subcircuit.
348 We can therefore write:

$$
\begin{aligned}
&\sigma_{\tau'}(\mathsf{rel}_X(C)) = \sigma_{x_{p+1}=d}\left(\{\tau\} \times \mathsf{rel}(f_\tau) \times D^{\{x_{p+1},\dots,x_n\} \setminus \mathsf{var}(f_\tau)}\right) && (349)\\
&\qquad \text{since } x_{p+1} \text{ only appears in the frontier :} && (350)\\
&\qquad = \{\tau\} \times D^{\{x_{p+1},\dots,x_n\} \setminus \mathsf{var}(f_\tau)} \times \sigma_{x_{p+1}=d}(\mathsf{rel}(v)) \times \bigtimes_{w \in f_\tau \setminus \{v\}} \mathsf{rel}(w) && (351)\\
&\qquad \text{from the previous relation:} && (352)\\
&\qquad = \{\tau\} \times D^{\{x_{p+1},\dots,x_n\} \setminus \mathsf{var}(f_\tau)} \times \{[x_{p+1} \leftarrow d]\} \times \mathsf{rel}(v_d) \times D^{\mathsf{var}(v) \setminus (\{x_{p+1}\} \cup \mathsf{var}(v_d))} && (353)\\
&\qquad\quad \times \bigtimes_{w \in f_\tau \setminus \{v\}} \mathsf{rel}(w) && (354)\\
&\qquad \text{from Lemma 3:} && (355)\\
&\qquad = \{\tau\} \times D^{\{x_{p+1},\dots,x_n\} \setminus \mathsf{var}(f_\tau)} \times \{[x_{p+1} \leftarrow d]\} \times \bigtimes_{w \in \mathsf{sink}(v_d)} \mathsf{rel}(w) && (356)
\end{aligned}
$$

$$\times D^{\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup\mathsf{var}(v_d))} \times \bigtimes_{w\in f_\tau\setminus\{v\}} \mathsf{rel}(w)$$

$$= \{\tau\} \times \{[x_{p+1} \leftarrow d]\} \times D^{\{x_{p+1},\ldots,x_n\}\setminus\mathsf{var}(f_\tau)} \times D^{\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup\mathsf{var}(v_d))}$$

$$\times \bigtimes_{w\in\mathsf{sink}(v_d)} \mathsf{rel}(w) \times \bigtimes_{w\in f_\tau\setminus\{v\}} \mathsf{rel}(w)$$

$$= \{\tau'\} \times \bigtimes_{w\in f_{\tau'}} \mathsf{rel}(w) \times D^{\{x_{p+2},\ldots,x_n\}\setminus\mathsf{var}(\tau')}$$

In the second case, there is no gate in f_τ labelled by x_{p+1}. Since the circuit is ordered, it means that $x_{p+1} \notin \mathsf{var}(f_\tau)$. We can therefore write:

$$\sigma_{\tau'}(\mathsf{rel}_X(C)) = \{\tau\} \times \bigtimes_{w\in f_\tau} \mathsf{rel}(w) \times \sigma_{x_{p+1}=d}(D^{\{x_{p+1},\ldots,x_n\}\setminus\mathsf{var}(f_\tau)})$$

since x_{p+1} does not appear in the frontier or τ:

$$= \{\tau\} \times \bigtimes_{w\in f_\tau} \mathsf{rel}(w) \times \{[x_{p+1} \leftarrow d]\} \times D^{\{x_{p+2},\ldots,x_n\}\setminus\mathsf{var}(f_\tau)}$$

since $f_\tau = f_{\tau'}$:

$$= \{\tau'\} \times \bigtimes_{w\in f_{\tau'}} \mathsf{rel}(w) \times D^{\{x_{p+2},\ldots,x_n\}\setminus\mathsf{var}(f_{\tau'})}$$

Since the property is true for the empty prefix and inductively true, we conclude that it is true for any prefix τ. ◀

In order to be useful in practice, building and using the frontier of a prefix assignment τ cannot be too expensive. We formulate the following complexity statement:

▶ **Lemma 6.** *Let τ be a prefix assignment over the set of variables $X = \{x_1, \ldots, x_p\}$. We can compute f_τ in time $\mathcal{O}(|X|)$.*

Proof. Let τ be a prefix assignment of size p. The frontier f_τ is built in a top-down fashion, by following the edges corresponding to the variable assignments in τ. For each variable x assigned by τ, we follow at most one edge from a decision gate v such that $\mathsf{decvar}(v) = x$ and the edge is labelled $\tau(x)$. This means p edges are followed for the assignments. Moreover, to get to v, we might have to follow edges from $\times$-gates. Since the variable sets underneath $\times$-gates are disjoint from one another, we have that the number of such edges is bounded by $|X|$. This implies the total cost of building the frontier f_τ is $\mathcal{O}(|X| + p) = \mathcal{O}(|X|)$. ◀

3.2 Direct Access for ordered $\{\times, \mathsf{dec}\}$-circuits

The main result of this section is an algorithm that allows for direct access for a ordered $\{\times, \mathsf{dec}\}$-circuit on domain D and variables X. More precisely, we prove the following:

▶ **Theorem 7.** *Let $\prec$ be an order on X and C be a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit on domain D and variables X, then we can have a direct access on $\mathsf{rel}(C)$ for the order $\prec_{\mathsf{lex}}$ with access time polynomial in $\mathcal{O}(\mathsf{poly}|X|\mathsf{polylog}|D|)$ and precomputation time $\mathcal{O}(|C| \cdot \mathsf{poly}|X|\mathsf{polylog}|D|)$.*

Precomputation. In this section, we assume that C is a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit with respect to X. The *count label of C*, denoted by nrel_C, is the mapping from $\mathsf{decision}(C) \times D$ to $\mathbb{N}$ such that $\mathsf{nrel}_C(v, d) = \#\sigma_{x\leqslant d}(\mathsf{rel}(v))$, that is, $\mathsf{nrel}_C(v, d)$ is the number of tuples from $\mathsf{rel}(v)$ that assign a value on x smaller or equal than d. The precomputation step aims to compute nrel_C so that we can access $\mathsf{nrel}_C(v, d)$ quickly.

Our algorithm performs a bottom-up computation of the number of satisfying tuples in $\mathsf{rel}(v)$ for every gate v of C. If v is a decision-gate on variable x, then $\mathsf{rel}(v)$ is defined as a disjoint extended union from the relation computed by its input. It is then easy to see that:

$$|\mathsf{rel}(v)| = \sum_{w \in \mathsf{input}(v)} |\mathsf{rel}(w)| \times |D|^{|\Delta(v,w)|} \text{ where } \Delta(v,w) = \mathsf{var}(v) \setminus (\{x\} \cup \mathsf{var}(w)). \tag{1}$$

Similarly, $\mathsf{nrel}_C(v,d)$ can be computed by restricting the previous relation on the inputs of v that set x to a value $d' \leqslant d$, that is:

$$\mathsf{nrel}_C(v,d) = \sum_{w \in \mathsf{input}(v), \ell(w,v) \leqslant d} |\mathsf{rel}(w)| \times |D|^{|\Delta(v,w)|}. \tag{2}$$

Observe that from this relation, we can deduce that for a decision gate v, if $d' \in D$ does not label any edge (w,v) then $\mathsf{nrel}_C(v,d') = \mathsf{nrel}_C(v,d)$ where d is the largest value such that $d < d'$. For a decision gate v of C, we will hence only compute $\mathsf{nrel}_C(v,d)$ for $d \in D$ such that there is an edge (w,v) labeled by d.

Finally, if v is a Cartesian product, we clearly have $|\mathsf{rel}(v)| = \bigtimes_{w \in \mathsf{input}(v)} |\mathsf{rel}(w)|$. Hence, one can compute nrel_C using a dynamic algorithm that inductively computes $\mathsf{nrel}_C(v,d)$ for each decision-gate v of the circuit and also $|\mathsf{rel}(v)|$ and $\mathsf{var}(v)$ for every gate.

More precisely, the dynamic programing algorithm works as follows: we start by performing a topological ordering of the gates of C that is compatible with the underlying DAG of the circuit. In particular, it means that for a gate v and an input w of v, the topological ordering has to place w before v. Moreover, we also add the following constraint: if w and w' are both inputs of a decision gate v and if the edge (w,v) is labeled by $d \in D$ and the edge (w',v) is labeled by $d' \in D$ such that $d < d'$, then we ask for the topological order to place w before w'. It easy easy to construct such an ordering by simply doing a topological order of the DAG of C augmented by edges (w,w') where w and w' are inputs of the same decision gate v and (w,v) has a label smaller than (w',v). This modified DAG has size at most $2|C|$ and since computing a topological ordering of a DAG can be done in linear time, we can construct it in time $O(|C|)$.

Now, we dynamically compute $|\mathsf{rel}(v)|$, $\mathsf{var}(v)$ for every gate v and $\mathsf{nrel}_C(v,d)$ for every gate $v \in C$ and $d \in D$ as follows: we start by allocating two tables T_{rel} and T_{var} of size $|C|$ and a table T_{nrel_C} of size $|C|$ where each entry of T_{nrel_C} is intialized with an array of size $|D|$ where each entry is initialized as -1. We then populate each entry of these tables following the previously constructed topological ordering and using the relations written above (see (1) and (2)) and the fact that $\mathsf{var}(v) = \bigcup_{w \in \mathsf{input}(v)} \mathsf{var}(w)$. To compute nrel_C for a decision-gate v, we let $(w_0,v),\dots,(w_k,v)$ be the incoming edge of v ordered by increasing labelled $d_0 < \dots < d_k$. We then initialize $T_{\mathsf{nrel}_C}[v,d_0]$ with $\mathsf{nrel}_C(v,d_0) = |\mathsf{rel}(w_0)| \cdot |D|^{|\Delta(v,w_0)|}$ and compute $T_{\mathsf{nrel}_C}[v,d_{i+1}]$ as $T_{\mathsf{nrel}_C}[v,d_i] + |\mathsf{rel}(w_{i+1})| \cdot |D|^{|\Delta(v,w_{i+1})|}$ using relation (2).

An example of an ordered $\{\times, \mathsf{dec}\}$-circuit that has been annotated by our algorithm is presented in Figure 3.

It is clear from the relations (2) that at the end of this precomputation, we have $T_{\mathsf{nrel}_C}[v,d]$ contains $\mathsf{nrel}_C(v,d)$ if d labels an incoming edge of v. In practice, we will not need to access $\mathsf{nrel}_C(v,d')$ for other values $d' \in D$ but we observe here that we can still compute it in time $\mathsf{polylog}(|D|)$ from T_{nrel_C} and C as follows: we can find the largest $d \in D$ such that d labels an incoming edge of v and $d < d'$ using a binary search on the input edges of v and return $T_{\mathsf{nrel}_C}[v,d]$ if such a d exists and 0 otherwise. We hence have:

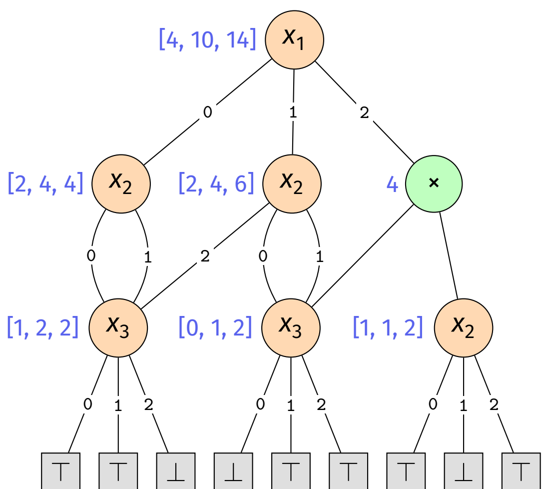


Figure 3 Example of a simple, annotated $\{\times, \mathsf{dec}\}$-circuit. The domain used is $\{0, 1, 2\}$ for variables x_1, x_2 and x_3. The lists shown to the left of the decision gates represent the values of nrel_C for those gates.

434 ▶ **Lemma 8** (Precomputation complexity). *Given a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit C, we can*
435 *compute a data structure in time $\mathcal{O}(|C| \cdot \mathsf{poly}|X|\mathsf{polylog}|D|)$ that allows us to access $\mathsf{var}(v)$,*
436 *$|\mathsf{rel}(v)|$ for every gate v of C in time $\mathcal{O}(1)$ and $\mathsf{nrel}_C(v, d)$ for every decision gate v and*
437 *$d \in D$ in time $\mathcal{O}(\mathsf{polylog}(|D|))$.*

438 **Proof.** The data structure simply consists in the three tables $T_{\mathsf{nrel}_C}, T_{\mathsf{rel}}$ and T_{var}. It is easy
439 to see that each entry of T_{var} can be computed in time $O(\mathsf{poly}(|X|)$ since we only have to
440 compute union of sets of elements in X hence one can compute T_{var} with $\mathcal{O}(|C|\mathsf{poly}(|X|))$.
441 Now observe that to compute T_{nrel_C}, one has to perform at most two arithmetic operations
442 for each edge of C. Indeed, to compute $T_{\mathsf{rel}}[v]$, one has to perform at most one addition and
443 one multiplication for each $w \in \mathsf{input}(v)$, whose cost can be associated to the edge (w, v).
444 Similarly, to compute $T_{\mathsf{nrel}_C}[v, d]$ as described in the previous paragraph, we do one addition
445 and one multiplication for each edge (w, v) in the circuit. Hence, we perform at most $\mathcal{O}(|C|)$
446 arithmetic operations. Now, the cost of these arithmetic operations is polynomial in the size
447 of the integer they are performed on. Since these integers are sizes of relation on domain
448 D and variable $Y \subseteq X$, their value is at most $|D|^{|X|}$ and they can be encoded on $|X|\mathsf{log}|D|$
449 bits. Hence, we have a total complexity of $\mathcal{O}(|C|\mathsf{poly}|X|\mathsf{polylog}|D|)$. ◀

450 **Direct access.** We now show how the precomputation from Lemma 8 allows us to get direct
451 access for ordered $\{\times, \mathsf{dec}\}$-circuits. We first show how one can solve a direct access task for
452 any relation as long as we have access to very simple counting oracles. We then show that
453 one can quickly simulate these oracle calls in ordered $\{\times, \mathsf{dec}\}$-circuits using precomputed
454 values and conclude.

455 ▶ **Lemma 9.** *Assume that we are given a relation $R \subseteq D^X$ with $X = \{x_1, \dots, x_n\}$ and*
456 *an oracle such that for every prefix assignment $\tau \in D^{\{x_1,\dots,x_p\}}$ and $d \in D$, it returns*
457 *$\sigma_{x_{p+1} \leqslant d}(\#\sigma_\tau(R))$. Then, for any $k \leqslant |R|$, we can compute $R[k]$ using $\mathcal{O}(n\mathsf{polylog}|D|)$ oracle*
458 *calls, where $n = |X|$.*

Proof. We prove this lemma by induction: we show that for every relation of R arity n, we can compute $R[k]$ using $n \cdot \lceil \log |D| \rceil$ oracle calls. We start by considering that R is a relation on one variable x. Let $\alpha = R[k]$. We are looking for $d \in D$ such that $\alpha(x) = d$. In this case, since $x = \min(\mathsf{var}(R))$, we know from Lemma 1 that d is the minimal value such that $\#\sigma_{x \leqslant d}(R) \geqslant k$. We can compute d by doing a dichotomic search on the domain values using $\lceil \log |D| \rceil$ calls to the oracle since the value $\#\sigma_{x \leqslant d}(R)$ increases when d increases.

Now, assume that the property holds for relations on variable sets of size n, that is, that we can find the k-th solution with $n \cdot \lceil \log |D| \rceil$ oracle calls. Let R be a relation on a set of variables $\{x_1, \dots, x_{n+1}\}$. From Lemma 1, we know that $R[k] = R'[k']$, where $R' = \sigma_{x_1 = d_1}(R)$, d_1 is the minimal value such that $\sigma_{x_1 \leqslant d_1}(R) \geqslant k$ and $k' = k - \#\sigma_{x_1 < d_1}(R)$. As we saw earlier, we can find d_1 using $\lceil \log |D| \rceil$ oracle calls of the form $\#\sigma_{x_1 \leqslant d}(R)$ and using a dichotomic search on d.

Now, by induction, we are able to compute $R'[k']$ using $n \cdot \lceil \log |D| \rceil$ oracle calls of the form $\#\sigma_{x_{p+1} \leqslant d}(\sigma_{\tau'}(R'))$ for τ' an assignment of $D^{\{x_2, \dots, x_p}$. However, observe that $\#\sigma_{x_{p+1} \leqslant d}(\sigma_{\tau'}(R')) = \#\sigma_{x_{p+1} \leqslant d}(\sigma_{\tau}(R))$ with $\tau = \tau' \times [x_1 \leftarrow d_1]$ since $R' = \sigma_{x_1 = d_1}(R)$. Hence we can compute $R[k]$ using $\lceil \log |D| \rceil + n \cdot \lceil \log |D| \rceil = (n+1)\lceil \log |D| \rceil$ oracle calls to relation R, which concludes the induction step. ◀

In order to evaluate the true complexity of answering direct access tasks, we now also have to evaluate the complexity of a single oracle call.

▶ **Lemma 10.** *Let C be a circuit such that $\mathsf{nrel}_C(v, d)$ and $\mathsf{var}(v)$ have been precomputed and can be access in time $\mathcal{O}(\mathsf{polylog}(|D|))$ for every gate v of C and $d \in D$. Let τ be a prefix assignment of $D^{\{x_1, \dots, x_p\}}$ and $d \in D$, then $\#\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C)))$ can be computed in time $\mathcal{O}(\mathsf{poly}(n)\mathsf{polylog}|D|)$, where $n = |X|$.*

Proof. We start by building the frontier f_τ associated with the prefix assignment τ. From Lemma 6, we know this can be done in time $\mathcal{O}(n)$. By Lemma 5:

We can rewrite:

$$\begin{aligned}\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C))) &= \sigma_{x_{p+1} \leqslant d}(\{\tau\} \times \mathsf{rel}(f_\tau) \times D^{\{x_{p+1}, \dots, x_n\} \setminus \mathsf{var}(f_\tau)}) \\ &= \{\tau\} \times \sigma_{x_{p+1} \leqslant d}(\mathsf{rel}(f_\tau) \times D^{\{x_{p+1}, \dots, x_n\} \setminus \mathsf{var}(f_\tau)})\end{aligned}$$

There are now two possible outcomes: either x_{p+1} is tested by f_τ or not. In the first case, since x_{p+1} only appears in the frontier, tested by a gate v, we have:

$$\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C))) = \{\tau\} \times \sigma_{x_{p+1} \leqslant d}(\mathsf{rel}(v)) \times \bigtimes_{w \in f_\tau \setminus \{v\}} \mathsf{rel}(w) \times D^{\{x_{p+1}, \dots, x_n\} \setminus \mathsf{var}(f_\tau)}$$

Hence $\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C)))$ can be computed as:

$$\begin{aligned}\#\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(\mathsf{rel}(C))) &= \#\sigma_{x_{p+1} \leqslant d}(\mathsf{rel}(v)) \times \prod_{w \in f_\tau \setminus \{v\}} \#\mathsf{rel}(w) \times |D|^{|\{x_{p+1}, \dots, x_n\} \setminus \mathsf{var}(f_\tau)|} \\ &= \mathsf{nrel}_C(v, d) \times \prod_{w \in f_\tau \setminus \{v\}} \#\mathsf{rel}(w) \times |D|^{|\{x_{p+1}, \dots, x_n\} \setminus \mathsf{var}(f_\tau)|}.\end{aligned}$$

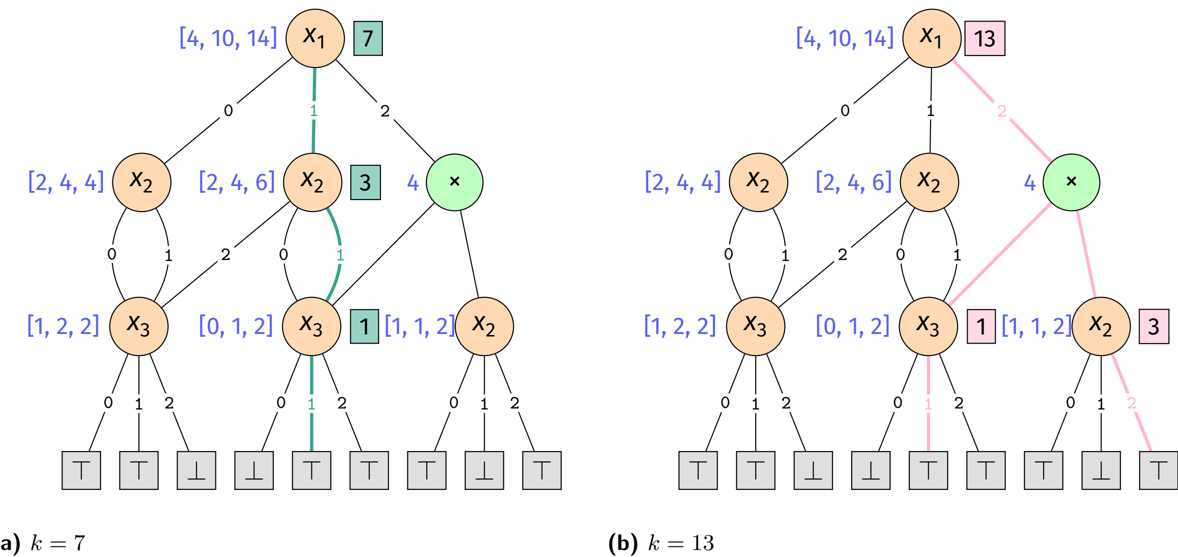
The values of $\mathsf{nrel}_C(v, d)$ and of $\#\mathsf{rel}(w)$ for $w \in f_\tau \setminus \{v\}$ have been precomputed and can be accessed in time $\mathcal{O}(\mathsf{polylog}(|D|))$. Now by definition $\mathsf{var}(f_\tau) = \bigcup_{v \in f_\tau} \mathsf{var}(v)$. Hence, since $\mathsf{var}(v)$ has been precomputed, we can compute $|\{x_{p+2}, \dots, x_n\} \setminus \mathsf{var}(f_\tau)|$ in $\mathcal{O}(n)$. The multiplication has at most n elements, the cost of this operation is therefore simply $\mathcal{O}(\mathsf{poly}(n)\mathsf{polylog}|D|)$.

In the second case where x_{p+1} is not tested by f_τ, we have that $x_{p+1} \notin \mathsf{var}(f_\tau)$ since the circuit is ordered. Hence, we apply a similar reasoning to obtain:

$$\#\sigma_{\tau \wedge x_{p+1} \leqslant d}(\mathsf{rel}(C)) = \#\sigma_{x_{p+1} \leqslant d}(D^{\{x_{p+1}\}}) \cdot |D|^{|\{x_{p+2},\dots,x_n\} \setminus \mathsf{var}(f_\tau)|} \cdot \prod_{w \in f_\tau} \#\mathsf{rel}(w)$$

The value of $\#\sigma_{x_{p+1} \leqslant d}(D^{\{x_{p+1}\}})$ is simply $\mathsf{rank}(d)$. As before, we can compute the multiplication in $\mathcal{O}(\mathsf{poly}(n)\mathsf{polylog}|D|)$. ◀

In short, we can follow the edges in the circuit by choosing the correct edge from the precomputed values in nrel_C. A short visual example of the followed paths for different direct access tasks over the same annotated circuit is presented in Figure 4. Notice how in the case where $k = 13$, the fact that we meet a $\times$-gate implies that we follow both paths at once. At one point of the algorithm, a frontier containing both the gates for x_2 and x_3 exists. The values shown at the right of the reached decision gates show how the index of the searched tuples evolves during a run of the search algorithm.



(a) $k = 7$ **(b)** $k = 13$

Figure 4 Examples of the paths followed during different direct access tasks on the same annotated ordered $\{\times, \mathsf{dec}\}$-circuit.

The proof of Theorem 7 is now an easy corollary of Lemmas 9 and 10. Indeed, after having precomputed nrel_C and var using 8, we can answer direct access tasks using the oracle based algorithm from Lemma 9 and Lemma 10 shows that these oracle accesses are in fact tractable in ordered circuits.

4 From join queries to ordered $\{\times, \mathsf{dec}\}$-circuits

In this section, we present a simple top-down algorithm, that can be seen as an adaptation of the exhaustive DPLL algorithm from [32], such that on input Q, $\prec$ and $\mathbf{D}$, it returns a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit C such that $\mathsf{rel}(C) = Q(\mathbf{D})$, where Q is a join query, $\prec$ an order on its variables and $\mathbf{D}$ a database. Exhaustive DPLL is an algorithm that has been originally devised to solve the #SAT problem. It has been observed by Huang and Darwiche [20] that the trace of this algorithm implicitly builds a Boolean circuit, corresponding to the

521 $\{\times, \mathsf{dec}\}$-circuits on domain $\{0,1\}$, enjoying interesting tractability properties. We show
522 how to adapt it in the framework of signed join queries. The algorithm itself is presented
523 in Section 4.1. We study the complexity of this algorithm in Section 4.3 depending on the
524 structure of Q and $\prec$, using hypergraph structural parameters introduced in Section 4.2.

525 4.1 Exhaustive DPLL for signed join queries

526 The main idea of DPLL for signed join queries is the following: given an order $\prec$ on the
527 variables of a join query Q and a database $\mathbf{D}$, we construct a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit
528 (where $x \succ y$ iff $y \prec x$)[3] computing $[\![Q]\!]^{\mathbf{D}}$ by successively testing the variables of Q with
529 decision gates, from the highest to the lowest wrt $\prec$. At its simplest form, the algorithm
530 picks the highest variable x of Q wrt $\prec$, creates a new decision gate v on x and then, for
531 every value $d \in D$, sets x to d and recursively computes a gate v_d computing the subset
532 of $[\![Q]\!]^{\mathbf{D}}$ where $x = d$. We then add v_d as an input of v and proceed with the next value
533 $d' \in D$. This approach is however not enough to get interesting tractability results. We
534 hence add the following optimizations. First, we keep a cache of already computed queries so
535 that if we recursively call the algorithm twice on the same input, we can directly return the
536 previously constructed gate. Moreover, if we detect that the answers of Q are the Cartesian
537 product of two or more subqueries $Q_1, \dots, Q_k$, then we create a new $\times$-gate v, recursively
538 call the algorithm on each component Q_i to construct a gate w_i and plug each w_i to v.
539 Detecting such cases is mainly done syntactically, by checking whether the query can be
540 partitionned into subqueries having disjoint variables. However, this approach would fail
541 to give good complexity bounds in the presence of negative atoms. To achieve the best
542 complexity, we also remove from Q every negative atom as soon as they are satisfied by
543 the current partial assignment. This allows us to discover more cases where the query has
544 connected components.

545 The theoretical performance of the previously described algorithm may however vary if
546 one is not careful in the way the recursive calls are actually made. We hence give a more
547 formal presentation the algorithm, whose pseudocode is presented in Algorithm 1, on which
548 we will be able to prove good upper bounds in Section 4.3. Since we are not yet insterested in
549 complexity analysis, we deliberately let the underlying datastructures for encoding relations
550 unspecified and delay this discussion to Section 4.3.

551 A few notations are used in Algorithm 1. Given a database $\mathbf{D}$ on domain D and a tuple
552 $\tau \in D^Y$, we denote by $[\![Q]\!]^{\mathbf{D}}_{\tau}$ the set of tuples $\sigma \in D^{\mathsf{var}(Q)\setminus Y}$ that are answers of Q when
553 extended with τ. More formally, $\sigma \in [\![Q]\!]^{\mathbf{D}}_{\tau}$ if and only if $(\sigma \times \tau)|_{\mathsf{var}(Q)} \in [\![Q]\!]^{\mathbf{D}}$. Given an
554 atom $R(\mathbf{x})$, a database $\mathbf{D}$ and a tuple $\tau \in D^Y$, we say that $R(\mathbf{x})$ *is inconsistent with τ wrt*
555 $\mathbf{D}$ (or simply inconsistent with τ when $\mathbf{D}$ is clear from context) if there is no $\sigma \in R^{\mathbf{D}}$ such
556 that $\tau \simeq \sigma$. Observe that if Q contains a positive atom $R(\mathbf{x})$ that is inconsistent with τ then
557 $[\![Q]\!]^{\mathbf{D}}_{\tau} = \emptyset$. Similarly, if Q contains a negative atom $\neg R(\mathbf{x})$ such that τ assigns every variable
558 of $\mathbf{x}$ and $\tau(\mathbf{x}) \in R$, then $[\![Q]\!]^{\mathbf{D}}_{\tau} = \emptyset$. If one of this case arises, we say that Q is *inconsistent*
559 *with* τ. Now observe that if $\neg R(\mathbf{x})$ is a negative atom of Q such that $R(\mathbf{x})$ is inconsistent
560 with τ, then $[\![Q]\!]^{\mathbf{D}}_{\tau} = [\![Q']\!]^{\mathbf{D}}_{\tau} \times D^W$ where $Q' = Q \setminus \{\neg R(\mathbf{x})\}$ and $W = \mathsf{var}(Q) \setminus \mathsf{var}(Q')$ (some
561 variables of Q may only appear in the atom $\neg R(\mathbf{x})$). This motivates the following definition:
562 the *simplification of Q wrt to τ and* $\mathbf{D}$, denoted by $Q \Downarrow \langle \tau, \mathbf{D} \rangle$ or simply by $Q \Downarrow \tau$ when $\mathbf{D}$
563 is clear from context, is defined to be the subquery of Q obtained by removing from Q every

[3] While one could easily change the algorithm so that it produces a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit instead, the structural parameters we will be considering for the tractability of DPLL in Section 4.2 are more naturally defined on $\prec$. We choose to present DPLL this way to ease the proofs later.

Algorithm 1 An algorithm to compute a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit representing $[\![Q]\!]^{\mathbf{D}}$

```
1:  procedure DPLL(Q, τ, D, ≺)
2:      if (Q, τ) is in cache then return cache(Q, τ)
3:      if Q is inconsistent with τ then return ⊥-gate
4:      if τ assigns every variable in Q then return ⊤-gate
5:      x ← max_≺ var(Q)
6:      for d ∈ D do
7:          τ' ← τ × [x ← d]
8:          if Q is inconsistent with τ' then  v_d ← ⊥-gate
9:          else
10:             Let Q_1, ..., Q_k be the τ'-connected components of Q ⇓ τ'
11:             for i = 1 to k do
12:                 w_i ← DPLL(Q_i, τ_i, D, ≺) where τ_i = τ'|var(Q_i)
13:             end for
14:             v_d ← new ×-gate with inputs w_1, ..., w_k
15:         end if
16:     end for
17:     v ← new dec-gate connected to v_d by a d-labelled edge for every d ∈ D
18:     cache(Q, τ) ← v
19:     return v
20: end procedure
```

negative atom $\neg R(\mathbf{x})$ of Q such that $R(\mathbf{x})$ is inconsistent with τ. From what precedes, we clearly have $[\![Q]\!]^{\mathbf{D}}_\tau = [\![Q']\!]^{\mathbf{D}}_\tau \times D^W$ where $Q' = Q \Downarrow \langle \tau, \mathbf{D}\rangle$ and $W = \mathsf{var}(Q) \setminus \mathsf{var}(Q')$.

For a tuple $\tau \in D^Y$ assigning a subset Y of variables of Q, the *τ-intersection graph $\mathcal{I}^Q_\tau$ of Q* is the graph whose vertices are the atoms of Q having at least one variable not in Y and there is an edge between two atoms a, b of Q if a and b share a variable that is not in Y. Observe that $\mathcal{I}^Q_\tau$ does not depend on the values of τ but only on the variables it sets. Hence it can be computed in polynomial time in the size of Q only. A connected component C of $\mathcal{I}^Q_\tau$ naturally induces a subquery Q_C of Q and is called a *τ-connected component*. Q is partitioned into its τ-connected components and the atoms whose variables are completely set by τ. More precisely, $Q = \bigcup_{C \in \mathcal{CC}} Q_C \cup Q'$ where $\mathcal{CC}$ are the connected component of $\mathcal{I}^Q_\tau$ and Q' contains every atom a of Q on variables $\mathbf{x}$ such that $\mathbf{x}$ only has variables in Y. Observe that if τ is an answer of Q', then $[\![Q]\!]^{\mathbf{D}}_\tau = \bigtimes_{C \in \mathcal{CC}} [\![Q_C]\!]^{\mathbf{D}}_{\tau_C}$ where $\tau_C = \tau|_{\mathsf{var}(Q_C)}$ since if C_1 and C_2 are two distinct τ-connected components of $\mathcal{I}^Q_\tau$, then $\mathsf{var}(Q_{C_1}) \cap \mathsf{var}(Q_{C_2}) \subseteq Y$.

We illustrate the previous definitions on the signed join query $Q(x_1, \dots, x_5)$ defined as $\neg R(x_1, \dots, x_5), S(x_1, x_2, x_3), T(x_1, x_4, x_5)$ and database $\mathbf{D}$ on domain $\{0, 1\}$ with $R^{\mathbf{D}} = \{(1, 1, 1, 1, 1)\}$. Let $\tau = [x_1 \leftarrow 0]$. The τ-intersection graph of Q is a path where $\neg R(x_1, \dots, x_5)$ is connected to $S(x_1, x_2, x_3)$ and $T(x_1, x_4, x_5)$. There is no edge between $S(x_1, x_2, x_3)$ and $T(x_1, x_4, x_5)$ since x_1 is their only common variable and it is assigned by τ. Hence, Q has one τ-connected component containing every atom of Q. Now, $Q \Downarrow \tau = S(x_1, x_2, x_3), T(x_1, x_4, x_5)$ since $R(0, x_2, \dots, x_5)$ is inconsistent over $\mathbf{D}$ and the τ-intersection graph of $Q \Downarrow \tau$ consists in two isolated vertices $S(x_1, x_2, x_3)$ and $T(x_1, x_4, x_5)$. Hence $Q \Downarrow \tau$ has two τ-connected components. This example also illustrates the role of simplification for discovering Cartesian products.

Algorithm 1 uses the previous observations to produce a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit. More precisely:

589 ▶ **Theorem 11.** *Let Q be a signed join query, $\mathbf{D}$ a database and $\prec$ an order on* $\mathsf{var}(Q)$*, then*
590 $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ *constructs a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit C and returns a gate v of C such*
591 *that* $\mathsf{rel}(v) = [\![Q]\!]^{\mathbf{D}}$*.*

592 **Proof.** The proof is by induction on the number of variables of Q that are not assigned
593 by τ. We claim that $\mathsf{DPLL}(Q, \tau, \mathbf{D}, \prec)$ returns a gate computing $[\![Q]\!]^{\mathbf{D}}_{\tau}$ which is stored into
594 *cache*(Q, τ). If every variable are assigned, then $\mathsf{DPLL}(Q, \tau, \mathbf{D}, \prec)$ returns either a $\top$-gate
595 or a $\bot$-gate depending on whether τ is inconsistent with Q or not, which clearly is $[\![Q]\!]^{\mathbf{D}}_{\tau}$.
596 Otherwise, it returns and add in the cache a decision gate v connected to a gate v_d by
597 a d-labelled edge for each $d \in D$. We claim that v_d computes $[\![Q]\!]^{\mathbf{D}}_{\tau\times[x\leftarrow d]}$. It is enough
598 since in this case, by definition of the relation computed by a decision-gate, v computes
599 $\bigcup_{d\in D} [\![Q]\!]^{\mathbf{D}}_{\tau\times[x\leftarrow d]} \times [x \leftarrow d] = [\![Q]\!]^{\mathbf{D}}_{\tau}$.
600 To prove that v_d computes $[\![Q]\!]^{\mathbf{D}}_{\tau'}$ where $\tau' = \tau \times [x \leftarrow d]$, we separate two cases: if τ'
601 is inconsistent with Q then $[\![Q]\!]^{\mathbf{D}}_{\tau'}$ is empty and v_d is a $\bot$-gate, which is what is expected.
602 Otherwise, let $Q_1, \dots, Q_k$ be the τ'-connected components of $Q \Downarrow \tau'$ and let $\tau_i = \tau'|_{\mathsf{var}(Q_i)}$.
603 From what precedes, we have $[\![Q]\!]^{\mathbf{D}}_{\tau'} = \bigtimes_{i=1}^{k} [\![Q_i]\!]^{\mathbf{D}}_{\tau_i}$. The algorithm uses a gate w_i from
604 Line 12, obtained from a recursive call to $\mathsf{DPLL}(Q_i, \tau_i, \mathbf{D}, \prec)$ where the number of variables
605 not assigned by τ_i in Q_i is less than the number of variables unassigned by τ in Q. Hence,
606 by induction, w_i computes $[\![Q_i]\!]^{\mathbf{D}}_{\tau_i}$ and since v_d is a $\times$-gate connected to each w_i, we indeed
607 have $\mathsf{rel}(v_d) = \bigtimes_{i=1}^{k} [\![Q_i]\!]^{\mathbf{D}}_{\tau_i}$. ◀

608 The worst case complexity of DPLL may be high when no cache hit occur which would
609 result in at least $|\mathbf{D}|^{fcn(Q)}$ recursive calls by Theorem 2. However, when $\prec$ has good
610 properties wrt Q, we can prove better bounds. Section 4.2 proposes a way of measuring
611 the complexity of an elimination order wrt Q and Section 4.3 gives upper bounds on the
612 complexity of DPLL depending on this measure.

613 4.2 Hyperorder width

614 In this section, we introduce the notion of width that is relevant to pinpoint the complexity
615 of the DPLL procedure previously described on signed join queries. Our decomposition
616 is not based on hypertree decompositions but rather on elimination orders. We introduce
617 several notions of widths for elimination orders on (signed) hypergraphs that will be used to
618 establish the following complexity bounds:

619 **Order based widths ($\mathsf{how}(\cdot), \mathsf{fhow}(\cdot)$).** A hypergraph $H = (V, E)$ and an order $\prec$ such that
620 $V = \{v_1, \dots, v_n\}$ with $v_1 \prec \dots \prec v_n$ induces a series of hypergraphs defined as $H_1^{\prec}, \dots, H_{n+1}^{\prec}$
621 as $H_1^{\prec} = H$ and $H_{i+1}^{\prec} = H_i^{\prec}/v_i$. The *hyperorder width* $\mathsf{how}(H, \prec)$ *of* $\prec$ *wrt* H is defined as
622 $\max_{i\leqslant n} cn(N_{H_i^{\prec}}(v_i), E)$. The *hyperorder width* $\mathsf{how}(H)$ *of* H is defined as the best possible
623 width using any elimination order, that is, $\mathsf{how}(H) = \min_{\prec} \mathsf{how}(H, \prec)$. We similarly define
624 the *fractional hyperorder width* $\mathsf{fhow}(H, \prec)$ *of* $\prec$ *wrt* H as $\max_{i\leqslant n} fcn(N_{H_i^{\prec}}(v_i), E)$ and the
625 *fractional hyperorder width* $\mathsf{fhow}(H)$ *of* H as $\mathsf{fhow}(H) = \min_{\prec} \mathsf{fhow}(H, \prec)$.
626 It has already been observed in many work ([23, Appendix C] or [16, 17, 24]) that
627 $\mathsf{how}(H)$ and $\mathsf{fhow}(H)$ are respectively equal to the generalized hypertree width and the
628 fractional hypertree width of H and that there is a natural correspondence between a tree
629 decomposition and an elimination order having the same width. However, to be able to
630 express our tractability results as function of the order, it is more practical to define the
631 width of orders instead of hypertree decompositions. In [8, Definition 9], $\mathsf{fhow}(H, \prec)$ is called
632 the incompatiblity number, though it is not formally defined on hypergraphs but directly on

conjunctive queries. The case $k = 1$, which corresponds to the α-acyclicity of the underlying hypergraph, has been also previously call an order without disruptive trio [12]. However, these notions are specifically used for the problem of direct access in conjunctive queries while the characterization of hypergraph measures in terms of elimination orders of hypergraphs predates by several years this terminology (see [6] for a survey). In this paper, we decided to have a terminology closer to the usual terminology for hypergraphs decompositions, where we replace the usual tree decompositions by order decompositions. It will be specifically useful for the hereditary order based widths.

Hereditary order based widths (β-$\mathsf{how}(\cdot)$, β-$\mathsf{fhow}(\cdot)$). One shortcoming of (fractional) hypertree width is that it is not hereditary. That is, the (fractional) hypertree width of a subhypergraph can be much bigger than the (fractional) hypertree width of the hypergraph itself. It makes it not well suited to discover tractable classes for signed join queries. Indeed, if a query Q contains a negative atom $\neg R(\mathbf{x})$ and if $R^{\mathbf{D}}$ is empty in the database $\mathbf{D}$, then $[\![Q]\!]^{\mathbf{D}}$ is equal to $[\![Q']\!]^{\mathbf{D}}$, where $Q' = Q \setminus \{\neg R(\mathbf{x})\}$. Hence if some aggregation problem for a fixed self-join free query Q on an input database $\mathbf{D}$ can be solved in $O(\mathsf{poly}(|\mathbf{D}|))$ for any database $\mathbf{D}$, it has to be tractable for every Q' obtained by removing a subset of the negative atoms from Q. This motivates the following definitions: for a hypergraph $H = (V, E)$ and an order $\prec$ on V, the *β-hyperorder width β-$\mathsf{how}(H, \prec)$ of $\prec$ wrt to H* is defined as $\max_{H' \subseteq H} \mathsf{how}(H', \prec)$. The *$\beta$-hyperorder width β-$\mathsf{how}(H)$ of H* is defined as the width of the best possible elimination order, that is, β-$\mathsf{how}(H) = \min_{\prec} \beta$-$\mathsf{how}(H, \prec)$. We define similarly the β-fractional hyperorder width of an order $\prec$ and of an hypergraph – β-$\mathsf{fhow}(H, \prec)$ and β-$\mathsf{fhow}(H)$ – by replacing $\mathsf{how}(\cdot)$ by $\mathsf{fhow}(\cdot)$ in the definitions.

Comparison with existing measures. The fact that fractional hypertree width is not hereditary has traditionnally been worked around by taking the largest width over every subhypergraph. In other words, the *β-fractional hypertree width* β-$\mathsf{fhtw}(H)$ of H is defined as β-$\mathsf{fhtw}(H) = \max_{H' \subseteq H} \mathsf{fhtw}(H')$. The β-hypertree width β-$\mathsf{htw}(H)$ is defined similarly. If one plugs the ordered characterisation of $\mathsf{fhtw}(H')$ in this definition, one can observe that β-$\mathsf{fhtw}(H) = \max_{H' \subseteq H} \min_{\prec} \mathsf{fhow}(H', \prec)$. Hence, the difference between β-$\mathsf{fhtw}(H)$ and β-$\mathsf{fhow}(H)$ boils down to inverting the min and the max in the definition. It directly gives that β-$\mathsf{fhtw}(H) \leqslant \beta$-$\mathsf{fhow}(H)$ and β-$\mathsf{htw}(H) \leqslant \beta$-$\mathsf{how}(H)$ for every H. The main advantage of the β-fractional hyperorder width is that it comes with a natural notion of decomposition — the best elimination order $\prec$ — that can be used algorithmically. This is not given by the definition of β-$\mathsf{fhtw}(\cdot)$ and has yet to be found.

The only exception is the case where β-$\mathsf{fhtw}(H) = 1$, known as β-acyclicity, where an order-based characterisation is known and has been used to show the tractability of many problems such as SAT [30], #SAT or #CQ for β-acyclic instances [11, 7]. The elimination order is based on the notion of nest points. In a hypergraph $H = (V, E)$, a *nest point* is a vertex $v \in V$ such that $E(v)$ is ordered by inclusion, that is, $E(v) = \{e_1, \dots, e_p\}$ with $e_1 \subseteq \dots \subseteq e_p$. A *$\beta$-elimination order $(v_1, \dots, v_n)$ for H* is an ordering of V such that for every $i \leqslant n$, v_i is a nest point of $H \setminus \{v_1, \dots, v_{i-1}\}$. A closer inspection of the definition of β-elimination order $\prec$ shows that β-$\mathsf{fhow}(H, \prec) = \beta$-$\mathsf{how}(H, \prec) = 1$, showing that iy corresponds to β-acyclicity. We can actually prove a more general result: the notion of β-acyclicity has been recently generalised by Lanzinger in [25] using a notion called nest sets. A set of vertices $S \subseteq V$ is a *nest set of H* if $\{e \setminus S \mid e \in E, e \cap S \neq \emptyset\}$ is ordered by inclusion. A *nest set elimination order* is a list $\Pi = (S_1, \dots, S_p)$ such that:

- $\bigcup_{i=1}^{p} S_i = 1$,

- $S_i \cap S_j = \emptyset$ and
- S_i is a nest set of $H \setminus \bigcup_{j<i} S_j$.

The width of a nest set elimination is $\mathsf{nsw}(H, \Pi) = \max_i |S_i|$ and the *nest set width* $\mathsf{nsw}(H)$ *of* H is defined to be the smallest possible width of a nest set elimination order of H. It turns out that our notion of width generalises the notion of nest set width, that is, we have $\beta\text{-}\mathsf{how}(H) \leqslant \mathsf{nsw}(H)$. More particularly, any order $\prec$ obtained from a nest set elimination order $\Pi = (S_1, \dots, S_p)$ by ordering each S_i arbitrarely verifies $\mathsf{nsw}(H, \Pi) \geqslant \beta\text{-}\mathsf{how}(H, \prec)$.

We summarise and give formal proofs of the above discussion in the following theorem:

▶ **Theorem 12.** *For every hypergraph* $H = (V, E)$*, we have:* $\beta\text{-}\mathsf{htw}(H) \leqslant \beta\text{-}\mathsf{how}(H) \leqslant \mathsf{nsw}(H)$*. In particular, if* H *is* β*-acyclic then* $\beta\text{-}\mathsf{how}(H) = 1$.

The proof mainly follows from the following lemma:

▶ **Lemma 13.** *Let* $H = (V, E)$ *be a hypergraph and* S *be a nest set of* H *of size* k*. We let* f *to be the maximal element (for inclusion) of* $\{e \setminus S \mid e \in E, e \cap S \neq \emptyset\}$*, which exists by definition and* $(s_1, \dots, s_k)$ *an ordering of* S*. For every* $i \leqslant k$ *and* e *an edge of* $H/s_1/\dots/s_i$*, then either* $e \cap S \neq \emptyset$ *and* $e \subseteq f \cup S$ *or* $e \cap S = \emptyset$ *and* e *is an edge of* $H \setminus \{s_1, \dots, s_i\}$.

Proof. We prove this lemma by induction on i. For $i = 0$, it is clear since if $e \cap S \neq \emptyset$, then $e \setminus S \subseteq f$ by definition of f. Hence $e \subseteq f \cup S$. Now, assuming the hypothesis holds for some i, let $H_i = H/s_1/\dots/s_i$ and $H_{i+1} = H_i/s_{i+1}$. By definition, the edges of H_{i+1} are the edges of H_i without the vertex s_{i+1} and with the additional edge $N^*_{H_i}(s_{i+1})$. Let e be an edge of H_{i+1} that is not $N^*_{H_i}(s_{i+1})$. Either e was in H_i in which case the induction hypothesis still holds. Or $e = e' \setminus \{s_{i+1}\}$ for some edge e' of H_i. By induction, since $e' \cap S \neq \emptyset$, $e' \subseteq f \cup S$. Hence $e = e' \setminus \{s_{i+1}\} \subseteq f \cup S$ and the induction hypothesis follows. Now assume $e = N^*_{H_i}(s_{i+1})$. By induction, every edge of H_i that contains s_{i+1} is contained in $f \cup S$ hence $N^*_{H_i}(s_{i+1}) \subseteq f \cup S$ and the induction follows. ◀

Proof of Theorem 12. The inequality $\beta\text{-}\mathsf{htw}(H) \leqslant \beta\text{-}\mathsf{how}(H)$ is straightforward using:

- $\beta\text{-}\mathsf{htw}(H) = \max_{H' \subseteq H} \min_{\prec} \mathsf{how}(H', \prec)$
- $\beta\text{-}\mathsf{how}(H) = \min_{\prec} \max_{H' \subseteq H} \mathsf{how}(H', \prec)$

Indeed, let $\prec_0$ be an elimination order that is minimal for $\beta\text{-}\mathsf{how}(H, \prec)$. By definition, for $H' \subseteq H$, $\mathsf{how}(H', \prec_0) \geqslant \min_{\prec} \mathsf{how}(H', \prec)$. Hence

$$\beta\text{-}\mathsf{how}(H) = \max_{H' \subseteq H} \mathsf{how}(H', \prec_0) \geqslant \max_{H' \subseteq H} \min_{\prec} \mathsf{how}(H', \prec) = \beta\text{-}\mathsf{htw}(H).$$

We now prove $\beta\text{-}\mathsf{how}(H) \leqslant \mathsf{nsw}(H)$. Let $k = \mathsf{nsw}(H)$ and $\Pi = (S_1, \dots, S_p)$ a nest set elimination of H of width k, that is, for every i, $|S_i| \leqslant k$. Let $\prec$ be an order on $V = (v_1, \dots, v_n)$ with $v_1 \prec \dots \prec v_n$, obtained from Π by ordering each S_i arbitrarely, that is, if $x \in S_i$ and $y \in S_j$ with $i < j$, we require that $x \prec y$. We claim that $\beta\text{-}\mathsf{how}(H, \prec) \leqslant \mathsf{nsw}(H, \Pi)$. First of all, we observe that if $(S_1, \dots, S_p)$ is a nest set elimination order for H, then it is also a nest set elimination order for every $H' \subseteq H$, which is formally proven in [25, Lemma 4][4]. Consequently, it is enough to prove that $\mathsf{how}(H, \prec) \leqslant k$. This follows from Lemma 13. Indeed, let $(v_1, \dots, v_t)$ be the prefix of $(v_1, \dots, v_n)$ such that $S_1 = \{v_1, \dots, v_t\}$. By Lemma 13, when v_{i+1} is removed from $H_i^{\prec} = H/v_1/\dots/v_i$, then $N_{i+1} = N_{H_i}(v_{i+1})$ is included in $f \cup S_1$ since $N_{i+1} \cap S_1 \neq \emptyset$ (both contain v_{i+1}). Hence N_{i+1} is covered by at most t edges: f – which

[4] Lemma 4 of [25] establishes the result for a connected subhypergraph of H but the same proof works for non-connected subhypergraphs.

contains at least one element of S_1 – plus at most one edge for each remaining element of S_1. Hence, up to the removing of v_t, the hyperorder width of $\prec$ is at most $t \leqslant k$. Now, when removing $(v_1, \dots, v_t)$ from H, by Lemma 13 again, $H_t^{\prec} = H \setminus \{v_1, \dots, v_t\}$ since no edge of $H_t^{\prec}$ has a non-empty intersection with S_1. It follows that S_2 is a nest set of $H_t^{\prec}$ and we can remove it in a similar way to S_1 and so on. Hence $\beta\text{-}\mathsf{how}(H, \prec) \leqslant k = \mathsf{nsw}(H, \Pi)$ which settles the inequality stated in the theorem.

It directly implies that if H is β-acyclic then $\beta\text{-}\mathsf{how}(H) = 1$ since if H is β-acyclic, then $\mathsf{nsw}(H) = 1$ and $\beta\text{-}\mathsf{how}(H) \leqslant \mathsf{nsw}(H) = 1$ by the previously established bound. ◀

The goal of this paper is not to give a thorough analysis of β-fractional hyperorder width so we leave for future research several questions related to it. We observe that we do not know the exact complexity of computing or approximating the β-fractional hyperorder width of an input hypergraph H. It is very likely hard to compute exactly since it is not too difficult to observe that when H is a graph, $\beta\text{-}\mathsf{fhow}(H)$ is sandwiched between the half of the treewidth of H and the treewidth of H itself and it is known that treewidth is NP-hard to compute [4]. We also leave open many questions concerning how β-fractional hyperorder width compares with other widths such as (incidence) treewidth, (incidence) cliquewidth or MIM-width. For these measures of width, #SAT, a problem close to computing the number of answers in signed join queries, is known to be tractable (see [11] for a survey). We leave open the most fundamental question of comparing the respective powers of $\beta\text{-}\mathsf{fhtw}(\cdot)$ and $\beta\text{-}\mathsf{fhow}(\cdot)$:

► **Open Question 14.** *Does there exist a family $(H_n)_{n\in\mathbb{N}}$ of hypergraphs such that $(\beta\text{-}\mathsf{fhtw}(H_n))_{n\in\mathbb{N}}$ is bounded by a constant $k \in \mathbb{N}$ while $(\beta\text{-}\mathsf{fhow}(H_n))_{n\in\mathbb{N}}$ is unbounded?*

One may wonder why the definition of β-hyperorder width has not appeared earlier in the literature, as it just boils down to swapping a min and a max in the definition of β-hypertree width while enabling an easier algorithmic treatment. We argue that the expression of hypertree width in terms of elimination orders – which is not the widespread way of working with this width in previous literature – is necessary to make this definition interesting. Indeed, if one swaps the min and max in the traditional definition of β-hypertree width, we get the following definition: $\beta\text{-}\mathsf{htw}'(H, T) = \min_T \max_{H' \subseteq H} \mathsf{htw}(H', T)$ where T runs over every tree decomposition of H and hence is valid for every $H' \subseteq H$ since, as every edge of H is covered by T, so are the edges of H'. This definition, while being obtained in the same way as $\beta\text{-}\mathsf{how}(\cdot)$, is not really interesting however because it does not generalise the notion of β-acyclicity:

► **Lemma 15.** *There exists a family of β-acyclic hypergraphs (H_n) such that for every $n \in \mathbb{N}$, $\beta\text{-}\mathsf{htw}'(H_n) = n$.*

Proof. Consider the hypergraph H_n whose vertex set is $[n]$ and edges are $\{0, i\}$ for $i > 0$ and $[n]$. That is H_n is a star centered in 0 and has an edge containing every vertex. H_n is clearly β-acyclic (any elimination order that ends with 0 is a β-elimination order) but we claim that $\beta\text{-}\mathsf{htw}'(H_n) = n$. Indeed, let T be a tree decomposition for H_n. By definition, it contains a bag that contains $[n]$. Now consider the subhypergraph H_n' of H_n obtained by removing the edge $[n]$. The hypertree width of T wrt H_n' is n since one needs the edge $\{i, 0\}$ to cover vertex i in the bag $[n]$ since i appears only in this edge. ◀

Signed hyperorder width. In the case of signed join queries, one can deal with positive and negative atoms differently, which is not reflected by the definition of $\beta\text{-}\mathsf{fhow}(\cdot)$. We generalise these widths to signed hypergraphs by taking subhypergraphs only on the negative

part, generalising a notion of acyclicity introduced by Brault-Baron in [5] that mixes β- and α-acyclicities for signed hypergraphs. Let $H = (V, E_+, E_-)$ be a signed hypergraph. Given an order $\prec$ on V, the *signed hyperorder width* $\mathsf{show}(H, \prec)$ *of* $\prec$ *wrt* H is defined as $\mathsf{show}(H, \prec) = \max_{E' \subseteq E_-} \mathsf{how}((V, E_+ \cup E'), \prec)$. The *signed hyperorder width* $\mathsf{show}(H)$ *of* H is defined as $\mathsf{show}(H) = \min_\prec \mathsf{show}((H, \prec)$. Fractional version of these widths could easily be defined but will not be needed in this paper. The following directly follows from the definition:

▶ **Theorem 16.** *For every signed hypergraph $H = (V, E_+, E_-)$ and elimination order $\prec$ of V:*

- *If $E_+ = \emptyset$ then $\mathsf{show}(H, \prec) = \beta\text{-}\mathsf{how}(H, \prec)$. In particular, $\mathsf{show}(H) = \beta\text{-}\mathsf{how}(H)$.*
- *If $E_- = \emptyset$ then $\mathsf{show}(H, \prec) = \mathsf{how}(H, \prec)$. In particular, $\mathsf{show}(H) = \mathsf{how}(H)$.*

4.3 Complexity of exhaustive DPLL

The complexity of DPLL on a conjunctive query Q and order $\prec$ can be bounded in terms of the hyperorder width of $H(Q)$ wrt $\prec$:

▶ **Theorem 17.** *Let Q be a signed join query, $\mathbf{D}$ a database over domain D and $\prec$ an order on $\mathsf{var}(Q)$. Then $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ produces a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit C of size $O(\mathsf{poly}_k(|Q|)|\mathbf{D}|^{k+1})$ such that $\mathsf{rel}(C) = [\![Q]\!]^{\mathbf{D}}$ and:*

- *$k = \mathsf{fhow}(H(Q), \prec)$ if Q is positive,*
- *$k = \mathsf{show}(H(Q), \prec)$ if Q otherwise.*

Moreover, the runtime of $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ is at most $\tilde{O}(\mathsf{poly}_k(|Q|)|\mathbf{D}|^{k+1})$.

This is dedicated to proving Theorem 17. In this section, we fix a signed join query Q that has exactly one $\langle\rangle$-component[5], a database $\mathbf{D}$ and an order $\prec$ on $\mathsf{var}(Q) = \{x_1, \dots, x_n\}$ where $x_1 \prec \cdots \prec x_n$. We let D be the domain of $\mathbf{D}$, n be the number of variables of Q and m be the number of atoms of Q. To ease notation, we will write X instead of $\mathsf{var}(Q)$. For $i \leqslant n$, we denote $\{x_1, \dots, x_i\}$ by $X_{\preceq i}$. Similarly, $X_{\prec i} = X_{\preceq i} \setminus \{x_i\}$, $X_{\succ i} = \mathsf{var}(Q) \setminus X_{\preceq i}$ and $X_{\succeq i} = \mathsf{var}(Q) \setminus X_{\prec i}$. Finally, we let $\mathbf{R}$ be the set of (K, σ) such that $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ makes at least one recursive call to $\mathsf{DPLL}(K, \sigma, \mathbf{D}, \prec)$. We start by bounding the size of the circuit and the runtime in terms of the number of recursive calls:

▶ **Lemma 18.** $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ *produces a circuit of size at most* $O(|\mathbf{R}| \cdot |D| \cdot \mathsf{poly}(|Q|))$ *in time* $\tilde{O}(|\mathbf{R}| \cdot |D| \cdot \mathsf{poly}(|Q|))$.

Proof. Given $(K, \sigma) \in \mathbf{R}$, we associate to it every edges created in the circuit by the first recursive call with these parameters. There are at most $m + 1$ such edges for each $d \in D$. Indeed, for a value $d \in D$, there are at most $m + 1$ σ'-connected components for $\sigma' = \sigma \cup [x \leftarrow d]$ hence the first recursive call creates at most m edges between v_d and w_i and one edge between v and v_d. Observe that any other recursive call with these parameters will not add any extra edges in the circuit since it will result in a cache hit. Hence, the size of the circuit produced in the end is at most $|\mathbf{R}| \cdot |D| \cdot (m + 1) = O(|\mathbf{R}| \cdot |D| \cdot \mathsf{poly}(Q))$.

Moreover, each operation in Algorithm 1 can be done in time polynomial in $|Q|$ if one stores the relation using the correct datastructure. Indeed, if one sees a relation R on variables $x_1 \prec \cdots \prec x_n$ as a set of words on an alphabet D whose first letter is x_n and last is x_1, one can store it as a trie of size $\tilde{O}(|R|)$ and project R on $x_1, \dots, x_{n-1}$ in $\tilde{O}(1)$. Hence, we can

[5] The case where Q has many $\langle\rangle$-component can be easily dealt with by constructing the Cartesian product of each $\langle\rangle$-component of Q.

805 test for inconsistency in time $\tilde{O}(|Q|)$ after having fixed the highest variables in Q to a value
806 $d \in D$ by going over every atom of Q. Moreover computing the σ'-connected component can
807 be done in polynomial time in $|Q|$ since it boils down to finding the connected components
808 of a graph having at most m nodes. Such a graph can be constructed in polynomial time in
809 $|Q|$ by testing intersections of variables in atoms. Finally, from the previous discussion, a
810 recursive call to Algorithm 1 creates at most $m+1$ edges for each $d \in D$. Moreover, reading
811 and writing values in the cache can be done in time $\tilde{O}(\mathsf{poly}(|Q|)$ by using a hash table. We
812 pay the cost of reading the cache in a recursive call directly on Line 12. Hence the time for
813 each $(K, \sigma) \in \mathbf{R}$ is $\tilde{O}(|D| \cdot \mathsf{poly}(|Q|))$, hence a total time of $\tilde{O}(|\mathbf{R}| \cdot |D| \cdot \mathsf{poly}(|Q|))$. ◀

814 It remains to bound the size of $\mathbf{R}$. Lemma 19 characterises the structure of the elements
815 of $\mathbf{R}$ and Lemma 20 shows connections with with the structure of the hypergraph of Q. We
816 need a few notations. Let $Q' \subseteq Q$ be a subquery of Q and x, y two variables of Q' such that
817 $y \prec x$. An *x-path to y in Q'* is a list $x_0, a_0, \ldots, x_p$ where $a_i \in \mathsf{atoms}(Q')$ is an atom of Q' on
818 variables $\mathbf{x_i}$, x_i is a variable of $\mathbf{x_i}$, $x_0 = x$, $x_p = y$ and $x_i \preceq x$ for every $i \leqslant p$. Intuitively, it
819 maps to a path in the hypergraph of Q' that starts from x and is only allowed to use vertices
820 smaller than x. The *x-component of Q'* is the set of atoms a of Q' such that there exists an
821 x-path to a variable y of a in Q'.

822 It turns out that the recursive calls performed by DPLL are x-components of some $Q' \subseteq Q$
823 and $x \in X$ where Q' is obtained from Q by removing negative atoms. Intuitively, these
824 removed atoms are the ones that cannot be satisfied anymore by the current assignment of
825 variables.

826 ▶ **Lemma 19.** *Let $(K, \sigma) \in \mathbf{R}$ and let x be the biggest variable of K not assigned by σ. There*
827 *exists $\tau \supset \sigma$, a partial assignment of $X_{\succ x}$ such that K is the x-component of $Q \Downarrow \tau$.*

828 **Proof.** The proof is by induction on the order of recursive calls. We start by the first call,
829 $(Q, \langle\rangle)$. Since Q has one $\langle\rangle$-component, the x_n-component of Q is Q itself. Moreover, since
830 $Q \Downarrow \langle\rangle = Q$, we have that Q is the x_n-component of $Q \Downarrow \langle\rangle$. Now let $(K, \sigma) \in \mathbf{R}$. By
831 definition, the recursive call is made during the execution of an other recursive call with
832 parameters (K', σ'). Assume by induction that for (K', σ'), the statement of Lemma 19
833 holds. In other words, let x' be the biggest variable of K'. Then K' is the x'-component
834 of $Q \Downarrow \tau'$ for some partial assignment $\tau' \supset \sigma'$ of $X_{\succ x'}$. Moreover, by definition of DPLL,
835 $\sigma = \sigma''|_{\mathsf{var}(K)}$ where $\sigma'' = \sigma \cup [x' \leftarrow d]$ for some $d \in D$ and K is a σ''-component of $K' \Downarrow \sigma''$.
836 We claim that K is the x-component of $Q \Downarrow \tau$, where $\tau = \tau' \cup [x' \leftarrow d]$. First observe
837 that every atom a of K are in $Q \Downarrow \tau$. Indeed, if a is positive, then a is also in $Q \Downarrow \tau$ by
838 definition. Now if $a = \neg R(\mathbf{x})$ is negative, we claim that a is not inconsistent with τ. Indeed,
839 by induction, a is in $Q \Downarrow \tau'$, hence it is not inconsistent with τ'. Now since a is also in
840 K and K is a subset of $K' \Downarrow \sigma''$ and $\sigma''(x') = d$, we know that a is not inconsistent with
841 $\tau' \cup [x \leftarrow d]$ which is τ by definition. Hence a is in $Q \Downarrow \tau$
842 Now let a be an atom of K. Since x is a variable of K, there is an atom a_0 in K that
843 contains x. Moreover, K is a σ''-connected component of $K' \Downarrow \sigma''$. Hence, by definition, we
844 have a path in K from x (starting with atom a_0) to some variable y in a that do not use any
845 variable assigned by σ'', which is equivalent to say that it does not use any variable assign by
846 σ since $\sigma = \sigma''|_{\mathsf{var}(K)}$ by definition. Since x is the biggest variable of K that is not assigned
847 σ by definition, the path from a_0 to a uses only variables smaller than x. In other words,
848 there is an x-path to y in K, that is, every atom a of K are in the x-component of $Q \Downarrow \tau$.
849 Now let a be an atom that is in the x-component of $Q \Downarrow \tau$. We first show that a in K'.
850 First of all, observe that a is in $Q \Downarrow \tau'$ since $Q \Downarrow \tau \subseteq Q \Downarrow \tau'$. Now, by definition, there is a
851 path from x to some variable y of a in $Q \Downarrow \tau$ that uses only variables smaller than x. Recall

852 that a_0 is an atom of K containing variable x. Since $K \subseteq K'$, a_0 is also an atom of K'.
853 Hence there is a path from atom a_0 to atom a using only variables smaller than x, hence
854 also smaller than x'. In other words, a is in the x'-component of $Q \Downarrow \tau'$, hence in K' by
855 definition. Now, since a is in $Q \Downarrow \tau$, it is also in $\sigma'' \Downarrow K'$. Hence, it is in the σ''-component
856 of $\sigma'' \Downarrow K'$ that contains x, that is, it is in K, which concludes the proof. ◀

857 The following lemma establishes a connection between x-components and the structure
858 of the underlying hypergraph. In essence, it allows us to bound the number of atoms needed
859 to cover $X_{\succ x}$ in an x-component using the signed hyperorder width.

860 ▶ **Lemma 20.** *Let Q be a signed join query on variables $X = \{x_1, \dots, x_n\}$, x_i a variable of*
861 *Q and K_i its x_i-component. We let H be the hypergraph of Q, $H_1 = H$ and $H_{j+1} = H_j/x_j$.*
862 *We have $N_{x_i}(H_i) = \mathsf{var}(K_i) \cap X_{\succeq x_i}$.*

863 **Proof.** The proof is by induction on i. We start by proving the equality for $i = 1$. Since there
864 are no variable of K_1 smaller than x_1, it is clear that $K_1 = \{a \in \mathsf{atoms}(Q) \mid x_1 \in \mathsf{var}(a)\}$.
865 Hence $\mathsf{var}(K_1) \cap X_{\succeq x_1} = V(K_1)$. Moreover, $N_H(x_1)$ is exactly the set of variables of atoms
866 of Q containing x_1 since there is on hyperedge in H per atom of Q. Hence $V(K_1) \cap X_{\succeq x_1} =$
867 $N_H(x_1) = N_{H_1}(v_1)$, the last equality following by definition: $H_1 = H$.
868 Now assume that the equality has been established up to x_{i-1}. We start by proving that
869 $V(K_i) \cap X_{\succeq x_i} \subseteq N_{H_i}(x_i)$. Let $w \in V(K_i) \cap X_{\succeq x_i}$. Either $w \in N_H(x_i)$ and then it is clear
870 that $w \in N_{H_i}(x_i)$. Otherwise, there is, by definition, a x_i-path from x_i to an atom a of K_i
871 containing w of length greater than 1. Let x_j be the biggest node on this path that is not
872 neither x_i nor w. By definition of a x_i-path, $j < i$. Moreover, the first part of this path from
873 x_i to x_j is an x_j-path and similarly, the second part from x_j to w is a x_j-path. By induction,
874 we thus have $w \in N_{H_j}(x_j)$ and $x_j \in N_{H_j}(x_j)$. Hence w and x_i are neighbours in H_{j+1} since
875 the edge $N_{H_j}(x_j) \setminus \{x_j\}$ has been added in H_{j+1}. In particular, it means that w and x_i are
876 neighbours in H_i since $i \geqslant j + 1$, hence $w \in N_{H_i}(x_i)$.
877 Now let $w \in N_{H_i}(v_i)$. By definition, $w \in X_{\succeq x_i}$ since $x_1, \dots, x_{i-1}$ have been removed in
878 H_i. It remains to prove that w is in an atom a such that there is x_i-path to a. If x_i and w
879 are neighbours in H then it means they appears together in an atom of Q and it is clear
880 that $w \in \mathsf{var}(K_i)$. Otherwise, let j be the smallest value for which $w \in N_{H_j}(x_i)$ which exists
881 since $w \in N_{H_i}(x_i)$ and $j > 1$ since x_i and w are not neighbours in $H = H_1$. The minimality
882 of j implies that x_i and w are not neighbours in H_{j-1}. Since the only edge added in H_j is
883 $N_{H_{j-1}}(x_{j-1}) \setminus \{x_{j-1}\}$, it means that both x_i and w are neighbours of x_{j-1} in H_{j-1}, that is,
884 $x_i \in N_{H_{j-1}}(x_{j-1})$ and $w \in N_{H_{j-1}}(x_{j-1})$. By induction, both x_i and w are variables of K_{j-1}.
885 In other words, there exists a x_{j-1}-path to an atom a containing x_i and an x_{j-1}-path to an
886 atom a' containing w w. By composing both paths, it gives a x_{j-1}-path – which is itself a
887 x_i-path – from x_i to w. Hence $w \in \mathsf{var}(K_i)$. ◀

888 We are now ready to prove the upperbound on $|\mathbf{R}|$ depending on the width of $\prec$.

889 ▶ **Lemma 21.** *Let m be the number of atoms of Q and n the number of variables. We have:*
890 - *if Q is a positive join query, $|\mathbf{R}| \leqslant n|\mathbf{D}|^k$ where $k = \mathsf{fhow}(H(Q), \prec)$.*
891 - *otherwise $|\mathbf{R}| \leqslant nm^{k+1}|\mathbf{D}|^k$ where $k = \mathsf{show}(H(Q), \prec)$.*

892 **Proof.** We start with the case where Q is a positive join query. Let $(K, \sigma) \in \mathbf{R}$. In this case,
893 we know by Lemma 19 that K is the x_i-component of $Q \Downarrow \tau$ for some $\tau \supset \sigma$. Now, since
894 Q does not have negative atoms, $Q = Q \Downarrow \tau$ since $Q \Downarrow \tau$ is obtained from Q by removing
895 negative atoms only. In other words, K is the x_i-component of Q and σ assigns the variables
896 of K that are greater than x_i. We also know that σ is not inconsistent with the atoms of

897 Q, otherwise, DPLL would return $\bot$. Hence, σ satisfies every atom of K when projected on
898 $X_{\succ x_i}$. By Lemma 20, $\mathsf{var}(K) \cap X_{\succ x_i} = N_{x_i}(H_i)$ where H_i is defined as in Lemma 20. Hence,
899 by definition, there exits a fractional cover of $N_{H_i}(x_i)$ using the atoms of Q with value at
900 most $k = \mathsf{fhow}(H(Q), \prec)$. Hence, σ can be seen as the projection on $\mathsf{var}(K) \cap X_{\succ x_i}$ of an
901 answer of the join of the atoms involved in the fractional cover. By Theorem 2, this join
902 query as at most $|\mathbf{D}|^k$ answers. Hence, there are at most $n|\mathbf{D}|^k$ possible elements in $\mathbf{R}$: there
903 are at most n x_i-component (one for each $i \leqslant n$), and at most $|db|^k$ associated σ.
904 Now, the case of signed query is a bit more complicated. Again, for $(K, \sigma) \in \mathbf{R}$, we
905 know that K is the x_i-component of $Q \Downarrow \tau$ for some $\tau \supset \sigma$ and, as before, τ is compatible
906 with every positive atom in $Q \Downarrow \tau$. Moreoever, if $\neg R(\mathbf{x})$ is an atom of $Q \Downarrow \tau$, then τ is
907 compatible with $R(\mathbf{x})$, since otherwise $\neg R(\mathbf{x})$ would not be in $Q \Downarrow \tau$. Now, let H' be the
908 hypergraph of $Q \Downarrow \tau$. By definition, it is a subhypergraph of $H(Q)$, where only negative
909 edges have been removed. Hence, by Lemma 20, $\mathsf{var}(K) \cap X_{\succ x_i} = N_{H'_i}(x_i)$ is covered by at
910 most $k = \mathsf{sfhow}(H')$ edges. Hence, σ, which corresponds to τ restricted to $\mathsf{var}(K) \cap X_{\succ x_i}$,
911 can be seen as the projection of an answer of a positive join query having at most k atoms.
912 Indeed, even if an edge used to cover $\mathsf{var}(K) \cap X_{\succ x_i}$ is associated to a negative atom, we
913 know that τ is compatible with the positive part of this atom. Hence, (K, σ) can be obtained
914 as follows: pick at most k atoms of Q, join their positive parts and take τ a solution of this
915 join projected of $X_{\succ x}$. Now K is the x-component of $Q \Downarrow \tau$ and $\sigma = \tau|_{\mathsf{var}(K)}$. Hence, there
916 are at most n choice of variable, m^{k+1} choice of subset of atoms of size at most k and each
917 join has at most $|\mathbf{D}|^k$ answers, which amounts to $nm^{k+1}|\mathbf{D}|^k$ possible $(K, \sigma) \in \mathbf{R}$, hence
918 $|\mathbf{R}| \leqslant nm^{k+1}|\mathbf{D}|^k$. ◀

919 Now Theorem 17 is a direct corollary of Lemmas 18 and 21. If Q is not $\langle\rangle$-connected, then
920 the first recursive call of DPLL will simply break Q into at most $O(m)$ connected component
921 and recursively called itself on each now $\langle\rangle$-components of Q.
922 One may wonder why we do not use fractional width when Q contains negative atoms.
923 The proof of Lemma 21 breaks in this case when we try to bound the number, for a given x,
924 of x-component K that can appear in recursive calls. In the proof of Lemma 21, we bound it
925 by taking at subset of at most k atoms of Q. To do it with fractional cover, one would need
926 to consider every combination of atoms of Q having fractional cover at most k which we
927 did not manage to bound by a polynomial in Q. We therefore leave this question open for
928 future research but observe that it would give a complexity of at most $\tilde{O}(2^m|\mathbf{D}|^{k+1})$ which is
929 polynomial wrt data complexity.
930 Another improvement that could be made in Theorem 17 is to have a dependency of
931 $|\mathbf{D}|^k$ instead of $|\mathbf{D}|^{k+1}$. The extra $|\mathbf{D}|$ comes from the for-loop on Line 6 that explores every
932 element of the domain. One could improve the complexity here by exploring only the values
933 $d \in D$ such that setting x to d does not make Q inconsistent. One could use the Leapfrog
934 join proposed in the Leapfrog Triejoin algorithm [34, Section 3.1 and 3.2] to explore these
935 candidates and we believe it would shave the extra $|\mathbf{D}|$ factor. However, the complexity
936 analysis is already complicated enough and we decided to leave this for future investigation.

937 5 Tractability results for queries

938 In this section, we connect the tractability result on direct access on ordered circuit of
939 Section 3 with the algorithm presented in Section 4 to obtain tractability results concerning
940 the complexity of direct access on signed join queries. We compare this results with previous
941 work.

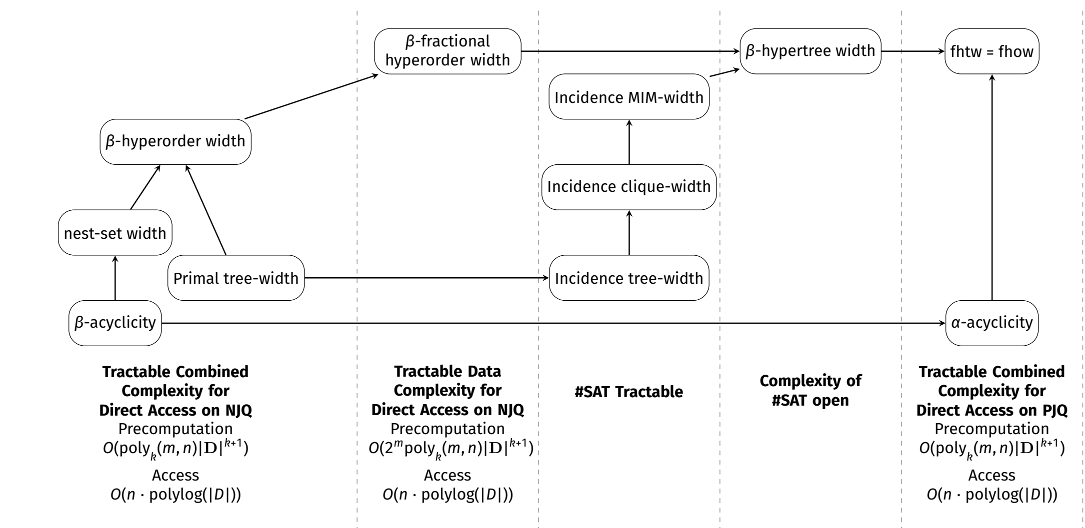


Figure 5 Landscape of hypergraph measures and known inclusions with tractability results for direct access on negative join queries (NJQ), direct access on positive join queries (PJQ) shown and #SAT on CNF formulas. Here n is the number of variables, m the number of atoms, $\mathbf{D}$ the database, D the domain and k the width measure ($k = 1$ for α- and β-acyclicity). In the case of CNF formulas, m stands for the number of clauses, the size of the database is at most m and the domain is $\{0, 1\}$. An arrow between two classes indicates inclusion.

942 ▶ **Theorem 22.** *Given a signed join query Q, an order $\prec$ on $\mathsf{var}(Q)$ and a database*
943 *$\mathbf{D}$ on domain D, we can solve the direct access problem for $\prec_{\mathsf{lex}}$ with precomputation*
944 *$\tilde{O}(|\mathbf{D}|^{k+1}\mathsf{poly}_k(|Q|))$ and access time $O(\mathsf{poly}(n) \cdot \mathsf{polylog}(|D|))$ where $n = |\mathsf{var}(Q)|$ and:*
945 - *if Q does not contain any negative atom, then $k = \mathsf{fhtw}(H(Q), \succ)$,*
946 - *Otherwise $k = \mathsf{show}(H(Q), \succ)$.*

947 **Proof.** It is a corollary of Theorems 7 and 17. Given Q and $\mathbf{D}$, we call $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \succ)$ to
948 construct a $\prec$-ordered circuit C on domain D and variables $X = \mathsf{var}(Q)$, computing $[\![Q]\!]^{\mathbf{D}}$.
949 The circuit is of size $O(|\mathbf{D}|^{k+1}\mathsf{poly}_k(|Q|))$ and is computed in time $\tilde{O}(|\mathbf{D}|^{k+1}\mathsf{poly}_k(|Q|))$
950 with k as given in the statement. Now, we execute the precomputation step described in
951 Section 3.2 in time $O(|C|\mathsf{poly}(n) \cdot \mathsf{polylog}(|D|)) = \tilde{O}(|\mathbf{D}|^{k+1}\mathsf{poly}_k(|Q|))$ to get a count-labelled
952 circuit C computing $[\![Q]\!]^{\mathbf{D}}$. This terminates the precomputation part which has indeed the
953 desired complexity.
954 Now, to find the ith solution in $[\![Q]\!]^{\mathbf{D}}$, we simply find the ith solution of $\mathsf{rel}(C)$ using the
955 algorithm of Section 3.2. By Theorem 7, the access time is hence $O(\mathsf{poly}(n) \cdot \mathsf{polylog}(|D|)) =$
956 $O(\mathsf{poly}(n) \cdot \mathsf{polylog}(|D|))$. ◀

957 Figure 5 summarizes our contributions for join queries with negations and summarizes
958 how our contribution is located in the landscape of known tractability results. Even if our
959 result applies to signed conjunctive query, we summarize our contribution only for negative
960 join queries and positive join queries since it allows to compare hypergraph measures (where
961 tractability of signed queries is stated using signed hypergraphs parameters). The two
962 left-most columns of the figure are contributions of this paper (Theorem 22), the right-most
963 column is known from [8] but can be recovered in our framework (see discussion below). The
964 third and fourth column states the complexity of #SAT and is discussed below. A complete
965 presentation of the results stated in this figure can be found in [10, Chapter 2].

Negative join queries and #SAT. Theorem 22 generalises many tractability results from the literature. First of all, our result can directly be applied to #SAT, the problem of counting the number of satisfying assignment of a CNF formula. A CNF formula F with m clauses can directly be transformed into a negative join query Q_F with m atoms having the same hypergraph and into a database $\mathbf{D}_F$ on domain $\{0,1\}$ and of size at most m such that $[\![Q_F]\!]^{\mathbf{D}_F}$ is the set of satisfying assignments of F. Indeed, a clause can be seen as the negation of a relation having exactly one tuple. For example, $x \vee y \vee \neg z$ can be seen as $\neg R(x,y,z)$ where R contains the tuple $(0,0,1)$. Hence, Theorem 22 generalizes both [11] and [7] by providing a compilation algorithm for β-acyclic queries to any domain size and to the more general measure of β-hyperorder width. It also shows that not only counting is tractable but also the more general direct access problem.

Theorem 22 also generalises the results of [25] which shows the tractability of the evaluation of negative join queries with bounded nest set width. Since a negative join query with nest set width k has β-hyperorder width at most k by Theorem 12, Theorem 22 implies that direct access is tractable for the class of queries with bounded nest set width. In particular, counting the number of answers is tractable for this class, a question left open in [25].

Direct access for positive conjunctive queries. Theorem 22 allows to recover the tractability of direct access for positive join queries with bounded fractional hypertree width proven in [12, 8]. Indeed, given an order $\prec$ on the vertices of a hypergraph, [8] introduces the notion of incompatibility number of $\prec$ which corresponds exactly to its fractional hyperorder width. Hence Theorem 22 implies the same tractability results for positive join query as [8, Theorem 10]. The complexity bounds from this paper are however better than ours and proven optimal since the preprocessing is of the form $\mathsf{poly}_k(Q)|\mathbf{D}|^k$ where we have $\mathsf{poly}_k(Q)|\mathbf{D}|^{k+1}$. We nevertheless believe that with a more careful analysis of the implementation of Algorithm 1, we could match this upper bound although this is not the focus of this paper. Another strong point of [12] (and also [9, Theorem 39] which is the arXiv version of [8]) is that it handles conjunctive queries, that is, join queries with projection which is not covered by Theorem 22. We demonstrate the versatility of the circuit-based approach by showing how one can also handle quantifiers directly on the circuit.

▶ **Theorem 23.** *Let C be a $\prec$-order circuit on domain D, variables $X = \{x_1, \dots, x_n\}$ such that $x_1 \prec \dots \prec x_n$ and $j \leqslant n$. One can compute in time $O(|C| \cdot \mathsf{poly}(n) \cdot \mathsf{polylog}(|D|))$ a circuit C' of size at most $|C|$ such that $\mathsf{rel}(C') = \mathsf{rel}(C)|_{\{x_1,\dots,x_j\}}$.*

Proof. Let v be a decision gate on variable x_k with $k > j$. By definition, every decision-gate in the circuit rooted at v tests a variable $y \in \{x_{k+1}, \dots, x_n\}$. Hence $\mathsf{rel}(v) \subseteq D^Y$ with $Y \subseteq \{x_k, \dots, x_n\}$. Moreover, by computing a count label of C in time $O(|C| \cdot \mathsf{poly}(n) \cdot \mathsf{polylog}(|D|))$ as in Lemma 8, we can decide whether $\mathsf{rel}(v)$ is the empty relation in time $O(1)$ by simply checking whether $\mathsf{nrel}_C(v, d_0) \neq 0$ where d_0 is the largest element of D. We construct C' by replacing every decision-gate v on a variable x_k with $k > j$ by a constant gate $\top$ if $\mathsf{rel}(v) \neq \emptyset$ and $\bot$ otherwise. We clearly have that $|C'| \leqslant |C|$ and from what precedes, we can compute C' in $O(|C| \cdot \mathsf{poly}(n) \cdot \mathsf{polylog}(|D|))$. Moreover, it is straightforward to show by induction that every gate v' of C' which corresponds to a gate v of C computes $\mathsf{rel}(C)|_{\{x_1,\dots,x_j\}}$, which concludes the proof. ◀

Now we can use Theorem 23 to handle conjunctive querie by first using Theorem 17 on the underlying join query to obtain a $\prec$-circuit and then by projecting the variables directly in the circuit. This approach works only when the largest variables in the circuits are the quantified variables. It motivates the following definition: given a hypergraph $H = (V, E)$,

an elimination order $(v_1, \dots, v_n)$ of V is S-connex if and only if there exists i such that $\{v_i, \dots, v_n\} = S$. In other words, the elimination order starts by eliminating $V \setminus S$ and then proceeds to S. Given a conjunctive query Q and an elimination order $\prec$ on $\mathsf{var}(Q)$, we say that the elimination is free-connex if it is a $\mathsf{free}(Q)$-connex elimination order of $H(Q)$ where $\mathsf{free}(Q)$ are the free variables of Q[6]. We directly have the following:

▶ **Theorem 24.** *Given a conjunctive query $Q(Y)$, a free-connex order $\succ$ on $\mathsf{var}(Q)$ and a database $\mathbf{D}$ on domain D, we can solve the direct access problem for $\prec_{\mathsf{lex}}$ with precomputation $\tilde{O}(|\mathbf{D}|^{k+1}\mathsf{poly}_k(|Q|))$ and access time $O(n \cdot \mathsf{polylog}(|D|))$ where $n = |\mathsf{var}(Q)|$ and:*

- *if Q does not contain any negative atom, then $k = \mathsf{fhtw}(H(Q), \succ)$,*
- *Otherwise $k = \mathsf{show}(H(Q), \succ)$.*

Proof. By running $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \succ)$, one obtains a $\prec$-ordered circuit computing $[\![Q]\!]^{\mathbf{D}}$. The size of the circuit is $O(|\mathbf{D}|^{k+1}\mathsf{poly}_k(|Q|))$ by Theorem 17. Now, $\succ$ is free-connex, that is, $\succ$ is of the form $z_1 \succ \cdots \succ z_n$ and there exists i such that $\{z_i, \dots, z_n\} = \mathsf{free}(Q)$. Hence, with respect to the relation $\prec$, we have that $\mathsf{var}(Q) \setminus \mathsf{free}(Q)$ are maximal. Hence by Theorem 23, we can construct a $\prec$-circuit of size at most $O(|\mathbf{D}|^{k+1}\mathsf{poly}_k(|Q|))$ computing $[\![Q]\!]^{\mathbf{D}}|_Y = [\![Q(Y)]\!]^{\mathbf{D}}$, which conclude the proof using Theorem 7. ◀

We observe that our notion of free-connex elimination order for Q is akind to [9, Definition 38] with two differences: first, in [9], it is allowed to only specify a preorder on $\mathsf{free}(Q)$ and the complexity of the algorithm is then stated with the best possible compatible ordering, which would be possible in our framework too. The second difference is that the order are presented in reverse, that is, in their definition, the order starts with free variables and ends with quantified variables. We decided to present free-connexity of elimination orders this way to make this notion corresponds to the existing notion of free-connexity using tree decomposition. Now, Theorem 24 constructs a direct access for $\prec_{\mathsf{lex}}$ when $\succ$ is free-connex, so Theorem 24 proves the same tractability result as [9, Theorem 39], again with an extra $|D|$ factor but compatible with negative and signed conjunctive queries.

6 Conclusion and Future Work

In this paper, we have proven new tractability results concerning the direct access of the answers of signed conjunctive queries. In particular, we have introduce a framework unifying the positive and the signed case using factorised representation of the answer sets of the query. This opens many new avenue of research. First of all, contrary to the positive query case, we do not yet have lower bounds parameterised by β-hyperorder width on the preprocessing and access time needed for solving direct access tasks. Having a better understanding of what happens on fractional relaxation of β-hyperorder width would be a first step toward proving such lower bounds. Another question remains concerning the complexity of the DPLL-style algorithm. We strongly believe that with the right data structures, the complexity of DPLL on queries having fractional hypertree width k should be of the order $|\mathbf{D}|^k$ instead of the $|\mathbf{D}|^{k+1}$, which would allow us to match the existing upper bounds exactly. We leave a more involved analysis of this algorithm for future work.

Finally, we belive that the circuit representation that we are using is promising for answering different kind of aggregation tasks and hence generalising existing results to

[6] The notion of S-connexity already exists for tree decompositions. We use the same name here as the existence of an S-connex tree decomposition of (fractional) hypertree width k is equivalent to the existence of an S-connex elimination order of (fractional) hyperorder width k.

1053 the case of signed conjunctive queries. For example, we believe that FAQ and AJAR
1054 queries [24, 21] could be solved using this data structure. Indeed, it looks possible to
1055 annotate the circuit with semi-ring elements and to project them out in a similar fashion as
1056 Theorem 23. Similarly, we believe that the framework of [15] for solving direct access tasks
1057 on conjunctive queries with aggregation operators may be generalised in a similar way to the
1058 class of ordered $\{\times, \mathsf{dec}\}$-circuits.

References

1. Guillaume Bagan. *Algorithmes et complexité des problèmes d'énumération pour l'évaluation de requêtes logiques. (Algorithms and complexity of enumeration problems for the evaluation of logical queries).* PhD thesis, University of Caen Normandy, France, 2009.
2. 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.
3. Nurzhan Bakibayev, Tomáš Kočiskỳ, Dan Olteanu, and Jakub Závodnỳ. Aggregation and ordering in factorised databases. *Proceedings of the VLDB Endowment*, 6(14):1990–2001, 2013.
4. Hans L. Bodlaender. A linear time algorithm for finding tree-decompositions of small treewidth. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 226–234. ACM, 1993.
5. Johann Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre.* PhD thesis, Université de Caen, 2013.
6. Johann Brault-Baron. Hypergraph acyclicity revisited. *ACM Computing Surveys (CSUR)*, 49(3):1–26, 2016.
7. Johann Brault-Baron, Florent Capelli, and Stefan Mengel. Understanding model counting for beta-acyclic CNF-formulas. In *32nd International Symposium on Theoretical Aspects of Computer Science*, volume 30 of *LIPIcs*, pages 143–156. Schloss Dagstuhl, 2015.
8. 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.
9. Karl Bringmann, Nofar Carmeli, and Stefan Mengel. Tight fine-grained bounds for direct access on join queries. *arXiv preprint arXiv:2201.02401*, 2022.
10. Florent Capelli. *Structural restrictions of CNF-formulas: applications to model counting and knowledge compilation.* PhD thesis, Université Paris Diderot, Sorbonne Paris Cité, 2016. URL: https://florent.capelli.me/publi/these_capelli.pdf.
11. Florent Capelli. Understanding the complexity of #SAT using knowledge compilation. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–10. IEEE Computer Society, 2017. doi:10.1109/LICS.2017.8005121.
12. 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.
13. 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.
14. 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*, STOC '77, pages 77–90, New York, NY, USA, 1977. ACM. doi:10.1145/800105.803397.
15. Idan Eldar, Nofar Carmeli, and Benny Kimelfeld. Direct access for answers to conjunctive queries with aggregation. *arXiv preprint arXiv:2303.05327*, 2023.
16. 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, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*, pages 109–127. Springer, 2018.
17. Robert Ganian, André Schidler, Manuel Sorge, and Stefan Szeider. Threshold treewidth and hypertree width. *Journal of Artificial Intelligence Research*, 74:1687–1713, 2022.
18. Georg Gottlob and Reinhard Pichler. Hypergraphs in model checking: Acyclicity and hypertree-width versus clique-width. *SIAM Journal on Computing*, 33(2):351–378, 2004.

1111 **19** Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Transac-*
1112 *tions on Algorithms (TALG)*, 11(1):4, 2014.
1113 **20** Jinbo Huang and Adnan Darwiche. DPLL with a Trace: From SAT to Knowledge Compilation.
1114 In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*,
1115 pages 156–162, 2005.
1116 **21** Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over
1117 annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium*
1118 *on Principles of Database Systems*, pages 91–106, 2016.
1119 **22** Jens Keppeler. *Answering Conjunctive Queries and FO+ MOD Queries under Updates*. PhD
1120 thesis, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät, 2020.
1121 **23** Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. Faq: questions asked frequently. *arXiv*
1122 *preprint arXiv:1504.04044*, 2015.
1123 **24** Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. Faq: questions asked frequently. In
1124 *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database*
1125 *Systems*, pages 13–28, 2016.
1126 **25** Matthias Lanzinger. Tractability beyond β-acyclicity for conjunctive queries with negation
1127 and sat. *Theoretical Computer Science*, 942:276–296, 2023.
1128 **26** Dan Olteanu. Factorized databases: A knowledge compilation perspective. In *AAAI Workshop:*
1129 *Beyond NP*, 2016.
1130 **27** Dan Olteanu and Jakub Závodnỳ. Factorised representations of query results: size bounds and
1131 readability. In *Proceedings of the 15th International Conference on Database Theory*, pages
1132 285–298. ACM, 2012.
1133 **28** Dan Olteanu and Jakub Závodnỳ. Size bounds for factorised representations of query results.
1134 *ACM Transactions on Database Systems (TODS)*, 40(1):1–44, 2015.
1135 **29** Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results.
1136 *ACM Transactions on Database Systems*, 40(1):1–44, March 2015.
1137 **30** S. Ordyniak, D. Paulusma, and S. Szeider. Satisfiability of acyclic and almost acyclic CNF
1138 formulas. *Theoretical Computer Science*, 481:85–99, 2013.
1139 **31** Reinhard Pichler and Sebastian Skritek. Tractable counting of the answers to conjunctive
1140 queries. *Journal of Computer and System Sciences*, 79:984–1001, September 2013.
1141 **32** Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. Combining
1142 component caching and clause learning for effective model counting. *Theory and Applications*
1143 *of Satisfiability Testing*, 4:7th, 2004.
1144 **33** Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over
1145 factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*,
1146 pages 3–18. ACM, 2016.
1147 **34** Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole
1148 Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International*
1149 *Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106.
1150 OpenProceedings.org, 2014. `doi:10.5441/002/icdt.2014.13`.
1151 **35** Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh*
1152 *International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 82–94.
1153 VLDB Endowment, 1981.