Logical Methods in Computer Science
Volume 22, Issue 1, 2026, pp. 24:1–24:53
https://lmcs.episciences.org/

Submitted Nov. 13, 2024
Published Mar. 12, 2026

DIRECT ACCESS FOR CONJUNCTIVE QUERIES WITH NEGATIONS

FLORENT CAPELLI [a], NOFAR CARMELI [b], OLIVER IRWIN [c], AND SYLVAIN SALVATI [c]

[a] Université d'Artois, CNRS, UMR 8188, Centre de Recherche en Informatique de Lens (CRIL), F-62300 Lens, France
e-mail address: capelli@cril.fr

[b] Inria, LIRMM, Université de Montpellier, CNRS, UMR 5506, F-34000 Montpellier, France
e-mail address: nofar.carmeli@inria.fr

[c] Université de Lille, CNRS, Inria, UMR 9189 - CRIStAL, F-59000 Lille, France
e-mail address: oliver.irwin@univ-lille.fr, sylvain.salvati@univ-lille.fr

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 k, the k^{th} 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 polynomial preprocessing time. Previous work has precisely characterised the tractable classes and given fine-grained lower bounds on the preprocessing 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.

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 was introduced by Bagan, Durand, Grandjean and Olive in [BDGO08]

Key words and phrases: Conjunctive queries, factorised databases, direct access, hypertree decomposition.
This paper is a longer version of [CI24] which appeared at ICDT 2024.
We would like to thank the anonymous reviewers of this paper for their numerous comments which helped improve the quality of this paper. This work was supported by project ANR KCODA, ANR-20-CE48-0004.

DOI:10.46298/LMCS-22(1:24)2026

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 [BDGO08] or sampling without repetition [CZB+20, Kep20] 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 that is cheaper to compute than the complete answer set. This approach is thus separated in two phases: a *preprocessing phase* where the data structure 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 possibly higher) and constant access time (in the RAM model which will be made precise in Section 2. 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 [BDGO08], Bagan, Durand, Grandjean and Olive give an algorithm for solving direct access tasks with linear preprocessing time and constant access time on a class of first order logic formulas and bounded degree databases. Bagan [Bag09] 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 over general databases. In [CZB+20], 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 [Yan81] to the tractability of counting the number of answers of conjunctive queries [PS13] 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 [CTG+23] for the special case of acyclic queries and by Bringmann, Carmeli and Mengel in [BCM22a] for any join queries. More recently, Eldar, Carmeli and Kimelfeld [ECK23] have 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 [OPS13, BB13] and the class of bounded nested-set width queries [Lan23]. The former has been shown to also support efficient (weighted) counting [BCM15, Cap17]. 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 [BCM22a] 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 unifies 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 [BCM22a] for the worst positive query we could construct by removing some negative atoms of Q and turning the others into positive atoms.

As a byproduct, we introduce a new notion of hypergraph width based on elimination order, called the β-hyperorder width. It is a hereditary width notion, meaning that the width of every subhypergraph does not exceed the width of the original hypergraph. It 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 [GP04], 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.

We give two different algorithms for solving direct access on signed queries: the first method uses the results on positive join queries from [BCM22a] as an oracle. We show that it has optimal data complexity but has an exponential dependency on the size of the query. The second algorithm 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 order $\prec$ chosen on the variables of Q, some orders being harder than other and we are able to measure their complexity. We then show that with a second light preprocessing step on the circuit itself, we can answer direct access tasks on the circuit in time $\text{poly}(n)\,\text{polylog}(D)$ 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 [OZ12], 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 [Olt16, SOC16, BKOZ13, OZ15]. 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 [Cap17] where they are useful to deal with β-acyclic CNF formulas.

Difference with conference version. This paper is a longer and improved version of [CI24]. The longer version contains every proof missing from [CI24] but also several new results. First, we improve the complexity of the preprocessing step by a $|\mathbf{D}|$ factor. This is achieved using a simple algorithmic trick during the computation of the circuit to avoid exploring every possible domain value when branching on a variable. This improvement allows us to match the complexity from [BCM22a] when running our algorithm on a join query without negative

atoms. Second, we match this improved upper bound with a lower bound, leveraging results from [BCM22a], which establishes the optimality of our approach. Indeed, we show that for every signed join query Q and order $\prec$ on the variables of Q, there exists k (depending on Q and $\prec$) such that direct access on signed join queries is possible with preprocessing time $\tilde{\mathcal{O}}(|\mathbf{D}|^k f(|Q|))$ and direct access $\mathcal{O}(f(|Q|)\operatorname{polylog}(|\mathbf{D}|))$ but if one is able to solve direct access for Q with preprocessing time $\mathcal{O}(|\mathbf{D}|^{k-\varepsilon} g(|Q|)\operatorname{polylog}(|\mathbf{D}|))$ for some g and $\varepsilon > 0$, and access time $\mathcal{O}(\operatorname{polylog}(|\mathbf{D}|))$, then the Zero-Clique conjecture, a widely believed conjecture from fine-grained complexity, is false.

Organisation of the paper. The paper is organised as follows: Section 2 introduces the notations and concepts necessary to understand the paper. We give in Section 3 a reduction of the problem of solving direct access on signed join query to the problem of solving direct access on positive queries. It directly gives us a full characterisation of the tractable cases of signed join queries but with exponential query complexity. We then present the family of circuits we use to represent database relations and the direct access algorithm in Section 4. Section 5 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 5.2 using hypergraph decompositions defined in Section 3.3. Finally Section 6 explicitly states the results we obtain by combining both techniques together and improves the combined complexity of the algorithm obtained in Section 3 by reduction. Moreover, it explains how one can go from join queries to conjunctive queries by existentially projecting variables directly in the circuit. Finally, Section 7 studies the special cases of negative join queries and of SAT and compares our result to the existing literature on these topics.

2. Preliminaries

General mathematical notations. Given $n \in \mathbb{N}$, we denote by $[n]$ the set $\{1, \dots, n\}$. When writing down complexity, we use the notation $\operatorname{poly}(n)$ to denote that the complexity is polynomial in n, $\operatorname{poly}_k(n)$ 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 $\operatorname{polylog}(n)$ to denote that the complexity is polynomial in $\mathsf{log}(n)$. Moreover, we use $\tilde{\mathcal{O}}(n)$ to indicate that polylogarithmic factors are ignored, that is, the complexity is $\mathcal{O}(n \operatorname{polylog}(n))$.

Tuples and relations. Let D and X be finite sets. A (named) *tuple* on domain D and variables set X is a mapping from X to D. We denote by D^X the set of all tuples on domain D and variables set X. A *relation R on domain D and variables set X* is a subset of tuples, that is, $R \subseteq D^X$. We will denote the variables set X of R by $\mathsf{var}(R)$. The number of tuples in a relation R, or the *size of R* will often be denoted by $\#R$. Given a tuple $\tau \in D^X$ and $Y \subseteq X$, we denote by $\tau|_Y$ the tuple on domain D and variables set 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 set $\{x\}$ 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 denote by $\tau_1 \bowtie \tau_2$ the tuple on domain D and variables set $X_1 \cup X_2$ 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 $\tau_1 \times \tau_2$ instead of $\tau_1 \bowtie \tau_2$. The *join* $R_1 \bowtie R_2$ *of* R_1 *and* R_2, for two relations R_1 and R_2 on domain D and variables set X_1 and 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 $R_1 \,\overline{\cup}\, R_2$, is the relation on domain D and variables set $X_1 \cup X_2$ 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 variables set X to a domain D. We denote $\sigma_F(R)$ as the subset of R where the formula F is true, where F is a conjunction of atomic formulas of the form $x = d, x < d, x \leqslant d, x > d, x \geqslant d$ for a variable $x \in X$ and domain value $d \in D$. For example, $\sigma_{x \leqslant d}(R)$ contains every tuple $\tau \in R$ such that $\tau(x) \leqslant d$. Throughout the paper, we will assume that both the domain D and the variable set X are totally ordered. The order on D will be denoted as $<$ and the order on X as $\prec$ and we will often write $D = \{d_1, \ldots, d_p\}$ with $d_1 < \cdots < d_p$ and $X = \{x_1, \ldots, x_n\}$ with $x_1 \prec \cdots \prec x_n$. For $Y \subseteq X$, we denote by $\min_\prec(Y)$ (or $\min(Y)$ when $\prec$ is clear from context), the minimal element of Y.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 k^{th} tuple in R for the $\prec_{\mathsf{lex}}$-order.

We will often use the following observation:

Lemma 2.1. *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. A visual representation of this proof and of the meaning of the statement is given in Fig. 1. A complete proof can be found in Appendix A. □

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

$$Q \coloneqq R_1(\mathbf{x_1}), \ldots, R_\ell(\mathbf{x}_\ell), \neg S_{\ell+1}(\mathbf{x}_{\ell+1}), \ldots, \neg S_m(\mathbf{x_m})$$

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 $a_i = |\mathbf{x_i}|$ and $a_j = |\mathbf{x_j}|$. The *answers of* Q *over* $\mathbf{D}$ is the relation $[\![Q]\!]^{\mathbf{D}} \subseteq D^{\mathsf{var}(Q)}$ defined as the set of $\sigma \in D^{\mathsf{var}(Q)}$ such that for every $i \leqslant \ell$, $\sigma(\mathbf{x_i}) \in R_i^{\mathbf{D}}$ and for every $\ell < i \leqslant m$,

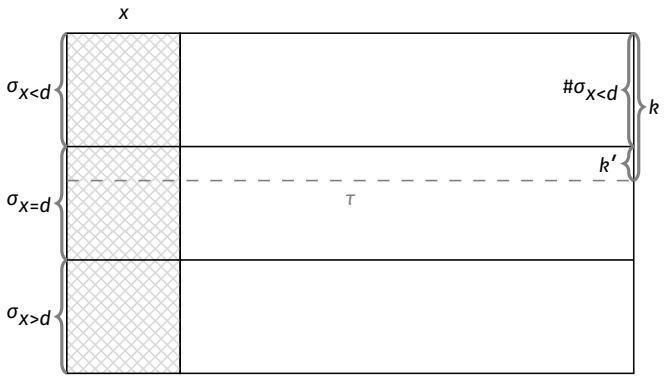


FIGURE 1. Visual representation of Lemma 2.1. The rows represent tuples in increasing order and the columns represent variables. The value on x of the k^{th} tuple in R, depicted by a dashed line, is the smallest value $d \in D$ such that R contains more than k tuples with a value smaller or equal to d. For a smaller d, we are below the dashed line because there are not enough tuple.

$\sigma(\mathbf{x_i}) \notin S_i^{\mathbf{D}}$ where $\sigma(\mathbf{x_i})$ is defined as $(\sigma(x_i^1), \ldots, \sigma(x_i^{a_i}))$ for $\mathbf{x_i} = (x_i^1, \ldots, x_i^{a_i})$. 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 signed 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.

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* [CTG+23] is the problem of returning, on input k, the k-th tuple $[\![Q]\!]^{\mathbf{D}}[k]$ for the order $\prec_{\mathsf{lex}}$ if $k \leqslant \#[\![Q]\!]^{\mathbf{D}}$ and failing otherwise. We are interested in answering direct access tasks using the same setting as [CTG+23]: we allow a *preprocessing* phase during which a data structure is constructed, followed by an *access* phase. Our goal is to obtain – with a preprocessing 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. 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.

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 words, 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 *neighbourhood of* v *in* H and by $N_H^*(v) = N_H(v) \setminus \{v\}$ the *open neighbourhood of* v. We will be interested in the following

[1]We follow the definition of [Lan23] 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.

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 edge of H and by adding a new edge that contains the open neighbourhood of v.

Given sets W, $S \subseteq W$ and $K \subseteq 2^W$, a *covering of S with K* is a subset $F \subseteq K$ such that $S \subseteq \bigcup_{e \in F} e$. The *cover number $\rho(S, K)$ of S with respect to K* is defined as the minimal size of a covering of S with K, that is, $\rho(S, K) = \min\{|F| \mid F \text{ is a covering of } S \text{ with } K\}$. A *fractional covering of S with K* is a function $c : K \to \mathbb{R}_+$ such that for every $s \in S$, $\sum_{e \in K, s \in e} c(e) \geqslant 1$. Observe that a covering is a fractional covering where c has values in $\{0, 1\}$. The *fractional cover number $\rho^*(S, K)$ of S wrt K* is defined as the minimal size of a fractional covering of S with K, that is, $\rho^*(S, K) = \min\{\sum_{e \in E} c(e) \mid c \text{ is a fractional covering of } S \text{ with } K\}$.

We will mostly use (fractional) covering for covering vertices of hypergraphs using a subset of its edges. Fractional covers are particularly interesting because of the following theorem by Grohe and Marx:

Theorem 2.2 [GM14]**.** *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 $|\mathbf{D}|^{\lambda}$.*

Signed hypergraphs. 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 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 $E_- = \emptyset$.

Model of computation. In this paper, we will always work in the word-RAM model of computation, with $\mathcal{O}(\mathsf{log}(n))$-bit words (where n is the size of the input, which, in this paper, is often the size of the database) and unit-cost operations. The main consequences of this choice are the following: we can perform arithmetic operations on integers of size at most n^k (hence encoded over $\mathcal{O}(k\mathsf{log}(n))$ bits) in time polynomial in k only. In particular, it is known that addition can be performed in time $\mathcal{O}(k)$ (using the usual addition algorithm on numbers represented in base $\mathsf{log}(n)$) and both multiplication and division can be performed in time $\mathcal{O}(k\mathsf{log}(k))$ using Harvey and van der Hoeven algorithm [HVDH21]. We will heavily use this fact in the complexity analysis of our algorithm since we will need to manipulate integers representing the sizes of relations on domain D and k variables, hence, of size at most $|D|^k$. We will hence assume that every arithmetic operation here can be performed in time $\mathcal{O}(k\mathsf{log}(k))$.

Moreover, we know that using radix sort, we can sort m values whose total size is bounded by n in time $O(m + n)$ [CLRS22, Section 6.3]. We will mostly use this fact to sort relations of a database $\mathbf{D}$ in time $O(|\mathbf{D}|)$.

3. An optimal yet inefficient algorithm

In this section, we reduce between the problem of answering direct access tasks on join queries with negated atoms and the problem of answering direct access tasks on join queries

without negated atoms. This will allow us to directly use results from [BCM22a] and get, for any given Q, an algorithm to answer direct access tasks with optimal data complexity. Both the algorithm and its optimality will directly follow from the algorithm and the lower bound of [BCM22a]. The algorithm however has exponential complexity in the size of the query. We present in the next sections a direct and optimal algorithm with better combined complexity.

3.1. **From join queries to positive join queries.** Here, we show that, in terms of data complexity, and if we ignore polylog factors, direct access for a signed query has the same complexity as direct access for the worst positive query obtained by considering some of the negated relations to be positive.

A signed join query Q is a join between two subqueries: the one consisting of the positive atoms and the one consisting of the negated atoms. With a slight abuse of notation, we denote this by $Q = P \wedge N$, where P is the join of the positive atoms and N the join of the negated atoms.

Let Q be a signed join query. For a set of relations $N \subseteq \mathsf{atoms}^-(Q)$, we denote by $\overline{N}$ the same relations but considered as positive atoms. Given a disjoint pair (N_1, N_2) of subsets of $\mathsf{atoms}^-(Q)$, we denote by Q_{N_1,N_2} the query computed by considering all the relations in N_1 as if they were positive atoms, keeping the negated atoms of N_2 and removing all other negated atoms. That is, $Q_{N_1,N_2} = P \wedge \overline{N_1} \wedge N_2$.

Example 3.1. For $Q = U(a,b) \wedge \neg R(x,y) \wedge \neg S(y,z) \wedge \neg T(z,x)$, $N_1 = \{\neg R(x,y)\}$ and $N_2 = \{\neg T(z,x)\}$, we have $Q_{N_1,N_2} = U(a,b) \wedge R(x,y) \wedge \neg T(z,x)$.

Our lower bound relies on the following strategy: we show that having direct access to Q is equivalent to having direct access to $Q_{N,\emptyset}$ for every $N \subseteq \mathsf{atoms}^-(Q)$, which is a positive query and for which we already know lower bounds. To do so, we actually prove that this is equivalent to having a direct access to Q_{N_1,N_2} for every disjoint $N_1, N_2 \subseteq \mathsf{atoms}^-(Q)$ by induction. The proof structure is summarized by Fig. 2.

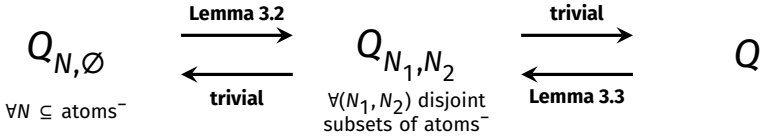


FIGURE 2. Relations between direct access for signed queries and direct access for positive queries

Lemma 3.2 (Subtraction Lemma)**.** *Let S be a set ordered by $\prec$ and two subsets S_1, S_2 of S such that $S_1 \subseteq S_2$. Assume that, for any value k, we can output the k^{th} element of S_1 (respectively of S_2) for the order $\prec$ in time t_1 (respectively in time t_2). Then, given any k, we can output the k^{th} element of $S_2 \setminus S_1$ for the order $\prec$ in time $C \cdot (t_2 + t_1 \cdot \log|S_1|) \cdot \log|S_2|$ for some constant C.*

Proof. We define *ranking* as the reverse problem to direct access. That is, given a tuple τ from an ordered set, finding its rank. If τ is not in the set, we return the rank of the largest smaller element or 0 if τ is smaller than the first element of S. We observe that having direct

access for a set of tuples S in time t implies having *ranking* on S in time $\mathcal{O}(\log|S| \cdot t)$. This holds since we can simply do a binary search on S to find the correct rank for a given tuple.

Next, we claim that, given an element τ of rank r_2 in S_2, we can find its rank in the subtraction of the sets $S_2 \setminus S_1$ in time $\mathcal{O}(\log|S_1| \cdot t_1)$. We use ranking for τ to find its rank r_1 in S_1. Then, we deduce that its rank in $S_2 \setminus S_1$ is $r_2 - r_1$. Now, we can simply do a binary search over the ranks of S_2. For each rank r_2, we access its element τ in time t_2, and check its ranking in $S_2 \setminus S_1$ as described above. This leads to direct access for $S_2 \setminus S_1$ in time $\mathcal{O}((t_2 + t_1 \cdot \log|S_1|) \cdot \log|S_2|)$. □

Lemma 3.3. *Let Q be a signed join query with n variables. Assume that for all $N \subseteq \mathsf{atoms}^-(Q)$ we have direct access for $Q_{N,\emptyset}$ with preprocessing time t_p and access time t_a. Then, we have direct access to Q_{N_1,N_2} for any disjoint pair N_1, N_2 of subsets of $\mathsf{atoms}^-(Q)$ with preprocessing time $2^{|\mathsf{atoms}^-(Q)|} \cdot t_p$ and access time $(An \cdot \log|D|)^{2|\mathsf{atoms}^-(Q)|} \cdot t_a$ for some constant A.*

Proof. The lemma follows by proving the following statement by induction over the size of N_2: for any subset $N_1 \subseteq \mathsf{atoms}^-(Q)$ such that N_1 and N_2 are disjoint, we have direct access for Q_{N_1,N_2} with preprocessing time $2^{|N_2|} \cdot t_p$ and access time $(2Cn \cdot \log|D|)^{2|N_2|} \cdot t_a$ where C is the constant from Lemma 3.2. The base case is when $N_2 = \emptyset$, and it is given by the hypothesis of the lemma statement.

Now let us consider N_2 to be a non-empty set. Since it is not empty, we can write it as $N_2 = N_2' \wedge \neg R(\mathbf{x})$, where $\neg R(\mathbf{x})$ is one arbitrarily chosen atom from N_2. From the induction hypothesis, since $|N_2'| < |N_2|$, for any subset $N_1' \subseteq \mathsf{atoms}^-(Q)$ such that $N_1' \cap N_2' = \emptyset$, we have direct access for the query $Q_{N_1',N_2'}$ with preprocessing time $2^{|N_2|-1} \cdot t_p$ and access time $(2Cn \cdot \log|D|)^{2|N_2|-1} \cdot t_a)$.

We can rewrite Q_{N_1,N_2} as:

$$\begin{aligned} Q_{N_1,N_2} &= P \wedge \overline{N_1} \wedge N_2' \wedge \neg R(\mathbf{x}) \\ &= (P \wedge \overline{N_1} \wedge N_2') \setminus (P \wedge \overline{N_1} \wedge N_2' \wedge R(\mathbf{x})) \\ &= Q_{N_1,N_2'} \setminus Q_{N_1 \cup \{R(\mathbf{x})\},N_2'} \end{aligned}$$

Notice that $(N_1 \cup \{R(\mathbf{x})\}, N_2')$ is a disjoint pair since $(N_1, N_2' \cup \{R(\mathbf{x})\})$ is a disjoint pair, and $R(\mathbf{x})$ does not appear in N_2 more than once since we consider self-join free queries. We know from the induction hypothesis that we have direct access to the answers of both $Q_{N_1,N_2'}$ and $Q_{N_1 \cup \{R(\mathbf{x})\},N_2'}$. Moreover, since $N_1 \subset N_1 \cup \{R(\mathbf{x})\}$ and adding positive atoms can only restrict the answer set of a query, we have that $[\![Q_{N_1 \cup \{R(\mathbf{x})\},N_2'}]\!]^{\mathbf{D}} \subseteq [\![Q_{N_1,N_2'}]\!]^{\mathbf{D}}$. We apply Lemma 3.2 to get direct access to the answers of Q_{N_1,N_2}. Preprocessing consists in doing the preprocessing for $Q_{N_1 \cup \{R(\mathbf{x})\},N_2'}$ and $Q_{N_1,N_2'}$. By induction, both of them takes $2^{|N_2|-1} \cdot t_p$ hence a preprocessing time of $2^{|N_2|} \cdot t_p$ time. For the access time, we apply Lemma 3.2 with $t_1 = t_2 = (2Cn \cdot \log|D|)^{2(|N_2|-1)} \cdot t_a$ and bound S_1, S_2 with $|D|^n$ and get:

$$2 \cdot (2Cn \cdot \log|D|)^{2(|N_2|-1)} \cdot t_a \cdot (Cn \cdot \log|D|)^2 \leqslant (2Cn \cdot \log|D|)^{2|N_2|} \cdot t_a$$

This concludes our induction step. □

We show the lower bound with the following statement:

Lemma 3.4. *Let Q be a signed join query with n variables and without self-joins. Assume we have direct access for Q with preprocessing time t_p and access time t_a. Then, for any*

disjoint pair (N_1, N_2) *of* $\mathsf{atoms}^-(Q)$*, we have direct access to* Q_{N_1,N_2} *with preprocessing time* $2^{|\mathsf{atoms}^-(Q)|} \cdot t_p$ *and access time* $(An \cdot \log|D|)^{2|\mathsf{atoms}^-(Q)|} \cdot t_a$ *for some constant* A*.*

Proof. We prove by induction over the size of N_1 that, for any disjoint pair (N_1, N_2), we have direct access for Q_{N_1,N_2} with preprocessing time $2^{|N_1|} \cdot t_p$ and access time $(2Cn \cdot \log|D|)^{2|N_1|} \cdot t_a$ where C is the constant from Lemma 3.2. The base case is when $N_1 = \emptyset$. If $N_2 = \mathsf{atoms}^-(Q)$, then since $Q_{\emptyset,\mathsf{atoms}^-(Q)} = Q$, it is given by the hypothesis of the theorem statement. If $N_2 \subset \mathsf{atoms}^-(Q)$, then we can use the algorithm of $Q_{\emptyset,\mathsf{atoms}^-(Q)}$ for answering Q by setting the negated relations that are not in N_2 to be empty. Notice that we can do so freely because the query has no self-joins.

Now, let us consider N_1 to be a non-empty set. We can then denote it as $N_1 = N_1' \wedge \neg R$. By induction, we have direct access for the answers of Q_{N_1',N_2} for every N_2 disjoint from N_1 with preprocessing time $2^{|N_1|-1} \cdot t_p$ and access time $(2Cn \cdot \log|D|)^{2(|N_1|-1)} \cdot t_a$.

We can rewrite Q_{N_1,N_2} as :

$$\begin{aligned} Q_{N_1,N_2} &= P \wedge \overline{N_1} \wedge N_2 \\ &= P \wedge \overline{N_1'} \wedge N_2 \wedge R \\ &= (P \wedge \overline{N_1'} \wedge N_2) \setminus (P \wedge \overline{N_1'} \wedge N_2 \wedge \neg R) \\ &= Q_{N_1',N_2} \setminus Q_{N_1',N_2 \cup \{\neg R\}} \end{aligned}$$

We can use the induction hypothesis on both Q_{N_1',N_2} and $Q_{N_1',N_2\cup\{\neg R\}}$. Since having more negated atoms implies that the answer set is more restricted, we have that $[\![Q_{N_1',N_2\cup\{\neg R\}}]\!]^{\mathbf{D}} \subseteq [\![Q_{N_1',N_2}]\!]^{\mathbf{D}}$. We apply Lemma 3.2 to get direct access to the answers of Q_{N_1,N_2}. Preprocessing takes $2 \cdot 2^{|N_1|-1} \cdot t_p = 2^{|N_1|} \cdot t_p$ time. For the access time, we apply Lemma 3.2 by bounded S_1 and S_2 by $|D|^n$ and by using $t_1 = t_2 = (2Cn \cdot \log|D|)^{2(|N_1|-1)}$ that we have by induction. It gives a total access time bounded by $(2Cn \cdot \log|D|)^{2(|N_1|-1)} \cdot (Cn\log|D|)^2$ which is bounded by $(2Cn \cdot \log|D|)^{2|N_1|}$. This concludes our induction step. □

In the following, we want to transfer the knowledge we have about positive queries to conclude the complexity of signed queries.

Theorem 3.5. *Let* Q *be a self-join free signed join query with* n *variables.*

- *If* Q *has direct access with preprocessing time* t_p *and access time* t_a*, then for all* $N \subseteq \mathsf{atoms}^-(Q)$ *we have direct access for* $Q_{N,\emptyset}$ *with preprocessing time* $2^{|\mathsf{atoms}^-(Q)|} \cdot t_p$ *and access time* $(An \cdot \log|D|)^{2|\mathsf{atoms}^-(Q)|} \cdot t_a$*.*
- *If for all* $N \subseteq \mathsf{atoms}^-(Q)$ *we have direct access for* $Q_{N,\emptyset}$ *with preprocessing time* t_p *and access time* t_a*, then* Q *has direct access with preprocessing time* $2^{|\mathsf{atoms}^-(Q)|} \cdot t_p$ *and access time* $(An \cdot \log|D|)^{2|\mathsf{atoms}^-(Q)|} \cdot t_a$*,*

where A *is a constant.*

Proof. The first part of the statement follows from applying Lemma 3.4 with $N_2 = \emptyset$. The second part follows from Lemma 3.3 applied with $N_1 = \emptyset$ and $N_2 = \mathsf{atoms}^-(Q)$ since $Q_{\emptyset,\mathsf{atoms}^-(Q)} = Q$.

□

3.2. **Optimally solving signed join queries.** Theorem 3.5 suggests that the (data) complexity of answering direct access tasks on a signed join query Q is as hard as the hardest positive query that can be obtained by keeping only a subset of negative atoms and turning them positive. The complexity of answering direct access tasks on positive join queries is well understood from [BCM22a]. In that paper, the following is shown:

Theorem 3.6 [BCM22a], Theorem 44**.** *There exists a polynomial p such that for every self-join free positive join query Q (considered constant) on variables set X and an order $\prec$ on X, there exists $\iota(Q, \prec) \in \mathbb{Q}_+$ such that:*

- *for every database $\mathbf{D}$, direct access tasks for $[\![Q]\!]^{\mathbf{D}}$ with order $\prec_{\mathsf{lex}}$ can be answered with preprocessing time $\mathcal{O}((p(|Q|) \cdot |\mathbf{D}|)^{\iota(Q,\prec)})$ and access time $\mathcal{O}(p(|Q|) \cdot \mathsf{log}(|\mathbf{D}|))$.*
- *if there exists $f\colon \mathbb{N} \to \mathbb{N}$ and $\varepsilon > 0$ such that for every database $\mathbf{D}$ and constant $\delta > 0$, direct access tasks for $[\![Q]\!]^{\mathbf{D}}$ with order $\prec_{\mathsf{lex}}$ can be solved with preprocessing time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{\iota(Q,\prec)-\varepsilon})$ and access time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{\delta})$, then the Zero-Clique Conjecture is false.*

The value $\iota(Q, \prec)$ is a function that only depends on Q and the chosen order and characterizes the optimal preprocessing time needed. We do not need the precise definition of this function yet as only its existence is necessary for the lower bound. More details on this parameter will be presented in Section 3.3. The Zero-Clique Conjecture is a widely believed conjecture in the domain of fine-grained complexity saying that for every k and $\varepsilon > 0$, there is no randomized algorithm with complexity $\mathcal{O}(n^{k-\varepsilon})$ able to decide whether there exists a k-clique whose edge weights sum to 0 in a edge-weighted graph. In other words, unless such algorithm exists, the best preprocessing one can hope for to solve direct access tasks of positive join queries is $\mathcal{O}(|\mathbf{D}|^{\iota(Q,\prec)})$. In [BCM22a], complexities in **??** 3.6 are given in terms of data complexity but it is not hard to see that the dependency on $|Q|$ is polynomial for constant $\iota(Q, \prec)$.

We now extend the definition of ι to signed join queries as follows, inspired by Theorem 3.5: let $Q = P \wedge N$ be a signed join query, where P are the positive atoms of Q and N the negative atoms of Q. Let X be the variables set of Q and $\prec$ an order on X. We define $\iota^s(Q, \prec) = \max_{N \subseteq \mathsf{atoms}^-(Q)} \iota(Q_{N,\emptyset}, \prec)$, where $\iota(\cdot, \cdot)$ is the incompatibility number defined in [BCM22a]. Observe that by definition, if Q is a positive join query, then $\iota(Q, \prec) = \iota^s(Q, \prec)$. Moreover, Theorem 3.5 and **??** 3.6 directly imply this generalisation of **??** 3.6 to signed join queries:

Theorem 3.7. *There exists a function $g\colon \mathbb{N} \to \mathbb{N}$ such that, given a self-join free signed join query Q on variables set X and an order $\prec$ on X, we have:*

- *for every database $\mathbf{D}$, direct access tasks for $[\![Q]\!]^{\mathbf{D}}$ and order $\prec_{\mathsf{lex}}$ can be answered with preprocessing time $g(|Q|) \cdot |\mathbf{D}|^{\iota^s(Q,\prec)}$ and access time $g(|Q|) \cdot \mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}(|\mathbf{D}|)$.*
- *if there exists $f\colon \mathbb{N} \to \mathbb{N}$ and $\varepsilon > 0$ such that for every database $\mathbf{D}$ and constant $\delta > 0$, direct access tasks for $[\![Q]\!]^{\mathbf{D}}$ with order $\prec_{\mathsf{lex}}$ can be solved with preprocessing time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{\iota^s(Q,\prec)-\varepsilon})$ and access time $\mathcal{O}(f(|Q|) \cdot |\mathbf{D}|^{\delta})$, then the Zero-Clique Conjecture is false.*

Proof. Let Q be a signed join query, m be the number of negative atoms of Q and let $k = \iota^s(Q, \prec)$. Moreover, let p be the polynomial from **??** 3.6. By definition, we have that for every $N \subseteq \mathsf{atoms}^-(Q)$, $\iota(Q_{N,\emptyset}, \prec) \leqslant k$. Hence by **??** 3.6, we can solve the direct access task for $[\![Q_{N,\emptyset}]\!]^{\mathbf{D}}$ with order $\prec_{\mathsf{lex}}$ with preprocessing time $\mathcal{O}((p(|Q|) \cdot |\mathbf{D}|)^k)$ and access time $\mathcal{O}(p(|Q|) \cdot \mathsf{log}(|\mathbf{D}|))$. By Lemma 3.3, we then have direct access for Q on

$\mathbf{D}$ with preprocessing time $\mathcal{O}(2^{|\mathsf{atoms}^-(Q)|}(p(|Q|)|\mathbf{D}|)^k)$ and access time $\mathcal{O}(p(|Q|)\cdot|X|\cdot(A|X|)^{2|\mathsf{atoms}^-(Q)|}\cdot\mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}(|\mathbf{D}|))$. Observe that A is a constant and $|X|\leqslant|Q|$, hence we can define $g(m)=K(p(m)\cdot m\cdot A|X|^{2m})$ for some large enough constant K and we have the desired bound: the preprocessing time can be done in $g(|Q|)|\mathbf{D}|^k$ and access time $g(|Q|)\cdot\mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}(|\mathbf{D}|)$

Now assume that there exists $f\colon\mathbb{N}\to\mathbb{N}$ and $\varepsilon>0$ such that for every $\delta'>0$, we have a direct access scheme for Q with preprocessing time $\mathcal{O}(f(|Q|)\cdot|\mathbf{D}|^{k-\varepsilon})$ and access time $\mathcal{O}(f(|Q|)\cdot|\mathbf{D}|^{\delta'})$. Let $N\subseteq\mathsf{atoms}^-(Q)$ be such that $\iota(Q_{N,\emptyset},\prec)=k$. Let $\delta>0$ and set $\delta'=\delta/2$. By Lemma 3.4, we have direct access to $[\![Q_{N,\emptyset}]\!]^{\mathbf{D}}$ with preprocessing time $\mathcal{O}(2^{|\mathsf{atoms}^-(Q)|}f(|Q|)\cdot|\mathbf{D}|^{k-\varepsilon})=\mathcal{O}(h(|Q_{N,\emptyset}|)\cdot|\mathbf{D}|^{k-\varepsilon})$ and access time $\mathcal{O}(f(|Q|)\cdot\mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}(|\mathbf{D}|)|\mathbf{D}|^{\frac{\delta}{2}})=\mathcal{O}(h(|Q_{N,\emptyset}|)\cdot|\mathbf{D}|^{\delta})$ for some $h\colon\mathbb{N}\to\mathbb{N}$. But then by **??** 3.6, we have an algorithm for $Q_{N,\emptyset}$ whose complexity implies that the Zero-Clique Conjecture is false. □

Observe that the complexity obtained to preprocess a signed query has an exponential dependency on the size of the query since it is obtained using Lemma 3.3 which intuitively does a direct access preprocessing for every subquery of Q. This exponential dependency in $|Q|$ is not present for positive queries in [BCM22a]. Sections 4 and 5 are dedicated to design a better algorithm, achieving the same data complexity as in Theorem 3.7 but with a better dependency on $|Q|$ and a more direct algorithm.

Let us conclude this section with a final observation regarding Theorem 3.7. It is proven in [BCM22a] that **??** 3.6 also holds when Q contains self-joins. This is not true for signed join queries which is why we explicitly assumed Q to be self-join free. Indeed, for any join query Q containing an atom $R(\vec{x})$ and order $\prec$, the signed query $Q'=Q\wedge\neg R(\vec{x})$ is such that $\iota^s(Q,\prec)=\iota^s(Q',\prec)$ but is not self-join free. That said, we clearly have $[\![Q']\!]^{\mathbf{D}}=\emptyset$ for every database $\mathbf{D}$, hence direct access for Q' can easily be answered with preprocessing and access time independent on $|\mathbf{D}|$, which shows that the lower bound from Theorem 3.7 does not hold for signed join queries with self-joins (the upper bound holds since we can always consider each occurrence of a relation as a unique relation). It is an interesting research direction to understand the complexity of signed conjunctive queries with self-join that we leave open for future work.

3.3. **Hypergraph decompositions and incompatibility numbers.** In this section, we give the definition of the incompatibility number $\iota(\cdot,\cdot)$ from [BCM22a] and connect ι and ι^s with existing and new hypergraph decomposition techniques.

Elimination orders. Given a hypergraph $H=(V,E)$ and an order $\prec$ such that $V=\{v_1,\dots,v_n\}$ with $v_1\prec\cdots\prec v_n$, we define a series of hypergraphs as $H_1^{\prec},\dots,H_{n+1}^{\prec}$ defined as follows: $H_1^{\prec}=H$ and $H_{i+1}^{\prec}=H_i^{\prec}/v_i$. The *hyperorder width of the elimination order* $\prec$ *for* H*, denoted by* $\mathsf{how}(H,\prec)$ is defined as $\max_{i\leqslant n}\rho(N_{H_i^{\prec}}(v_i),E)$. The *hyperorder width* $\mathsf{how}(H)$ *of* H is defined as the best possible width using any elimination order, that is, $\mathsf{how}(H)=\min_{\prec}\mathsf{how}(H,\prec)$. We similarly define the *fractional hyperorder width of* $\prec$ *for* H*, denoted by* $\mathsf{fhow}(H,\prec)$ as $\max_{i\leqslant n}\rho^*(N_{H_i^{\prec}}(v_i),E)$ and the *fractional hyperorder width* $\mathsf{fhow}(H)$ *of* H as $\mathsf{fhow}(H)=\min_{\prec}\mathsf{fhow}(H,\prec)$.

The definition of fractional hyperorder width of an elimination order $\prec$ matches the definition of the $\iota(Q,\succ)$ from [BCM22a] and used in **??** 3.6 and Theorem 3.7. However, the

incompatibility number is stated for the reversed of the elimination order. In this paper, we choose to make the connection with the existing literature on elimination order for hypergraphs more explicit and hence use the reversed order. We hence have:

Theorem 3.8. *For every positive join query Q on variables set X and order $\prec$ on X, it holds that $\iota(Q, \succ) = \mathsf{fhow}(H(Q), \prec)$.*

It has already been observed many times ([AKNR15, Appendix C] or [FHLS18, GSSS22, AKNR16] but also in Proposition 14 of [BCM22a], or via the notion of static width in [KNOZ25]) that $\mathsf{how}(H)$ and $\mathsf{fhow}(H)$ are respectively equal to the generalised hypertree width and the fractional hypertree width of H and that there is a natural correspondence between a tree decomposition and an elimination order having the same width. However, to be able to express our tractability results as a function of the order, it is more practical to define the width of orders instead of hypertree decompositions[2].

The case $\iota(Q, \succ) = 1$ hence corresponds to $\mathsf{fhow}(H(Q), \prec) = 1$, that is, $H(Q)$ is α-acyclic and $\prec$ is an α-elimination order witnessing this acyclicity, as defined in [BB16]. (Reversed) orders witnessing this has been also previously called orders without disruptive trio [CTG+23] which exactly corresponds to the classical notion of elimination ordering witnessing α-acyclicity (see [BB16] for a survey, where they are called α-elimination order). In this paper, we will now mostly use a terminology based on hypergraph decomposition of the query rather than incompatibility number since we also aim at comparing our results with other results on negative join queries stated in terms of hypergraphs.

Signed hyperorder width. In the case of signed join queries, we can naturally generalise the notion of hyperorder width so that it matches our extended definition of incompatibility number of signed join query $\iota^s(Q, \succ)$.

Let $H = (V, E_+, E_-)$ be a signed hypergraph. We say that H' is *a negative subhypergraph of H* and write $H' \subseteq^- H$ if it is a hypergraph of the form $H' = (V, E_+ \cup E')$ for some $E' \subseteq E_-$. Given an order $\prec$ on V, we define:

- the *signed hyperorder width* $\mathsf{show}(H, \prec)$ *of* $\prec$ *for* H as
$$\mathsf{show}(H, \prec) = \max_{H' \subseteq^- H} \mathsf{how}(H, \prec)$$
and the corresponding *signed hyperorder width* $\mathsf{show}(H)$ *of* H, defined as $\mathsf{show}(H) = \min_{\prec} \mathsf{show}(H, \prec)$,
- the *fractional signed hyperorder width* $\mathsf{sfhow}(H, \prec)$ *of* $\prec$ *for* H as
$$\mathsf{sfhow}(H, \prec) = \max_{H' \subseteq^- H} \mathsf{fhow}(H, \prec)$$
and the corresponding *fractional signed hyperorder width* $\mathsf{sfhow}(H)$ *of* H defined as $\mathsf{sfhow}(H) = \min_{\prec} \mathsf{sfhow}(H, \prec)$.

By definition and by Theorem 3.8, we immediately have:

Theorem 3.9. *For every signed join query Q on variables set X and $\prec$ an order on X, we have $\iota^s(Q, \succ) = \mathsf{sfhow}(H(Q), \prec)$.*

[2]Strictly speaking, the definition of $\mathsf{how}(\cdot)$ and $\mathsf{fhow}(\cdot)$ in [AKNR16] differ slightly in that the elimination step of v removes every edge containing v and replace it by the neighborhood of v, where in our definition, we keep them. This does not change the notion of neighborhood at each step so it results in the same widths but with a slightly different definition.

From now on, we will be working with hypergraph measures instead of incompatibility number when dealing with join queries. For future comparison, let us restate Theorem 3.7 in terms of hypergraph measure:

Theorem 3.10. *There exists a polynomial p such that given a self-join free signed join query Q on variables set X and an order $\prec$ on X, we have:*

- *for every database $\mathbf{D}$, direct access tasks for $[\![Q]\!]^{\mathbf{D}}$ and order $\prec_{\mathsf{lex}}$ can be answered with preprocessing time $\mathcal{O}(2^{|\mathsf{atoms}^-(Q)|}(p(|Q|)\cdot|\mathbf{D}|)^{\mathsf{sfhow}(H(Q),\succ)})$ and access time $\mathcal{O}(p(|Q|)\cdot|X|\cdot\mathsf{log}^{2|\mathsf{atoms}^-(Q)|+1}(|\mathbf{D}|))$.*
- *if there exists $f\colon\mathbb{N}\to\mathbb{N}$ and $\varepsilon>0$ such that for every database $\mathbf{D}$ and constant $\delta>0$, direct access tasks for $[\![Q]\!]^{\mathbf{D}}$ with order $\prec_{\mathsf{lex}}$ can be solved with preprocessing time $\mathcal{O}(f(|Q|)\cdot|\mathbf{D}|^{\mathsf{sfhow}(H(Q),\succ)-\varepsilon})$ and access time $\mathcal{O}(f(|Q|)\cdot|\mathbf{D}|^{\delta})$, then the Zero-Clique Conjecture is false.*

Observe again that the dependency on $|Q|$ in the preprocessing phase given by Theorem 3.10 is exponential. In the next two sections, we propose another, more direct, approach for solving direct access on signed conjunctive queries which also reduces the combined complexity to a polynomial in both $|Q|$ and $|\mathbf{D}|$ when parametrizing by the non-fractional version of hyperorder width of Q (that is $\mathsf{show}(H(Q),\prec)$). Moreover, the algorithm presented in the next two sections has a better access time than the one we obtained in Theorem 3.10.

4. Direct Access and 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 [OZ15], 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 [Cap17, Theorem 9].

4.1. Relational circuits.

A $\{\bowtie,\mathsf{dec}\}$-circuit C on variables set $X=\{x_1,\dots,x_n\}$ and domain D is a multi-directed[3] acyclic graph (DAG), where we refer to the vertices as gates. For a given gate g, every g' with a directed edge $g'\to g$ will be called *an input of g*. We denote by $\mathsf{input}(g)$ the set of inputs of g. The circuit has one distinguished gate $\mathsf{out}(C)$ called the *output* of C. Moreover, the circuit is labelled as follows:

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

The set of all the decision gates in a circuit C is denoted by $\mathsf{decision}(C)$. Given a gate v of C, we denote by C_v the *subcircuit of C rooted in v* to be the circuit whose output is v and which contains every gate g such that there is a directed path from g to v. We define the *variable set of v*, denoted by $\mathsf{var}(v)\subseteq X$, to be the set of variables x labelling a decision

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

gate in C_v. The variable labeling a decision gate v is denoted by $\mathsf{decvar}(v)$. The *size* $|C|$ of a $\{\bowtie, \mathsf{dec}\}$-circuit is defined to be the number of edges of its underlying DAG.

The intended semantics of a $\{\bowtie, \mathsf{dec}\}$-circuit is that each gate of the circuit computes a relation $\mathsf{rel}(v)$ on variables set $\mathsf{var}(v)$. Intuitively, join nodes computes the joins of the relations computed by its input and decision nodes computes the union of the relation computed by its input with an additional variable x whose value depends on the label of the edge between the decision node and its input.

More formally, we define the *relation* $\mathsf{rel}(v) \subseteq D^{\mathsf{var}(v)}$ *computed at gate* v inductively as follows: if v is an input labelled by $\bot$, then $\mathsf{rel}(v) = \emptyset$. If v is an input labelled by $\top$, then $\mathsf{rel}(v) = D^{\emptyset}$, that is, $\mathsf{rel}(v)$ is the relation containing only the empty tuple. Otherwise, let $v_1, \ldots, v_k$ be the inputs of v. If v is a $\bowtie$-gate, then $\mathsf{rel}(v)$ is defined to be $\mathsf{rel}(v_1) \bowtie \ldots \bowtie \mathsf{rel}(v_k)$. If v is a decision gate labelled by a variable x, $\mathsf{rel}(v) = R_1 \cup \cdots \cup R_k$ where $R_i = [x \leftarrow d_i] \bowtie \mathsf{rel}(v_i) \bowtie D^{\mathsf{var}(v) \setminus (\mathsf{var}(v_i) \cup \{x\})}$ and d_i is the label of the incoming edge (v_i, v). It is readily verified that $\mathsf{rel}(v)$ is a relation on domain D and variables set $\mathsf{var}(v)$. The *relation computed by* C *over a set of variables* X (assuming $\mathsf{var}(C) \subseteq X$), denoted by $\mathsf{rel}_X(C)$, is defined to be $\mathsf{rel}(\mathsf{out}(C)) \times D^{X \setminus \mathsf{var}(\mathsf{out}(C))}$.

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

Deciding whether the relation computed by a $\{\bowtie, \mathsf{dec}\}$-circuit is non-empty is NP-hard by a straightforward reduction from model checking of conjunctive queries [CM77]. Such circuits are hence of little use to get tractability results. We are therefore more interested in the following restriction of $\{\bowtie, \mathsf{dec}\}$-circuits: a $\{\times, \mathsf{dec}\}$-circuit C is a $\{\bowtie, \mathsf{dec}\}$-circuit such that: (i) for every $\bowtie$-gate v of C with inputs $v_1, \ldots, v_k$ and $i < j \leqslant k$, it holds that $\mathsf{var}(v_i) \cap \mathsf{var}(v_j) = \emptyset$, (ii) for every decision gate v of C labelled by x with inputs $v_1, \ldots, v_k$ and $i \leqslant k$, it holds that $x \notin \mathsf{var}(v_i)$. This restriction is akin to the one found in the literature of knowledge compilation and $\{\times, \mathsf{dec}\}$-circuits can actually be seen as a generalization of decision-DNNF to non-Boolean domains [DM02]. Cartesian product corresponds to decomposable $\wedge$-gates. Checking whether the relation computed by a $\{\times, \mathsf{dec}\}$-circuit C is non-empty can be done in time $\mathcal{O}(|C|)$ by a dynamic programming algorithm propagating in a bottom-up fashion whether $\mathsf{rel}(v)$ is empty. Similarly, given a $\{\times, \mathsf{dec}\}$-circuit C, one can compute the size of $\mathsf{rel}(C)$ in polynomial time in $|C|$ by a dynamic programming algorithm propagating in a bottom-up fashion $|\mathsf{rel}(v)|$. Proof of this fact can straightforwardly be adapted from existing literature on restricted Boolean circuits [DM02].

Example 4.1. Fig. 3 shows an example of a $\{\times, \mathsf{dec}\}$-circuit on variables $\{x_1, \ldots, x_5\}$ and domain $\{0, 1, 2\}$. Observe that the Cartesian product has variables $\{x_3, x_5\}$ on its left side and variables $\{x_2, x_4\}$ on its right side, which are indeed disjoint variable sets. The relation computed by the circuits includes for example the tuples $\langle x_1 \leftarrow 2, x_2 \leftarrow 2, x_3 \leftarrow 1, x_4 \leftarrow 0, x_5 \leftarrow 0\rangle$, $\langle x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 0, x_4 \leftarrow 0, x_5 \leftarrow 0\rangle$, but no tuple setting both x_1 to 0 and x_2 to 2. Indeed, every tuple with $x_1 \leftarrow 0$ in the relation computed by C must be extended by a tuple computed by the leftmost decision gate on x_2 but this decision gate has no edge labeled with 2.

If v is the only Cartesian product in Fig. 3, one can observe that C_v is the circuit highlighted with thick red border and $\mathsf{rel}(v)$ is the relation over variables $\{x_2, x_3, x_4, x_5\}$ and domain $\{0, 1, 2\}$ containing tuples $\langle x_2 \leftarrow 2, x_3 \leftarrow 1, x_4 \leftarrow 0, x_5 \leftarrow 0\rangle, \langle x_2 \leftarrow 2, x_3 \leftarrow 1, x_4 \leftarrow 0, x_5 \leftarrow 2\rangle, \langle x_2 \leftarrow 2, x_3 \leftarrow 1, x_4 \leftarrow 1, x_5 \leftarrow 0\rangle, \langle x_2 \leftarrow 2, x_3 \leftarrow 1, x_4 \leftarrow 1, x_5 \leftarrow 2\rangle$.

Ordered Relational Circuits. Let X be a set of variables and $\prec$ an order on X. We say that a $\{\times, \mathsf{dec}\}$-circuit C on domain D and variables set X is a $\prec$-*ordered* $\{\times, \mathsf{dec}\}$-*circuit* if for every decision gate v of C labelled with $x \in X$, it holds that for every $y \in \mathsf{var}(v) \setminus \{x\}$, $x \prec y$. We simply say that a circuit C is an ordered $\{\times, \mathsf{dec}\}$-circuit if there exists some order $\prec$ on X such that C is a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit. Observe that the example from Fig. 3 is ordered for the order $(x_1, x_2, x_3, x_4, x_5)$. Observe that, according to the definition, nothing prevents variables under a Cartesian product to be interleaved as depicted on the example: x_2 and x_4 are on one side while x_3 and x_5 are on the other.

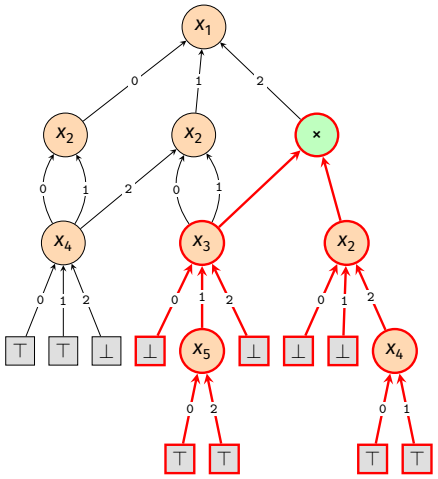


FIGURE 3. Example of a $\{\times, \mathsf{dec}\}$-circuit. The domain used is $\{0, 1, 2\}$ and the variable set is $\{x_1, x_2, x_3, x_4\}$.

4.2. **Direct access for ordered relational circuits.** Given a relational circuit C, we can define a notion of direct access tasks for circuits as follows: given a relational circuit C on variables X and an order $\prec$ on X, an algorithm solves direct access for C for order $\prec_{\mathsf{lex}}$ with precomputation time t_p and access time t_a if there exists an algorithm that runs in time t_p which constructs a data structure C' and then, using C', there is an algorithm that runs in time t_a such that, on input k, returns the k^{th} tuple of $\mathsf{rel}(C)$ for the order $\prec_{\mathsf{lex}}$.

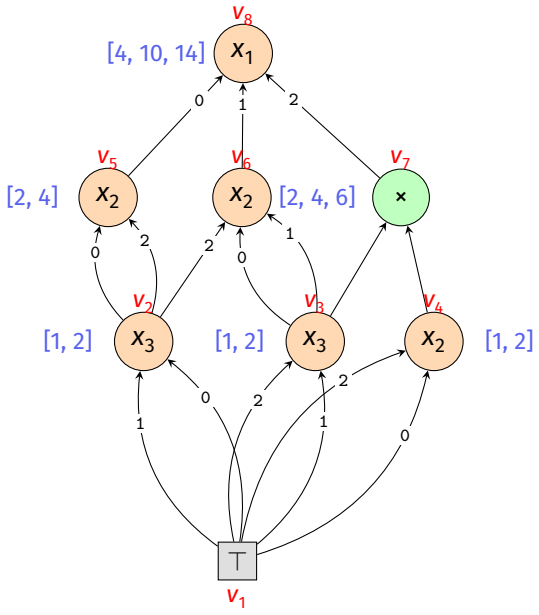
The main result of this section is an algorithm that allows for direct access for an ordered $\{\times, \mathsf{dec}\}$-circuit on domain D and variables set X. More precisely, we prove the following:

Theorem 4.2. *Let $\prec$ be an order on X and C be a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit on domain D and variables set X, then we can solve direct access tasks on $\mathsf{rel}(C)$ for order $\prec_{\mathsf{lex}}$ with access time $\mathcal{O}(|X|^3 \mathsf{log}(|X|) + |X|^2 \mathsf{log}|D|)$ and precomputation time $\mathcal{O}(|X| \mathsf{log}(|X|) \cdot |C|)$.*

4.2.1. *Precomputation.* In this section, we assume that C is a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit with respect to X. Moreover, for every decision gate v, we assume that its ingoing edges are stored in a list sorted by increasing value of their label. More precisely, we assume that the ingoing edges of v are a list $[e_1, \ldots, e_k]$ such that $d_1 < \cdots < d_k$ where d_i is the label of e_i.

Since we are working in the unit-cost model, we can always preprocess C in linear time to sort the ingoing edges of v.

The $\mathsf{nrel}_C(\cdot,\cdot)$ values. We are interested in the following values: for every decision gate v of C labelled with variable x and ingoing edges $[e_1,\ldots,e_k]$ with respective labels $[d_1,\ldots,d_k]$, we define for every $i \leqslant k$, $\mathsf{nrel}_C(v,d_i)$ as $\#\sigma_{x\leqslant d_i}(\mathsf{rel}(v))$. That is, $\mathsf{nrel}_C(v,d_i)$ is the number of tuples from $\mathsf{rel}(v)$ that assign a value on x smaller or equal than d_i. The precomputation step aims to compute nrel_C so that we can access $\mathsf{nrel}_C(v,d_i)$ quickly for every v and d_i.



x_1	x_2	x_3
0	0	0
0	0	1
0	2	0
0	2	1
1	0	1
1	0	2
1	1	1
1	1	2
1	2	0
1	2	1
2	0	1
2	2	2
2	0	1
2	2	2

FIGURE 4. Example of a $\{\times, \mathsf{dec}\}$-circuit annotated with nrel_C values. 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. The relation computed by the circuit is given on the right. The full annotation of the circuit needed for direct access is a bit more complicated and is described later in this section.

Example 4.3. We give an example of an annotated circuit on Fig. 4 where each decision-gate is labeled with a list. For a decision-gate v in the circuit with incoming edge $e_1,\ldots,e_k$ labeled with $d_1,\ldots,d_k$ with $d_1 < \cdots < d_k$, the i^{th} entry of the list is $\mathsf{nrel}_C(v,d_i)$. In this example, gate v_2 is labelled by $[1,2]$. This is because the relation computed by v_2 is on variable $\{x_3\}$ and contains two tuples: $\langle x_3 \leftarrow 0\rangle$ and $\langle x_3 \leftarrow 1\rangle$. Hence, there is one tuple in this relation which sets x_3 to a value smaller than 0, two tuples which set x_3 to a value smaller than 1. Since there is no incoming edge labeled by 2, we do not need to compute $\mathsf{nrel}_C(v_2,2)$. That is, $\mathsf{nrel}_C(v_2,0) = 1$ and $\mathsf{nrel}_C(v_2,1) = 2$. Observe that gate v_3 is also labeled by $[1,2]$ but this corresponds to $\mathsf{nrel}_C(v_3,1) = 1$ and $\mathsf{nrel}_C(v_3,2) = 2$ since there is no incoming edge labeled by 0.

The gate v_6 is labeled by $[2, 4, 6]$. Indeed, it can be seen that there are two tuples with $x_2 = 0$ in $\mathsf{rel}(v_6)$. Both tuples come from following the 0-labeled incoming edge of v_6 to v_3. The size of the relation computed by v_3 is 2, which can be read as value $\mathsf{nrel}_C(v_3, 2)$, that is, $\mathsf{nrel}_C(v_6, 0) = 2$. Similarly, by following the 1-labeled incoming edge to v_3, we know that there are 2 tuples with $x_2 = 1$ in $\mathsf{rel}(v_6)$, hence 4 tuples with $x_2 \leqslant 1$, that is, $\mathsf{nrel}_C(v_6, 1) = 2 + \mathsf{nrel}_C(v_6, 0) = 4$. We finally get 2 tuples with $x_2 = 2$ from v_2, giving $\mathsf{nrel}_C(v_6, 2) = 2 + \mathsf{nrel}_C(v_6, 1) = 6$.

Dynamic programming for computing $\mathsf{nrel}_C(\cdot)$. As hinted in the previous example, our algorithm performs a bottom-up computation to compute $\mathsf{nrel}_C(\cdot, \cdot)$ values. However, some other values need to be computed along them to ease later computation. The first one we need is the number of tuples in $\mathsf{rel}(v)$ for every gate v of C. We get this value inductively as follows. If v is an input gate labelled by $\bot$ then we obviously have $|\mathsf{rel}(v)| = 0$ and if it is an input gate labelled by $\top$, we have $|\mathsf{rel}(v)| = 1$. Now, if v is a decision-gate on variable x with (sorted) ingoing edges $e_1 = (v_1, v), \ldots, e_k = (v_k, v)$ labelled respectively by $d_1 = \ell(e_1) < \cdots < d_k = \ell(e_k)$, then $|\mathsf{rel}(v)|$ can inductively be computed as follows:

$$|\mathsf{rel}(v)| = \sum_{i=1}^{k} |\mathsf{rel}(v_i)| \times |D|^{|\Delta(v,v_i)|} \text{ where } \Delta(v, w) = \mathsf{var}(v) \setminus (\{x\} \cup \mathsf{var}(w)). \tag{4.1}$$

Similarly, $\mathsf{nrel}_C(v, d_i)$ can be computed by restricting the previous relation on the inputs of v:

$$\mathsf{nrel}_C(v, d_i) = \sum_{j=1}^{i} |\mathsf{rel}(v_j)| \times |D|^{|\Delta(v,v_j)|} \tag{4.2}$$

In particular, observe that $\mathsf{nrel}_C(v, d_1) = |\mathsf{rel}(v_1)| \times |D|^{|\Delta(v,v_1)|}$ and $\mathsf{nrel}_C(v, d_{i+1}) = \mathsf{nrel}_C(v, d_i) + |\mathsf{rel}(v_{i+1})| \times |D|^{|\Delta(v,v_{i+1})|}$.

Finally, if v is a Cartesian product, we clearly have $|\mathsf{rel}(v)| = \prod_{w \in \mathsf{input}(v)} |\mathsf{rel}(w)|$. Hence, one can compute nrel_C using a dynamic programming algorithm that inductively computes $\mathsf{nrel}_C(v, d)$ for each decision-gate v of the circuit with an ingoing edge labelled by d. Observe however that in order to efficiently compute these values, we also need to have easy access to $|\mathsf{rel}(v)|$ and $\mathsf{var}(v)$ for every gate v. Fortunately, as hinted above, these values are also easy to compute inductively.

More precisely, the dynamic programing algorithm works as follows: we start by performing a topological ordering $(v_1, \ldots, v_N)$ 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. This can be computed in time $\mathcal{O}(|C|)$.

We dynamically compute, for every gate v, the values $|\mathsf{rel}(v)|$, the sets $\mathsf{var}(v)$, and if v is a decision gate with an ingoing edge labelled by $d \in D$, we also compute $\mathsf{nrel}_C(v, d)$.

We proceed as follows: we allocate a table T_{rel} of size N such that for $i \leqslant N$, $T_{\mathsf{rel}}[i]$ is initialized to 0. At the end of the algorithm, $T_{\mathsf{rel}}[i]$ will contain $|\mathsf{rel}v_i|$. We also allocate a table T_{var} such that for $i \leqslant N$, $T_{\mathsf{var}}[i]$ is initialized with a $|X|$-bitvector containing on 0. At the end of the algorithm, $T_{\mathsf{var}}[v_i][k]$ is 1 if and only if $x_k \in \mathsf{var}v_i$. This initialization step takes $\mathcal{O}(N \cdot |X|) = \mathcal{O}(|C| \cdot |X|)$.

We now initialize T_{nrel_C} of size N where for $i \leqslant N$, the entry $T_{\mathsf{nrel}_C}[i]$ is intialized with an array of size $|\mathsf{input}(v_i)|$ containing only -1 values. Clearly, T_{nrel_C} has size $\mathcal{O}(|C|)$ since

it contains one entry per edge of C. At the end of the algorithm, for $i \leqslant N$, if v_i is a decision-gate, then $T_{\mathsf{nrel}_C}[i][p]$ contains $\mathsf{nrel}_C(v_i, d_p)$ where d_p is the label of the p^{th} input of v_i. If v_i is a Cartesian product or an input, $T_{\mathsf{nrel}_C}[i]$ is not relevant and is not modified after the initialization.

We then populate each entry of these tables following the previously constructed topological ordering and using the relations written above (see (4.1) and (4.2)) 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 $(w_1, v), \ldots, (w_k, v)$ be the incoming edges of v ordered by increasing labels $d_1 < \cdots < d_k$. We set $T_{\mathsf{nrel}_C}[v, d_1]$ to $\mathsf{nrel}_C(v, d_1)$ which is equal to $|\mathsf{rel}(w_1)| \cdot |D|^{|\Delta(v,w_1)|}$. We then 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 (4.2).

It is clear from the relations (4.2) 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. Pseudocode for this algorithm can be found in Algorithm 1.

Algorithm 1 An algorithm to annotate a circuit C on variables X and domain D

procedure ANNOTATE(C, X, D)
  $v_1, \ldots, v_N \leftarrow$ a topological ordering of the gates of C
  $T_{\mathsf{rel}} \leftarrow$ table of size N, initialized to 0
  $T_{\mathsf{var}} \leftarrow$ table of size N, initialized with 0-bitvectors of size $|X|$
  $T_{\mathsf{nrel}_C} \leftarrow$ table of size N;
  for $i = 1$ to N **do**
    if v_i is $\top$-input **then** $T_{\mathsf{rel}}[i] \leftarrow 1$; **continue**
    if v_i is $\bot$-input **then** $T_{\mathsf{rel}}[i] \leftarrow 0$; **continue**
    if v_i is a $\times$-gate **then**
      $T_{\mathsf{rel}}[i] \leftarrow \prod_{v_j \in \mathsf{input}(v_i)} T_{\mathsf{rel}}[j]$
      $T_{\mathsf{var}}[i] \leftarrow \bigcup_{v_j \in \mathsf{input}(v_i)} T_{\mathsf{var}}[j]$
    else if v_i is a decision gate on x_k **then**
      $T_{\mathsf{var}}[i] \leftarrow \{x_k\} \cup \bigcup_{v_j \in \mathsf{input}(v_i)} T_{\mathsf{var}}[j]$
      $T_{\mathsf{rel}}[i] \leftarrow \sum_{v_j \in \mathsf{input}(v_i)} T_{\mathsf{rel}}[j] \times |D|^{|T_{\mathsf{var}}[j] \setminus T_{\mathsf{var}}[i]| - 1}$
      Order incoming edges $(v_{a_1}, v_i), \ldots, (v_{a_k}, v_i)$ of v_i by increasing label
      $T_{\mathsf{nrel}_C}[i] \leftarrow$ new array of size k initialized to 0
      $T_{\mathsf{nrel}_C}[i][1] \leftarrow T_{\mathsf{rel}}[a_1]$
      for $j = 2$ to k **do**
        $T_{\mathsf{nrel}_C}[i][k] \leftarrow T_{\mathsf{nrel}_C}[i][j-1] + T_{\mathsf{rel}}[a_j]$
      end for
    end if
  end for
  return $T_{\mathsf{rel}}, T_{\mathsf{var}}, T_{\mathsf{nrel}_C}$
end procedure

Example 4.4. From the circuit in Fig. 4, assume we have populated T_{rel}, T_{var} and T_{nrel_C} up to $i = 6$. We now need to compute the seventh entry of each table. Since v_7 is a $\times$-gate, we do not need to do anything for $T_{\mathsf{nrel}_C}[7]$. Now $T_{\mathsf{rel}}[7]$ must contain the size of the relation computed by v_7. Since v_7 is a $\times$-gate, this is $|\mathsf{rel}(v_3)| \times |\mathsf{rel}(v_4)|$ which is equal to $T_{\mathsf{rel}}[3] \times T_{\mathsf{rel}}[4]$ by induction. So we set $T_{\mathsf{rel}}[7] = T_{\mathsf{rel}}[3] \times T_{\mathsf{rel}}[4]$. Similarly, $T_{\mathsf{var}}[7] = T_{\mathsf{var}}[3] \cup T_{\mathsf{var}}[4]$. Since $T_{\mathsf{var}}[7]$ is encoded as a bitvector, the union boils down to making a bitwise OR operation.

Finally, we need to compute $T_{\mathsf{rel}}[8], T_{\mathsf{var}}[8], T_{\mathsf{nrel}_C}[8]$. Since v_8 is a decision gate, $T_{\mathsf{var}}[8] = T_{\mathsf{var}}[5] \cup T_{\mathsf{var}}[6] \cup T_{\mathsf{var}}[7] \cup \{x_1\}$. Now $T_{\mathsf{rel}}[8] = T_{\mathsf{rel}}[5] + T_{\mathsf{rel}}[6] + T_{\mathsf{rel}}[7]$. Observe that in this case, we do not need to add corrective factor since v_5, v_6 and v_7 define relations on the same variables set. If the variables sets were different, we would have to adjust the value following Eq. (4.1). Now, it remains to compute $T_{\mathsf{nrel}_C}[8]$. Since v_8 has three incoming edges, we will have to compute $T_{\mathsf{nrel}_C}[8][1], T_{\mathsf{nrel}_C}[8][2], T_{\mathsf{nrel}_C}[8][3]$. We order the edges by increasing label and find that: $T_{\mathsf{nrel}_C}[8][1] = T_{\mathsf{rel}}[5] = 4$, $T_{\mathsf{nrel}_C}[8][2] = T_{\mathsf{nrel}_C}[8][1] + T_{\mathsf{rel}}[6] = 4 + 6 = 10$ and $T_{\mathsf{nrel}_C}[8][3] = T_{\mathsf{nrel}_C}[8][2] + T_{\mathsf{rel}}[7] = 10 + 4 = 14$.

The following lemma follows directly from the previously described algorithm:

Lemma 4.5 (Precomputation complexity)**.** *Given a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit C, we can compute a data structure in time $\mathcal{O}(|X|\mathsf{log}(|X|) \cdot |C|)$ that allows us to access in constant time to the following:*

- $\mathsf{var}(v)$, $|\mathsf{rel}(v)|$ *for every gate v of C,*
- $\mathsf{nrel}_C(v, d)$ *for every decision gate v having an ingoing edge labelled by $d \in D$.*

Proof. The data structure simply consists of the three tables $T_{\mathsf{nrel}_C}, T_{\mathsf{rel}}$ and T_{var}. It is easy to see that each entry of T_{var} can be computed in time $\mathcal{O}(|X|)$ since we only have to compute the union of sets of elements in X and we can use a bitmask of size $|X|$ to do it efficiently. Indeed, we can represent $\mathsf{rel}(v)$ as a bitvector $T_{\mathsf{var}}[v]$ with $|X|$ entries for each v such that $T_{\mathsf{var}}[v][i] = 1$ if and only if $x_i \in \mathsf{rel}(v)$. Now computing $T_{\mathsf{var}}[v][i]$ can be done by looking at every input w of v and check whether $T_{\mathsf{var}}[w][i] = 1$. For each gate w, we will have one such look up for each incoming edge and each variable, hence one can compute T_{var} in time $\mathcal{O}(|C| \cdot |X|)$.

Now observe that to compute T_{nrel_C}, one has to perform at most two arithmetic operations for each edge of C. Indeed, to compute $T_{\mathsf{rel}}[v]$, one has to perform at most one addition and one multiplication for each $w \in \mathsf{input}(v)$, whose cost can be associated to the edge (w, v). Similarly, to compute $T_{\mathsf{nrel}_C}[v, d]$ as described in the previous paragraph, we do one addition and one multiplication for each edge (w, v) in the circuit. Hence, we perform at most $\mathcal{O}(|C|)$ arithmetic operations. Now, the cost of these arithmetic operations is at most $\mathcal{O}(|X|\mathsf{log}(|X|))$ in the unit-cost RAM model (see Section 2 for a discussion). Indeed, all operations are performed on integers representing sizes of relations on domain D and variables set $Y \subseteq X$. Hence, their value cannot exceed $|D|^{|X|}$.

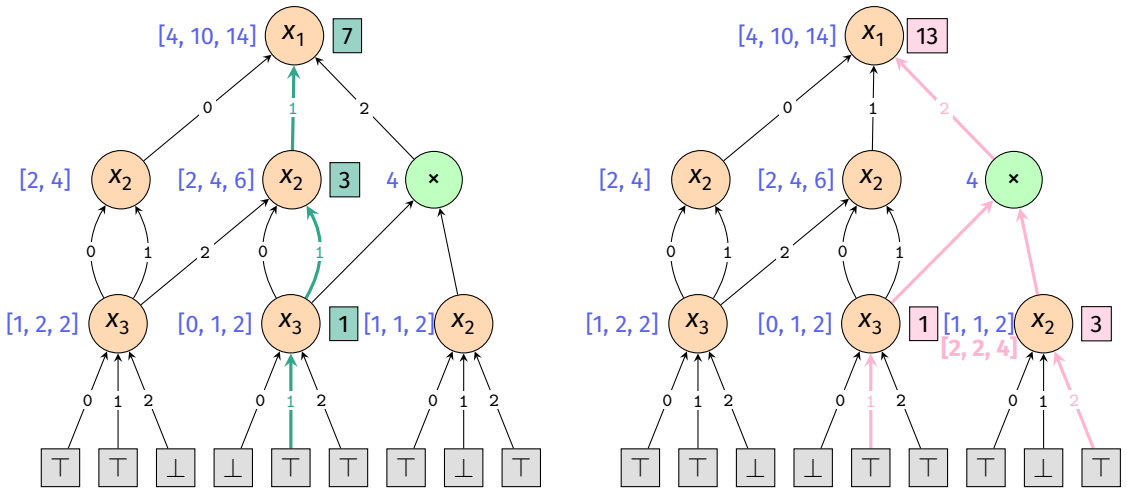
Hence, we have a total complexity of $\mathcal{O}(|X|\mathsf{log}(|X|) \cdot |C|)$. □

4.2.2. *Direct access.* We now show how the precomputation from Lemma 4.5 allows us to get direct access for ordered $\{\times, \mathsf{dec}\}$-circuits. We first show how one can solve a direct access task for any relation as long as we have access to very simple counting oracles. We then show that one can quickly simulate these oracle calls in ordered $\{\times, \mathsf{dec}\}$-circuits using precomputed values.

Example 4.6. An example of how the algorithm works is shown in Fig. 5. Fig. 5a shows the paths taken in the circuit to evaluate the 7^{th} tuple in the circuit. The main idea resides in the fact that we can connect direct access with counting tasks. If we want the 7^{th} tuple in the circuit, we cannot assign $x_1 \leftarrow 0$, since we know that there are only 4 possible extensions in that case. We should however assign $x_1 \leftarrow 1$, since there are 10 possible assignments with $x_1 \leqslant 1$. We then move down the circuit and search in the new gate, but we have to

adapt the index since 4 of the possible extensions have been discarded (because they involve assigning $x_1 \leftarrow 0$). We are now therefore looking for the 3rd tuple in this subcircuit. Again, assign $x_2 \leftarrow 0$ will only give 2 possible extensions, which is not enough to find the third solution. Assign $x_2 \leftarrow 1$ however is enough since there are 4 assignments with $x_2 \leqslant 1$. We hence set $x_2 \leftarrow 1$ and proceed with the last decision-gate. By assigning $x_2 \leftarrow 1$, we discarded 2 extensions, so we are now looking for the first answer in this last decision-gate, which is obtained by setting $x_3 \leftarrow 1$. We continue the search in this manner until reaching a terminal gate.

A more involved example can be found in Fig. 5b. In this case, we are looking for the 13th tuple of the circuit. However, after assigning (with the same method as before) $x_1 \leftarrow 2$, we reach a $\times$-gate, and therefore do not assign a value to a given variable. What happens is we are now looking for the $13 - 10 = 3^{\text{rd}}$ solution of a set of circuits comprised of the bottom-centre gate and the bottom-right gate. We choose which of the variables from the roots of the set of circuits to assign first based on the order, so in this case, we start with x_2. We have to remember that for each possible solution of the circuit rooted in x_2, there are two possible extensions from the circuit rooted in x_3 (see Lemma 4.8). This is the same thing as imagining the gate being virtually relabelled by $[2, 2, 4]$. In other words, $x_2 \leftarrow 0$ provides two tuples for the Cartesian product, which is not enough to get the third answer. The same happens for $x_2 \leftarrow 1$ but $x_2 \leftarrow 2$ has enough extension since there are 4 assignments for the Cartesian product that have $x_2 \leqslant 2$. We hence assign $x_2 \leftarrow 2$. We discarded 2 assignments with this choice, we are hence looking for the first assigmnent for x_3 which is found by setting x_3 to 1.



(A) $k = 7$ (B) $k = 13$

FIGURE 5. Examples of the paths followed during different direct access tasks on the same annotated ordered $\{\times, \mathsf{dec}\}$-circuit for order (x_1, x_2, x_3).

The following lemma connects direct access tasks with counting tasks. A similar claim can be found in [BCM22a, Proposition 3.5]:

Lemma 4.7. *Assume that we are given a relation $R \subseteq D^X$ with $X = \{x_1, \dots, x_n\}$ and an oracle $\mathcal{A}$ such that for every prefix assignment $\tau \in D^{\{x_1,\dots,x_p\}}$ and $N \in \mathbb{N}$, it returns the smallest value $d \in D$ such that $\#\sigma_{x_{p+1} \leqslant d}(\sigma_\tau(R)) \geqslant N$ and the value $\#\sigma_{x_{p+1}<d}(\sigma_\tau(R))$. Then, for any $k \leqslant |R|$, we can compute $R[k]$ using $\mathcal{O}(|X|)$ calls to oracle $\mathcal{A}$.*

Proof. Let $\alpha = R[k]$ be the answer we are looking for. We iteratively find the value of α on X by applying Lemma 2.1. More precisely, assume that we already know $\alpha(x_1), \dots, \alpha(x_p)$ for some $p < n$. Moreover, we know k_p such that $\alpha = \sigma_\beta(R)[k_p]$ where β is the prefix assignment of $\{x_1, \dots, x_p\}$ such that $\beta(x_i) = \alpha(x_i)$. We know by Lemma 2.1 that $\alpha(x_{p+1}) = \min\{d \mid \#\sigma_{x_{p+1} \leqslant d}(\sigma_\beta(R)) \geqslant k_p\}$ and $k_{p+1} = k_p - \#\sigma_{x_{p+1}<d}(\sigma_\beta(R))$, which we can compute with one call to oracle $\mathcal{A}$. Hence, by induction, we can fully reconstruct α with $\mathcal{O}(|X|)$ calls to oracle $\mathcal{A}$. □

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. To do that, we need to understand how an assignment of the form $D^{\{x_1,\dots,x_p\}}$ for some p behaves with respect to the circuit. We hence define a *prefix assignment* of size p to be an assignment $\tau \in D^{\{x_1,\dots,x_p\}}$ with $p \leqslant n$. Moreover, for a gate v in C, we define $\mathsf{sink}(v)$ as:

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

We have the following property:

Lemma 4.8. *For any gate v, we have $\mathsf{rel}(v) = \bigtimes_{w \in \mathsf{sink}(v)} \mathsf{rel}(w)$.*

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, $\mathsf{rel}(w) = \bigtimes_{g \in \mathsf{sink}(w)} \mathsf{rel}(g)$ for all inputs w of v. 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 a prefix assignment τ, we want to characterize the set of tuples in C that are compatible with τ. To do so, we can follow every decision-gate whose variable are set by τ and follow every input of $\times$-gates. This gives a set of gates in the circuit that we call the frontier of τ such that any assignment of $\mathsf{rel}(C)$ extending τ must be made from assignments of each gate in the frontier. More formally, we introduce the *frontier f_τ of τ* in C as follows:

(1) instantiate a set $F = \{\mathsf{out}(C)\}$, containing only the root of the circuit
(2) as long as it is possible, transform F as follows:
   - if $v \in F$ is a $\times$-gate, remove v from F and add $\mathsf{sink}(v)$ to F,
   - if $v \in F$ is a decision gate and the variable x labelling v is assigned in the prefix, *i.e.* $x \in \{x_1, \dots, x_p\}$, remove v from F and add v_d to F, where v_d is the input of v such that edge (v_d, v) is labelled by $\tau(x)$.
(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 represent the following relation: the *frontier relation* denoted by $\mathsf{rel}(f_\tau)$ is defined as the relation $\bigtimes_{v\in f_\tau} \mathsf{rel}(v)$. It is a relation defined on variables set $\mathsf{var}(f_\tau) := \bigcup_{v\in f_\tau} \mathsf{var}(v)$.

For a prefix τ on variables set $\{x_1,\dots,x_p\}$, we denote by $\sigma_\tau(R)$ the relation $\sigma_{x_1=\tau(x_1),\dots,x_p = \tau(x_p)}(R)$. We have the following connection between the relation of a frontier and prefix assignments:

Lemma 4.9. *Let C be an ordered $\{\times,\mathsf{dec}\}$-circuit on variables set $X=\{x_1,\dots,x_n\}$ and τ be a prefix assignment on variables set $\{x_1,\dots,x_p\}$. Then $\sigma_\tau(\mathsf{rel}_X(C)) = \{\tau\}\times \mathsf{rel}(f_\tau)\times D^{\{x_{p+1},\dots,x_n\}\setminus \mathsf{var}(f_\tau)}$.*

Proof. We prove the lemma by induction on the size of the prefix. For an empty prefix $\tau=\langle\rangle$, we have $f_\tau = \mathsf{sink}(\mathsf{out}(C))$. Indeed, if $\mathsf{out}(C)$ is a decision gate or input, then it is trivial since $\mathsf{sink}(\mathsf{out}(C)) = \{\mathsf{out}(C)\}$. Otherwise we simply sink through the $\times$-gate since no variable is assigned by τ. We have that $\sigma_{\langle\rangle}(\mathsf{rel}_X(C)) = \mathsf{rel}_X(C)$, which is itself by definition equal to $\mathsf{rel}(\mathsf{out}(C))\times D^{X\setminus \mathsf{var}(\mathsf{out}(C))}$. From Lemma 4.8, we know that $\mathsf{rel}(\mathsf{out}(C)) = \bigtimes_{w\in\mathsf{sink}(\mathsf{out}(C))}\mathsf{rel}(w)$. We know that $\mathsf{var}(\mathsf{out}(C)) = \mathsf{var}(f_{\langle\rangle})$. Thus, we have that $\sigma_{\langle\rangle}(\mathsf{rel}_X(C)) = \bigtimes_{w\in\mathsf{sink}(\mathsf{out}(C))}\mathsf{rel}(w)\times D^{\{x_1,\dots,x_n\}\setminus\mathsf{var}(f_{\langle\rangle})}$.

Now suppose the property holds for any prefix τ of size p. We now show that it also holds for a prefix $\tau' = \tau\times[x_{p+1}\leftarrow d]$.

We can rewrite $\sigma_{\tau'}(\mathsf{rel}_X(C))$ as $\sigma_{x_{p+1}=d}(\sigma_\tau(\mathsf{rel}_X(C)))$. Applying the induction hypothesis to this equality gives:

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

From here, we have two possibilities: either there exists a decision gate $v\in f_\tau$ such that $\mathsf{decvar}(v) = x_{p+1}$ or not.

First case: there exists $v\in f_\tau$ such that $\mathsf{decvar}(v)=x_{p+1}$. In this case, we have by definition: $f_{\tau'} = f_\tau\setminus\{v\}\cup\mathsf{sink}(v_d)$.

We start by pointing out that for a decision gate v with $x_{p+1}=\mathsf{decvar}(v)$ and $d\in D$, we have $\sigma_{x_{p+1}=d}(\mathsf{rel}(v)) = \{[x_{p+1}\leftarrow d]\}\times\mathsf{rel}(v_d)\times D^{\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup\mathsf{var}(v_d))}$. In other words, every tuple in the relation computed by v where $x_{p+1}=d$ is of the form $[x_{p+1}\leftarrow d]\times\tau'\times\sigma'$ where $\tau'\in\mathsf{rel}(v_d)$ and σ' is any tuple on domain D and variables set $\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup\mathsf{var}(v_d))$. We can therefore write:

$$\begin{aligned}
\sigma_{\tau'}(\mathsf{rel}_X(C)) &= \sigma_{x_{p+1}=d}\left(\{\tau\}\times\mathsf{rel}(f_\tau)\times D^{\{x_{p+1},\dots,x_n\}\setminus\mathsf{var}(f_\tau)}\right)\\
&\quad\text{since } x_{p+1} \text{ only appears in the frontier :}\\
&= \{\tau\}\times D^{\{x_{p+1},\dots,x_n\}\setminus\mathsf{var}(f_\tau)}\times\sigma_{x_{p+1}=d}(\mathsf{rel}(v))\times\bigtimes_{w\in f_\tau\setminus\{v\}}\mathsf{rel}(w)\\
&\quad\text{from the previous relation:}\\
&= \{\tau\}\times D^{\{x_{p+1},\dots,x_n\}\setminus\mathsf{var}(f_\tau)}\times\{[x_{p+1}\leftarrow d]\}\times\mathsf{rel}(v_d)\times D^{\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup\mathsf{var}(v_d))}\\
&\qquad\times\bigtimes_{w\in f_\tau\setminus\{v\}}\mathsf{rel}(w)\\
&\quad\text{from Lemma 4.8:}
\end{aligned}$$

$$
\begin{aligned}
&= \{\tau\} \times D^{\{x_{p+1},\dots,x_n\}\setminus \mathsf{var}(f_\tau)} \times \{[x_{p+1} \leftarrow d]\} \times \bigtimes_{w\in \mathsf{sink}(v_d)} \mathsf{rel}(w) \\
&\qquad \times D^{\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup \mathsf{var}(v_d))} \times \bigtimes_{w\in f_\tau\setminus\{v\}} \mathsf{rel}(w) \\
&= \{\tau\} \times \{[x_{p+1} \leftarrow d]\} \times D^{\{x_{p+1},\dots,x_n\}\setminus \mathsf{var}(f_\tau)} \times D^{\mathsf{var}(v)\setminus(\{x_{p+1}\}\cup \mathsf{var}(v_d))} \\
&\qquad \times \bigtimes_{w\in \mathsf{sink}(v_d)} \mathsf{rel}(w) \times \bigtimes_{w\in f_\tau\setminus\{v\}} \mathsf{rel}(w) \\
&= \{\tau'\} \times \bigtimes_{w\in f_{\tau'}} \mathsf{rel}(w) \times D^{\{x_{p+2},\dots,x_n\}\setminus \mathsf{var}(f'_\tau)}
\end{aligned}
$$

Second case: there are no $v \in f_\tau$ such that $\mathsf{decvar}(v) = x_{p+1}$. In this case, since the circuit is ordered, it means that $x_{p+1} \notin \mathsf{var}(f_\tau)$. Moreover $f_\tau = f_{\tau'}$. We can therefore write:

$$
\begin{aligned}
\sigma_{\tau'}(\mathsf{rel}_X(C)) &= \{\tau\} \times \bigtimes_{w\in f_\tau} \mathsf{rel}(w) \times \sigma_{x_{p+1}=d}(D^{\{x_{p+1},\dots,x_n\}\setminus \mathsf{var}(f_\tau)}) \\
&\quad \text{since } x_{p+1} \text{ does not appear in the frontier or } \tau: \\
&= \{\tau\} \times \bigtimes_{w\in f_\tau} \mathsf{rel}(w) \times \{[x_{p+1} \leftarrow d]\} \times D^{\{x_{p+2},\dots,x_n\}\setminus \mathsf{var}(f_\tau)} \\
&= \{\tau'\} \times \bigtimes_{w\in f_{\tau'}} \mathsf{rel}(w) \times D^{\{x_{p+2},\dots,x_n\}\setminus \mathsf{var}(f_{\tau'})} \\
&\quad \text{since } f_\tau = f_{\tau'}.
\end{aligned}
$$

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 4.10. *Let τ be a prefix assignment over the set of variables set $X = \{x_1, \dots, x_p\}$. 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 $\mathcal{O}(|X| + p) = \mathcal{O}(|X|)$. □

We are now ready to prove the main connection between frontiers and Lemma 4.7:

Lemma 4.11. *Let C be an ordered $\{\times, \mathsf{dec}\}$-circuit on variables set $X = \{x_1, \dots, x_n\}$ such that $\mathsf{nrel}_C(v, d)$ and $\mathsf{var}(v)$ have been precomputed as in Lemma 4.5. Let τ be a prefix assignment of $D^{\{x_1,\dots,x_p\}}$. Then, for every N, one can compute the smallest value $d \in D$ such that $\#\sigma_{x_{p+1}\leqslant d}(\sigma_\tau(\mathsf{rel}(C))) \geqslant N$ and the value $\#\sigma_{x_{p+1}<d}(\sigma_\tau(\mathsf{rel}(C)))$ in time $\mathcal{O}(|X|^2 \mathsf{log}|X| + |X|\mathsf{log}(|D|))$.*

Proof. We start by building the frontier f_τ associated with the prefix assignment τ. From Lemma 4.10, we know this can be done in time $\mathcal{O}(|X|)$. By Lemma 4.9:

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 there is a decision gate v labelled by variable x_{p+1} in f_τ or not. In the first case, since x_{p+1} only appears in the frontier, we have for every $d \in D$:

$$\sigma_{x_{p+1}\leqslant d}(\sigma_\tau(\mathsf{rel}(C))) = \{\tau\}\times \sigma_{x_{p+1}\leqslant d}(\mathsf{rel}(v))\times \bigtimes_{w\in f_\tau\setminus\{v\}} \mathsf{rel}(w)\times D^{\{x_{p+2},\dots,x_n\}\setminus \mathsf{var}(f_\tau)}$$

Hence we have:

$$\begin{aligned}\#\sigma_{x_{p+1}\leqslant d}(\sigma_\tau(\mathsf{rel}(C))) &= \#\sigma_{x_{p+1}\leqslant d}(\mathsf{rel}(v))\times \prod_{w\in f_\tau\setminus\{v\}} \#\mathsf{rel}(w)\times |D|^{|\{x_{p+2},\dots,x_n\}\setminus \mathsf{var}(f_\tau)|}\\ &= \mathsf{nrel}_C(v,d)\times \prod_{w\in f_\tau\setminus\{v\}} \#\mathsf{rel}(w)\times |D|^{|\{x_{p+2},\dots,x_n\}\setminus \mathsf{var}(f_\tau)|}\\ &= \mathsf{nrel}_C(v,d)P\end{aligned}$$

where P is a product containing at most $|X|$ precomputed integers since , all of them of size at most $|D|^{|X|}$ and hence, it can be evaluated in time $\mathcal{O}(|X|^2\mathsf{log}|X|)$. Indeed, every value $\#\mathsf{rel}(w)$ for $w \in f_\tau \setminus \{v\}$ have been precomputed and can be accessed in constant time. Moreover, $\mathsf{var}(f_\tau) = \bigcup_{v\in f_\tau} \mathsf{var}(v)$. Hence, since $\mathsf{var}(v)$ has been precomputed too, we can compute $e := |\{x_{p+2},\dots,x_n\}\setminus \mathsf{var}(f_\tau)|$ in $\mathcal{O}(|X|)$ and then $|D|^e$ in $\mathcal{O}(|X|^2\mathsf{log}|X|)$.

Now observe that the smallest value d such that $\#\sigma_{x_{p+1}\leqslant d}(\sigma_\tau(\mathsf{rel}(C))) \geqslant N$ has to be the label of one ingoing edge of v. Indeed, if d does not label any ingoing edge of v, then $\mathsf{nrel}_C(v,d) = \mathsf{nrel}_C(v,d_0)$ where $d_0 < d$ is the greatest value smaller than d, which labels one ingoing edge of v (or 0 if no such d_0 exists). Hence the smallest value d we are looking for has to be the label of one ingoing edge of v and is the smallest one among them such that $\mathsf{nrel}_C(v,d) \geqslant \frac{N}{P}$. Since P has been evaluated already, we can evaluate $\frac{N}{P}$ in time $\mathcal{O}(|X|^2\mathsf{log}|X|)$.

To find the smallest value d where $\mathsf{nrel}_C(v,d)$ is greater than $\frac{N}{P}$, it remains to do a binary search among the ingoing edges of v, which have already been sorted by increasing label. Comparing $\mathsf{nrel}_C(v,d)$ to $\frac{N}{P}$ can be done in $\mathcal{O}(|X|)$ for any d since the values do not exceed $|D|^{|X|}$. There are at most $|D|$ ingoing edge, meaning this binary search can be performed in time $\mathcal{O}(|X|\mathsf{log}|D|)$. Hence finding the value d we are interested in has a total complexity of $\mathcal{O}(|X|^2\mathsf{log}|X| + |X|\mathsf{log}|D|)$.

Now that we know the value $d \in D$ we are looking for, it remains to compute $\#\sigma_{x_{p+1}<d}(\sigma_\tau(\mathsf{rel}(C)))$. This value is equal to $\#\sigma_{x_{p+1}\leqslant d'}(\sigma_\tau(\mathsf{rel}(C)))$ where d' is the value labeling the ingoing edge of v just before d. If no such value exist, then $\#\sigma_{x_{p+1}<d}(\sigma_\tau(\mathsf{rel}(C)))$ is 0. Otherwise, we can compute it as before in time $\mathcal{O}(|X|^2\mathsf{log}|X|)$.

In the second case where there is no decision-gate labeled by x_{p+1} in 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:

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

where $\mathsf{rank}(d)$ is the number of elements smaller than d in D and P a product whose elements have been precomputed and which can hence be evaluated in time $\mathcal{O}(|X|^2\mathsf{log}|X|)$. Hence, the smallest value $d \in D$ such that $\#\sigma_{x_{p+1}\leqslant d}(D^{\{x_{p+1}\}}) \geqslant N$ is the one whose rank is $\lceil \frac{N}{P} \rceil$. Moreover, $\#\sigma_{x_{p+1}<d}(\sigma_\tau(\mathsf{rel}(C))) = \max(0, (\lceil \frac{N}{P} \rceil - 1)P)$. □

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 Fig. 5. 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.

The proof of Theorem 4.2 is now an easy corollary of Lemmas 4.7 and 4.11. Indeed, after having precomputed nrel_C and var using Lemma 4.5, we can answer direct access tasks using the oracle based algorithm from Lemma 4.7. Lemma 4.11 shows that these oracle accesses are in fact tractable in ordered circuits.

We conclude this section with a high level description of the direct access algorithm using pseudo code. The first procedure we need is the procedure Find-Frontier from Algorithm 2. Given a circuit C and a prefix assignment τ of $\{x_1, \ldots, x_p\}$, this procedure returns the frontier of τ by following every decision-gates whose variable is assigned by τ and sinking every $\times$-gate, implementing the procedure described in Lemma 4.10. Procedure Count-Extensions then computes the number of tuples in $\mathsf{rel}(C)$ compatible with τ and such that $x_{p+1} \leqslant d$ for some domain value $d \in D$. It uses the frontier of τ and the equality described in Lemma 4.11. Observe that one needs access to $\mathsf{nrel}_C(v, d)$, $\mathsf{var}(v)$ and $|\mathsf{rel}(v)|$ for this, which can be accessed from the annotations of the circuit described in the previous section.

The last function is the direct access function. The Next-Value function takes a circuit, a prefix assignment τ of $x_1, \ldots, x_{p-1}$, an index k and finds the value on x_p of the k^{th} tuple of $\sigma_\tau(\mathsf{rel}(C))$. It does so by using calls to the Count-Extensions function and does a binary search over domain value. We then implements the approach described in 4.7 in the Direct-Access function by iteratively constructing τ using Next-Value and also updating the index to take into account discarded tuples from $\mathsf{rel}(C)$.

Remark 4.12. Observe that in Next-Value and Lemma 4.11, we implemented the oracle from Lemma 4.7 using a binary search on the ingoing edges of decision gates, which is the origin of the $\mathcal{O}(\mathsf{log}|D|)$ factor in the access time. Actually, the oracle call that we use boils down to solving a problem known as $\mathsf{FindNext}$ in priority queues: given values $v_1 < \cdots < v_p$ and N, find the smallest j such that $v_j \geqslant N$. Some data structures such as van Emde Boas trees [vEB75] support this operation in time $\mathsf{log\ log}\ p$ which would allow us to improve the direct access to $\mathcal{O}(\mathrm{poly}(|Q|)\mathsf{log\ log}\ |D|)$ time for a small sacrifice in the preprocessing (inserting in van Emde Boas trees takes time $\mathcal{O}(\mathsf{log\ log}\ |D|)$) if we store values $\mathsf{nrel}_C(v, d)$ in such data structures instead of ordered lists.

Remark 4.13. We observe that our use of the Find-Frontier function is not optimal. Indeed, each call to Find-Frontier are dealt with independently while they are actually only updating a given frontier by iteratively adding new values in τ. Hence, in practice, we could get better performances by updating the frontier at each iteration of the for-loop in Direct-Access instead of recomputing it from scratch.

Algorithm 2 Given an ordered $\{\times, \mathsf{dec}\}$-circuit C on domain D, a prefix assignment τ assigning $\{x_1, \ldots, x_p\}$ and a domain value d, Count-Extensions returns the number of tuples in $\mathsf{rel}(C)$, compatible with τ such that $x_{p+1} \leqslant d$.

procedure Find-Frontier(C, τ)
  $tmp \leftarrow \{\mathsf{out}(C)\}$
  $F \leftarrow \emptyset$
  while $tmp \neq \emptyset$ **do**
    for $v \in tmp$ **do**
      if v is a $\bot$-gate **then return** $\emptyset$
      if v is a $\times$-gate **then** $tmp \leftarrow (tmp \setminus \{v\}) \cup \mathsf{input}(v)$;
      if v is a decision-gate on x and $\tau(x)$ is defined **then**
        if v has no incoming edge labeled by $\tau(x)$ **then return** $\emptyset$
        $(w, v) \leftarrow$ the incoming edge of v labeled by $\tau(x)$
        $tmp \leftarrow (tmp \setminus \{v\}) \cup \{w\}$
      else
        $F \leftarrow F \cup \{v\}$
        $tmp \leftarrow tmp \setminus \{v\}$
      end if
    end for
  end while
  return F
end procedure
procedure Count-Extensions(C, τ)
  $F \leftarrow$ Find-Frontier(C,τ)
  $Y \leftarrow \{x_{p+2}, \ldots, x_n\} \setminus \bigcup_{v \in F} \mathsf{var}(v)$
  if F contains a decision-gate w on variable x_{p+1} **then**
    return $|D|^{|Y|} \times \prod_{w \in F \setminus \{v\}} |\mathsf{rel}(w)| \times \mathsf{nrel}_C(v, d)$
  else
    return $|D|^{|Y|} \times \prod_{w \in F} |\mathsf{rel}(w)| \times \mathsf{rank}(d)$
  end if
end procedure

5. 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 [SBB+04], such that given a signed join query Q, $\prec$ an order on its variables and $\mathbf{D}$ a database, it returns a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit C such that $\mathsf{rel}(C) = Q(\mathbf{D})$. Exhaustive DPLL is an algorithm that was originally devised to solve the #SAT problem. It was observed by Huang and Darwiche [HD05] that the trace of this algorithm implicitly builds a Boolean circuit, corresponding to the $\{\times, \mathsf{dec}\}$-circuits on domain $\{0, 1\}$, enjoying interesting tractability properties. We show how to adapt it to the framework of signed join queries. The algorithm itself is presented in Section 5.1. We study the complexity of this algorithm in Section 5.2 depending on the structure of Q and $\prec$, using hypergraph structural parameters introduced in Section 3.3.

Algorithm 3 Given a ordered $\{\times, \mathsf{dec}\}$-circuit C on variable $(x_1, \ldots, x_n)$, domain $D = (d_1, \ldots, d_{|D|})$ and an index k, return the k^{th} tuple in $\mathsf{rel}(C)$

```
1: procedure NEXT-VALUE(C, τ, k, m, M)
2:     if m = M then
3:         if COUNT-EXTENSIONS(C,τ,d_m) ⩾ k then return d_m Else return ⊥
4:     else
5:         mi = ⌊(m+M)/2⌋
6:         if COUNT-EXTENSIONS(C, τ, d_mi) ⩾ k then return NEXT-VALUE(C, τ, k, m, mi)
7:         else return NEXT-VALUE(C, τ, k, mi, M)
8:     end if
9: end procedure
1: procedure DIRECT-ACCESS(C, k)
2:     p ← 1
3:     τ ← ⟨⟩
4:     while p ⩽ n do
5:         d ← NEXT-VALUE(C, τ, k, 1, |D|)
6:         if d = ⊥ then return ⊥
7:         τ ← τ × [x_p ← d]
8:         p ← p + 1
9:         i ← rank(d) (that is, d = d_i)
10:        if i > 1 then k ← k−COUNT-EXTENSIONS(C, τ, d_{i−1})
11:    end while
12: end procedure
```

5.1. **Exhaustive DPLL for signed join queries.** The main idea of DPLL for signed join queries is the following: given an order $\prec$ on the variables of a join query Q and a database $\mathbf{D}$, we construct a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit (where $x \succ y$ is defined as a shorthand for $y \prec x$)[4] computing $[\![Q]\!]^{\mathbf{D}}$ by successively testing the variables of Q with decision gates, from the highest to the lowest wrt $\prec$. At its simplest form, the algorithm picks the highest variable x of Q wrt $\prec$, creates a new decision gate v on x and then, for every value $d \in D$, sets x to d and recursively computes a gate v_d computing the subset of $[\![Q]\!]^{\mathbf{D}}$ where $x = d$. We then add v_d as an input of v. This approach is however not enough to get interesting tractability results. We hence add the following optimizations. First, we keep a cache of already computed queries so that if we recursively call the algorithm twice on the same input, we can directly return the previously constructed gate. Moreover, if we detect that the answers of Q are the Cartesian product of two or more subqueries $Q_1, \ldots, Q_k$, then we create a new $\times$-gate v, recursively call the algorithm on each component Q_i to construct a gate w_i and plug each w_i to v. Detecting such cases is mainly done syntactically, by checking whether the query can be partitioned into subqueries having disjoint variables. However, this approach would fail to give good complexity bounds in the presence of negative atoms. To achieve the best complexity, we also remove from Q every negative atom as soon as it is

[4]It may look strange at this point that the order given in the input differs from the order used by the output circuit. We will see later that $\prec$ corresponds to a structural parameters, generalizing existing results Section 3.3 but we have to follow it in reverse to build the circuit.

satisfied by the current partial assignment. This allows us to discover more cases in which the query has more than one connected component.

The complexity of the previously described algorithm may however vary if one is not careful in the way the recursive calls are actually made. We hence give a more formal presentation of the algorithm, whose pseudocode is given in Algorithm 4. We prove upper bounds on the runtime of Algorithm 4 parametrised by the structure of the query in Section 5.2. Since we are not insterested in complexity analysis yet, we deliberately let the underlying data structures for encoding relations unspecified and delay this discussion to Section 5.2.

Algorithm 4 An algorithm to compute a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit computing $[\![Q]\!]^{\mathbf{D}}$

procedure $\mathsf{DPLL}(Q, \tau, \mathbf{D}, \prec)$
  if (Q, τ) is in cache **then return** $\mathsf{cache}(Q, \tau)$
  if Q is inconsistent with τ **then return** $\bot$-gate
  if τ assigns every variable in Q **then return** $\top$-gate
  $x \leftarrow \max_{\prec} \mathsf{var}(Q)$
  for $d \in D$ **do**
    $\tau' \leftarrow \tau \times [x \leftarrow d]$
    if Q is inconsistent with τ' **then** $v_d \leftarrow \bot$-gate
    else
      Let $Q_1, \ldots, Q_k$ be the τ'-connected components of $Q \Downarrow \tau'$
      for $i = 1$ *to* k **do**
        $w_i \leftarrow \mathsf{DPLL}(Q_i, \tau_i, \mathbf{D}, \prec)$ where $\tau_i = \tau'|_{\mathsf{var}(Q_i)}$
      end for
      if $k = 1$ **then**
        $v_d \leftarrow w_1$
      else
        $v_d \leftarrow$ **new** $\times$-gate with inputs $w_1, \ldots, w_k$
      end if
    end if
  end for
  $v \leftarrow$ **new** dec-gate connected to v_d by a d-labelled edge for every $d \in D$
  $\mathsf{cache}(Q, \tau) \leftarrow v$
  return v
end procedure

A few notations are used in Algorithm 4. Given a database $\mathbf{D}$ on domain D and a tuple $\tau \in D^Y$ (where Y may contain variables not present in Q), 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 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 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 *with respect to* $\mathbf{D}$ (or simply inconsistent with τ when $\mathbf{D}$ is clear from context) if there is no $\sigma \in R^{\mathbf{D}}$ such that $\tau \simeq \sigma$. Observe that if Q contains a positive atom $R(\mathbf{x})$ that is inconsistent with τ then $[\![Q]\!]^{\mathbf{D}}_{\tau} = \emptyset$. Similarly, if Q contains a negative atom $\neg R(\mathbf{x})$ such that τ assigns every variable of $\mathbf{x}$ to a domain value and $\tau(\mathbf{x}) \in R$, then $[\![Q]\!]^{\mathbf{D}}_{\tau} = \emptyset$. If one of these cases arises, we say that Q is *inconsistent with* τ. Now observe that if $\neg R(\mathbf{x})$ is a negative atom of Q such that $R(\mathbf{x})$ is inconsistent with

τ, then $[\![Q]\!]^{\mathbf{D}}_\tau = [\![Q']\!]^{\mathbf{D}}_\tau \times D^W$ where $Q' = Q \setminus \{\neg R(\mathbf{x})\}$ and $W = \mathsf{var}(Q) \setminus (\mathsf{var}(Q') \cup \mathsf{var}(\tau))$ (some variables of Q may only appear in the atom $\neg R(\mathbf{x})$). This motivates the following definition: the *simplification of Q with respect to τ and* $\mathbf{D}$, denoted by $Q \Downarrow \langle \tau, \mathbf{D}\rangle$ or simply by $Q \Downarrow \tau$ when $\mathbf{D}$ is clear from context, is defined to be the subquery of Q obtained by removing from Q every 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 $W = \mathsf{var}(Q) \setminus (\mathsf{var}(Q') \cup \mathsf{var}(\tau))$.

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 set of 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 components of $\mathcal{I}^Q_\tau$ and Q' contains every atom a of Q on variables set $\mathbf{x}$ 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 $\mathsf{var}(Q_{C_1}) \cap \mathsf{var}(Q_{C_2}) \subseteq Y$.

Example 5.1. We illustrate the previous definitions and a run of Algorithm 4 on the following query: $Q(x_1, x_2, x_3, x_4) = \neg S(x_1, x_2, x_3, x_4) \wedge T(x_1, x_3) \wedge R(x_2, x_4)$ and relations given in Fig. 6a on domain $\{0, 1\}$.[5] The circuit built by Algorithm 4 is depicted in Fig. 6b. Observe that the first recursive call happens on assignment $\tau_0 = [x_1 \leftarrow 0]$ which is still consistent with S. Hence, the τ_0-intersection graph of Q is a path connecting T, S and R with S between T and R. It only has one connected component, hence no Cartesian product gate is produced and a decision gate on x_2 is created. The next recursive call happens on assignment $\tau_1 = [x_1 \leftarrow 0, x_2 \leftarrow 0]$. The τ_1-intersection graph of Q is the same as before and has only one connected component. A decision gate on x_3 is then created and a recursive call happens with $\tau_2 = [x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 0]$. Now, $\neg S$ and τ_2 are inconsistent, hence a $\bot$-gate is created and the algorithm backtracks and sets x_3 to 1 to build the partial assignment $\tau' = [x_1 \leftarrow 0, x_2 \leftarrow 0, x_3 \leftarrow 1]$ on Line 7. Now observe that $Q \Downarrow \tau'$ only contains atom $T(x_2, x_4)$ because $S(x_1, x_2, x_3, x_4)$ is inconsistent with τ' and hence atom $\neg S$ is removed. Moreover, all variables of $R(x_1, x_3)$ are set by τ' and it is also removed in $Q \Downarrow \tau'$. The τ'-intersection graph of Q hence has only one vertex $T(x_2, x_4)$. A recursive call is then issued with input $(T(x_2, x_4), [x_2 \leftarrow 0])$ since $\tau'|_{x_2, x_4} = [x_2 \leftarrow 0]$. A decision gate is created for x_4 and both values $0, 1$ give answers of Q which creates two $\top$-gates.

Now, the algorithm backtracks to the decision gate labelled by x_2 and deals with this part of the recursive call. Later, the algorithm will eventually backtrack to the first call in the stack and set $\tau' = [x_1 \leftarrow 1]$ on Line 7. As before, $\neg S$ is simplified in $Q \Downarrow \tau'$. Hence the τ'-intersection graph of $Q \Downarrow \tau'$ has only one vertex for T and one vertex of R. But both atoms do not share variables, hence a Cartesian product gate is created and two recursive calls happen. The first one has parameter $(R(x_1, x_3), [x_1 \leftarrow 1])$ and the other

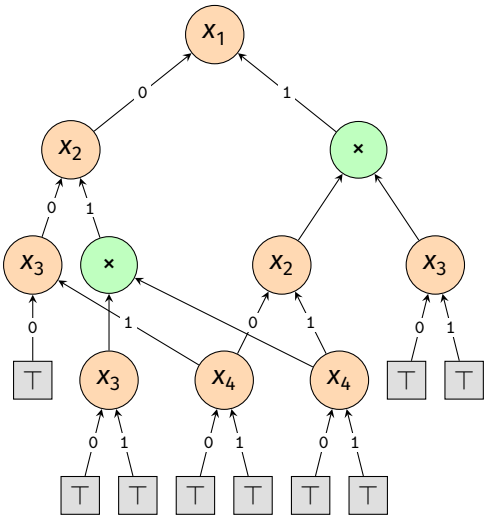
[5] An interactive version of this example and others can be found at `https://florent.capelli.me/algorithms/dpll`.

S	x_1	x_2	x_3	x_4
	0	0	0	0

R	x_2	x_4
	0	0
	0	1
	1	1
	1	0

T	x_1	x_3
	0	1
	1	1

(A) Database on domain $\{0, 1\}$.



(B) Produced circuit.

FIGURE 6. A run of Algorithm 4 with $Q(x_1, x_2, x_3, x_4) = \neg S(x_1, x_2, x_3, x_4) \wedge R(x_2, x_4) \wedge T(x_1, x_3)$. Details are given in Example 5.1

one $(T(x_2, x_4), [])$. We focus on this second recursive call. The first call it makes is on $(T(x_2, x_4), [x_2 \leftarrow 0])$. But now, recall that this call has already been made (see the end of previous paragraph). Hence the gate computing it is stored in the cache and a cache hit occurs, leading to sharing in the circuit.

Algorithm 4 uses the previous observations to produce a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit. More precisely:

Theorem 5.2. *Let Q be a signed join query, $\mathbf{D}$ a database and $\prec$ an order on $\mathsf{var}(Q)$, then $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ constructs a $\succ$-ordered $\{\times, \mathsf{dec}\}$-circuit C and returns a gate v of C such that $\mathsf{rel}_{\mathsf{var}(Q)}(v) = [\![Q]\!]^{\mathbf{D}}$.*

Proof. The proof is by induction on the number of variables of Q that are not assigned by τ. We claim that $\mathsf{DPLL}(Q, \tau, \mathbf{D}, \prec)$ returns a gate computing $[\![Q]\!]^{\mathbf{D}}_{\tau}$ which is stored into $\mathit{cache}(Q, \tau)$. If every variable is assigned, then $\mathsf{DPLL}(Q, \tau, \mathbf{D}, \prec)$ returns either a $\top$-gate or a $\bot$-gate depending on whether τ is inconsistent with Q or not, which clearly is $[\![Q]\!]^{\mathbf{D}}_{\tau}$. Otherwise, it returns and adds in the cache a decision gate v connected to a gate v_d by 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 since in this case, by definition of the relation computed by a decision-gate, v computes $\bigcup_{d \in D} [\![Q]\!]^{\mathbf{D}}_{\tau \times [x \leftarrow d]} \times [x \leftarrow d] = [\![Q]\!]^{\mathbf{D}}_{\tau}$.

To prove that v_d computes $[\![Q]\!]^{\mathbf{D}}_{\tau'}$ where $\tau' = \tau \times [x \leftarrow d]$, we separate two cases: if τ' is inconsistent with Q then $[\![Q]\!]^{\mathbf{D}}_{\tau'}$ is empty and v_d is a $\bot$-gate, which is what is expected. Otherwise, let $Q_1, \ldots, Q_k$ be the τ'-connected components of $Q \Downarrow \tau'$ and let $\tau_i = \tau'|_{\mathsf{var}(Q_i)}$. 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 Line 12, obtained from a recursive call to $\mathsf{DPLL}(Q_i, \tau_i, \mathbf{D}, \prec)$ where the number of variables not assigned by τ_i in Q_i is less than the number of variables unassigned by τ in Q. Hence,

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 have $\mathsf{rel}(v_d) = \bigtimes_{i=1}^{k} [\![Q_i]\!]^{\mathbf{D}}_{\tau_i}$. □

The worst case complexity of DPLL may be high when no cache hit occurs which would result in at least $|[\![Q]\!]^{\mathbf{D}}|$ recursive calls. However, when $\prec$ has good properties with respect to Q, we can prove better bounds. Section 5.2 gives upper bounds on the complexity of DPLL depending on $\prec$, using measures defined in Section 3.3.

5.2. **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 5.3. *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 $\mathcal{O}((\mathrm{poly}_k\,|Q|) \cdot |\mathbf{D}|^k \cdot |D|)$ such that $\mathsf{rel}(C) = [\![Q]\!]^{\mathbf{D}}$ and:*

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

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

We now proceed to prove Theorem 5.3. In this section, we fix a signed join query Q that has exactly one $\langle\rangle$-component. This can be done without loss of generality since the case where Q has many $\langle\rangle$-components can be easily dealt with by constructing the Cartesian product of each $\langle\rangle$-component of Q. We also fix a database $\mathbf{D}$ and an order $\prec$ on $\mathsf{var}(Q) = \{x_1, \ldots, x_n\}$ 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 $\{x_1, \ldots, x_i\}$ by $X_{\preceq x_i}$. Similarly, $X_{\prec x_i} = X_{\preceq x_i} \setminus \{x_i\}$, $X_{\succ x_i} = \mathsf{var}(Q) \setminus X_{\preceq x_i}$ and $X_{\succeq x_i} = \mathsf{var}(Q) \setminus X_{\prec x_i}$.

Finally, we let $\mathbf{R}^{\mathbf{D}}_Q$ be the set of (K, σ) such that $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ makes at least one recursive call to $\mathsf{DPLL}(K, \sigma, \mathbf{D}, \prec)$ and such that K is consistent with σ. We start by bounding the size of the circuit and the runtime in terms of the number of recursive calls:

Lemma 5.4. *$\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ produces a circuit of size at most $\mathcal{O}(|\mathbf{R}^{\mathbf{D}}_Q| \cdot |D| \cdot \mathrm{poly}(|Q|))$ in time $\tilde{\mathcal{O}}(|\mathbf{R}^{\mathbf{D}}_Q| \cdot |D| \cdot \mathrm{poly}(|Q|))$.*

Proof. Given $(K, \sigma) \in \mathbf{R}^{\mathbf{D}}_Q$, we bound the number of edges created in the circuit during 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 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. Moreover, any recursive call of the form (K, σ) with K inconsistent with σ will return a $\bot$-gate without creating any new edge. Hence, the size of the circuit produced in the end is at most $|\mathbf{R}^{\mathbf{D}}_Q| \cdot |D| \cdot (m + 1) = \mathcal{O}(|\mathbf{R}^{\mathbf{D}}_Q| \cdot |D| \cdot \mathrm{poly}(Q))$.

Moreover, each operation in Algorithm 4 can be done in time polynomial in $|Q|$ if one stores the relation using a well-chosen data structure. 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 $\tilde{\mathcal{O}}(|R|)$ and project R on $x_1, \ldots, x_{n-1}$ in $\tilde{\mathcal{O}}(1)$ (this is the encoding used in the Triejoin algorithm from [Vel14]). Hence, we can test for inconsistency in time $\tilde{\mathcal{O}}(|Q|)$ after having fixed the highest variables in Q to a value $d \in D$

by going over every atom of Q. During a recursive call (K, σ) where K is inconsistent with σ, we have to check for this inconsistency before returning $\bot$ and cannot really do it for free. To simplify the analysis, we assume that this check is performed before calling the function and hence we can assume every recursive call (K, σ) where K is inconsistent with σ is done for free. In other words, we incur the cost of this call to the calling function.

Moreover computing the σ'-connected components can be done in polynomial time in $|Q|$ since it boils down to finding the connected components of a graph having at most m nodes. Such a graph can be constructed in polynomial time in $|Q|$ by testing intersections of variables in atoms. Finally, from the previous discussion, a recursive call to Algorithm 4 creates at most $m + 1$ edges for each $d \in D$. Moreover, reading and writing values in the cache can be done in time $\tilde{\mathcal{O}}(\text{poly}(|Q|))$ by using a hash table. Indeed, the cached values are subqueries of Q together with partial variable assignments, hence they can be encoded with $\mathcal{O}(|Q|\mathsf{log}|D|)$ bits. If we account for the cost of reading the cache in a recursive call and the cost of checking inconsistency (as explained before) directly on Line 12, the time for each $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$ is $\tilde{\mathcal{O}}(|D| \cdot \text{poly}(|Q|))$ and the total time is $\tilde{\mathcal{O}}(|\mathbf{R}_Q^{\mathbf{D}}| \cdot |D| \cdot \text{poly}(|Q|))$. □

It remains to bound the size of $\mathbf{R}_Q^{\mathbf{D}}$ which is done in two steps. Lemma 5.5 characterises the structure of the elements of $\mathbf{R}_Q^{\mathbf{D}}$ and Lemma 5.6 shows connections with the structure of the hypergraph of Q. We need a few notations. Let $Q' \subseteq Q$ be a subquery of Q and x, y two variables of Q' such that $y \prec x$. An *x-path to y in Q'* is a list $x_0, a_0, x_1, a_1, \ldots, a_{p-1}, x_p$ where a_i is an atom of Q' on variables $\mathbf{x_i}$, x_i and x_{i+1} are variables of $\mathbf{x_i}$, $x_0 = x$, $x_p = y$ and $x_i \preceq x$ for every $i \leqslant p$. Thus, x-paths to y in the hypergraph of Q' are simply paths that start from x, end in y and are only allowed to go through variables smaller than x. By extension, given an atom a of Q', an *x-path to a in Q'* is an x-path to some variable y in a. The *x-component of Q'* is the set of atoms a of Q' such that there exists an x-path to a in Q'. An *x-access to y in Q'*, is an x-path to some atom a such that is a variable of a. Notice that there is an x-access to y in Q' iff y occurs in the x-component of Q'. Notice that when there is an x-access to y in Q', it may be the case that $x \prec y$.

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

This observation is stated formally in Lemma 5.5 whose proof is quite technical, due in part to heavy notations and concepts that are necessary to truly formalise what is happening. A picture illustrating the proof of Lemma 5.5 can be found on Fig. 7.

Lemma 5.5. *Let $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$ and let x be the biggest variable of K not assigned by σ. There exists τ a partial assignment of $X_{\succ x}$ such that: $\sigma = \tau|_{\mathsf{var}(K)}$ and K is the x-component of $Q \Downarrow \tau$.*

Proof. The proof is by induction on the order of recursive calls. We start by the first call, $(Q, \langle\rangle)$. Since Q has one $\langle\rangle$-component, the x_n-component of Q is Q itself. Moreover, since $Q \Downarrow \langle\rangle = Q$, we have that Q is the x_n-component of $Q \Downarrow \langle\rangle$. Now let $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$. By definition, the recursive call is made during the execution of another recursive call with parameters (K', σ'). Assume by induction that for (K', σ'), the statement of Lemma 5.5 holds. In other words, let x' be the biggest variable of K'. Then K' is the x'-component of $Q \Downarrow \tau'$ for some partial assignment τ' of $X_{\succ x'}$ such that $\tau'|_{\mathsf{var}(K')} = \sigma'$. The recursive call

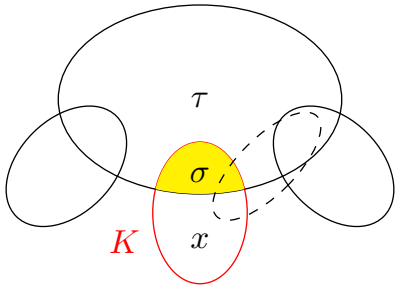


Figure 7. Illustration of Lemma 5.5: τ is the assignments of variables so far when the first recursive call to (K, σ) occurs and it separates Q into disjoint τ-components. Some negative atoms of Q (represented with dashed edges) have been simplified away in $Q \Downarrow \tau$. K, in red, is a τ-connected component of $Q \Downarrow \tau$: it is connected only through the atoms not simplified in $Q \Downarrow \tau$. σ is the part of τ that sets variables in K, and x is the largest variable of K not assigned by σ. Lemma 5.5 proves that K is exactly the x-component of $Q \Downarrow \tau$, that is, every atom that can be reached from x using atoms in $Q \Downarrow \tau$ and variables smaller than x.

then made to (K, σ) has the following form: $\sigma = \sigma''|_{\mathsf{var}(K)}$ where $\sigma'' = \sigma' \cup [x' \leftarrow d]$ for some $d \in D$ and K is a σ''-component of $K' \Downarrow \sigma''$.

We claim that K is the x-component of $Q \Downarrow \tau$, where $\tau = \tau' \cup [x' \leftarrow d]$. First observe that every atom a of K is in $Q \Downarrow \tau$. Indeed, if a is positive, then a is also in $Q \Downarrow \tau$ by definition. Now if $a = \neg R(\mathbf{x})$ is negative, we claim that a is not inconsistent with τ. Indeed, by induction, a is in $Q \Downarrow \tau'$, hence it is not inconsistent with τ'. Now since a is also in K and K is a subset of $K' \Downarrow \sigma''$ and $\sigma''(x') = d$, we know that a is not inconsistent with $\tau' \cup [x \leftarrow d]$ which is τ by definition. Hence a is in $Q \Downarrow \tau$.

Now let a be an atom of K. K is a σ''-connected component of $K' \Downarrow \sigma''$. Hence, by definition, we have a path in K from x to some variable y in a that does not use any variable assigned by σ'', which is equivalent to saying that it does not use any variable assigned by σ since, by definition, $\sigma = \sigma''|_{\mathsf{var}(K)}$. As x is defined to be the biggest variable of K that is not assigned by σ, the path from x to y only uses variables smaller than x. In other words, there is an x-path to y in K. That is, every atom a of K is in the x-component of $Q \Downarrow \tau$.

Now let a be an atom that is in the x-component of $Q \Downarrow \tau$. Let y be a variable in a such that there is an x-path to y in $Q \Downarrow \tau$. We call this path p. We first show that p is also in K'. Since $Q \Downarrow \tau \subseteq Q \Downarrow \tau'$, p is also an x-path to y of $Q \Downarrow \tau'$. As x is occurring in K and $K \subseteq K'$, x occurs in some atom of K'. Moreover, as K' is the x'-component of $Q \Downarrow \tau'$, there is an x'-path to x, call it p', in K'. Since $K' \subseteq Q \Downarrow \tau'$, p' is also an x'-path to x in $Q \Downarrow \tau'$. The path $p'p$ obtained by concatenating the p' and p in $Q \Downarrow \tau'$ is an x'-path to y in $Q \Downarrow \tau'$. Since K' is the x'-component of $Q \Downarrow \tau'$, $p'p$ is in K' and hence p is in K'. We now show that p is in K. By definition, K is the σ''-connected component of $K' \Downarrow \sigma''$ that contains x. In particular, it contains every path from x to other variables in K' that does not pass through x'. As p is an x-path to y, it cannot pass through x' (since $x \prec x'$) and is thus in K'. This finally shows that a is in K and concludes the proof.

□

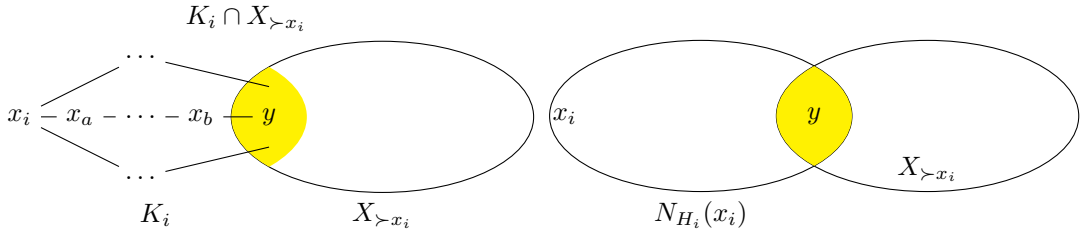


FIGURE 8. Illustration of Lemma 5.6: in the first picture, K_i is pictured on the left and the yellow area represents the variables in K_i that are greater than x_i, that is, that can be reached by following paths of variables smaller than x_i. When removing variables smaller than x_i, these paths are progressively merged into the neighborhood of x_i resulting in the second picture: in H_i, the neighborhood of x_i is exactly the yellow area from the first picture, that is, the vertices of K_i greater than x_i. Every other variable from K_i have been removed. The proof of Lemma 5.6 consists in proving by induction on paths length that both yellow areas are the same.

The following lemma establishes a connection between x-components and the structure of the underlying hypergraph. In essence, it allows us to bound the number of atoms needed to cover $X_{\succ x}$ in an x-component using the signed hyperorder width. Fig. 8 illustrates the lemma and give an idea of the proof.

Lemma 5.6. *Let Q be a signed join query on variables set $X = \{x_1, \dots, x_n\}$. Let $1 \leqslant i \leqslant n$ and K_i the x_i-component of Q. We let H be the hypergraph of Q, $H_1 = H$ and $H_{j+1} = H_j/x_j$. We have $N_{H_i}(x_i) = \mathsf{var}(K_i) \cap X_{\succeq x_i}$.*

Proof. The proof is by induction on i. We start by proving the equality for $i = 1$. Since there are no variables of K_1 smaller than x_1, it is clear that $K_1 = \{a \in \mathsf{atoms}(Q) \mid x_1 \in \mathsf{var}(a)\}$. Hence $\mathsf{var}(K_1) \cap X_{\succeq x_1} = \mathsf{var}(K_1)$. Moreover, $N_H(x_1)$ is exactly the set of variables of atoms of Q containing x_1 since there is one hyperedge in H per atom of Q. Hence, $\mathsf{var}(K_1) \cap X_{\succeq x_1} = N_H(x_1) = N_{H_1}(x_1)$, the last equality being a consequence of $H_1 = H$.

Now assume that the equality has been established up to x_{i-1}. We start by proving that $\mathsf{var}(K_i) \cap X_{\succeq x_i} \subseteq N_{H_i}(x_i)$. Let $w \in \mathsf{var}(K_i) \cap X_{\succeq x_i}$. First assume $w \in N_H(x_i)$. That is, there is an edge e in H such that both w and x_i are in e. Moreover, it is easy to see that $e \setminus \{x_1, \dots, x_{i-1}\}$ is an edge of H_i. Hence it is clear that $w \in N_{H_i}(x_i)$. Otherwise, if $w \notin N_H(x_i)$, as $w \in \mathsf{var}(K_i)$, by definition of K_i, there is an x_i-access of length greater than 1 to w in K_i. Let x_j be the biggest node on that path that is different from x_i. By definition, an x_i-access is an x_i-path and $j < i$. Moreover, splitting that of x_i-access at x_j gives an x_j-access to x_i and an x_j-access to w in K_i. Thus, x_i and w occur in the x_j-component of K_i. Since $K_i \subseteq H$, they also occur in K_j, the x_j-component of H. By induction, we thus have $w \in N_{H_j}(x_j)$ and $x_i \in N_{H_j}(x_j)$. Hence w and x_i are neighbours in H_{j+1} since the edge $N_{H_j}(x_j) \setminus \{x_j\}$ has been added in H_{j+1}. In particular, since $i > j$, it means that H_i contains the edge $N_{H_j}(x_j) \setminus \{x_j, \dots, x_{i-1}\}$, which contains both w and x_i, hence $w \in N_{H_i}(x_i)$. This establishes the first part of the proof, that is, $\mathsf{var}(K_i) \cap X_{\succeq x_i} \subseteq N_{H_i}(x_i)$.

We now prove the converse inclusion. Let $w \in N_{H_i}(x_i)$. By definition, $w \in X_{\succeq x_i}$ since $x_1, \dots, x_{i-1}$ have been removed in H_i. It remains to prove that w is in an atom a such that there is an x_i-path to a. If x_i and w are neighbours in H, then it means that they

appear together in an atom of Q and it is clear 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 since $w \in N_{H_i}(x_i)$. We must have $j > 1$ since x_i and w are not neighbours in $H = H_1$. The minimality of j implies that x_i and w are not neighbours in H_{j-1}. Since the only edge added in H_j is $N_{H_{j-1}}(x_{j-1}) \setminus \{x_{j-1}\}$, it means that both x_i and w are neighbours of x_{j-1} in H_{j-1}, that is, $x_i \in N_{H_{j-1}}(x_{j-1})$ and $w \in N_{H_{j-1}}(x_{j-1})$. By induction, both x_i and w are variables of K_{j-1}. In other words, there is an x_{j-1}-access to x_i and an x_{j-1}-access to w in K_{j-1}. Since $j - 1 < i$, the paths that constitute these x_{j-1}-accesses only pass through variables y so that $y \prec x_{j-1} \prec x_i$. This shows that that there is an x_i-access to w in K_{j-1}. So w is in the x_i-component of K_{j-1} and thus in the x_i-component of H. Hence $w \in \mathsf{var}(K_i)$. □

We are now ready to prove the upper bound on $|\mathbf{R}_Q^{\mathbf{D}}|$ depending on the width of $\prec$.

Lemma 5.7. *Let m be the number of atoms of Q and n the number of variables. We have:*

- *if Q is a positive join query, $|\mathbf{R}_Q^{\mathbf{D}}| \leqslant n|\mathbf{D}|^k$ where $k = \mathsf{fhow}(H(Q), \prec)$.*
- *otherwise $|\mathbf{R}_Q^{\mathbf{D}}| \leqslant nm^{k+1}|\mathbf{D}|^k$ where $k = \mathsf{show}(H(Q), \prec)$.*

Proof. We start with the case where Q is a positive join query. Let $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$. In this case, we know by Lemma 5.5 that K is the x_i-component of $Q \Downarrow \tau$ for some $\tau \supseteq \sigma$. Now, since Q does not have negative atoms, $Q = Q \Downarrow \tau$ since $Q \Downarrow \tau$ is obtained from Q by removing negative atoms only. In other words, K is the x_i-component of Q and σ assigns the variables of K that are greater than x_i. We also know that σ is not inconsistent with the atoms of K by definition of $\mathbf{R}_Q^{\mathbf{D}}$. Hence, σ satisfies every atom of K when projected on $X_{\succ x_i}$. By Lemma 5.6, $\mathsf{var}(K) \cap X_{\succ x_i} = N_{x_i}(H_i)$ where H_i is defined as in Lemma 5.6. Thus, by definition, there exits a fractional cover of $N_{H_i}(x_i)$ using the atoms of Q with value at most $k = \mathsf{fhow}(H(Q), \prec)$. In other words, σ can be seen as the projection on $\mathsf{var}(K) \cap X_{\succ x_i}$ of an answer of the join of the atoms involved in the fractional cover. By **??** 2.2, this join query has at most $|\mathbf{D}|^k$ answers. Hence, there are at most $n|\mathbf{D}|^k$ possible elements in $\mathbf{R}_Q^{\mathbf{D}}$: there are at most n x_i-components (one for each $i \leqslant n$), and at most $|\mathbf{D}|^k$ associated assignments σ to each component.

The case of signed queries is similar but requires more attention. Again, for $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$, we know that K is the x_i-component of $Q \Downarrow \tau$ for some $\tau \supseteq \sigma$ and, as before, τ is consistent with every positive atom in $Q \Downarrow \tau$. Moreoever, if $\neg R(\mathbf{x})$ is an atom of $Q \Downarrow \tau$, then τ is consistent with $R(\mathbf{x})$, since otherwise $\neg R(\mathbf{x})$ would not be in $Q \Downarrow \tau$. Now, let H' be the hypergraph of $Q \Downarrow \tau$. By definition, it is a subhypergraph of $H(Q)$, where only negative edges have been removed. Hence, by Lemma 5.6, $\mathsf{var}(K) \cap X_{\succ x_i} = N_{H'_i}(x_i)$ is covered by at most $\mathsf{show}(H', \prec) \leqslant \mathsf{show}(H(Q), \prec) = k$ edges. Hence, σ, which corresponds to τ restricted to $\mathsf{var}(K) \cap X_{\succ x_i}$, can be seen as the projection to $\mathsf{var}(K) \cap X_{\prec x_i}$ of an answer of a positive join query having at most k atoms of Q. Indeed, even if an edge of $H(Q)$ used to cover $\mathsf{var}(K) \cap X_{\succ x_i}$ corresponds to a negative atom $\neg R$ of Q, we know that τ is consistent with R, otherwise, the atom $\neg R$ would have been simplified by τ in $Q \Downarrow \tau$.

Consider $I_k(Q)$ to be the following set: $I_k(Q)$ contains every pair (L, α) such that L is a subset of at most k atoms $P_1, \dots, P_{k_1}, \neg N_1, \dots, \neg N_{k_2}$ of Q (that is $k_1 + k_2 \leqslant k$) and α is a solution of the join query with atoms $P_1, \dots, P_{k_1}, N_1, \dots, N_{k_2}$ on $\mathbf{D}$. By definition, it is easy to see that $|I_k(Q)| \leqslant m^{k+1}|\mathbf{D}|^k$. Indeed, there are at most $\sum_{j=1}^{k} \binom{m}{j} \leqslant m^{k+1}$ choices for L and since α is an answer of a positive join query having at most k relations, there are at most $|\mathbf{D}|^k$ such answers.

From what precedes, we know that for every $(K, \sigma) \in \mathbf{R}_Q^{\mathbf{D}}$, there exists some α such that $(K, \alpha) \in I_k(Q)$ and $\sigma = \alpha|_{\mathsf{var}(K) \cap X_{\succ x_i}}$ for some $x_i \in X$. Hence there are at most $nm^{k+1}|\mathbf{D}|^k$ elements in $\mathbf{R}_Q^{\mathbf{D}}$, the extra n-factor originating from the choice of x_i. □

Now Theorem 5.3 is a direct corollary of Lemmas 5.4 and 5.7. If Q is not $\langle\rangle$-connected, then the first recursive call of DPLL will simply break Q into at most $\mathcal{O}(m)$ connected components and recursively call itself on each now $\langle\rangle$-component of Q.

Observe that our result is based on signed hyperorder width, which is not the fractional version we had in Section 3. This is because with our current understanding of the algorithm, we are not able to prove a polynomial combined complexity bound on the runtime of DPLL when only the signed fractional hyperorder width is bounded. Indeed, the proof of Lemma 5.7 breaks in this case. The proof would be analoguous up to the definition of $I_k(Q)$. Indeed, to account for fractional cover, we should not consider $I_k(Q)$ anymore but $J_k(Q)$ defined as follows: $J_k(Q)$ contains every pair (L, α) such that L is a subset of atoms $P_1, \dots, P_{k_1}, \neg N_1, \dots, \neg N_{k_2}$ of Q **having a fractional cover of at most** k and α is a solution of the join query with atoms $P_1, \dots, P_{k_1}, N_1, \dots, N_{k_2}$ on $\mathbf{D}$. Now, bounding the number of α once L is fixed by $|\mathbf{D}|^k$ can still be done by **??** 2.2 but we cannot bound the number of distinct L by m^{k+1} anymore. It is not clear to us whether this number can be bounded by a polynomial (where k is considered a constant) in m. The best that we can do is to bound this number by 2^m, that is, just saying that L is one subset of atoms of Q. By doing this, we can still bound $|\mathbf{R}_Q^{\mathbf{D}}|$ by $2^m n|\mathbf{D}|^k$ which gives:

Theorem 5.8. *Let Q be a signed join query with n variables and m atoms, $\mathbf{D}$ 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 $\mathcal{O}((2^m \operatorname{poly}(|Q|) \cdot |\mathbf{D}|^k \cdot |D|)$ such that $\mathsf{rel}(C) = [\![Q]\!]^{\mathbf{D}}$ where $k = \mathsf{sfhow}(H(Q), \prec)$. Moreover, the runtime of $\mathsf{DPLL}(Q, \langle\rangle, \mathbf{D}, \prec)$ is at most $\tilde{\mathcal{O}}(2^m \operatorname{poly}(|Q|) \cdot |\mathbf{D}|^k \cdot |D|)$.*

5.3. **Binarisation.** In this section, we explain how one can remove a $|D|$ factor in the complexity of DPLL. This is desirable since this extra factor shows a gap between our approach and the optimal bound. We start by observing that this factor is not an overestimation in the analysis of the algorithm but that it can effectively appear on a concrete example.

Example 5.9. Consider the query $Q := A(x_1) \wedge B(x_2) \wedge \neg R(x_1, x_2)$ (which has already been considered in [ZFOK24] for similar reasons). Consider order x_1, x_2 and take $A(x_1) = [d]$, $B(x_2) = [d]$ and $R(x_1, x_2) = \{(v, v) \mid v \in [d]\}$ (that is $\neg R(x_1, x_2)$ enforces $x_1 \neq x_2$). The size of the database is $\mathcal{O}(d)$, and the signed hyperorder width of Q is 1, but DPLL will produce a quadratic size circuit. Indeed, $\neg R(x_1, x_2)$ prevents DPLL from creating any Cartesian product. Moreover, for $d' \in [d]$, $Q \Downarrow [x_1 \leftarrow d']$ is $\big(B(x_2) \wedge x_1 \neq x_2\big)$. Hence, for $d' \neq d''$, $(Q \Downarrow [x_1 \leftarrow d'], [x_1 \leftarrow d'])$ and $(Q \Downarrow [x_1 \leftarrow d''], [x_1 \leftarrow d''])$ are syntactically distinct since they do not assign the same value to x_1. It means that recursive calls on $x_1 \leftarrow d'$ for $d' \in [d]$ never hit the cache. Hence, the resulting circuit will have size $\mathcal{O}(d^2)$ since the returned circuit is essentially a tree with one path for each solution of the form $\{x_1 \leftarrow d_1, x_2 \leftarrow d_2\}$ and $d_1 \neq d_2$.

To overcome this inefficiency, we rely on a simple trick that was already used in the design of worst-case optimal join algorithms [CIS24] which consists in encoding the domain

A	x_1
	0
	1
	2

B	x_2
	1
	2
	3

R	x_1	x_2
	0	0
	1	1
	2	2
	3	3

(A) Original database

$\widetilde{A}^2$	x_1^1	x_1^2
	0	0
	1	0
	0	1

$\widetilde{B}^2$	x_2^1	x_2^2
	1	0
	0	1
	1	1

$\widetilde{R}^2$	x_1^1	x_1^2	x_2^1	x_2^2
	0	0	0	0
	1	0	1	0
	0	1	0	1
	1	1	1	1

(B) Binarised database with $b = 2$

FIGURE 9. Binarisation of a database

over the Boolean domain and transforming the query and the database accordingly. It turns out that this transformation can be done in a way that preserves the structure of the query. Moreover, the isomorphism between the answers of the original query and the answers of the transformed one preserves the order. This allows us to get direct access to the answers of the original query from the answers of the transformed one.

In this section we fix a query Q on variables set X, a database $\mathbf{D}$ on domain $D = [d]$ and let $b = \lceil \mathsf{log}(d) \rceil$. For $v \in [d]$, we denote by $\widetilde{v}^b$ the binary encoding of v over b bits and let $\widetilde{v}^b[i]$ be the i^{th} bit of $\widetilde{v}^b$ for every $1 \leqslant i \leqslant b$. For a variable $x \in X$, we introduce fresh variables $x^1, \ldots, x^b$ and let $\widetilde{X}^b = \{x^i \mid x \in X, 1 \leqslant i \leqslant b\}$. For a tuple $\tau \in D^Y$, we let $\widetilde{\tau}^b$ be the tuple in $D^{\widetilde{Y}^b}$ such that for every $y \in Y$ and $1 \leqslant i \leqslant b$, $\widetilde{\tau}^b(y^i) = \widetilde{\tau(y)}^b[i]$. For a relation $R \subseteq D^Y$, we let $\widetilde{R}^b = \{\widetilde{\tau}^b \mid \tau \in R\} \subseteq D^{\widetilde{Y}^b}$. For an atom A (positive or negative) on variables set Y, we let $\widetilde{A}^b$ be the corresponding atom on variables set $\widetilde{Y}^b$.

We let $\widetilde{Q}^b$ be the query on variables set $\widetilde{X}^b$ whose atoms are $\widetilde{A}^b$ for each atom A of Q. Given a database $\mathbf{D}$ for Q, we let $\widetilde{\mathbf{D}}^b$ be a database for $\widetilde{Q}^b$ where every relation symbol R appearing in Q and interpreted by $R^{\mathbf{D}}$ in $\mathbf{D}$ is now interpreted by $\widetilde{R^{\mathbf{D}}}^b$ in $\widetilde{\mathbf{D}}^b$.

Example 5.10. Let $Q := A(x_1), B(x_2), \neg R(x_1, x_2)$ and $D = [3]$. We have $\widetilde{Q}^2$ is

$$\widetilde{A}^2(x_1^1, x_1^2), \widetilde{B}^2(x_2^1, x_2^2), \neg \widetilde{R}^2(x_1^1, x_1^2, x_2^1, x_2^2).$$

The binarised version $\widetilde{\mathbf{D}}^b$ of the database $\mathbf{D}$ for Q given in Fig. 9a is given in Fig. 9b.

Binarisation is interesting since there is a natural isomorphism between $[\![Q]\!]^{\mathbf{D}}$ and $[\![\widetilde{Q}^b]\!]^{\widetilde{\mathbf{D}}^b}$. Indeed, given $\tau \in [\![Q]\!]^{\mathbf{D}}$, it is easy to see that $\widetilde{\tau}^b$ is an answer of $\widetilde{Q}^b$ on $\widetilde{\mathbf{D}}^b$. Similarly given $\tau \in \{0,1\}^{\widetilde{X}}$, we let $\bar{\tau}^b \in D^X$ be the tuple defined as for every $x \in X$, $\bar{\tau}^b = \sum_{i=1}^{b} 2^{i-1} \tau(x^i)$. It is clear that for every $\tau \in [\![\widetilde{Q}^b]\!]^{\widetilde{\mathbf{D}}^b}$, $\bar{\tau}^b \in [\![Q]\!]^{\mathbf{D}}$. Moreover, for an order $\prec$ on X, we denote by $\prec^b$ the order on $\widetilde{X}^b$ defined as $x^i \prec^b y^j$ if and only if $x \prec y$ or $x = y$ and $i > j$. In other words, if $x_1, \ldots, x_n$ is the order on X, the order $\prec^b$ on $\widetilde{X}^b$ is $x_1^b, \ldots, x_1^1, \ldots, x_n^b, \ldots, x_n^1$. The order $\prec^b$ naturally induces an order $\prec^b_{\mathsf{lex}}$ on $\{0,1\}^{\widetilde{X}^b}$. For $\tau, \tau' \in [d]^X$, we have that $\tau \prec_{\mathsf{lex}} \tau'$ if and only if $\widetilde{\tau}^b \prec^b_{\mathsf{lex}} \widetilde{\tau'}^b$. This is because the natural ordering on $[d]$ can be seen as the lexicographical order on its bit representation, starting from the most significant bit of $\tau(x)$ which is $\widetilde{\tau}^b(x^b)$ to the least significant bit of $\tau(x)$ which is $\widetilde{\tau}^b(x^1)$. Hence, we have the following:

Proposition 5.11. *Let Q be a join query on variables set X, $\prec$ an order on X, and $\mathbf{D}$ a database for Q on domain $[d]$ for some $d \in \mathbb{N}$. Let $b = \lceil \mathsf{log}(d) \rceil$. The function $\bar{\cdot}^b$ is an*

isomorphism between $[\![\widetilde{Q}^b]\!]^{\widetilde{\mathbf{D}}^b}$ *and* $[\![Q]\!]^{\mathbf{D}}$ *that can be computed in time* $\mathcal{O}(nb)$ *such that if* τ *is the* k^{th} *answer of* $\widetilde{Q}^b$ *on* $\widetilde{\mathbf{D}}^b$ *for the order* $\prec^b_{\mathsf{lex}}$ *then* $\bar{\tau}^b$ *is the* k^{th} *solution of* Q *for the order* $\prec_{\mathsf{lex}}$.

A consequence of Proposition 5.11 is that if one has direct access for the answers of $\widetilde{Q}^b$ on database $\widetilde{\mathbf{D}}^b$ for the order $\prec_b$ with preprocessing time P and access time A, then we directly get direct access for Q on database $\mathbf{D}$ with preprocessing time P and access time $A + nb$. Indeed, we can perform the preprocessing as for $\widetilde{Q}^b$. To get the k^{th} answer of Q, one computes the k^{th} answer τ of $\widetilde{Q}^b$ and returns $\bar{\tau}^b$.

This is interesting since the analysis of DPLL made in Theorem 5.3 has a linear dependency in the size of the domain, which is bounded by 2 when considering $\widetilde{\mathbf{D}}^b$ instead of $\mathbf{D}$. Since the number of atoms does not change and the number of variables only increases by a logarithmic factor $\mathsf{log}|D|$, running DPLL on $\widetilde{Q}^b$ and $\widetilde{\mathbf{D}}^b$ would offer an improvement on the complexity, as long as the hyperorder width of Q for $\prec$ and of $\widetilde{Q}^b$ for $\prec^b$ are the same, without negatively affecting our ability to solve direct access tasks. This is indeed true and we now show:

Theorem 5.12. *For every query* Q *and* $b \in \mathbb{N}$, $\mathsf{show}(H(\widetilde{Q}^b), \prec^b) = \mathsf{show}(H(Q), \prec)$ *and* $\mathsf{sfhow}(H(\widetilde{Q}^b), \prec^b) = \mathsf{sfhow}(H(Q), \prec)$.

The rest of this section is dedicated to the proof of Theorem 5.12. It is based on the following notion: given a signed hypergraph $H = (V, E_+, E_-)$ and a vertex $u \in V$, we define the signed hypergraph $\mathsf{clone}(H, u)$ to be the hypergraph obtained by *cloning* u *in* H, that is, by adding a new vertex u' in every edge where u appears. More formally, the vertices of $\mathsf{clone}(H, u)$ are $V \cup \{u'\}$ where u' is a fresh vertex not in V and the positive (resp. negative) edges of $\mathsf{clone}(H, u)$ are $\{e \cup \{u'\} \mid e \in E_+, u \in e\} \cup \{e \in E_+ \mid u \notin e\}$ (resp. $\{e \cup \{u'\} \mid e \in E_-, u \in e\} \cup \{e \in E_- \mid u \notin e\}$). We will use $\mathsf{clone}(e, u)$ as a shorthand for $e \cup \{u'\}$ if $u \in e$ and for e otherwise.

It is easy to see that $H(\widetilde{Q}^b)$ can be obtained from $H(Q)$ by iteratively cloning $b - 1$ times each variable x of Q. Proving Theorem 5.12 boils down to proving that cloning a vertex u does not change the width of a hypergraph as long as the copy u' of u is removed right after u. In other words, we will show that $\mathsf{show}(H, \prec) = \mathsf{show}(H', \prec_u)$ and $\mathsf{sfhow}(H, \prec) = \mathsf{sfhow}(H', \prec_u)$ where $H' = \mathsf{clone}(H, u)$ and $\prec_u$ is the order obtained from $\prec$ by inserting u' after u. The first thing needed for the proof is to observe that cloning and removing vertices commute in the following way:

Lemma 5.13. *Let* $H = (V, E)$ *be a hypergraph and* $u \in V$. *For every* $v \neq u$, $\mathsf{clone}(H/v, u) = \mathsf{clone}(H, u)/v$.

Proof. Since $u \neq v$, $\mathsf{clone}(H/v, u)$ and $\mathsf{clone}(H, u)/v$ have the same vertex set. It is then enough to show that they have the same edges. Let f be an edge of $\mathsf{clone}(H/v, u)$. By definition, either $f = \mathsf{clone}(e \setminus \{v\}, u)$ for e an edge of H or $f = \mathsf{clone}(N^*_H(v), u)$. In the first case, $f = \mathsf{clone}(e, u) \setminus \{v\}$ since $u \neq v$ and so it is an edge of $\mathsf{clone}(H, u)/v$. In the second case, we claim that $f = N^*_{\mathsf{clone}(H,u)}(v)$, and hence f is also an edge of $\mathsf{clone}(H, u)/v$.

Indeed, we first show $f \subseteq N^*_{\mathsf{clone}(H,u)}(v)$. Let $w \in f$ and u' be the vertex added in $\mathsf{clone}(H, u)$. If $w \neq u'$, it means that $w \in N^*_H(v)$. Hence there is an edge $e' \in E$ such that $\{w, v\} \subseteq e'$. In this case, w is in $\mathsf{clone}(e', u)$. In particular, $w \in N^*_{\mathsf{clone}(H,u)}(v)$. Now if $w = u'$, then it can only happen if $u \in f$, that is, $u \in N^*_H(v)$. In this case, there is $e' \in E$ such that $\{u, v\} \subseteq e'$ and in particular $\{v, u'\} \subseteq \mathsf{clone}(e', u)$. It means that $u' \in N^*_{\mathsf{clone}(H,u)}(v)$.

We now show $N^*_{\mathsf{clone}(H,u)}(v) \subseteq f$. Let $w \in N^*_{\mathsf{clone}(H,u)}(v)$ and u' be the vertex added to $\mathsf{clone}(H,u)$. As before, if $w \neq u'$, then $w \in N^*_H(v)$ and $w \in \mathsf{clone}(N^*_H(v),u)$. If $w = u'$, then u must be in $N^*_{\mathsf{clone}(H,u)}(v)$, but then it means it is in $N^*_H(v)$ too. Hence $u' \in \mathsf{clone}(N^*_H(v),u)$.

It remains to show that every edge f of $\mathsf{clone}(H,u)/v$ is also an edge of $\mathsf{clone}(H/v,u)$. This is completely symmetrical to the proof above. Indeed, either $f = \mathsf{clone}(e,u) \setminus \{v\}$ for some e and since $u \neq v$, we have $f = \mathsf{clone}(e \setminus \{v\},u)$ and it proves that f is an edge of $\mathsf{clone}(H/v,u)$, or we have $f = N^*_{\mathsf{clone}(H,u)}(v)$ but we just proved that $f = \mathsf{clone}(N^*_H(v),u)$ which is an edge of $\mathsf{clone}(H/v,u)$. □

We now address the missing case from Lemma 5.13 where $v = u$:

Lemma 5.14. *Let $H = (V,E)$ be a hypergraph and $u \in V$. Let $H' = \mathsf{clone}(H,u)/u$. Then $H/u = H' \setminus \{u'\}$ and $N^*_{H'}(u') = N^*_H(u)$.*

Proof. Let f be an edge of H'. We show that $f \setminus \{u'\}$ is an edge of H/u. By definition, f is either of the form $\mathsf{clone}(e,u) \setminus \{u\}$ for e an edge of H or $N^*_{\mathsf{clone}(H,u)}(u)$. In the first case, $f \setminus \{u'\} = e \setminus \{u\}$ which is an edge of H/u. In the second case, we clearly have $f = N^*_{\mathsf{clone}(H,u)}(u) = N^*_H(u) \cup \{u'\}$. Hence $f \setminus \{u'\} = N^*_H(u)$ is an edge of H/u.

Similarly, let f be an edge of H/u. We show that either f or $f \cup \{u'\}$ is an edge of H'. By definition, f is either equal to $e \setminus \{u\}$ for some edge e of H or $f = N^*_H(u)$. In the first case, there are two subcases depending on whether $u \notin e$ or $u \in e$. Assume $u \notin e$ first. Then $f = e$ is also an edge of $\mathsf{clone}(H,u)$, and hence of $\mathsf{clone}(H,u)/u = H'$. Now if $u \in e$, then $\mathsf{clone}(e,u) = e \cup \{u'\}$ is an edge of $\mathsf{clone}(H,u)$ and then $(e \setminus \{u\}) \cup \{u'\} = f \cup \{u'\}$ is an edge of H'.

In the second case, we have $f = N^*_H(u)$. But then, as before $N^*_{\mathsf{clone}(H,u)}(u) = f \cup \{u'\}$. Hence $f \cup \{u'\}$ is an edge of $\mathsf{clone}(H,u)/u = H'$.

From this analysis, observe that u' in H' only appears in $N^*_{\mathsf{clone}(H,u)}(u)$ or in edge f of the form $(e \setminus \{u\}) \cup \{u'\}$ where e is an edge of H such that $u \in e$. Hence, $N^*_{H'}(u') = N^*_H(u)$. □

We are now ready to prove the following, which implies Theorem 5.12:

Lemma 5.15. *Let $H = (V,E)$ be a hypergraph, $\prec$ an order on V and $u \in V$. Let u' be the clone of u in $H' = \mathsf{clone}(H,u)$ and $\prec_u$ be the order on $V \cup \{u'\}$ where u' is put right after u. Then $\mathsf{how}(H,\prec) = \mathsf{how}(H',\prec_u)$ and $\mathsf{fhow}(H,\prec) = \mathsf{fhow}(H',\prec_u)$.*

Proof. We write V as $u_1,\dots,u_n$, with $u_1 \prec \cdots \prec u_n$. Assume $u = u_j$ and let $H_0 = H$ and $H_i = H/u_1/\dots/u_i$. Moreover, let $H'_0 = H'$ and for $i < j$, let $H'_i = H'/u_1/\dots/u_i$. For $i \geqslant j$, let $H'_i = H'/u_1/\dots/u_j/u'/u_{j+1}/\dots/u_i$. By Lemma 5.13, for $i < j$, $H'_i = \mathsf{clone}(H_i,u_j)$. Hence, it directly follows that $N_{H_i}(u_{i+1}) = N_{H'_i}(u_{i+1}) \setminus \{u'\}$ and that $\rho(N_{H_i}(u_{i+1}),H) = \rho(N_{H'_i}(u_{i+1}),H')$ and $\rho^*(N_{H_i}(u_{i+1}),H) = \rho^*(N_{H'_i}(u_{i+1}),H')$.

Up to $i = j$, removing u_i from H_i or H'_i results in the same effect on the width. Now, consider the case where u' is removed from H'_j. By Lemma 5.14, the neighbourhood of u' in H'_j is the same as the neighbourhood of u_j in H_{j-1}. Thus it can be covered with the same (fractional) number of edges and hence removing u' from H'_j does not increase the width of the order.

Finally, we observe that by Lemma 5.14 again, $H'_j/u' = H_j$. Indeed, since $N^*_{H'_j}(u') = N^*_{H_{j-1}}(u_j)$, H'_j/u' does not introduce any new edge and then $H'_j/u' = H'_j \setminus \{u'\}$ which is equal to $H_{j-1}/u_j = H_j$ by Lemma 5.14. Hence by a direct induction, $H'_i = H_i$ for $i > j$. Hence both elimination orders have the same width. □

It only remains to generalise Lemma 5.15 to signed hyperorder width:

Lemma 5.16. *Let $H = (V, E_+ \cup E_-)$ be a signed hypergraph, $\prec$ an order on V and $u \in V$. Let u' be the clone of u in $H' = \mathsf{clone}(H, u)$ and $\prec_u$ be the order on $V \cup \{u'\}$ where u' is put right after u. Then $\mathsf{show}(H, \prec) = \mathsf{show}(H', \prec_u)$ and $\mathsf{sfhow}(H, \prec) = \mathsf{sfhow}(H', \prec_u)$.*

Proof. Let $H_0 \subseteq^- H$ be a negative subhypergraph of H and let $H'_0 = \mathsf{clone}(H_0, u)$. It is easy to see that H'_0 is a negative subhypergraph of H'. By Lemma 5.15, $\mathsf{how}(H_0, \prec) = \mathsf{how}(H'_0, \prec_u)$ and $\mathsf{fhow}(H_0, \prec) = \mathsf{fhow}(H'_0, \prec_u)$. Hence $\mathsf{show}(H, \prec) \leqslant \mathsf{show}(H', \prec_u)$ and $\mathsf{sfhow}(H, \prec) \leqslant \mathsf{sfhow}(H', \prec_u)$. Similarly, for a negative subhypergraph H'_0 of H', there is a negative subhypergraph H_0 of H such that $H'_0 = \mathsf{clone}(H_0, u)$. We hence similarly get $\mathsf{show}(H, \prec) \geqslant \mathsf{show}(H', \prec_u)$ and $\mathsf{sfhow}(H, \prec) \geqslant \mathsf{sfhow}(H', \prec_u)$. □

Theorem 5.12 is a direct consequence of Lemma 5.16 since $H(\widetilde{Q}^b)$ is obtained by cloning each vertex of $H(Q)$ b times and the copies x^i of vertex x are consecutive in $\prec^b$.

Remark 5.17. Consider again the instance $Q = A(x_1) \wedge B(x_2) \wedge (x_1 \neq x_2)$ from Example 5.9 where we proved that caching would never occur with this instance, resulting in a circuit of size at least $\tilde{\mathcal{O}}(d^2)$. However, when running DPLL on $\widetilde{Q}^b$, we can see that caching will now often happen. Indeed, assume that the algorithm is in a state where every copy $x_1^b, \ldots, x_1^1$ of x_1 was set already to values $v_b, \ldots, v_1$ and the algorithm is now setting variables $x_2^b, \ldots, x_2^1$. Observe that as soon as some copy x_2^i is set to the value $1 - v_i$, then $\neg\widetilde{R}^b$ is satisfied since the value assigned to x_1 is necessarely different from the value assigned to x_2. Hence, the atom $\neg\widetilde{R}^b$ is now simplified away and caching can occur. Each subset of $\widetilde{B}^b$ where some prefix of $x_2^b, \ldots, x_2^1$ will be cached, which boils down to $d \cdot \log d$ values, less than the $\tilde{\mathcal{O}}(d^2)$ from Example 5.9.

We conclude this section by stating how one can use binarisation as a means of improving the construction of ordered $\{\times, \mathsf{dec}\}$-circuits representing answers of signed conjunctive queries.

Theorem 5.18. *Let Q be a signed join query, $\mathbf{D}$ a database over domain D and $\prec$ an order on $\mathsf{var}(Q)$. Then $\mathsf{DPLL}(\widetilde{Q}^b, \langle\rangle, \widetilde{\mathbf{D}}^b, \prec_b)$ produces a $\succ_b$-ordered $\{\times, \mathsf{dec}\}$-circuit C of size $\tilde{\mathcal{O}}((\mathrm{poly}_k |Q|) \cdot |\mathbf{D}|^k)$ on domain $\{0, 1\}$ such that $\mathsf{rel}(C) = [\![\widetilde{Q}^b]\!]^{\mathbf{D}}$ and:*

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

Moreover, the runtime of this construction is at most $\tilde{\mathcal{O}}(\mathrm{poly}_k(|Q|) \cdot |\mathbf{D}|^k)$.

Proof. This is simply Theorem 5.3 applied to $\widetilde{Q}^b$ with $|D| = 2$ together with the fact that $\mathsf{sfhow}(H(\widetilde{Q}^b), \prec_b) = \mathsf{sfhow}(H(Q), \prec)$ and $\mathsf{fhow}(H(\widetilde{Q}^b), \prec_b) = \mathsf{fhow}(H(Q), \prec)$ from Theorem 5.12. □

6. Tractability results for join and conjunctive queries

In this section, we connect the tractability result on direct access on ordered circuit of Section 4 with the algorithm presented in Section 5 to obtain tractability results concerning the complexity of direct access on signed join queries. Moreover, we show how our approach can be used to get direct access for signed conjunctive queries, that is, signed join queries with projected variables.

Theorem 6.1. *Given a signed join query Q, an order $\prec$ on $\mathsf{var}(Q)$ and a database $\mathbf{D}$ on domain D, we can solve the direct access problem for $\prec_{\mathsf{lex}}$ with preprocessing time $\tilde{\mathcal{O}}(|\mathbf{D}|^k \operatorname{poly}_k(|Q|))$ and access time $\mathcal{O}(\operatorname{poly}(|Q|) \cdot (\mathsf{log}|D|)^3(\mathsf{loglog}|D|))$ and:*

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

Proof. We construct a $\prec_b$ ordered-circuit computing $[\![\widetilde{Q}^b]\!]^{\mathbf{D}}$ using Theorem 5.18 for $b = \lceil \mathsf{log}|D| \rceil$ on domain of size 2 and variables set $\widetilde{X}^b = \mathsf{var}(\widetilde{Q}^b)$. We preprocess this circuit as in Theorem 4.2. Constructing the circuit and preprocessing it constitutes the preprocessing phase of our direct access algorithm and can be executed in $\tilde{\mathcal{O}}(|\mathbf{D}|^k \operatorname{poly}_k(|Q|))$ by Theorem 5.18.

Now, to find the i^{th} solution in $[\![Q]\!]^{\mathbf{D}}$, we simply find the i^{th} solution of $\mathsf{rel}(C)$ using the algorithm of Section 4.2, which is the i^{th} solution of $[\![\widetilde{Q}^b]\!]^{\widetilde{\mathbf{D}}^b}$ and by Proposition 5.11, we can reconstruct from it the i^{th} solution of $[\![Q]\!]^{\mathbf{D}}$ in time $\mathcal{O}(\mathsf{log}|D|)$. By Theorem 4.2, the access time is hence $\mathcal{O}(\operatorname{poly}(|Q|) \cdot (\mathsf{log}|D|)^3(\mathsf{loglog}|D|))$ because the number of variables of C is $\mathcal{O}(|\mathsf{var}(Q)|\mathsf{log}|D|)$. □

The main difference between Theorem 6.1 and Theorem 3.10 is that the complexity given in Theorem 6.1 is polynomial in $|Q|$ for bounded signed hyperorder width or for bounded fractional hypertree width, which matches **??** 3.6 in this case. If we are now interested in signed *fractional* hyperorder width, then we are not able to prove that DPLL produces a circuit that is polynomial in $|Q|$. However, by Theorem 5.8, we know that DPLL on the binarisation of Q produces a circuit of size $\tilde{\mathcal{O}}(2^m n|\mathbf{D}|^k)$, which then matches the complexity given in **??** 3.6 for preprocessing but has a better access time since the degree of $\mathsf{log}|D|$ does not depend on $|Q|$ anymore. Formally, we get this improved version of the upper bound of Theorem 3.10. In particular, observe that even if preprocessing is exponential in the query size, this is not the case for the access time:

Theorem 6.2. *Given a signed join query Q, an order $\prec$ on $\mathsf{var}(Q)$ and a database $\mathbf{D}$ on domain D, we can solve the direct access problem for $\prec_{\mathsf{lex}}$ with preprocessing time $\tilde{\mathcal{O}}(|\mathbf{D}|^{\mathsf{sfhow}(H(Q),\succ)} 2^{|\mathsf{atoms}(Q)|} \operatorname{poly}(|Q|))$ and access time $\mathcal{O}(\operatorname{poly}(|Q|) \cdot (\mathsf{log}|D|)^3(\mathsf{loglog}|D|))$.*

Direct access for conjunctive queries. As mentioned, Theorem 6.1 allows to recover the tractability of direct access for positive join queries with bounded fractional hypertree width proven in [CTG+23, BCM22a] and restated here in **??** 3.6. However, [CTG+23] (see also [BCM22b, Theorem 39] which is the arXiv version of [BCM22a]) also generalises the algorithm to conjunctive queries, that is, join queries with projections. Here, we demonstrate the versatility of the circuit-based approach by showing how one can also handle quantifiers directly on the circuit:

Theorem 6.3. *Let C be a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit on domain D, variables set $X = \{x_1, \ldots, x_n\}$ such that $x_1 \prec \cdots \prec x_n$ and $j \leqslant n$. One can compute in time $\mathcal{O}(|C| \cdot \operatorname{poly}(n) \cdot \operatorname{polylog}(|D|))$ a $\prec$-ordered $\{\times, \mathsf{dec}\}$-circuit C' of size at most $|C|$ such that $\mathsf{rel}(C') = \mathsf{rel}(C)|_{\{x_1,\ldots,x_j\}}$.*

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 $y \in \{x_{k+1}, \ldots, x_n\}$. Hence $\mathsf{rel}(v) \subseteq D^Y$ with $Y \subseteq \{x_k, \ldots, x_n\}$. Moreover, one can compute $|\mathsf{rel}(v)|$ for every gate v of C in time

$\mathcal{O}(|C| \cdot \text{poly}(n) \cdot \text{polylog}(|D|))$ as explained in Lemma 4.5. Hence, after this preprocessing, we can decide whether $\mathsf{rel}(v)$ is the empty relation in time $\mathcal{O}(1)$ by simply checking whether $|\mathsf{rel}(v)| = \mathsf{nrel}_C(v, d_0) \neq 0$ 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 $\mathcal{O}(|C| \cdot \text{poly}(n) \cdot \text{polylog}(|D|))$. 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 6.3 to handle conjunctive queries by first using Theorem 5.3 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 6.4. *Given a signed conjunctive query $Q(Y)$, a free-connex order $\prec$ on $\mathsf{var}(Q)$ and a database $\mathbf{D}$ on domain D, we can solve the direct access problem for $\prec_{\mathsf{lex}}$ with preprocessing time $\tilde{\mathcal{O}}(|\mathbf{D}|^k \,\text{poly}_k(|Q|))$ and access time $\mathcal{O}(\text{poly}(|Q|) \cdot (\mathsf{log}|D|)^3 \mathsf{loglog}|D|)$ 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 $\mathsf{DPLL}(\widetilde{Q}^b, \langle\rangle, \widetilde{\mathbf{D}}^b, \prec^b)$ for $b = \lceil \mathsf{log}|D| \rceil$, one obtains a $\succ^b$-ordered circuit computing $[\![\widetilde{Q}^b]\!]^{\widetilde{\mathbf{D}}^b}$. The size of the circuit is $\tilde{\mathcal{O}}(|\mathbf{D}|^k \,\text{poly}_k(|Q|))$ by Theorem 5.3. Now, $\prec^b$ is free-connex, hence $\succ^b$ is of the form $z_1^b \succ \cdots \succ z_1^1 \succ \cdots \succ z_n^b \succ \cdots \succ z_n^1$ and there exists i such that $\{z_1, \dots, z_j\} = \mathsf{free}(Q)$. Hence by Theorem 6.3, we can construct a $\succ^b$-ordered $\{\times, \mathsf{dec}\}$-circuit of size at most $\mathcal{O}(|\mathbf{D}|^k \,\text{poly}_k(|Q|))$ computing $[\![\widetilde{Q}^b]\!]^{\widetilde{\mathbf{D}}^b}|_{\widetilde{\mathsf{free}(Q)}^b}$, which concludes the proof using Theorem 4.2. □

We observe that our notion of free-connex elimination order for Q is akin to [BCM22b, Definition 38] with one main difference: in [BCM22b], 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. Now, Theorem 6.4 constructs a direct access for $\prec_{\mathsf{lex}}$ when $\prec$ is free-connex, so Theorem 6.4 proves the same tractability result as [BCM22b, Theorem 39] with the same complexity.

Observe also that there is another way of recovering Theorem 6.4. We can modify DPLL so that when only projected variables remain, instead of compiling, it calls an efficient join algorithm that can decide whether the answer set of the query is empty or not and returns either $\top$ or $\bot$ in the circuit depending on the answer of this join algorithm. This approach could be more efficient because it is able to exploit more complex measures

[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.

such as the submodular width of the resulting hypergraph, using a join algorithm such as PANDA [AKNS17].

7. Negative join queries and SAT

One particularly interesting application of our result is when it is applied to negative join queries, that is, join queries where every atom is negated. Tractability results for negative join queries have been established previously in the literature. One of the first results in this direction was the study of β-acyclic negative join queries. It was shown in [Bra12] that the negative conjunctive queries where evaluation can be done in linear time in the database are exactly the β-acyclic conjunctive queries. A slight generalisation of this result to acyclic signed join queries can be found in Brault Baron's thesis [BB13]. This result was generalised for counting in [BCM15] and recently to queries with more complex agregation in [ZFOK24]. A generalisation of β-acyclic queries, namely negative join queries with bounded nest set width, was proposed by Lazinger in [Lan23] where evaluation of such queries was shown to be tractable.

Another interesting application of our result is to offer new tractability results for aggregation problems in SAT solving. Indeed, the SAT problem inputs are CNF (Conjunctive Normal Form) formulas, which can be seen as a particular case of negative join queries. Indeed, 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, a database $\mathbf{D}_F$ on domain $\{0,1\}$ of size at most m such that $[\![Q_F]\!]^{\mathbf{D}_F}$ is the set of satisfying assignments of F: 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, any polynomial time algorithm in *combined complexity* on negative join queries directly transfers to CNF formulas, where many tractability results for SAT and #SAT are known when the hypergraph of the input CNF is restricted [OPS13, BCMS15, PSS13, SS13, SS10, STV14] (see [Cap16, Chapter 2] for a survey).

In this section, we show that Theorem 6.1 generalises many of these results because most of the hypergraph families known to give tractability results for negative join queries can be shown to have bounded signed hyperorder width. We now study the notion of signed hyperorder width restricted to negative join queries and compare it to existing measures from literature.

7.1. Hyperorder width for negative join queries.

If a join query is negative, then we can consider its signed hypergraph to be simply the hypergraph induced by its negative atoms. Hence, the notion of signed hyperorder width can be simplified without separating between positive and negative edges. We hence use the following definitions: for a hypergraph $H = (V, E)$ and an order $\prec$ on V, the *β-hyperorder width* $\beta\text{-}\mathsf{how}(H, \prec)$ *of* $\prec$ *for* H is defined as $\max_{H' \subseteq H} \mathsf{how}(H', \prec)$. The *$\beta$-hyperorder width* $\beta\text{-}\mathsf{how}(H)$ *of* H is defined as the width of the best possible elimination order, that is, $\beta\text{-}\mathsf{how}(H) = \min_{\prec} \beta\text{-}\mathsf{how}(H, \prec)$. We define similarly the β-fractional hyperorder width of an order $\prec$ and of a hypergraph – $\beta\text{-}\mathsf{fhow}(H, \prec)$ and $\beta\text{-}\mathsf{fhow}(H)$ – by replacing $\mathsf{how}(\cdot)$ by $\mathsf{fhow}(\cdot)$ in the definitions. Observe that for every negative join query Q, $\iota^s(Q, \succ)$ corresponds to $\beta\text{-}\mathsf{fhow}(H(Q), \prec)$ where $H(Q)$ is seen as a (non signed) hypergraph.

We now turn to comparing the notion for β-hyperorder width with other measures from the literature. The definition of β-hyperorder width can be seen as a hereditary closure of

generalised hypertree width. Indeed, it is a well known fact that hypertree width is not hereditary, that is, a subhypergraph may have a greater hypertree width than the hypergraph itself. Take for example the hypergraph consisting of a triangle on vertices $\{1, 2, 3\}$ together with the hyperedge $\{1, 2, 3\}$. This hypergraph is α-acyclic, and hence has hypertree width 1 while its subhypergraph consisting of the triangle has fractional hypertree width $3/2$ and generalised hypertree width 2.

The fact that fractional hypertree width is not hereditary has traditionally been worked around by taking the largest width over every subhypergraph. This led to the definition of the *β-fractional hypertree width* $\beta\text{-}\mathsf{fhtw}(H)$ of H, defined as $\beta\text{-}\mathsf{fhtw}(H) = \max_{H' \subseteq H} \mathsf{fhtw}(H')$ [GP04]. The β-hypertree width $\beta\text{-}\mathsf{htw}(H)$ is defined similarly by replacing $\mathsf{fhtw}(\cdot)$ by $\mathsf{htw}(\cdot)$. If one plugs the ordered characterisation of $\mathsf{fhtw}(H')$ in this definition, we have the following equivalent way of defining fractional β-hypertree width: $\beta\text{-}\mathsf{fhtw}(H) = \max_{H' \subseteq H} \min_{\prec} \mathsf{fhow}(H', \prec)$. Now recall that $\beta\text{-}\mathsf{fhow}(H) = \min_{\prec} \max_{H' \subseteq H} \mathsf{fhow}(H', \prec)$. Hence, the difference between $\beta\text{-}\mathsf{fhtw}(H)$ and $\beta\text{-}\mathsf{fhow}(H)$ boils down to inverting the min and the max in the definition. It is easy to see then that $\beta\text{-}\mathsf{fhtw}(H) \leqslant \beta\text{-}\mathsf{fhow}(H)$ and $\beta\text{-}\mathsf{htw}(H) \leqslant \beta\text{-}\mathsf{how}(H)$ for every H (see Theorem 7.1 for details). 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. No such decomposition for $\beta\text{-}\mathsf{fhtw}(\cdot)$ that can be used algorithmically has yet been found.

The case where $\beta\text{-}\mathsf{fhtw}(H) = 1$, known as β-acyclicity, is the only one for which tractability results are known, for example SAT [OPS13], #SAT or #CQ for β-acyclic instances [Cap17, BCM15]. This is due to the fact that in this case, an order-based characterisation is known. 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 $E(v)$ can be ordered by inclusion, that is, $E(v) = \{e_1, \dots, e_p\}$ with $e_1 \subseteq \cdots \subseteq e_p$. 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 $H \setminus \{v_1, \dots, v_{i-1}\}$. We show in Theorem 7.1 that β-elimination orders correspond exactly to elimination orders having β-hyperorder width 1, hence proving that our width notion generalises β-acyclicity.

We actually prove a more general result: the notion of β-acyclicity has been recently generalised by Lanzinger in [Lan23] using a notion called nest sets. A set of vertices $S \subseteq V$ is a *nest set of* H if $\{e \setminus S \mid e \in E, e \cap S \neq \emptyset\}$ can be ordered by inclusion. A *nest set elimination order* is a list $\Pi = (S_1, \dots, S_p)$ such that:

- $\bigcup_{i=1}^{p} S_i = V$,
- $S_i \cap S_j = \emptyset$ and
- S_i is a nest set of $H \setminus \bigcup_{j<i} S_j$.

The width of a nest set elimination order is defined as $\mathsf{nsw}(H, \Pi) = \max_i |S_i|$ 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 arbitrarily 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 7.1. *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, H is β-acyclic if and only if $\beta\text{-}\mathsf{how}(H) = 1$.*

The proof mainly follows from the following lemma. Intuitively, it says that if S is a nest set of H of size k, then if we iteratively remove the vertices of S in H, then the edges it

introduces can always be covered by k edges from the original hypergraph. This is because they are made of some vertices from S (at most k of them) and some other vertices that are all covered by the maximal edge covering $\{e \setminus S \mid e \cap S \neq \emptyset\}$.

Lemma 7.2. *Let $H = (V, E)$ be a hypergraph and S be a nest set of H of size k. We let f 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 edge e of $H/s_1/\dots/s_i$, 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 (i) the edges of H_i without the vertex s_{i+1} or (ii) the additional edge $N^*_{H_i}(s_{i+1})$. Let e be an edge of H_{i+1} that is not $N^*_{H_i}(s_{i+1})$. We have two cases: either e is already in H_i, in which case the induction hypothesis still holds. Or e is not in H_i, which means that $e = e' \setminus \{s_{i+1}\}$ for some edge e' of H_i containing s_{i+1}. In particular, $s_{i+1} \in e' \cap S$ and then $e' \cap S \neq \emptyset$. By induction, we have $e' \subseteq f \cup S$ and then $e = e' \setminus \{s_{i+1}\} \subseteq f \cup S$ and the induction hypothesis follows. Finally, assume $e = N^*_{H_i}(s_{i+1})$, that is, e is obtained by taking the union of every e' in H_i where s_{i+1} is in e' and removing s_{i+1}. But then $e' \cap s_{i+1} \neq \emptyset$. By induction again, $e' \subseteq f \cup S$ hence $N^*_{H_i}(s_{i+1}) \subseteq f \cup S$ and the induction hypothesis follows. □

Proof of Theorem 7.1. 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 arbitrarily, 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 [Lan23, Lemma 4][7]. Consequently, it is enough to prove that $\mathsf{how}(H, \prec) \leqslant k$. Indeed, if we prove it, then we know that for every hypergraph $H' \subseteq H$, $\prec$ is also a nest set elimination order for H', and hence, $\mathsf{how}(H', \prec) \leqslant k$ too.

This follows from Lemma 7.2. 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 7.2, 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 sets contain v_{i+1}). Hence N_{i+1} is covered by at most t edges: f – which 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 7.2 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

[7] Lemma 4 of [Lan23] establishes the result for a connected subhypergraph of H but the same proof works for non-connected subhypergraphs.

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 H is β-acyclic if and only if $\beta\text{-}\mathsf{how}(H) = 1$. Indeed, if H is β-acyclic, then $\mathsf{nsw}(H) = 1$ (the definition of nest set width elimination order of width 1 directly corresponds to the definition of β-elimination orders) 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 [Bod93]. This does not rule out the possibility that deciding whether $\beta\text{-}\mathsf{how}(H) \leqslant k$ is tractable for every k as it is the case for treewidth [Bod93], but we observe that deciding whether $\mathsf{fhow}(H) \leqslant 2$ is known to be NP-hard [FGP18]. 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 [Cap16, Chapter 2] 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 7.3. *Does there exist a family $(H_n)_{n\in\mathbb{N}}$ of hypergraphs such that there exists $k \in \mathbb{N}$ such that for every n, $\beta\text{-}\mathsf{fhtw}(H_n) \leqslant k$ while $\beta\text{-}\mathsf{fhow}(H_n)$ 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:

Proposition 7.4. *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 additionnally 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 covers $[n]$. Now consider the subhypergraph H'_n of H_n obtained by removing the edge $[n]$. The hypertree width of T with respect to H'_n is n since one needs the edge $\{i, 0\}$ for every i to cover vertex i in the bag $[n]$ since i appears only in this edge. □

7.2. **Applications.** In the previous section, we have shown that β-hyperorder width generalises β-acyclicity and nest set width. Hence, Theorem 6.1 can be used to show 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 [Lan23]:

Theorem 7.5. *Let Q be a negative join query of bounded nest set width and $\mathbf{D}$ be a database. Then we can compute $|[\![Q]\!]^{\mathbf{D}}|$ in polynomial time.*

Proof. Let k be the nest set width of Q. It is proven in [Lan23, Theorem 15] that a nest set elimination order for Q can be found in time $2^{\mathcal{O}(k^2)}\operatorname{poly}(|H(Q)|)$ which induces an elimination order $\prec$ for $H(Q)$ of β-hyperoder width k. Now, using DPLL on this order and on the binarised form of Q, we can construct a circuit computing $[\![Q]\!]^{\mathbf{D}}$ in time $\tilde{\mathcal{O}}((\operatorname{poly}(H(Q))|\mathbf{D}|)^k)$ and extract $|[\![Q]\!]^{\mathbf{D}}|$ from it using Lemma 4.5 in time $\tilde{\mathcal{O}}((\operatorname{poly}(H(Q))|\mathbf{D}|)^k)$. □

Similarly, our result can directly be applied to #SAT, the problem of counting the number of satisfying assignments of a CNF formula. Hence, Theorem 6.1 generalises both [Cap17] and [BCM15] 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.

Fig. 10 summarises our contributions for join queries with negations and summarises how our contribution is located in the landscape of known tractability results. The two left-most columns of the figure are contributions of this paper (Theorem 6.1), the right-most column is known from [BCM22a] but can be recovered in our framework (see discussion below). A complete presentation of the results stated in this figure can be found in [Cap16, Chapter 2].

8. 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 introduced a framework unifying the positive and the signed case using a factorised representation of the answer set of the query. Our complexity bounds, when restricted to the positive case, match the existing optimal one and we have proven that our algorithm remains optimal for signed join queries without self-join (in terms of data complexity), closing the two main open questions of the conference version of this paper [CI24]. Our approach opens many new avenues of research. In particular, we believe that the circuit representation that we use is promising for answering different kinds of aggregation tasks and hence generalising existing results on conjunctive queries to the case of signed conjunctive queries. For example, we believe that FAQ and AJAR queries [AKNR16, JPR16] could be solved using this data structure. Indeed, it looks possible to annotate the circuit with semi-ring elements and to project them out in a similar fashion as Theorem 6.3. Similarly, we believe that the framework of [ECK23] for solving direct access tasks on conjunctive queries with aggregation operators may be generalised to the class of ordered $\{\times, \mathsf{dec}\}$-circuits. Finally, contrary to the positive case, we do not yet know what is the complexity of solving direct access tasks on signed join queries with self-joins. We know that self-joins can only make things easier and have exhibited an example at the end of Section 3.2 where having a self-join between the positive and the negative part leads to drastic improvement in the preprocessing complexity. We do

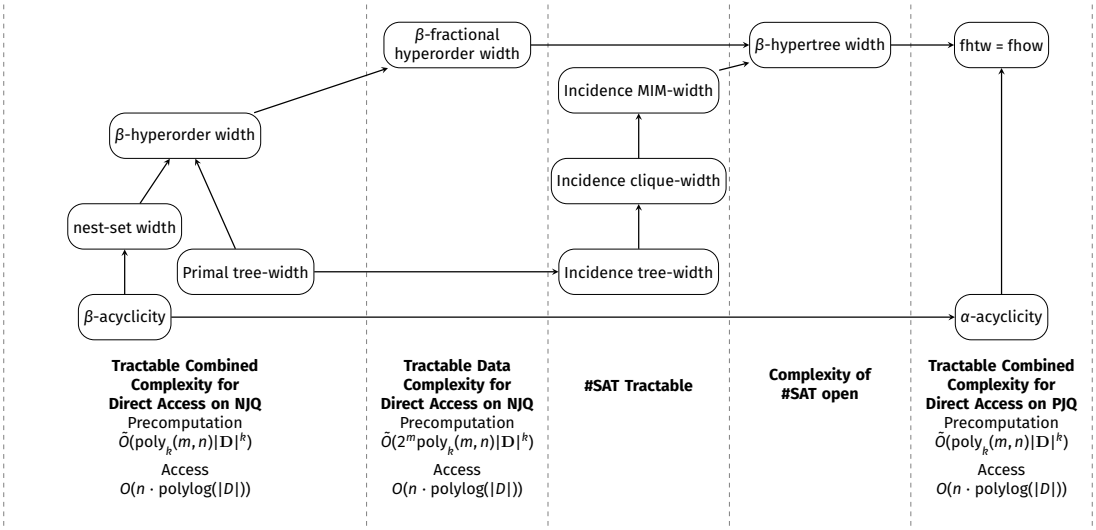


FIGURE 10. Landscape of hypergraph measures and known inclusions with tractability results for direct access on negative join queries (NJQ), direct access on positive join queries (PJQ), and #SAT on CNF formulas. Here, n is the number of variables, m the number of atoms, $\mathbf{D}$ the database, D 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 for a fixed k.
All tractability results for direct access are consequences of Theorem 6.4 applied to either fractional hyperorder width in the positive case or β-hyperorder width in the negative case. The β-fractional hyperorder width is a consequence of Theorem 3.7. Comparison between β-hyperorder width, nest-set width and β-hypertree width is a consequence of Theorem 7.1. Tractability of #SAT for MIM-width is from [STV14], where most inclusions below can be found. The inclusion between MIM-width and β-hypertree width can be found in [Cap16, Theorem 2.33].

not know however whether there are other more subtle cases where self-joins help, as it was shown for enumeration [CS23].

References

[AKNR15] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. Faq: questions asked frequently. *arXiv preprint arXiv:1504.04044*, 2015.

[AKNR16] Mahmoud Abo Khamis, Hung Q Ngo, and Atri Rudra. Faq: questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 13–28, 2016.

[AKNS17] Mahmoud Abo Khamis, Hung Q Ngo, and Dan Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 429–444, 2017.

[Bag09] 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.

[BB13] Johann Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.

[BB16] Johann Brault-Baron. Hypergraph acyclicity revisited. *ACM Computing Surveys (CSUR)*, 49(3):1–26, 2016.

[BCM15] 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.

[BCM22a] 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.

[BCM22b] Karl Bringmann, Nofar Carmeli, and Stefan Mengel. Tight fine-grained bounds for direct access on join queries. *arXiv preprint arXiv:2201.02401*, 2022.

[BCMS15] Simone Bova, Florent Capelli, Stefan Mengel, and Friedrich Slivovsky. On Compiling CNFs into Structured Deterministic DNNFs. In *Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science, pages 199–214. Springer International Publishing, September 2015.

[BDGO08] 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.

[BKOZ13] 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.

[Bod93] 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*, pages 226–234. ACM, 1993.

[Bra12] Johann Brault-Baron. A negative conjunctive query is easy if and only if it is beta-acyclic. In Patrick Cégielski and Arnaud Durand, editors, *Computer Science Logic - 26th International Workshop/21st Annual Conference of the EACSL, September 3-6, 2012, Fontainebleau, France*, volume 16 of *LIPIcs*, pages 137–151. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

[Cap16] 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`.

[Cap17] Florent Capelli. Understanding the complexity of #SAT using knowledge compilation. In *32nd Annual ACM/IEEE Symposium on Logic in Computer Science, 2017, Reykjavik, Iceland, June 20-23, 2017*, pages 1–10. IEEE Computer Society, 2017. `doi:10.1109/LICS.2017.8005121`.

[CI24] Florent Capelli and Oliver Irwin. Direct access for conjunctive queries with negations. In Graham Cormode and Michael Shekelyan, editors, *27th International Conference on Database Theory, March 25-28, 2024, Paestum, Italy*, volume 290 of *LIPIcs*, pages 13:1–13:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024. `doi:10.4230/LIPICS.ICDT.2024.13`.

[CIS24] Florent Capelli, Oliver Irwin, and Sylvain Salvati. A simple algorithm for worst-case optimal join and sampling, 2024. URL: `https://arxiv.org/abs/2409.14094`, `arXiv:2409.14094`.

[CLRS22] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2022.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, pages 77–90, New York, NY, USA, 1977. ACM. doi:10.1145/800105.803397.

[CS23] Nofar Carmeli and Luc Segoufin. Conjunctive queries with self-joins, towards a fine-grained enumeration complexity analysis. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 277–289, 2023.

[CTG+23] 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.

[CZB+20] 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.

[DM02] Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*, 17:229–264, 2002.

[ECK23] Idan Eldar, Nofar Carmeli, and Benny Kimelfeld. Direct access for answers to conjunctive queries with aggregation. *arXiv preprint arXiv:2303.05327*, 2023.

[FGP18] Wolfgang Fischl, Georg Gottlob, and Reinhard Pichler. General and fractional hypertree decompositions: Hard and easy cases. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 17–32. ACM, 2018.

[FHLS18] Johannes K Fichte, Markus Hecher, Neha Lodha, and Stefan Szeider. An smt approach to fractional hypertree width. In *Principles and Practice of Constraint Programming: 24th International Conference, Lille, France, August 27-31, 2018, Proceedings 24*, pages 109–127. Springer, 2018.

[GM14] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms (TALG)*, 11(1):4, 2014.

[GP04] 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.

[GSSS22] Robert Ganian, André Schidler, Manuel Sorge, and Stefan Szeider. Threshold treewidth and hypertree width. *Journal of Artificial Intelligence Research*, 74:1687–1713, 2022.

[HD05] Jinbo Huang and Adnan Darwiche. DPLL with a Trace: From SAT to Knowledge Compilation. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence*, pages 156–162, 2005.

[HVDH21] David Harvey and Joris Van Der Hoeven. Integer multiplication in time o(nlog\,n). *Annals of Mathematics*, 193(2):563–617, 2021.

[JPR16] Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 91–106, 2016.

[Kep20] Jens Keppeler. *Answering Conjunctive Queries and FO+ MOD Queries under Updates*. PhD thesis, Humboldt-Universität zu Berlin, Mathematisch-Naturwissenschaftliche Fakultät, 2020.

[KNOZ25] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Conjunctive queries with free access patterns under updates. *Logical Methods in Computer Science*, 21, 2025.

[Lan23] Matthias Lanzinger. Tractability beyond β-acyclicity for conjunctive queries with negation and sat. *Theoretical Computer Science*, 942:276–296, 2023.

[Olt16] Dan Olteanu. Factorized databases: A knowledge compilation perspective. In *AAAI Workshop: Beyond NP*, 2016.

[OPS13] S. Ordyniak, D. Paulusma, and S. Szeider. Satisfiability of acyclic and almost acyclic CNF formulas. *Theoretical Computer Science*, 481:85–99, 2013.

[OZ12] Dan Olteanu and Jakub Závodnỳ. Factorised representations of query results: size bounds and readability. In *Proceedings of the 15th International Conference on Database Theory*, pages 285–298. ACM, 2012.

[OZ15] Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *ACM Transactions on Database Systems*, 40(1):1–44, March 2015.

[PS13] Reinhard Pichler and Sebastian Skritek. Tractable counting of the answers to conjunctive queries. *Journal of Computer and System Sciences*, 79:984–1001, September 2013.

[PSS13] D. Paulusma, F. Slivovsky, and S. Szeider. Model Counting for CNF Formulas of Bounded Modular Treewidth. In *30th International Symposium on Theoretical Aspects of Computer Science*, pages 55–66, 2013.

[SBB+04] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A Kautz, and Toniann Pitassi. Combining component caching and clause learning for effective model counting. *Theory and Applications of Satisfiability Testing*, 4:7th, 2004.

[SOC16] Maximilian Schleich, Dan Olteanu, and Radu Ciucanu. Learning linear regression models over factorized joins. In *Proceedings of the 2016 International Conference on Management of Data*, pages 3–18. ACM, 2016.

[SS10] M. Samer and S. Szeider. Algorithms for propositional model counting. *Journal of Discrete Algorithms*, 8(1):50–64, 2010.

[SS13] F. Slivovsky and S. Szeider. Model Counting for Formulas of Bounded Clique-Width. In *Algorithms and Computation - 24th International Symposium, ISAAC*, pages 677–687, 2013.

[STV14] S. Hortemo Sæther, J.A. Telle, and M. Vatshelle. Solving MaxSAT and #SAT on structured CNF formulas. In *Theory and Applications of Satisfiability Testing*, pages 16–31, 2014.

[vEB75] Peter van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*, pages 75–84. IEEE, 1975.

[Vel14] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory, Athens, Greece, March 24-28, 2014*, pages 96–106. OpenProceedings.org, 2014. doi:10.5441/002/icdt.2014.13.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, pages 82–94. VLDB Endowment, 1981.

[ZFOK24] Hangdong Zhao, Austen Z. Fan, Xiating Ouyang, and Paraschos Koutris. Conjunctive queries with negation and aggregation: A linear time characterization. *Proc. ACM Manag. Data*, 2(2):75, 2024. doi:10.1145/3651138.

Appendix A. Proof of Lemma 2.1

Lemma 2.1. *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. We now formally prove this claim. 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 k^{th} tuple of R), we have $\#\sigma_{x \leqslant \tau(x)}(R) \geqslant k$.

We now show that, given a value $d' < \tau(x)$, we have that $d' \notin A$ and therefore $\tau(x)$ 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 k^{th} tuple, the latter set has less than $k-1$ elements. Hence $d' \notin A$. 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 $k' = k - \#\sigma_{x<d}(R)$. □