Conjunctive Queries with Negation and Aggregation: A Linear Time Characterization

HANGDONG ZHAO, University of Wisconsin–Madison, USA
AUSTEN Z. FAN, University of Wisconsin–Madison, USA
XIATING OUYANG, University of Wisconsin–Madison, USA
PARASCHOS KOUTRIS, University of Wisconsin–Madison, USA

In this paper, we study the complexity of evaluating Conjunctive Queries with negation ($\mathsf{CQ}^\neg$). First, we present an algorithm with linear preprocessing time and constant delay enumeration for a class of CQs with negation called free-connex signed-acyclic queries. We show that no other queries admit such an algorithm subject to lower-bound conjectures. Second, we extend our algorithm to Conjunctive Queries with negation and aggregation over a general semiring, which we call Functional Aggregate Queries with negation ($\mathsf{FAQ}^\neg$). Such an algorithm achieves constant delay enumeration for the same class of queries but with a slightly increased preprocessing time, which includes an inverse Ackermann function. We show that this surprising appearance of the Ackermmann function is probably unavoidable for general semirings but can be removed when the semiring has a specific structure. Finally, we show an application of our results to computing the difference of CQs.

CCS Concepts: • **Theory of computation → Database query processing and optimization (theory)**; **Data structures and algorithms for data management**.

Additional Key Words and Phrases: Query Optimization, Signed-acyclicity, Negation, Aggregation, Range Queries.

ACM Reference Format:
Hangdong Zhao, Austen Z. Fan, Xiating Ouyang, and Paraschos Koutris. 2024. Conjunctive Queries with Negation and Aggregation: A Linear Time Characterization. *Proc. ACM Manag. Data* 2, 2 (PODS), Article 75 (May 2024), 19 pages. https://doi.org/10.1145/3651138

1 INTRODUCTION

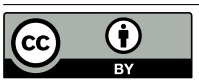
This paper focuses on the query evaluation problem for Conjunctive Queries with negation ($\mathsf{CQ}^\neg$), a fundamental class of relational queries. We will think of a $\mathsf{CQ}^\neg$ as having the following form:

$$Q(\mathbf{x}_F) \leftarrow \bigwedge_{K \in \mathcal{E}^+} R_K(\mathbf{x}_K) \wedge \bigwedge_{K \in \mathcal{E}^-} \neg R_K(\mathbf{x}_K) \tag{1}$$

where the variables are of the form x_i, $i \in [n] = \{1, 2, \ldots, n\}$, $\mathcal{E}^+, \mathcal{E}^-$ are two sets of hyperedges that are subsets of $[n]$, and $F \subseteq [n]$ are the free variables. We will call the triple $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ the *signed hypergraph* of Q. We will consider only *safe queries*, where $\bigcup_{K \in \mathcal{E}^+} K = [n]$. When $\mathcal{E}^- = \emptyset$, then Q is a Conjunctive Query (CQ) with the associated hypergraph $([n], \mathcal{E}^+)$.

The complexity of query evaluation for CQs is well-understood. Yannakakis [24] first showed that a Boolean CQ (i.e., $F = \emptyset$) can be evaluated on a database $\mathcal{D}$ of size $|\mathcal{D}|$ in time $O(|Q| \cdot |\mathcal{D}|)$ if the

Authors' addresses: Hangdong Zhao, University of Wisconsin–Madison, Madison, USA, hangdong@cs.wisc.edu; Austen Z. Fan, University of Wisconsin–Madison, Madison, USA, afan@cs.wisc.edu; Xiating Ouyang, University of Wisconsin–Madison, Madison, USA, xouyang@cs.wisc.edu; Paraschos Koutris, University of Wisconsin–Madison, Madison, USA, paris@cs.wisc.edu.



ACM 2836-6573/2024/5-ART75
https://doi.org/10.1145/3651138

hypergraph $([n], \mathcal{E}^+)$ is α-acyclic ($|Q|$ denotes the size of the query). Further work [3, 4] generalized this result to show that if Q is *free-connex α-acyclic* then the tuples in $Q(\mathcal{D})$ can be enumerated with constant delay $O(|Q|)$ after a linear-time preprocessing step. Free-connex α-acyclicity means that both $([n], \mathcal{E}^+)$ and $([n], \mathcal{E}^+ \cup \{F\})$ are α-acyclic hypergraphs. It was also shown [3] that this tractability result is tight under widely believed lower-bound conjectures.

1.1 CQs with Negation

Our first goal in this paper is to generalize the above classic result to the case where $\mathcal{E}^- \neq \emptyset$. Prior work has looked into this problem, but without achieving a complete answer.

To explain the current progress, let us first consider the case of a Boolean CQ$^\neg$ and attempt to solve the (seemingly harder) problem of counting the number of valuations that satisfy the body of Q with input a database $\mathcal{D}$, which we will denote as $\#Q(\mathcal{D})$. We should note here that $\#Q$ is solvable in $O(|\mathcal{D}|)$ time if Q is an α-acyclic CQ. Brault-Baron [6] had the insight that we can compute $\#Q$ using the inclusion-exclusion principle. Indeed, let Q_S be the Boolean CQ with hypergraph $([n], \mathcal{E}^+ \cup S)$ for any $S \subseteq \mathcal{E}^-$. Then we can write:

$$\#Q(\mathcal{D}) = \sum_{S \subseteq \mathcal{E}^-} (-1)^{|S|} \#Q_S(\mathcal{D}) \tag{2}$$

Hence, if every Q_S is α-acyclic, then $\#Q$ (and thus Q) can be computed with data complexity $O(|\mathcal{D}|)$. This naturally leads to the notion of *signed acyclicity*, introduced in [6]: a Boolean $CQ^\neg$ is signed-acyclic if the hypergraph $([n], \mathcal{E}^+ \cup S)$ is α-acyclic for every $S \subseteq \mathcal{E}^-$. Thus, a Boolean CQ$^\neg$ can be computed in linear time (data complexity) if it is signed-acyclic. Interestingly, if $\mathcal{E}^+$ consists only of singleton hyperedges, then signed acyclicity is equivalent to β-acyclicity of the hypergraph $([n], \mathcal{E}^-)$. However, there are two issues with applying the inclusion-exclusion approach. First, it has an exponential dependency on $|Q|$ and thus does not give a polynomial-time algorithm in combined complexity. Second, it cannot be used to provide any delay guarantees for the enumeration problem in non-Boolean queries.

The second issue was partially addressed by Brault-Baron [5, 6], who proposed an enumeration algorithm for any free-connex signed-acyclic CQ$^\neg$. However, this algorithm either achieves constant delay with a $O(|\mathcal{D}| \log^{|Q|} |\mathcal{D}|)$ preprocessing time, or achieves logarithmic delay with linear preprocessing time. The logarithmic factor is a consequence of the technique used, which translates a database instance to an instance over the Boolean domain.

Our first main result shows that the translation to the Boolean domain is not necessary and in fact we can achieve both constant delay and linear time preprocessing for free-connex signed-acyclic queries. Moreover, our algorithm has only a polynomial dependence on the size of the query.

Theorem 1.1. *Let Q be a free-connex signed-acyclic* CQ$^\neg$. *Then there is an algorithm that can enumerate the results of $Q(\mathcal{D})$ with $O(|Q|^3 + |Q| \cdot |\mathcal{D}|)$ preprocessing time and $O(|Q|)$ delay.*

1.2 FAQ with Negation

Our second goal is to study the evaluation of CQ$^\neg$ in the presence of aggregation. We do this by studying a more general problem, that of computing a CQ$^\neg$ under a general semiring, following the approach of FAQs [19]. More precisely, given a commutative semiring $\mathbf{S} = (\boldsymbol{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$, we define an FAQ$^\neg$ as an expression of the form:

$$\varphi(\mathbf{x}_F) = \bigoplus_{\mathbf{x}_{[n]\setminus F}} \bigotimes_{K \in \mathcal{E}^+} R_K(\mathbf{x}_K) \otimes \bigotimes_{N \in \mathcal{E}^-} \overline{R}_N(\mathbf{x}_N), \tag{3}$$

Here, a *positive factor* R_K can be viewed as a table of entries of the form $\langle \mathbf{x}_K, R_K(\mathbf{x}_K) \rangle$ (where the weight of tuple $\mathbf{x}_K$ is the value $R_K(\mathbf{x}_K) \in \boldsymbol{D}$), and for entries not in the table, the weight is implicitly

$\mathbf{0}$. On the other hand, a *negative factor* $\overline{R}_N$ is a table of entries of the form $\langle \mathbf{x}_K, R_N(\mathbf{x}_N)\rangle$, and for entries not in the table, the weight is a default constant value $\mathbf{c}_N \neq \mathbf{0}$. To recover the CQ$^\neg$ setting, we choose the Boolean semiring, and encode the values of the negative factor such that it is $\mathbf{0}$ if the tuple is in the table, otherwise $\mathbf{1}$. Our semiring formulation is though much more general: the only difference between a negative and a positive factor in our setting is whether the "default" value of a tuple not in the table is $\mathbf{0}$. This observation offers a novel angle to the semantics of negation, and is critical to our algorithms. For FAQ$^\neg$, we can show the following.

Theorem 1.2. *Let φ be a free-connex signed-acyclic* FAQ$^\neg$ *over a commutative semiring* S. *Then there is an algorithm that can enumerate φ with $O(|\varphi|^3 + |\varphi| \cdot |\mathcal{D}| \cdot \alpha(14 \cdot |\mathcal{D}|, |\mathcal{D}|))$ preprocessing time and $O(|\varphi|)$ delay.*

Here, $|\varphi|$ denotes the size of the query φ and $\alpha(m, n)$ denotes the inverse Ackermann function, which grows extremely slowly as a bi-variate function of m, n. The appearance of this function in the runtime expression is surprising in our opinion. It occurs because the aggregation problem reduces to the well-studied problem of computing interval sums over an arbitrary semiring, called RangeSum [12, 25]. If the semiring structure allows for linear preprocessing and constant-time answering for its RangeSum problem, then we can drop the Ackermann factor. Examples of such semirings are the Boolean semiring $(\{\text{true}, \text{false}\}, \vee, \wedge, \text{false}, \text{true})$, tropical semiring $(\mathbb{R}, \min, +, +\infty, 0)$, and semirings with additive inverse (e.g. the counting ring over integers, i.e. $(\mathbb{Z}, +, \times, 0, 1)$ which we use to count solutions).

Theorem 1.3. *Let φ be a free-connex signed-acyclic* FAQ$^\neg$ *over a commutative semiring* S *with additive inverse. Then there is an algorithm that can enumerate φ with $O(|\varphi|^3 + |\varphi| \cdot |\mathcal{D}|)$ preprocessing time and $O(|\varphi|)$ delay.*

1.3 Lower Bounds

Our third goal is to match our linear-time upper bounds with lower bounds. In this direction, we show that under believable conjectures, any CQ$^\neg$ that is not free-connex signed-acyclic does not admit an algorithm that can emit the first result in linear time (hence matching the upper bound). Our conditional lower bounds are somewhat weaker than the ones obtained for CQs because the presence of negation means that we cannot use the sparse version of some problems (e.g., detecting a triangle, or Boolean matrix multiplication). For FAQ$^\neg$, we show stronger conditional lower bounds over the tropical semiring and counting ring based on weaker lower bound conjectures. Finally, we provide some evidence that the inverse Ackermann factor in the runtime for general semirings is unavoidable. In particular, we show that the query $\varphi(x) = \bigoplus_y A(x) \otimes B(y) \otimes \overline{R}(x, y)$ corresponds to a variant of the offline RangeSum problem over a general $\oplus$ operator. Using this observation, we can modify a construction of Chazelle [12] to show a superlinear lower bound on the number of $\oplus$ operations necessary to compute φ.

1.4 Difference of CQs

Finally, we show that our algorithm for CQ$^\neg$ can be applied to obtain optimal algorithms with linear-time preprocessing and constant delay for the problem of computing the difference between two CQs of the form $Q_1 - Q_2$, which was recently studied by Hu and Wang [17].

2 PRELIMINARIES

Hypergraphs. A hypergraph is a pair $\mathcal{H} = ([n], \mathcal{E})$ where $[n] = \{1, \ldots, n\}$ is the set of vertices of $\mathcal{H}$ and $\mathcal{E}$ is a multiset[1] of hyperedges where each $K \in \mathcal{E}$ is a nonempty subset of $[n]$.

[1]A multiset is a collection of elements each of which can occur multiple times

A signed hypergraph is a tuple $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ where $[n]$ is the set of vertices of $\mathcal{H}$, $\mathcal{E}^+$ and $\mathcal{E}^-$ are two multisets of hyperedges where each hyperedge $K \in \mathcal{E}^+$ (resp. $N \in \mathcal{E}^-$) is a subset of $[n]$. We consider only *safe* signed hypergraphs, where every vertex in $[n]$ occurs in some hyperedge $K \in \mathcal{E}^+$.

Semigroups and semirings. A triple $\mathbf{S} = (\boldsymbol{D}, \oplus)$ is a semigroup if $\oplus$ is an associative binary operator over $\boldsymbol{D}$. A tuple $\mathbf{S} = (\boldsymbol{D}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$ is a (commutative) semiring if $\oplus$ and $\otimes$ are commutative binary operators over $\boldsymbol{D}$ satisfying the following conditions: (1) $(\boldsymbol{D}, \oplus)$ is a commutative monoid[2]with an additive identity, denoted by $\mathbf{0}$; (2) $(\boldsymbol{D}, \otimes)$ is a commutative monoid with a multiplicative identity, denoted by $\mathbf{1}$; (3) $\otimes$ distributes over $\oplus$; and (4) For any element $e \in \boldsymbol{D}$, we have $e \otimes \mathbf{0} = \mathbf{0} \otimes e = \mathbf{0}$.

Enumeration and Complexity. In this paper, we study the enumeration problem for $\mathsf{FAQ}^\neg$, $\mathsf{Enum}(\varphi, \mathcal{D})$, which takes as input a $\mathsf{FAQ}^\neg$ φ and a database instance $\mathcal{D}$ and outputs a sequence of answers such that every *answer* in $\varphi(\mathcal{D})$ is printed precisely once. An enumeration algorithm for $\mathsf{Enum}(\varphi, \mathcal{D})$ may consist of two phases:

- (preprocessing phase) it constructs efficient data structures from φ and $\mathcal{D}$; and
- (enumeration phase) it may access the data structures built during preprocessing, and emit the answers of $\varphi(\mathcal{D})$ one by one, without repetitions.

We say that an enumeration algorithm enumerates with delay $O(\tau)$ if the time between the emission of any two consecutive answers (and the time to emit the first answer, and the time from the last answer to the end) is bounded by $O(\tau)$. In particular, we say that an enumeration algorithm is *constant-delay* if it enumerates with delay independent of the input database size $|\mathcal{D}|$.

Model of computation. We adopt the *random-access machine (RAM)* as our computation model with $O(\log n)$-bit words, which is standard in fine-grained complexity. The machine has read-only input registers and it contains the database and the query, read-write work memory registers, and write-only output registers. It is assumed that each register can store any tuple, and each tuple is stored in one register. The machine can perform all "standard" operations [3].

In this paper, we consider the *combined complexity* of $\mathsf{CQ}^\neg$ and $\mathsf{FAQ}^\neg$, i.e. the complexity is measured in the size of both the query and the database instance. For a $\mathsf{FAQ}^\neg$ φ, we define $|\varphi|$ to be the sum of the arity of all factors in φ. For a database $\mathcal{D}$, we define $|\mathcal{D}|$ as sum of the size of all relations in $\mathcal{D}$.

Ackermann function. The Ackermann function $A(m, n)$ for integers $m, n \geq 0$ is recursively defined as follows:

(1) $A(0, n) = n + 1$;
(2) $A(m + 1, 0) = A(m, 1)$; and
(3) $A(m + 1, n + 1) = A(m, A(m + 1, n))$.

It is known that the Ackermann function grows faster than any primitive recursive function and therefore is not itself primitive recursive. The inverse Ackermann function $\alpha(m, n)$ is defined as

$$\alpha(m, n) = \min\{i \geq 1 : A(i, \lfloor \frac{m}{n} \rfloor) \geq \log_2(n)\}$$

and $\alpha(m, n)$ grows very slowly. For example, we have $\alpha(n, n) < 5$ for any practical integer n.

[2]In the usual semiring definition, we do not need the multiplicative monoid to be commutative.
[3]This includes all arithmetic (e.g. $+, -, \div, *$) and logical operations (see [15, 16, 22]).

3 SIGNED ACYCLICITY

Before we introduce the definition of signed acyclicity, we first go over the notions of α- and β-acyclicity.

A hypergraph $\mathcal{H} = ([n], \mathcal{E})$ is α-acyclic if there is a tree $\mathcal{T} = (V(\mathcal{T}), E(\mathcal{T}))$ and a bijective function $\chi : \mathcal{E} \to V(\mathcal{T})$ such that for every vertex $v \in [n]$, the set of nodes $\{\chi(K) \mid v \in K, K \in \mathcal{E}\}$ induces a connected component in $\mathcal{T}$. A vertex $v \in [n]$ is an α-leaf of $\mathcal{H}$ if the multiset $\{K \in \mathcal{E} \mid v \in K\}$ contains a maximal element K_v with respect to $\subseteq$. It is known that every α-acyclic hypergraph has an α-leaf [7].

A hypergraph $\mathcal{H} = ([n], \mathcal{E})$ is β-acyclic if for any subset $\mathcal{E}' \subseteq \mathcal{E}$, $\mathcal{H}' = ([n], \mathcal{E}')$ is α-acyclic. A vertex $v \in [n]$ is a β-leaf of $\mathcal{H}$ if the multiset $\{K \in \mathcal{E} \mid x \in K\}$ can be linearly ordered by $\subseteq$. It is known that every β-acyclic hypergraph has a β-leaf [7, 8].

We can now introduce the notion of signed acyclicity, slightly modified from [6] to take multisets into account.

Definition 3.1 (Signed Acyclicity). A signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ is *signed-acyclic* if $([n], \mathcal{E}^+ \cup \mathcal{E}')$ is α-acyclic for every multiset $\mathcal{E}' \subseteq \mathcal{E}^-$.

Similar to α and β-leaves, we can define signed-leaves.

Definition 3.2 (signed-leaf). Let $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be a signed hypergraph. We say that $x \in [n]$ is a *signed-leaf* if there exists a hyperedge $U \in \mathcal{E}^+$, called the pivot of x, such that:

(1) (α-property) $K \subseteq U$ for every $K \in \mathcal{E}^+$ that contains x; and
(2) (β-property) the multiset $\{N \in \mathcal{E}^- \mid x \in N, N \not\subseteq U\} \cup \{U\}$ can be linearly ordered by $\subseteq$ with U being the minimal element.

The notion of a signed-leaf degenerates to an α-leaf when $\mathcal{E}^- = \emptyset$ (since the β-property holds trivially) and reduces to a β-leaf when $\mathcal{E}^+$ contains only singleton hyperedges (since the α-property holds trivially and the pivot is a singleton set). Recall that every α-acyclic hypergraph has an α-leaf and every β-acyclic hypergraph has a β-leaf. The next proposition should not be surprising.

Proposition 3.3. *Every signed-acyclic signed hypergraph has a signed-leaf.*

Next, we generalize the notion of an elimination sequence [19] to signed hypergraphs. Given $\mathcal{H}$ and a signed-leaf v with a pivot U, we define $\langle \mathcal{H}, v \rangle$ to be the hypergraph that (i) removes from $\mathcal{H}$ any (positive or negative) hyperedge V that contains v such that $V \subseteq U$, and (ii) removes v from any hyperedge that contains it.

Definition 3.4 (signed-elimination sequence). Let $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be a signed hypergraph. A vertex ordering $\sigma = (v_1, v_2, \ldots, v_n)$ of all vertices in $[n]$ is a signed-elimination sequence of $\mathcal{H}$ if v_i is a signed-leaf of $\mathcal{H}_i$ for every $i \in [n]$, where $\mathcal{H}_n = \mathcal{H}$, and $\mathcal{H}_j = \langle \mathcal{H}_{j+1}, v_{j+1} \rangle$ for $j = n-1, n-2, \ldots, 1$.

For a vertex ordering $\sigma = (v_1, v_2, \ldots, v_{n-1}, v_n)$, we often denote $\sigma = \sigma' \cdot v_n$ where $\sigma' = (v_1, v_2, \ldots, v_{n-1})$. By definition, if $\sigma = \sigma' \cdot v$ is a signed-elimination sequence of a signed hypergraph $\mathcal{H}$, then σ' is a signed-elimination sequence of $\langle \mathcal{H}, v \rangle$.

Proposition 3.5. *Let $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be a signed hypergraph. If $\mathcal{H}$ is signed-acyclic, then*

(1) $\mathcal{H}$ has a signed-elimination sequence; and
(2) for every hyperedge $\{u_1, u_2, \ldots, u_k\} \in \mathcal{E}^-$, $\mathcal{H}$ has a signed-elimination sequence of the form $(u_1, u_2, \ldots, u_k, v_{k+1}, \ldots, v_n)$.

Example 3.6. Consider the following $\mathrm{CQ}^\neg$, which will function as our running example.

$$Q(x_1, x_2, x_3, x_4) \leftarrow A(x_1, x_2, x_3) \wedge U(x_3, x_4) \wedge \neg V(x_4) \wedge \neg R(x_2, x_3, x_4) \wedge \neg S(x_1, x_2, x_3, x_4)$$

The vertex 4 is a signed-leaf for the hypergraph $\mathcal{H}$ of Q with pivot the hyperedge $\{3, 4\}$ that corresponds to the atom U. The query corresponding to the resulting hypergraph $\langle \mathcal{H}, 4 \rangle$ is:

$$Q'(x_1, x_2, x_3) = A(x_1, x_2, x_3) \wedge U(x_3) \wedge \neg R(x_2, x_3) \wedge \neg S(x_1, x_2, x_3)$$

In fact, Q is signed-acyclic with the signed-elimination sequence $\sigma = (1, 2, 3, 4)$.

We remark here that our notion of signed-leaf is equivalent to the notion of a leaf for signed hypergraphs as defined in [6] (we show this in Appendix A in the full version of this paper [27]). However, for our purposes we need to define a slightly different elimination sequence, since the hypergraph after the removal of a vertex x is defined differently.

4 ENUMERATION OF FULL CQ$^\neg$

In this section, we present an algorithm that can enumerate the answers of a signed-acyclic full CQ$^\neg$ (where $F = [n]$) with constant delay after linear preprocessing time. The correctness and runtime analysis of the algorithm is included at Appendix B in the full version of this paper [27].

Let Q be a full signed-acyclic CQ$^\neg$ with a signed-acyclic signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ and $\mathcal{D}$ a database instance. Let $\sigma = \sigma' \cdot v$ be a signed-elimination sequence of $\mathcal{H}$.

Our preprocessing phase recursively eliminates a variable v in the signed-elimination sequence. When eliminating v, the key idea is to construct in linear time a database $\mathcal{D}'$, and reduce the problem of computing the answers of $Q(\mathcal{D})$ to computing the answers of $Q'(\mathcal{D}')$, where the signed hypergraph of Q' is $\mathcal{H}' = \langle \mathcal{H}, v \rangle$ and has a signed-elimination sequence σ'. The reduction needs to ensure that $\Pi_{[n]\setminus\{v\}} Q(\mathcal{D}) = Q'(\mathcal{D}')$. [4] To achieve constant delay enumeration, we construct (also in linear time) a data structure over the domain of the variable x_v, that can, given an answer to $Q'(\mathcal{D}')$, extend it to an answer to $Q(\mathcal{D})$ with constant delay.

Example 4.1. We will continue with the query Q in Example 3.6, the database $\mathcal{D}$ in Figure 1(a) and a signed-elimination sequence $\sigma = (1, 2, 3, 4)$. The result $Q(\mathcal{D})$ is depicted in Figure 1(d). After eliminating x_4, we obtain Q' and the instance $\mathcal{D}'$ in Figure 1(b). We have that $Q'(\mathcal{D}') = \{(\mathsf{a}_1, \mathsf{b}_1, \mathsf{c}_1), (\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2), (\mathsf{a}_3, \mathsf{b}_3, \mathsf{c}_3)\}$. The preprocessing phase reduces computing $Q(\mathcal{D})$ to $Q'(\mathcal{D}')$ and produces a data structure as shown in Figure 1(c), such that given any tuple $\mathbf{a}' \in Q'(\mathcal{D}')$, we may use the data structure to enumerate all answers of the form $(\mathbf{a}', a_4)$ in $Q(\mathcal{D})$, a_4 being a value of x_4.

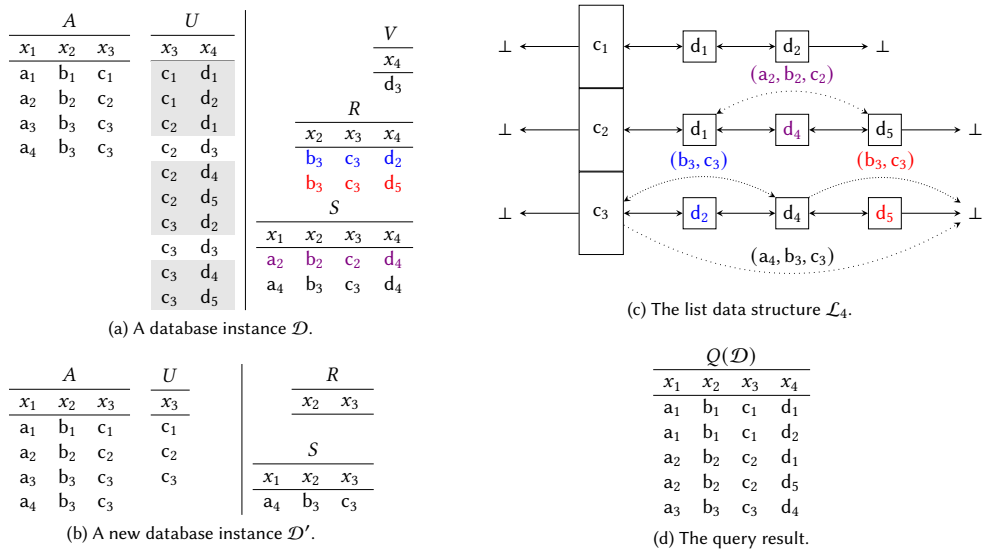
4.1 Preprocessing phase

Algorithm 1 describes the preprocessing phase that takes as input the signed hypergraph $\mathcal{H}$, a database instance $\mathcal{D}$, and a signed elimination sequence σ of $\mathcal{H}$. Let U be a pivot hyperedge of the signed-leaf v in $\mathcal{H}$ and $N_0 = \{v\} \subseteq U \subseteq N_1 \subseteq \cdots \subseteq N_m$ the linear order of the negative hyperedges that contain v. The preprocessing phase outputs a data structure $\mathcal{L}_v$ that maps tuples of the form $\Pi_{U\setminus\{v\}} \mathbf{a}_U$ to a doubly linked list of elements from the domain of x_v, with some extra tuple-labeled skipping links that we will introduce later. We explain how the algorithm eliminates the variable x_v in two steps: α-step and β-step.

α-step. The algorithm first performs an α-step (that mimics Yannakakis' algorithm) in which we simply remove tuples from R_U that will not contribute to a query answer of $Q(\mathcal{D})$ by filtering R_U with any positive (or negative) atom R_K with $v \in K$ and $K \subseteq U$.

Then, Algorithm 2 builds a hash table $\mathcal{L}_v$ that maps each tuple $\mathbf{a}_{U\setminus\{v\}} \in \Pi_{U\setminus\{v\}} R_U$ to a doubly linked list of the set $\{a_v \mid (\mathbf{a}_{U\setminus\{v\}}, a_v) \in R_U\}$, with a slight modification that every pointer is now parameterized/labeled with $\emptyset$.

[4] For a tuple $\mathbf{a}_N$ and $X \subseteq N$, we denote $\Pi_X \mathbf{a}_N$ as the projection of $\mathbf{a}_N$ on X.



(a) A database instance $\mathcal{D}$.

(c) The list data structure $\mathcal{L}_4$.

(b) A new database instance $\mathcal{D}'$.

(d) The query result.

Fig. 1. Given the query Q in Example 4.1, the database instance in (a) and the signed-elimination sequence $\sigma = (1, 2, 3, 4)$ as inputs, Algorithm 1 first produces the data structure $\mathcal{L}_4$ in (c), and then constructs a new query Q' in Example 4.1, a new database as in (b), and the sequence $\sigma' = (1, 2, 3)$ as inputs to the recursive call.

The extra label on the pointer allows us to add pointers with different labels in later steps. If Q contains no negative atoms, this data structure is sufficient to achieve constant delay enumeration by traversing the list from the head to the end. Finally, R_U is replaced with its projection $\Pi_{U\setminus\{v\}}R_U$.

Example 4.2. For our running example $U(x_3, x_4)$ is chosen as the pivot, and the tuples highlighted in gray in Figure 1(a) denotes the tuples in U that survive the semijoin step with the negative atom $\neg V(x_4)$. The linked lists of the hash table correspond to the data structure in Figure 1(c) if we ignore the dotted edges.

β-step. Assume $\mathbf{a} \in Q^+(\mathcal{D})$, where $Q^+(\mathbf{x}_{[n]}) \leftarrow \bigwedge_{K\in\mathcal{E}^+} R_K(\mathbf{x}_K)$. When negative atoms are present, for $\mathbf{a}$ to be an answer to $Q(\mathcal{D})$, we also need that $\Pi_{N_i}\mathbf{a} \notin R_{N_i}$ for every $i \in [m]$. Therefore, we need to augment $\mathcal{L}_v$ such that we can skip the enumeration of any value in the linked list that does not contribute to the answer. The linear order on the schemas of negative atoms allows us to construct this data structure in a dynamic fashion. If only one negative atom $\neg R_{N_1}$ is present (i.e., $m = 1$), for each tuple $\mathbf{a}_{N_1} \in R_{N_1}$ we can add a *skipping link* labeled with $\Pi_{N_1\setminus\{v\}}\mathbf{a}_{N_1}$ that bypasses the node with value $\Pi_{\{v\}}\mathbf{a}_{N_1}$ in the doubly linked list $\mathcal{L}_v[\Pi_{U\setminus\{v\}}\mathbf{a}_{N_1}]$. If $m = 2$, then there is another negative atom $\neg R_{N_2}$ with $N_1 \subseteq N_2$. For each tuple $\mathbf{a}_{N_2} \in R_{N_2}$, we only need to add a skipping link labeled with $\Pi_{N_2\setminus\{v\}}\mathbf{a}_{N_2}$ that bypasses the node with value $\Pi_{\{v\}}\mathbf{a}_{N_2}$ in the doubly linked list $\mathcal{L}_v[\Pi_{U\setminus\{v\}}\mathbf{a}_{N_2}]$ if that node has not been bypassed by a skipping link with label $\Pi_{N_1\setminus\{v\}}\mathbf{a}_{N_2}$ yet (i.e., $\Pi_{N_1}\mathbf{a}_{N_2} \notin R_{N_1}$). In general, for every tuple $\mathbf{a}_{N_i} \in R_{N_i}$ such that $\Pi_{\{v\}}\mathbf{a}_{N_i}$ has not been bypassed by any skipping link labeled with $\Pi_{N_j\setminus\{v\}}\mathbf{a}_{N_i}$ (i.e., for every $1 \le j < i \le m$, $\mathbf{a}_{N_j} \notin R_{N_j}$), we add a skipping link labeled with $\Pi_{N_i\setminus\{v\}}\mathbf{a}_{N_i}$ that bypasses $\Pi_{\{v\}}\mathbf{a}_{N_i}$, as well as every skipping links labeled with $\Pi_{N_j\setminus\{v\}}\mathbf{a}_{N_i}$ with $1 \le j < i$ starting in $\Pi_{\{v\}}\mathbf{a}_{N_i}$.

Algorithm 3 implements this idea using the parameterized/labeled pointers to support the skipping links. For any node in the doubly linked list, its next node given a tuple $\mathbf{a}_W$ with $W \supseteq U\setminus\{v\}$ is fetched by following $\text{next}[\Pi_{N_i\setminus\{v\}}\mathbf{a}_W]$ for the largest possible $i \in \{0, 1, \ldots, m\}$ in that node.

Algorithm 1: PreprocessFullCQ($\mathcal{H}, \mathcal{D}, \sigma$)

Input: signed hypergraph $\mathcal{H}$, instance $\mathcal{D}$, signed-elimination sequence $\sigma = \sigma' \cdot v$
Construct: $\mathcal{L}_v$ for every vertex v in σ

if σ *is empty* **then**
  terminate
$U \leftarrow$ a pivot hyperedge for the signed-leaf v in $\mathcal{H}$
foreach $K \in \mathcal{E}^+ \cup \mathcal{E}^-$ *such that* $v \in K \subseteq U$ **do** ▷ α-step
  if $K \in \mathcal{E}^+$ **then**
    $R_U \leftarrow R_U \ltimes R_K$
  else
    $R_U \leftarrow R_U \setminus (R_U \ltimes R_K)$
  remove relation R_K from $\mathcal{D}$
$\mathcal{L}_v \leftarrow \mathsf{BuildList}(v, U)$
replace R_U with $R_{U\setminus\{v\}} \leftarrow \Pi_{U\setminus\{v\}} R_U$
Let $N_1, \ldots, N_m \in \mathcal{E}^-$ s.t. $U \subseteq N_1 \subseteq \cdots \subseteq N_m$ ▷ β-step
foreach $i = 1, 2, \ldots, m$ **do**
  $\mathsf{ExtendList}(\mathcal{L}_v, N_i)$
  replace R_{N_i} with $R_{N_i\setminus\{v\}}$ that contains all $\mathbf{a}_{N_i\setminus\{v\}} \in \Pi_{N_i\setminus\{v\}} R_{N_i}$ such that $\Pi_{U\setminus\{v\}} \mathbf{a}_{N_i\setminus\{v\}} \in \mathcal{L}_v$
  and $\mathcal{L}_v[\Pi_{U\setminus\{v\}} \mathbf{a}_{N_i\setminus\{v\}}].\mathsf{nextM}(\mathbf{a}_{N_i\setminus\{v\}}) = \perp$
PreprocessFullCQ($\langle \mathcal{H}, v \rangle, \mathcal{D}, \sigma'$)

Algorithm 2: BuildList(v, U)

Input: $U \in \mathcal{E}^+, v \in U$
Output: a data structure $\mathcal{L}_v$

$\mathcal{L}_v \leftarrow$ an empty hashtable (returns $\perp$ always)
foreach $\mathbf{a}_U \in R_U$ **do**
  if $\mathcal{L}_v[\Pi_{U\setminus\{v\}} \mathbf{a}_U] = \perp$ **then**
    newHead.prev$[\emptyset] \leftarrow \perp$
    newHead.next$[\emptyset] \leftarrow \perp$
    $\mathcal{L}_v[\Pi_{U\setminus\{v\}} \mathbf{a}_U] \leftarrow$ newHead
  head $\leftarrow \mathcal{L}_v[\Pi_{U\setminus\{v\}} \mathbf{a}_U]$
  Node.val $\leftarrow \Pi_{\{v\}} \mathbf{a}_U$
  Node.prev$[\emptyset] \leftarrow$ *head*
  Node.next$[\emptyset] \leftarrow$ *head*.next$[\emptyset]$
  (*head*.next$[\emptyset]$).prev$[\emptyset] \leftarrow$ Node
return $\mathcal{L}_v$

Recall that $N_0 = \{v\}$ and thus pointers next$[\emptyset]$ and prev$[\emptyset]$ are always present for each node. This operation is denoted as $\mathsf{nextM}(\mathbf{a}_W)$, and we define $\mathsf{prevM}(\mathbf{a}_W)$ similarly. The formal definitions for the operations currNode.$\mathsf{nextM}(\mathbf{a}_W)$ and currNode.$\mathsf{prevM}(\mathbf{a}_W)$ are defined in Algorithm 4 and 5, respectively.

Algorithm 3: ExtendList($\mathcal{L}_v, N_i$)

Input: a data structure $\mathcal{L}_v$, $N_i \in \mathcal{E}^-$
Global variables: $U \in \mathcal{E}^+$, $N_1, N_2 \ldots, N_i \in \mathcal{E}^-$ s.t. $v \in U$, $U \subseteq N_1 \subseteq N_2 \subseteq \cdots \subseteq N_i$

foreach $\mathbf{a}_{N_i} \in R_{N_i}$ **do**
  if $\Pi_U \mathbf{a}_{N_i} \notin R_U$ *or* $\exists j \in \{1, 2, \ldots, i-1\}$ *such that* $\Pi_{N_j} \mathbf{a}_{N_i} \in R_{N_j}$ **then**
    continue
  currNode $\leftarrow v \in \mathcal{L}_v[\Pi_{U\setminus\{v\}} \mathbf{a}_{N_i}]$ with v.val $= \Pi_{\{v\}} \mathbf{a}_{N_i}$
  prevNode $\leftarrow$ currNode.prevM($\mathbf{a}_{N_i\setminus\{v\}}$)
  nextNode $\leftarrow$ currNode.nextM($\mathbf{a}_{N_i\setminus\{v\}}$)
  prevNode.next[$\mathbf{a}_{N_i\setminus\{v\}}$] $\leftarrow$ nextNode
  if nextNode $\neq \perp$ **then**
    nextNode.prev[$\mathbf{a}_{N_i\setminus\{v\}}$] $\leftarrow$ prevNode

Algorithm 4: currNode.nextM($\mathbf{a}_W$)

Input: a list node currNode from $\mathcal{L}_v$, a tuple $\mathbf{a}_W$ with $W \supseteq U \setminus \{v\}$
Global variables of $\mathcal{L}_v$**:** $U \in \mathcal{E}^+$, $\{v\} = N_0 \subseteq N_1, N_2 \ldots, N_i \in \mathcal{E}^-$ s.t. $v \in U$, $U \subseteq N_1 \subseteq N_2 \subseteq \cdots \subseteq N_i$
Output: the next node of currNode following $\mathbf{a}_W$

$\ell \leftarrow$ the largest in $\{0, 1, \ldots, m\}$ such that currNode.next[$\Pi_{N_i\setminus\{v\}} \mathbf{a}_W$] is defined
return currNode.next[$\Pi_{N_\ell\setminus\{v\}} \mathbf{a}_W$]

Algorithm 5: currNode.prevM($\mathbf{a}_W$)

Input: a list node currNode from $\mathcal{L}_v$, a tuple $\mathbf{a}_W$ with $W \supseteq U \setminus \{v\}$
Global variables of $\mathcal{L}_v$**:** $U \in \mathcal{E}^+$, $\{v\} = N_0 \subseteq N_1, N_2 \ldots, N_i \in \mathcal{E}^-$ s.t. $v \in U$, $U \subseteq N_1 \subseteq N_2 \subseteq \cdots \subseteq N_i$
Output: the previous node of currNode following $\mathbf{a}_W$

$\ell \leftarrow$ the largest in $\{0, 1, \ldots, m\}$ such that currNode.prev[$\Pi_{N_i\setminus\{v\}} \mathbf{a}_W$] is defined
return currNode.prev[$\Pi_{N_\ell\setminus\{v\}} \mathbf{a}_W$]

Algorithm 6: $\mathcal{L}_v$.Iterate($\mathbf{a}_W$)

Input: a tuple $\mathbf{a}_W$ such that $(U \setminus \{v\}) \subseteq W$

currNode $\leftarrow \mathcal{L}_v[\Pi_{U\setminus\{v\}} \mathbf{a}_W]$
while currNode.nextM($\Pi_{W\setminus\{v\}} \mathbf{a}_W$) $\neq \perp$ **do**
  currNode $\leftarrow$ currNode.nextM($\Pi_{W\setminus\{v\}} \mathbf{a}_W$)
  emit currNode.val
emit $\perp$

To traverse $\mathcal{L}_v$, we implement an iterator (Algorithm 6) that takes a tuple $\mathbf{a}_W$ with $W \supseteq U \setminus \{v\}$, and then traverses the linked list $\mathcal{L}_v[\Pi_{U\setminus\{v\}} \mathbf{a}_W]$ using nextM($\mathbf{a}_W$).

Example 4.3. Algorithm 3 takes as input the basic linked list data structure and all tuples in the negative atoms $\neg R(x_2, x_3, x_4)$ and $\neg S(x_1, x_2, x_3, x_4)$ in Figure 1(a) to produce the data structure in

Algorithm 7: EnumerationFullCQ(σ)

Input: a signed-elimination sequence $\sigma = (v_1, v_2, \ldots, v_n)$
Output: an enumeration of the query answers $Q(\mathcal{D})$

```
foreach a_{v_1} ∈ L_{v_1}.Iterate(∅) do
    foreach a_{v_2} ∈ L_{v_2}.Iterate((a_{v_1})) do
        foreach a_{v_3} ∈ L_{v_3}.Iterate((a_{v_1}, a_{v_2})) do
            ...
            foreach a_{v_n} ∈ L_{v_n}.Iterate((a_{v_1}, ..., a_{v_{n-1}})) do
                emit (a_{v_1}, a_{v_2}, ..., a_{v_n})
```

Figure 1(c). Note that the skipping link bypassing d_4 with label (a_4, b_3, c_4) also needs to bypass both skipping links labeled by (b_3, c_3) and (b_3, c_3) that bypass d_2 and d_5 respectively.

Finally, for each negative atom $\neg R_{N_i}$, we need to keep precisely the subset of $\Pi_{N_i \setminus \{v\}} R_{N_i}$ so that we can *avoid* emitting an answer to $Q'(\mathcal{D}')$ that cannot be extended to an answer to $Q(\mathcal{D})$. This is done by keeping the tuple $\mathbf{a}_{N_i \setminus \{v\}} \in \Pi_{N_i \setminus \{v\}} R_{N_i}$ only if $\Pi_{U \setminus \{v\}} \mathbf{a}_{N_i \setminus \{v\}} \in \mathcal{L}_v$, but the tuple cannot be extended to any answer to $Q(\mathcal{D})$, i.e., the traversal on $\mathcal{L}_v$ using $\mathbf{a}_{N_i \setminus \{v\}}$ does not lead to any value of x_v.

Example 4.4. After removing x_4 from relations $A(x_1, x_2, x_3, x_4)$ and $U(x_3, x_4)$, we end up with relations $A(x_1, x_2, x_3)$ and $U(x_3)$ in Figure 1(b). The query containing only the positive atoms of Q' would return $\{(a_1, b_1, c_1), (a_2, b_2, c_2), (a_3, b_3, c_3), (a_4, b_3, c_3)\}$, but we wish to skip enumerating (a_4, b_3, c_3), since it cannot be extended to an answer to $Q(\mathcal{D})$. Indeed, the traversal on the linked list $\mathcal{L}_4[c_3]$ will simply follow the black skipping link (a_4, b_3, c_3) to reach the end of the list. We achieve this by adding (a_4, b_3, c_3) to $S(x_1, x_2, x_3)$ so that this tuple is not returned in the recursive call.

4.2 Enumeration phase

The enumeration phase is shown in Algorithm 7. It follows the reverse order of $\sigma = \sigma' \cdot v$ to first recursively enumerate a tuple $\mathbf{a}'_{\sigma'} \in Q'(\mathcal{D}')$, and then (with a slight abuse of notation), use every $a_v \in \mathcal{L}_v$.Iterate($\mathbf{a}'_{\sigma'}$) to obtain an answer $(\mathbf{a}'_{\sigma'}, a_v)$ to $Q(\mathcal{D})$. The preprocessing phase guarantees that the iterator is nonempty for every answer $\mathbf{a}'_{\sigma'}$.

4.3 A Comprehensive Example

This section demonstrates Algorithm 1 (pre-processing phase) and Algorithm 7 (enumeration phase) end-to-end on our running example.

Example 4.5. We illustrate on the running example query $Q(= Q_4)$ in Example 3.6, the database $\mathcal{D}(= \mathcal{D}_4)$ in Figure 2(a) and the signed-elimination sequence $\sigma = (1, 2, 3, 4)$.

$$Q_4(x_1, x_2, x_3, x_4) \leftarrow A(x_1, x_2, x_3) \wedge U(x_3, x_4) \wedge \neg V(x_4) \wedge \neg R(x_2, x_3, x_4) \wedge \neg S(x_1, x_2, x_3, x_4)$$

Preprocessing phase We dive into the recursive steps of Algorithm 1 on input $(\mathcal{H}_4, \mathcal{D}_4, (1, 2, 3, 4))$:

(1) First, we remove the signed leaf x_4 from $\mathcal{H}_4$, which yields the following query $Q_3(x_1, x_2, x_3)$ whose hypergraph corresponds to the hypergraph $\mathcal{H}_3 = \langle \mathcal{H}_4, x_4 \rangle$ and emits a skipping list data structure $\mathcal{L}_4$ as shown in Figure 2(e):

$$Q_3(x_1, x_2, x_3) \leftarrow A(x_1, x_2, x_3) \wedge U(x_3) \wedge \neg R(x_2, x_3) \wedge \neg S(x_1, x_2, x_3).$$

The database is changed to $\mathcal{D}_3$ as in Figure 2(b). This is a high-level summary of the previous examples where $\mathcal{D}_3 = \mathcal{D}'$ (Example 4.1 through Example 4.4). Next, we keep following the signed-elimination sequence $(1, 2, 3)$, i.e. on input $(\mathcal{H}_3, \mathcal{D}_3, (1, 2, 3))$.

(2) We remove the signed leaf x_3 from $\mathcal{H}_3$, yielding $Q_2(x_1, x_2)$ whose hypergraph corresponds to the hypergraph $\mathcal{H}_2 = \langle \mathcal{H}_3, x_3 \rangle$ and emits $\mathcal{L}_3$ as shown in Figure 2(f):

$$Q_2(x_1, x_2) \leftarrow A(x_1, x_2).$$

Note that in this step, the positive atom $U(x_3)$ and all negated atoms $\neg R(x_2, x_3)$ and $\neg S(x_1, x_2, x_3)$ are removed from Q_3 since their corresponding negative hyperedges are contained by the positive hyperedge corresponding to $A(x_1, x_2, x_3)$. Further, the skipping list $\mathcal{L}_3$ does not contain any skipping edge. The database is changed to $\mathcal{D}_2$ as in Figure 2(c).

(3) We further remove the signed leaf x_2 from $\mathcal{H}_2$, finally yielding $Q_1(x_1)$ with a hypergraph corresponding to $\mathcal{H}_1 = \langle \mathcal{H}_2, x_2 \rangle$ and emits $\mathcal{L}_2$ shown in Figure 2(g):

$$Q_1(x_1) \leftarrow A(x_1).$$

Note that $\mathcal{L}_2$ also does not contain any skipping edge. The database is changed to $\mathcal{D}_1$ as in Figure 2(d).

(4) Finally, we remove the signed leaf x_1 from $\mathcal{H}_1$, and this step essentially creates a linked-list $\mathcal{L}_1$ (shown in Figure 2(h)) on the remaining elements in the relation A in Figure 2(d).

Enumeration phase For the enumeration step, we first enumerate every element in $\mathcal{L}_1$ (yielding $\mathsf{a}_1, \mathsf{a}_2$ and a_3). Then we use that element enumerated in $\mathcal{L}_1$ as a probing tuple in the enumeration process of every element in $\mathcal{L}_2$. For example, $\mathcal{L}_2.\mathsf{nextM}(\mathsf{a}_1)$ leads to b_1, and $\mathcal{L}_2.\mathsf{nextM}(\mathsf{a}_2)$ leads to b_2. We continue this step using the combined tuple enumerated from $\mathcal{L}_1$ and $\mathcal{L}_2$ (say, $(\mathsf{a}_1, \mathsf{b}_1)$), to enumerate the elements in $\mathcal{L}_3$. For example, $\mathcal{L}_3.\mathsf{nextM}((\mathsf{a}_1, \mathsf{b}_1))$ gives c_1 and $\mathcal{L}_3.\mathsf{nextM}((\mathsf{a}_2, \mathsf{b}_2))$ gives c_2.

Finally, we use the combined tuple enumerated from $\mathcal{L}_1$, $\mathcal{L}_2$, $\mathcal{L}_3$ (say, $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2)$), to enumerate the elements in $\mathcal{L}_4$. This step uses the skipping links: for example, $\mathcal{L}_4.\mathsf{nextM}((\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2))$ would locate the doubly linked-list stored at the key c_2, and then we traverse that doubly linked-list using the tuple $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2)$. We first yield d_1, but since the skipping links leaving d_1 contain $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2)$, we follow that skipping-link and reach d_5, bypassing d_4 and correctly enumerating $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2, \mathsf{d}_1)$ and $(\mathsf{a}_2, \mathsf{b}_2, \mathsf{c}_2, \mathsf{d}_5)$ as answers.

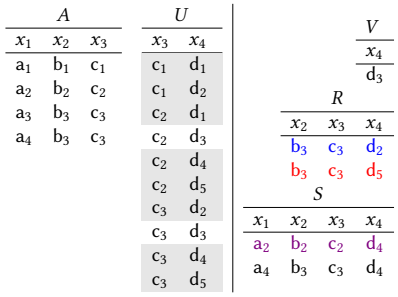
5 ENUMERATION OF FAQ$^\neg$

In this section, we give a high-level description of the enumeration algorithm for FAQ$^\neg$ queries as (3) that constructively proves Theorem 1.2. The formal version of the algorithm is presented at Appendix C of the full version of this paper [27], and runs on a more general class of queries, called NestFAQ$^\neg$ queries, with similar notions of free-connex signed-acyclicity.

Let φ be a free-connex signed-acyclic FAQ$^\neg$ (3) with free variables $F = [f]$ and $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$ be its associated signed-acyclic signed hypergraph. Assume w.l.o.g that $\mathcal{H}$ accepts a signed-elimination sequence $\sigma = (1, 2, \ldots, f \ldots, n)$. Let $\mathcal{D}$ be a database instance of φ. In the preprocessing phase, the algorithm executes a sequence of *signed-elimination steps on the signed-leaf* $i+1$, for each $i = n-1, \ldots, f$ (in order). A signed-elimination step returns an intermediate tuple $(\varphi_i, \mathcal{D}_i)$, where

- φ_i is a signed-acyclic NestFAQ$^\neg$ query with free variables $[i]$ associated with $\mathcal{H}_i = \langle \mathcal{H}_{i+1}, i+1 \rangle$, the signed hypergraph after eliminating $n, n-1, \ldots, i+1$ from $\mathcal{H}$, (e.g. $\mathcal{H}_{n-1} = \langle \mathcal{H}, n \rangle$)
- $\mathcal{D}_i$ is an instance of φ_i such that $\varphi_i(\mathcal{D}_i) = \varphi_{i+1}(\mathcal{D}_{i+1})$.

The sequence ends up with a full signed-acyclic NestFAQ$^\neg$ query φ_f with free variables F and $\mathcal{D}_f$ such that $\varphi(\mathcal{D}) = \varphi_f(\mathcal{D}_f)$. Next, for a full signed-acyclic NestFAQ$^\neg$ query, we show in Appendix



A		
x_1	x_2	x_3
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3
a_4	b_3	c_3

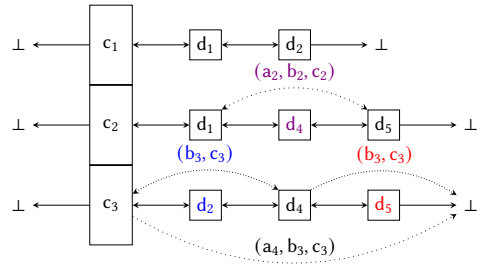
U	
x_3	x_4
c_1	d_1
c_1	d_2
c_2	d_1
c_2	d_3
c_2	d_4
c_2	d_5
c_3	d_2
c_3	d_3
c_3	d_4
c_3	d_5

V
x_4
d_3

R		
x_2	x_3	x_4
b_3	c_3	d_2
b_3	c_3	d_5

S			
x_1	x_2	x_3	x_4
a_2	b_2	c_2	d_4
a_4	b_3	c_3	d_4

(a) A database instance $\mathcal{D}(= \mathcal{D}_4)$.



(e) The list data structure $\mathcal{L}_4$.

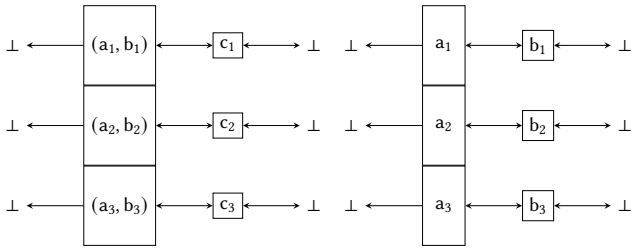
A		
x_1	x_2	x_3
a_1	b_1	c_1
a_2	b_2	c_2
a_3	b_3	c_3
a_4	b_3	c_3

U
x_3
c_1
c_2
c_3

R	
x_2	x_3

S		
x_1	x_2	x_3
a_4	b_3	c_3

(b) A new database instance $\mathcal{D}_3$.



(f) The list data structure $\mathcal{L}_3$.

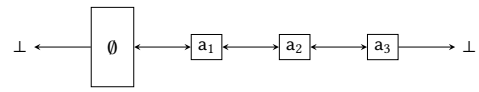
(g) The list data structure $\mathcal{L}_2$.

A	
x_1	x_2
a_1	b_1
a_2	b_2
a_3	b_3

(c) $\mathcal{D}_2$.

A
x_1
a_1
a_2
a_3

(d) $\mathcal{D}_1$.



(h) The list data structure $\mathcal{L}_1$.

Fig. 2. Intermediate databases and list data structures produced for Example 4.5.

C.1 of the full version of this paper [27] a reduction from the enumeration problem $\mathsf{Enum}(\varphi_f, \mathcal{D}_f)$ into $\mathsf{Enum}(Q_f^*, \mathcal{D}_f^*)$, where Q_f^* is a full signed-acyclic CQ$^\neg$ and $\mathcal{D}_f^*$ is a database instance of Q_f^*. At this point, we simply apply the preprocessing algorithm for a full CQ$^\neg$. In the enumeration phase, as we emit each answer $\mathbf{a}_F$ of $Q_f^*(\mathcal{D}_f)$, we plug the emitted tuple $\mathbf{a}_F$ into φ_f to recover its weight $\varphi_f(\mathbf{a}_F) = \varphi(\mathbf{a}_F)$.

We now take a closer look at a signed-elimination step via a concrete running example. The formal (and rather technical) description is deferred to Appendix C.2 up to Appendix C.6 in the full version of this paper [27].

Example 5.1. Consider the following signed-acyclic FAQ$^\neg$ $\varphi(x_1, x_2)$, where 3 is a signed-leaf of its associated hypergraph with a pivot hyperedge $\{3\}$ (corresponding to the positive factor R_3):

$$\bigoplus_{x_3} R_1(x_1) \otimes R_2(x_2) \otimes R_3(x_3) \otimes \overline{R}_{123}(x_1, x_2, x_3) \otimes \overline{R}_{23}(x_2, x_3)$$

The signed-elimination step for the signed-leaf 3 is much more involved than the elimination step for a signed-acyclic CQ$^\neg$. A naive attempt may aim for a reduction to the follwing φ'

$$\varphi'(x_1, x_2) = R_1(x_1) \otimes R_2(x_2) \otimes \overline{R'}_{12}(x_1, x_2) \otimes \overline{R'}_2(x_2),$$

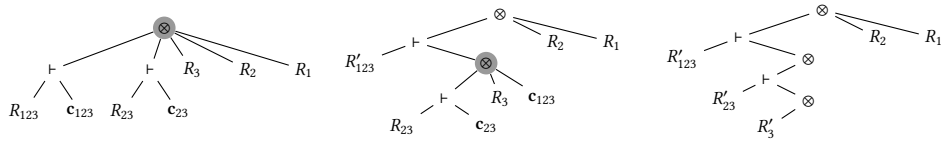


Fig. 3. The refactor steps on AST of φ for the signed-leaf 3, recursively from left to right. The shaded $\otimes$ nodes indicate the subtrees to be recursively refactored. The left is the original AST representation of φ, and the right is its refactored AST.

asking for $\overline{R'}_{12}(x_1, x_2) \otimes \overline{R'}_2(x_2) = \bigoplus_{x_3} R_3(x_3) \otimes \overline{R}_{123}(x_1, x_2, x_3) \otimes \overline{R}_{23}(x_2, x_3)$. However, it is not attainable by the semantics of FAQ$^\neg$ over a general semiring. Indeed, we examine the weight $\overline{R'}_2(a_2)$, for some $x_2 = a_2$ such that $\Pi_{x_3}\overline{R}_{23}(a_2, x_3) \neq \emptyset$: if $x_1 = a_1$ with $\Pi_{x_1,x_2}\overline{R}_{123}(a_1, a_2, x_3) = \emptyset$, then it should encode $\bigoplus_{x_3} R_3(x_3) \otimes \mathbf{c}_{123} \otimes \overline{R}_{23}(a_2, x_3)$, where $\mathbf{c}_{123}$ is the constant value for entries out of the table of $\overline{R}_{123}$; otherwise, we want $\overline{R'}_2(a_2) = \mathbf{1}$ and $\overline{R'}_{12}(a_1, a_2)$ to store $\bigoplus_{x_3} R_3(x_3) \otimes \overline{R}_{123}(a_1, a_2, x_3) \otimes \overline{R}_{23}(a_2, x_3)$. The two weights may not coincide over an arbitrary semiring.

This insufficiency calls for lifting FAQ$^\neg$ to a more expressive class of queries NestFAQ$^\neg$, where we exhibit φ as an *abstract syntax tree* (AST) depicted in Figure 3 (left), in which we decompose the negative factors as

$$\overline{R}_{123} = R_{123} \oplus \mathbb{1}_{\neg R_{123}} \otimes \mathbf{c}_{123}, \qquad \overline{R}_{23} = R_{23} \oplus \mathbb{1}_{\neg R_{23}} \otimes \mathbf{c}_{23}$$

where $\mathbb{1}_{\neg R_K}$ is an indicator factor that maps to $\mathbf{0}$ if $\mathbf{x}_K \in R_K$ and $\mathbf{1}$ otherwise. As a shorthand, we write $x \vdash y := x \oplus \mathbb{1}_{\neg x} \otimes y$. Now we illustrate the signed-elimination step on the signed-leaf 3 (corresponding to x_3) in Example 5.1. A signed-elimination step can be further decomposed into three steps: (i) refactor step, (ii) oracle-construction step, and (iii) aggregation step.

5.1 The Refactor Step

We first illustrate the refactor step: it reorganizes the AST of φ to make x_3 present only in one subtree rooted at a child of the root $\otimes$ node. First, it replaces R_{123} with a new factor $R'_{123} := R_{123} \otimes (R_{23} \vdash \mathbf{c}_{23}) \otimes R_3$, the entry table of which is exactly that of R_{123}, except for multiplying the extra weight $(R_{23} \vdash \mathbf{c}_{23}) \otimes R_3$ to each table entry. Then, we get a new AST in Figure 3 (middle) since

$$(R_{123} \vdash \mathbf{c}_{123}) \otimes (R_{23} \vdash \mathbf{c}_{23}) \otimes R_3 = R'_{123} \vdash ((R_{23} \vdash \mathbf{c}_{23}) \otimes R_3 \otimes \mathbf{c}_{123}) .$$

We keep recursing on the subtree for $(R_{23} \vdash \mathbf{c}_{23}) \otimes R_3 \otimes \mathbf{c}_{123}$, thus leading to the desired AST in Figure 3 (right) with new factors $R'_{23} := R_{23} \otimes (R_3 \vdash \mathbf{c}_3)$ and $R'_3 := R_3 \otimes \mathbf{c}_{123} \otimes \mathbf{c}_{23}$ computed. Indeed,

$$(R_{23} \vdash \mathbf{c}_{23}) \otimes R_3 \otimes \mathbf{c}_{123} = R'_{23} \vdash (R_3 \otimes \mathbf{c}_{123} \otimes \mathbf{c}_{23}) = R'_{23} \vdash R'_3.$$

As we go, R_3 sinks down and the linear inclusion $[3] \supseteq \{2, 3\} \supseteq \{3\}$ surfaces along the subtrees that contain x_3. In the end, R_3 absorbs the constants $\mathbf{c}_{123}$ and $\mathbf{c}_{23}$. The refactor step runs in linear time with no query (or database) size blow-up, as proven in Appendix C.3 of the full version of this paper [27].

5.2 The Oracle-construction Step

Next, we illustrate the oracle-construction step. Chazelle and Rosenberg [11] showed that a semigroup RangeSum[5] data structure (oracle) on an array of size w can be constructed in time $O(w)$

[5]The (semigroup) RangeSum problem is defined as follows: preprocess an array of w elements from a semigroup, and then for any query range, return the semigroup range sum.

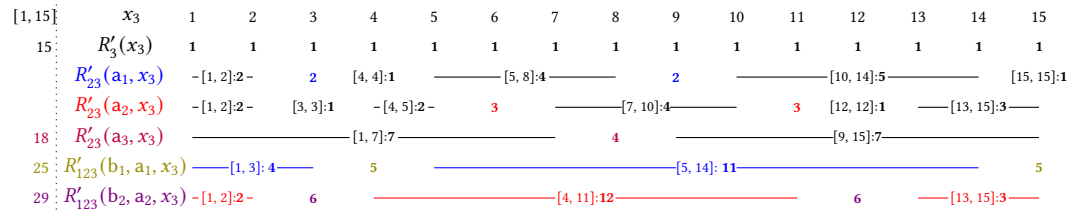


Fig. 4. The oracle-construction steps on the refactored AST. The oracle of $R'_{23}(a_1, x_3)$, $R'_{23}(a_2, x_3)$ and $R'_{23}(a_3, x_3)$ are built atop the oracle of $R'_3(x_3)$; the oracle of $R'_{123}(b_1, a_1, x_3)$ and $R'_{123}(b_2, a_2, x_3)$ are built atop the oracle of $R'_{23}(a_1, x_3)$ and $R'_{23}(a_2, x_3)$, respectively. The leftmost column shows the requested range sums over $[1, 15]$ in the aggregation step.

to support a semigroup sum over any range in $O(\alpha(14w, w))$ time, where α is the inverse Ackermann function. The oracle-construction step uses it as a black-box and builds a RangeSum oracle bottom-up for each subtree containing x_3, i.e. the subtree rooted at $\otimes$ nodes in Figure 3 (right). We demonstrate via a simple database instance.

Example 5.2. We assume the counting ring and on the refactored tree in Figure 3 (right), let R'_3 store $\{\langle i, 1\rangle, i \in [15]\}$, R'_{23} store $\{\langle(a_1, 3), 2\rangle, \langle(a_1, 9), 2\rangle, \langle(a_2, 6), 3\rangle, \langle(a_2, 11), 3\rangle, \langle(a_3, 8), 4\rangle\}$ and R'_{123} store $\langle(b_1, a_1, 4), 5\rangle, \langle((b_1, a_1, 15), 5\rangle, \langle((b_2, a_1, 3), 6\rangle, \langle(b_2, a_2, 12), 6\rangle$.

The oracle-construction step starts from an array representation of the database instance drawn in Figure 4, indexed by all possible values of x_3. As an example, the tuple $\langle(b_1, a_1, 4), 5\rangle$ of R'_{123} can be accessed as the 4-th element of the array $R'_{123}(b_1, a_1, x_3)$. The necessity of RangeSum becomes natural: after factoring out the term $R_1(b_1) \otimes R_2(a_1)$ (independent of x_3), the aggregation for $x_1 = b_1, x_2 = a_1$, from the lens of the array $R'_{123}(b_1, a_1, x_3)$, is

$$\bigoplus_{x_3} R'_{123}(b_1, a_1, x_3) \vdash (R'_{23}(a_1, x_3) \vdash R'_3(x_3)) = \sum_{1 \le x_3 \le 2} 1 + \sum_{x_3=3} 2 + \sum_{x_3=4} 5 + \sum_{5 \le x_3 \le 8} 1 + \sum_{x_3=9} 2 + \sum_{10 \le x_3 \le 14} 1 + \sum_{x_3=15} 5 \tag{4}$$

where each term is a sum over a range. To support fast aggregation, we construct a RangeSum oracle on the array of $R'_3(x_3)$. Then, we build a RangeSum oracle on the array of $R'_{23}(a_1, x_3)$ and $R'_{23}(a_2, x_3)$. In particular, we query the oracle of $R'_3(x_3)$ to fill in the partial sums (as in the 3rd, 4th row in Figure 4) and then construct RangeSum oracles over the partial sums. Take $R'_{23}(a_1, x_3)$ as the example, $[5, 8] : \mathbf{4}$ denotes querying the oracle $R'_3(x_3)$ with $5 \le x_3 \le 8$ as range and getting back the value $\mathbf{4}$. Then the RangeSum oracle for $R'_{23}(a_1, x_3)$ is constructed over the array (of partial sums):

$$[1, 2] : \mathbf{2} \quad \mathbf{2} \quad [4, 4] : \mathbf{1} \quad [5, 8] : \mathbf{4} \quad \mathbf{2} \quad [10, 14] : \mathbf{5}. \quad [15, 15] : \mathbf{1}$$

where the ranges above split at $x_3 = 3, 9$ and $x_3 = 4, 15$ in account for $R'_{23}(a_1, x_3)$ and $R'_{123}(b_1, a_1, x_3)$, respectively. The rationale here is to avoid mis-aligned ranges when querying the oracle of $R'_{23}(a_1, x_3)$ during the oracle-construction of $R'_{123}(b_1, a_1, x_3)$. The details are in the 6th row of Figure 4. The oracle-construction of $R'_{23}(a_2, x_3)$ and $R'_{123}(b_2, a_2, x_3)$ are similarly depicted in the 4th and 7th row of Figure 4. It is worth noting that the splits of ranges should be choreographed for query range alignment, and yet can not be arbitrarily fine-grained due to possible blow-ups in time and space. Indeed, we formally show in Appendix C.5 in the full version of this paper [27] that desired oracles can be carefully constructed in linear time and space.

5.3 The Aggregation Step

Finally, we go over the aggregation step that effectively peels off x_3 from the refactored AST. To that end, it pushes the aggregation operator $\bigoplus_{x_3}$ downward the refactored AST from root to $R'_3(x_3)$, i.e. the opposite direction of the oracle-construction step. Start with the root $\otimes$ node, we will construct new factors $R''_{12}(x_1, x_2)$ and $R''_2(x_2)$ such that (let $\bigoplus_{x_3} R'_3 = 15$):

$$\bigoplus_{x_3} R'_{123} \vdash (R'_{23} \vdash R'_3) = R''_{12} \vdash \bigoplus_{x_3} (R'_{23} \vdash R'_3) = R''_{12} \vdash (R''_2 \vdash \bigoplus_{x_3} R'_3) = R''_{12} \vdash (R''_2 \vdash 15).$$

To realize the first equality, we ask the table of $R''_{12}(x_1, x_2)$ to store two entries, $\langle(\mathsf{b}_1, \mathsf{a}_1), 25\rangle$ and $\langle(\mathsf{b}_2, \mathsf{a}_2), 29\rangle$, where 25 is the aggregation for $x_1 = \mathsf{b}_1, x_2 = \mathsf{a}_1$ in (4), that can be obtained by a simple range query $[1, 15]$ on the oracle of $R'_{123}(\mathsf{b}_1, \mathsf{a}_1, x_3)$. Similarly for 29 from a $[1, 15]$ query on the oracle of $R'_{123}(\mathsf{b}_2, \mathsf{a}_2, x_3)$. Indeed, the first equality holds because for all values of (x_1, x_2) not encoded in the table of R''_{12}, we always have $R'_{123} = \mathbf{0}$ and $\mathbb{1}_{\neg R'_{123}} = \mathbf{1}$, therefore for those (x_1, x_2)s, $R'_{123} \vdash (R'_{23} \vdash R'_3) = \mathbf{0} \oplus \mathbf{1} \otimes (R'_{23} \vdash R'_3) = R'_{23} \vdash R'_3$.

Now that we have pushed $\bigoplus_{x_3}$ to $(R'_{23} \vdash R'_3)$ (i.e. the $\otimes$ node on the 2nd level of the refactored AST) and we insert a single tuple $\langle(\mathsf{a}_3), 18\rangle$ into the new factor $R''_2(x_2)$, where 18 is yet another $[1, 15]$ query on the oracle of $R'_{23}(\mathsf{a}_3, x_3)$. The second equality holds because for all values of x_2 not encoded in the table of R''_2, we have $R'_{23} = \mathbf{0}$ and $\mathbb{1}_{\neg R'_{23}} = \mathbf{1}$. Thus, $R'_{23} \vdash R'_3 = \mathbf{0} \oplus \mathbf{1} \otimes R'_3 = R'_3$. The resulting NestFAQ$^\neg$ query φ'' after the signed-elimination on 3 is

$$\varphi''(x_1, x_2) = R_1(x_1) \otimes R_2(x_2) \otimes \left(R''_{12}(x_1, x_2) \vdash (R''_2(x_2) \vdash 15)\right)$$

where the AST of φ'' is exactly the refactored AST in Figure 3 (right), except that x_3 is being eliminated. Now the signed-elimination step is complete. We refer the reader to the full version of this paper [27] (see Appendix C.4 and Appendix C.6) for a formal analysis of the aggregation step.

5.4 The Reduction to Enumeration of Full CQ$^\neg$

Finally, we cast the enumeration of the (full) NestFAQ$^\neg$ query φ'' into the enumeration of the full signed-acyclic CQ$^\neg$ Q^*, by directly extracting factors from φ'' as relations:

$$Q^*(x_1, x_2) \leftarrow R^*_1(x_1) \wedge R^*_2(x_2) \wedge \neg R^{**}_{12}(x_1, x_2) \wedge \neg R^{**}_2(x_2)$$

where R^*_1 stores $a_1 \in R_1$ with $R_1(a_1) \neq \mathbf{0}$ (similarly for R^*_2), and R^{**}_{12} stores $(a_1, a_2) \in R''_{12}$ with $R''_{12}(a_1, a_2) = \mathbf{0}$ (similarly for R^{**}_2). Intuitively, Q^* forbids exactly the tuples (a_1, a_2) such that $\varphi''(x_1, x_2) = \mathbf{0}$ to be emitted. Thus, we can safely enumerate answers of Q^* using techniques in Section 4 and use φ'' to recover the non-$\mathbf{0}$ weights. Formal arguments can be found in Appendix C.1 in the full version of this paper [27].

6 LOWER BOUNDS

In this section, we present conditional and unconditional lower bounds which complement our upper bounds. All lower bounds here refer to *self-join-free* queries, which means that each relation name appears at most once in the body of the query. Technical proofs of this section can be found in Appendix D of the full version of this paper [27].

6.1 Lower Bounds for CQ$^\neg$

In this section, we show lower bound results for any CQ$^\neg$ that is not free-connex signed-acyclic. We split this result into two theorems that use different conditional lower bounds.

Problem $(k+1, k)$-Hyperclique

Input a k-uniform hypergraph[6] (for $k > 2$)
Output does it contain a hyperclique of size $k+1$, i.e. a set of $k+1$ vertices where every subset of size k forms a hyperedge

Conjecture 6.1 ($(k+1,k)$-Hyperclique). *There is no algorithm that solves $(k+1,k)$-Hyperclique in $O(m)$ time, where m is the number of edges in the input hypergraph.*

Readers are referred to [4] for evidence why Conjecture 6.1 is believable. When $k = 2$, the (3,2)-Hyperclique problem is the problem of finding a triangle in a graph.

Conjecture 6.2 (Triangle). *There is no algorithm that decides whether a graph with n nodes contains a triangle in $O(n^2)$ time.*

The following is a combination of results from [4, 6].

Theorem 6.3. *Let Q be a $\mathsf{CQ}^\neg$ that is not signed-acyclic. Assuming Conjecture 6.1 and Conjecture 6.2, then there is no algorithm for Q that has linear preprocessing time and $O(1)$ delay.*

To show a lower bound for queries that are not free-connex (but are signed-acyclic), we will use a weaker lower bound conjecture that implies Triangle.

Conjecture 6.4 (BMM). *There is no algorithm that computes the product $A \times B$ of two $n \times n$ Boolean matrices A and B in $O(n^2)$ time.*

Evidence for Conjecture 6.4 can be found in [21]. Bagan et al. [3] reduce the BMM problem to the non-free-connex acyclic query $Q(x,y) \leftarrow \bigvee_z A(x,z) \wedge B(z,y)$ and apply Conjecture 6.4 to obtain a conditional lower bound. The *matrix multiplication exponent* ω is the smallest number such that for any $\epsilon > 0$, there is an algorithm that multiplies two n-by-n matrices with at most $O(n^{\omega+\epsilon})$ operations (assuming RAM model). The best bound known so-far on ω is (roughly) $\omega < 2.373$ in [14, 23]. We note that Conjecture 6.4 does not violate the common belief that $\omega = 2$, since that only implies that BMM can be computed in time $n^{2+o(1)}$. For non-free-connex CQs, a weaker lower bound conjecture was used, sparse BMM (the matrices have m non-zero entries and no $O(m)$ algorithm exists). However, this conjecture cannot be applied in our case because we need to take the complement of the matrix to populate a negated atom, and that means that a sparse matrix becomes dense.

Theorem 6.5. *Let Q be a $\mathsf{CQ}^\neg$ that is signed-acyclic and not free-connex. Assuming Conjecture 6.4, there is no algorithm with linear preprocessing time and $O(1)$ delay.*

6.2 Lower Bounds for $\mathsf{FAQ}^\neg$

We present next lower bounds for $\mathsf{FAQ}^\neg$ when restricted to queries with head $\varphi()$, which we denote as $\mathsf{SumProd}^\neg$. To show these bounds, we will use weaker conjectures than the ones used in the previous section.

Conjecture 6.6 (Minimum-Weight k-Clique). *There is no algorithm that computes the minimum weight of a k-clique in a edge-weighted graph with n nodes in $O(n^k)$ time.*

Our reduction from Minimum-Weight k-Clique [1] is an application of the clique embedding power technique introduced in [13].

Theorem 6.7. *Assuming Conjecture 6.6, a $\mathsf{SumProd}^\neg$ φ over the tropical semiring can be solved in linear time iff φ is signed-acyclic.*

[6] A hypergraph is said to be *k-uniform* if every hyperedge contains exactly k vertices.

Over the counting ring, we can instead show that any lower bound for counting Boolean CQs transfers immediately to SumProd$^\neg$.

Theorem 6.8. *Suppose that no linear-time algorithm can count the solutions of a non α-acyclic Boolean CQ. Then, there is no linear-time algorithm for a* SumProd$^\neg$ *query over the counting ring that is not signed-acyclic.*

6.3 An Unconditional Lower Bound for FAQ$^\neg$

Finally, we show an unconditional lower bound that provides some evidence on the necessity of the inverse Ackermann factor in the runtime of Theorem 1.2.

The lower bound is based on the additive structure of the underlying semiring for FAQ$^\neg$, i.e. a commutative semigroup with operator $\oplus$. It uses the *arithmetic model of computation* [12, 26], which charges one unit of computation for every $\oplus$ operation performed, while all other computation is free. Essentially, the computation can be viewed as a sequence of instructions of the form $z_i = az_j \oplus bz_k$, where $\{z_i\}_i$ form an unbounded set of variables. Moreover, this sequence should be agnostic to the actual values of the semiring. The only thing we need is that the semigroup is *faithful* [12, 26], meaning that for every $T_1, T_2 \subseteq \{1, 2, \ldots, n\}$, and integers $\delta_i, \delta'_j > 0$, $\bigoplus_{i \in T_1} \delta_i \cdot a_i = \bigoplus_{j \in T_2} \delta'_j \cdot a_j$ cannot be an identity for all $a_1, a_2, \ldots, a_n \in S$ unless $T_1 = T_2$. This is essentially saying that there is no "magical shortcut" to compute the sums.

Theorem 6.9. *Under the arithmetic model of computation, any constant delay enumeration algorithm (in data complexity) for*

$$\varphi(x) = \bigoplus_y A(x) \otimes B(y) \otimes \mathbb{1}_{\neg R}(x, y)$$

on factors A, B each of size n and factor R of size 2m must require $\Omega(m \cdot \alpha(m, m))$ preprocessing time for every $m \leq n$.

7 DIFFERENCE OF CQS

As an application of our results, we consider the class of queries of the form $Q_1 - Q_2$, where Q_1, Q_2 are full CQs with the same set of free variables. It is shown in a recent paper [17] that $Q_1 - Q_2$ can be computed in time $O(|\mathcal{D}| + \mathsf{OUT})$, where OUT is the output size of $Q_1 - Q_2$, if Q_1 is α-acyclic and $Q_1 \wedge R_e$ is α-acyclic for every R_e in Q_2. We use our main theorem to strengthen this result by providing a constant-delay enumeration guarantee.

Theorem 7.1. *Let $Q = Q_1 - Q_2$, where Q_1, Q_2 are full CQs over the same set of free variables. If Q_1 is α-acyclic and $Q_1 \wedge R_e$ is α-acyclic for every R_e in Q_2, then the result Q can be enumerated with constant delay after $O(|\mathcal{D}|)$ preprocessing time.*

Proof of Theorem 7.1. Following [17], we can write Q as follows:

$$Q = \bigcup_{R_e \in Q_2} \Big(\bigwedge_{S_e \in Q_1} S_e \wedge \neg R_e \Big)$$

Hence, Q can be viewed as a union of CQ$^\neg$, each CQ$^\neg$ of the form $Q_1 \wedge \neg R_e$. Since Q_1 is α-acyclic and $Q_1 \wedge R_e$ is α-acyclic, $Q_1 \wedge \neg R_e$ is signed-acyclic and free-connex (trivially, since it is full) and thus by Theorem 1.1 can be enumerated with linear-time preprocessing and constant delay. We can now apply the Cheater's Lemma [10] to obtain linear-time preprocessing and constant-delay enumeration for Q. □

As a corollary, we obtain the following generalization to differences of non-full CQs.

Corollary 7.2. *Let $Q = Q_1 - Q_2$, where Q_1, Q_2 are CQs with the same set of free variables. If Q is difference-linear (see [17] Definition 2.3), then the output of Q can be enumerated with constant delay after $O(|\mathcal{D}|)$ preprocessing time.*

Proof of Corollary 7.2. From [17], the property of being difference-linear implies a linear-time reduction from Q to a difference of CQs $Q_1' - Q_2'$ (i.e. $Q = Q_1' - Q_2'$) such that Q_1', Q_2' are full CQs, and satisfy the properties of Theorem 7.1. We then simply apply Theorem 7.1. □

8 CQ¬ BEYOND LINEAR TIME

Now, what can we reason for evaluating a CQ¬ if it is not signed-acyclic? Lanzinger [20] defined a width measure of a CQ¬ called *nest-set width* (nsw) and showed that a Boolean CQ¬ with bounded nsw can be evaluated in polynomial time (combined complexity). One consequence of (2) is that we can obtain in a straightforward way a much tighter upper bound in terms of data complexity.

Definition 8.1. Let Q be a CQ¬ with signed hypergraph $\mathcal{H} = ([n], \mathcal{E}^+, \mathcal{E}^-)$. Then, β-#subw$(Q) = \max_{S \subseteq \mathcal{E}^-} \#\mathsf{subw}(Q_S)$, where Q_S is the CQ associated with hypergraph $([n], \mathcal{E}^+ \cup S)$ and #subw(Q) is the *counting* version of the submodular width defined by Abo Khamis et al. [18].

As shown in [2, 18], for any Boolean CQ Q, we can evaluate $\#Q$ in time $\widetilde{O}(|\mathcal{D}|^{\#\mathsf{subw}(Q)})$, where $\widetilde{O}$ hides a polylogarithmic factor in $|\mathcal{D}|$. Combining this with (2), we have:

Theorem 8.2. *Let Q be a Boolean* CQ¬*. Then, we can evaluate $\#Q$ (and thus Q) in time $\widetilde{O}(|\mathcal{D}|^{\beta\text{-}\#\mathsf{subw}(Q)})$.*

From Lanzinger [20], we know that β-#subw(Q) is always at most its nest-set width (see [20] Theorem 4.1); moreover, there are queries were nest-set width is unbounded but β-#subw is bounded (see [20] Lemma 4.3).

9 CONCLUSION

CQ¬ is a fundamental class of relational queries and yet recent literature [6, 9, 17, 20] has demonstrated that its complexity is far from being well-understood. This paper has made an initial foray into a novel way of interpreting CQ¬ from the perspective of semiring and FAQs [19]. We presented a constant-delay enumeration algorithm for the class of free-connex signed-acyclic FAQ¬, after linear preprocessing (modulo a surprising inverse Ackermann factor), and showed conditional and unconditional lower bounds for FAQ¬ out of this class. As for semirings of special structures, the algorithm needs no extra inverse Ackermann overhead and such semirings include the Boolean (i.e. CQ¬), tropical and counting semiring. We leave as an intriguing open question the parameterized complexity of general CQ¬ and FAQ¬.

REFERENCES

[1] Amir Abboud, Karl Bringmann, Holger Dell, and Jesper Nederlof. More consequences of falsifying SETH and the orthogonal vectors conjecture. In *STOC*, pages 253–266. ACM, 2018.

[2] 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*, PODS '17, page 429–444, New York, NY, USA, 2017. ACM.

[3] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[4] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020.

[5] Johann Brault-Baron. A negative conjunctive query is easy if and only if it is beta-acyclic. In *CSL*, volume 16 of *LIPIcs*, pages 137–151. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012.

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

[7] Johann Brault-Baron. Hypergraph acyclicity revisited. *ACM Comput. Surv.*, 49(3), Dec 2016.

[8] Andries E Brouwer and Antoon W. J. Kolen. A super-balanced hypergraph has a nest point. *Technical report, Math. centr. report ZW146, Amsterdam*, 1980.

[9] Florent Capelli and Oliver Irwin. Direct access for conjunctive queries with negation. *27th International Conference on Database Theory, ICDT 2024, March 25-28, 2024, Paestum, Italy*, pages 13:1–13:20, 2024.

[10] Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. *ACM Trans. Database Syst.*, 46(2), May 2021.

[11] B. Chazelle and B. Rosenberg. Computing partial sums in multidimensional arrays. In *Proceedings of the Fifth Annual Symposium on Computational Geometry*, SCG '89, page 131–139, New York, NY, USA, 1989. ACM.

[12] Bernard Chazelle and Burton Rosenberg. The complexity of computing partial sums off-line. *Int. J. Comput. Geom. Appl.*, 1(1):33–45, 1991.

[13] Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. The fine-grained complexity of boolean conjunctive queries and sum-product problems. In *ICALP*, volume 261 of *LIPIcs*, pages 127:1–127:20, 2023.

[14] François Le Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303. ACM, 2014.

[15] Etienne Grandjean and Louis Jachiet. Which arithmetic operations can be performed in constant time in the RAM model with addition? *CoRR*, abs/2206.13851, 2022.

[16] Juris Hartmanis and Janos Simon. On the power of multiplication in random access machines. In *SWAT*, pages 13–23. IEEE Computer Society, 1974.

[17] Xiao Hu and Qichen Wang. Computing the difference of conjunctive queries efficiently. *Proc. ACM Manag. Data*, 2023.

[18] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, and Maximilian Schleich. Functional aggregate queries with additive inequalities. *ACM Trans. Database Syst.*, 45(4), 2020.

[19] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. In Tova Milo and Wang-Chiew Tan, editors, *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 13–28. ACM, 2016.

[20] Matthias Lanzinger. Tractability beyond ß-acyclicity for conjunctive queries with negation. In *PODS 2021*, pages 355–369. ACM, 2021.

[21] Ran Raz. On the complexity of matrix product. *SIAM J. Comput.*, 32(5):1356–1369, 2003.

[22] Janos Simon and Mario Szegedy. On the complexity of RAM with various operation sets. In *STOC*, pages 624–631. ACM, 1992.

[23] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898, 2012.

[24] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94. IEEE Computer Society, 1981.

[25] Andrew Chi-Chih Yao. Space-time tradeoff for answering range queries (extended abstract). In *STOC*, pages 128–136. ACM, 1982.

[26] Andrew Chi-Chih Yao. On the complexity of maintaining partial sums. *SIAM J. Comput.*, 14(2):277–288, 1985.

[27] Hangdong Zhao, Austen Z. Fan, Xiating Ouyang, and Paraschos Koutris. Conjunctive queries with negation and aggregation: A linear time characterization. *CoRR*, abs/2310.05385, 2023.

Received June 2023; revised August 2023; accepted September 2023