Enumerating answers of acyclic conjunctive queries with self-joins (Report)

Clément Rouvroy
ENS Paris
rouvroy@phare.normalesup.org

May 2, 2025

Abstract

This work studies the enumeration of answers to conjunctive queries. Research on this topic typically focuses on conjunctive queries without self-joins. We want to work towards a dichotomy that allows us to know if a conjunctive query with self-joins can be enumerated with optimal time guarantees: linear time before the first answer and constant delay between answers. In this work, we have found a sufficient and necessary conditions. However, we have examples that we can not classify and can lead to new work on the subject.

DISCLAIMER: This report is yet to be polished. It is composed of the work I've made during summer 2024 at INRIA, supervised by Nofar Carmeli and David Carral, and during the first semester 2024-2025 at ENS supervised by Luc Segoufin.

1 Introduction

Querying data efficiently is one of the main topics of database theory. One of the most common ways of storing data is relational databases. For example, you can have a database that represents a hospital's patients' files, treatments, surgeries, and rooms. You can then query this database to get all the rooms of the patients who have surgery in the next 24 hours to be sure not to feed them.

Many query languages exist, and many are expressive but may be too hard to study. Hence, researchers often focus on a language called conjunctive queries which is powerful enough to express many queries and simple enough to be studied and to have some meaningful results.

Queries can have a number of results that is polynomial in the size of the input database, hence we need to define what is an efficient query evaluation algorithm. If we compute all results and then return all of them at once we cannot hope for a linear time algorithm, hence we prefer to print the answers one by one and to measure the delay between two prints. The best we can hope for is to begin with a linear time computation (required to read the input) and then enumerate the answers with a constant delay between them. This defines the complexity class $\mathrm{CD} \circ \mathrm{Lin}$.

We have a dichotomy to decide whether a conjunctive query without self-join is in $\mathrm{CD} \circ \mathrm{Lin}$ [1]. From this dichotomy, we know that the two main difficulties in the

enumeration of a conjunctive query are cyclicity and projections. For conjunctive queries with self-joins, we know that this dichotomy does not extend if the query is cyclic [6]. That is, if a conjunctive query with self-joins is cyclic, it is not always hard. We have found that the dichotomy does not extend to acyclic queries with projections either. Hence, we want to make progress toward a dichotomy to decide whether acyclic conjunctive queries with self-join are in CD $\circ$ Lin.

In this work, we focus on conjunctive queries with self-joins that only have atoms of arity at most 2. This choice allows us to have a clear definition of the structure whose presence might make the query hard, called a free-tree. We would like to remove those structures and replace each of them with only one atom without projections (of arity possibly more than 2), and we define patching as a way of doing so. Using this idea of patching we find a sufficient condition. We have found two necessary conditions, however, we also build queries that are not captured by any of our conditions (neither sufficient nor necessary).

2 Preliminaries

A relational schema is a set of relation names with an arity for each of them. In this work, we will always assume implicitly the existence of a relational schema σ. A database D is a finite relational structure consisting of σ, a finite domain $\mathrm{Dom}(D)$, and for each relational symbol R of σ of arity r, a subset R^D of $\mathrm{Dom}(D)^r$. If $\vec{y} \in R^D$, then $R(\vec{y})$ is called a fact of D. If $\vec{y}$ is a vector of variables and R is a relation name of arity $|\vec{y}|$, then $R(\vec{y})$ is called an atom.

A conjunctive query (CQ) is an expression of the form $q(\vec{x}) \leftarrow \exists \vec{z}.\psi(\vec{x}, \vec{z})$ where ψ is called the body of the query and is a conjunction of atoms using variables in $\vec{x}, \vec{z}$, i.e. $\psi(\vec{x}, \vec{z}) = \bigwedge_{i=1}^{k} R_i(\vec{y_i})$. If $q(\vec{x})$ has for body ψ, we often write $R_i(\vec{y_i}) \in q$ which means that $R_i(\vec{y_i})$ is in the conjunction that defines ψ. The variables in $\vec{x}$ (resp. $\vec{z}$) are free (resp. quantified). In a CQ, all free variables should appear in at least one atom. We use $\mathrm{Vars}(q)$ to denote $\vec{x}, \vec{z}$ and $\mathrm{Free}(q)$ to denote $\vec{x}$. Often we see these vectors as a set and write $x \in \vec{x}$ which means $\exists i.\vec{x}_i = x$. A query q is said to be full (resp. boolean) if $\vec{z} = \emptyset$ (resp. $\vec{x} = \emptyset$). From now on, we will always use $x_1, x_2, x_3, \ldots$ for free variables and $z_1, z_2, z_3, \ldots$ for quantified variables.

An assignment for a query q over a database D is a function $\mu : \mathrm{Vars}(q) \to \mathrm{dom}(D)$ s.t. for every $R(\vec{y}) \in q$, $R(\mu(\vec{y}))$ is a fact of D. A solution to a CQ q is a $\mu : \mathrm{Free}(q) \to D$ such that there exists an assignment μ' for q over D s.t. $\mu'_{|\mathrm{Free}(q)} = \mu$. The evaluation of a CQ q over a database D, denoted $q(D)$ consists of returning all solutions of q over D. The enumeration of a CQ q over D consists of printing all solutions in $q(D)$ one by one without repetition.

A homomorphism from $q(\vec{x})$ to $q'(\vec{x'})$ is a function $h : \mathrm{Vars}(q) \to \mathrm{Vars}(q')$ such that for all atoms $R(\vec{y_i}) \in q$, there exists an atom $R(\vec{y_i}') \in q'$ with $h(\vec{y_i}) = \vec{y_i}'$. Let $h : \mathrm{Vars}(q) \to \mathrm{Vars}(q)$, if h is a homomorphism from q to q, then it is an endomorphism.

Queries have an associated hypergraph. This hypergraph has the variables for vertices and for each atom $R(\vec{x})$ there is a hyperedge of set $\{y \in \mathrm{Vars}(q) \mid y \in \vec{x}\}$. We color hyperedges with one color for each atom. Hyperedges of similar colors mean that they are using the same relation symbol.

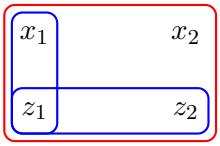

Figure 1: $q(x_1, x_2) \leftarrow \exists z_1, z_2.R(x_1, z_1), R(z_1, z_2), T(x_1, x_2, z_1, z_2)$

See Fig. 1 for an example of a hypergraph associated to a query. Note that in this representation we lose the order in the atom. We only use this representation when we are speaking of cyclicity. When we want to draw queries to find solutions to them, we do not use this representation but an oriented hypergraph, just as in Fig. 4. A CQ is acyclic if its associated hypergraph is α-acyclic, otherwise it is cyclic. A CQ q is free-connex if its hypergraph is α-acyclic when we add to it a hyperedge that contains all free variables.

A path in a hypergraph is a finite sequence of vertices $u_1, \ldots, u_k$ such that for each $1 \leq i \leq k-1$, u_i and u_{i+1} are neighbors. Two vertices (u, v) are neighbours if there exists a hyperedge h such that $\{u; v\} \subseteq h$. A free-path in a CQ q is a chordless path in the hypergraph of the query that only uses quantified variables except for the endpoints that are free, i.e. $(x_1, z_1, \ldots, z_n, x_2)$ with $n \geq 1$. A CQ has a self-join if it has two atoms that use the same relational symbol.

We are using the RAM model for all computations and complexity results. When we study problems on a query we can either look at their complexities regarding q and D (combined complexity) or only D (data complexity): all this work is placed in data complexity. We have $q \in \text{Enum}(f, g)$ if there is an algorithm that enumerates all solutions of q with an $O(g(|D|))$ delay after an $O(f(|D|))$ time precomputation. Note that Enum is a complexity class that takes in consideration the number of solutions, but not just by asking a total running time of $O(f(|\text{input}|)+g(|\text{solution}|))$, it also implies that we start by using $O(f(|\text{input}|))$ time, and then we will print every $O(g(|\text{input}|))$. That is a better complexity class because in the case where $g = o(f)$, we can only do smaller computations between answers once we have started to enumerate. Hence, Enum is more suitable if we look at complexity because we can classify problems that may take time to have a first answer but then print the other with a smaller delay.

For example, $q \in \text{Enum}(\mathit{Linear}, \mathit{Constant})$, also written $q \in \text{CD} \circ \text{Lin}$, with $\text{Linear} : \begin{cases} \mathbb{N} \to \mathbb{N} \\ n \mapsto n \end{cases}$ and $\text{Constant} : \begin{cases} \mathbb{N} \to \mathbb{N} \\ n \mapsto 1 \end{cases}$ means that we are able for any database D, after an $O(|D|)$ computation, to enumerates all the solutions of $q(D)$ with an $O(1)$ delay between each answer and without repeting of solutions.

A common linear pre-computation is the removal of dangling tuples (atoms that are not used in any solutions). This can be done in linear time on acyclic queries according to [2].

Lemma 2.1 (Cheater's lemma [6]). *Let q be a CQ. There is an equivalence between:*

- *q is in $CD \circ Lin$*
- *There is an algorithm that enumerates all solutions of q with constant delay,*

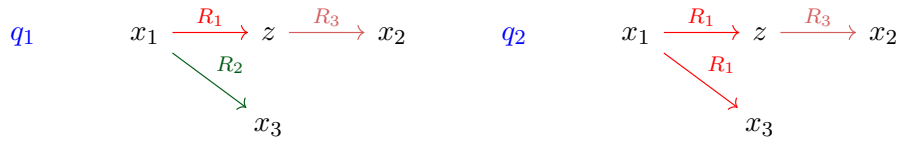

Figure 2: (R) An easy acyclic not free-connex CQ with self-join, and (L) its associated join-free CQ

except a constant number of times where the delay is linear. This algorithm can print with duplicates, but there is a constant c such that no solutions is outputed more than c times.

We will often use three complexity hypotheses:

- sBMM: Given two boolean matrices A, B represented by the list of their non-zero entries, one can not compute AB in $O(m)$ with m being the number of 1 in the matrices A, B and AB.
- sHyperclique: For all $k \geq 3$, it is not possible to determine the existence of a k-hyperclique in a $(k-1)$-uniform hypergraph with m hyperedges in time $O(m)$.
- VUTD: for any $\alpha \in]0;1]$, given a tripartie graph V_1, V_2, V_3 such that $|V_1| = n$,$|V_2| = |V_3| = O(n^\alpha)$, one can not detect if there is a triangle in V_1, V_2, V_3 in time $O(n^{1+\alpha})$.

Theorem 2.2 ([1], [4]). *Assuming sBMM and sHyperclique we have that a CQ without self-join is in CD $\circ$ Lin if and only if it is acyclic free-connex.*

This work will focus on a classification for the class CD $\circ$ Lin for acyclic query with self-joins. Hence, in this work, a query is easy, if it is in this class and hard otherwise.

3 Proving easiness

3.1 An easy non-free-connex CQ

Query q_2 of Fig. 2 is an example that shows that self-joins affect the known dichotomy for conjunctive queries without self-joins. We can see that q_1 is hard using Theorem 2.2 because it has no self-join and is acyclic but not free-connex. However, q_2 has a self-join, so we can not come to the same conclusion. We have that:

Lemma 3.1. *q_2 has a free-path, but it is in CD $\circ$ Lin.*

Proof. Consider the query $q_2'(x_1, x_2, x_3) \leftarrow R(x_1, x_3), R(x_3, x_2)$ that is the same as q_2 but z has been replaced by x_3. One can verify easily that those claims hold:

- q_2' is in CD $\circ$ Lin, thanks to proposition 3 of [5].
- For any solutions $(a_1, a_2, a_3) \in q_2'(D)$, $(a_1, a_2, a_3) \in q_2(D)$.

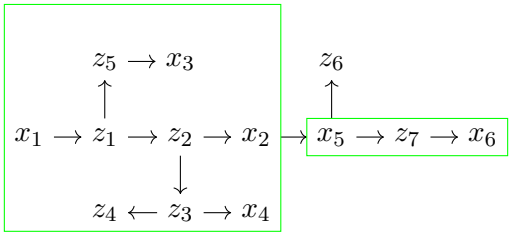

Figure 3: Example of a query with two free-trees. Remember that x_i are free and z_i are quantified.

- For any (a_1, a_2) solution of the free-path x_1, z, x_2 in q_2, there exists a' such that (a_1, a', a_2) is a solution to $q_2'(D)$.

Combining all these claims, one can derive an algorithm that starts by doing the linear pre-computations to enumerate q_2', then enumerate it with a constant delay. For each answer (a_1, a_2, a_3) of $q_2'(D)$, the algorithm keep in memory (a_1, a_2) and prints (a_1, a_2, a_3) (hence it prints with constant delay). Once the algorithm has finished to enumerate $q_2'(D)$, it enumerates all the (a_1, a_2) that it kept in memory and for all $R(a_1, a_3) \in D$, it prints $(a_1, a_2, a_3) \in q_2(D)$. The whole algorithm and proof can be found in appendix Lemma .1.

□

According to Lemma 3.1, q_2 is an acyclic not free-connex CQ with self-join that is in CD $\circ$ Lin.

3.2 What is hard in a query with self-join ?

According to Theorem 2.2, an acylic conjunctive query without self-join can only be hard because it has a free-path. Lemma 3.1 demonstrates that free-path can be overcome for queries with self-joins. Hence, we need a new hard structure to study: we have found free-trees, that are motivated by Proposition 3.8.

Let q and q' be two CQs, q' is a subquery of q if all atoms of q' are in q. Let q' be a subquery of q, q' is maximal for some property P if it is maximal for the inclusion of atoms over all subquery of q that satisfies P.

Definition 3.2 (Binary-atoms)**.** *A CQ is binary-atoms if every atom in it is of arity* ≤ 2.

Definition 3.3 (Free-tree)**.** *Let q be an acyclic CQ, a free-tree is a subquery of q obtained in such a way:*

- *Take S, a maximal set of connected quantified variables.*
- *If there are at least two free variables that are connected to S, then it is associated with a free-tree which is the maximal subquery of q that uses only variables in S and free variables connected to S by an atom.*

One can find two examples of free-trees on Fig. 3. The main intuition behind free-trees is that two free-trees $T_1 \neq T_2$ of a CQ q do not share any quantified variables

compared to a free-path. Hence, if we have all the informations needed to enumerate the free-trees of a query q, it might be easier to concatenate all these informations into an algorithm. This intuition can be formalized into Proposition 3.8, that will need some definitions.

Definition 3.4 (Query restricted to a subquery)**.** *Let q be a conjunctive query, and q' a subquery of q. $q_{|q'}$ denotes the query q where every variables are quantified, except those free in q'.*

Lemma 3.5 (Extension of restricted solution)**.** *Let q be an acyclic conjunctive query, and q' be the restriction of q to T a free-tree. If dangling tuples have been removed of the database, then for each solution μ' of q', one can find in $O(1)$ time a solution μ of q such that $\mu_{|Free(q')} = \mu'$.*

Proof Sketch. Let $d_T : \mathrm{Vars}(q) \to \mathbb{N}$ denotes the distance from a variable v to T in the hypergraph associated to q. One can proof by induction $P(n)$: "If all variables with $d_T(v) < n$ has received a value in μ, then all variables with d_T equals n can receive a value in $O(1)$ by looking at the only neighbors they have in $\{v \in \mathrm{Vars}(Q) \mid d_T(v) < n\}$.", where the unicity of the neighbor comes from the acylicity of q. This induction holds because dangling tuples are removed from the database, hence if we have a partial solution it can always be completed to a full solution of q.

Corollary 3.6 (Enumeration of restricted solution)**.** *Let q be an acyclic conjunctive query, and q' be the restriction of q to T a free-tree. If dangling tuples have been removed of the database and one can enumerate q' in $CD \circ Lin$, then it can be done before an algorithm that enumerates q in $CD \circ Lin$ without changing its complexity.*

Proof. One can enumerate q' in $\mathrm{CD} \circ \mathrm{Lin}$ and print a unique solution of q every time using Lemma 3.5. We conclude using Cheater's Lemma 2.1. □

This corollary motivates the following definition, which will be useful when we will have one algorithm per free-tree :

Definition 3.7 ($\mathrm{CD} \circ \mathrm{Lin}_q$)**.** *Let q, q' be two conjunctive queries. We say that $q' \in CD \circ Lin_q$ if there is an algorithm in $CD \circ Lin$ that enumerates all solutions of q' but sometimes enumerate some solutions of q (it resets the delay and also is without duplicates).*

Looking at Corollary 3.6, one could use algorithms to enumerate the free-trees solutions and store them before printing all solutions of the original query. This is the idea of the following proof.

Proposition 3.8 (Tracting query from free-trees)**.** *Let q be a binary-atom, acyclic, conjunctive queries with self-joins that has free-trees $T_1, \ldots, T_k$. If $q_{|T_1}, \ldots, q_{|T_k}$ are in $CD \circ Lin_q$, then q is in $CD \circ Lin$.*

Proof. Let $T_1, \ldots, T_k$ be the free-trees of q and let $A_1, \ldots, A_k$ be the algorithms that enumerates $q_{|T_1}, \ldots, q_{|T_k}$ in $\mathrm{CD} \circ \mathrm{Lin}_q$.

One can execute $\mathcal{A}_1, \ldots, \mathcal{A}_k$ to build k fresh relation names $R_{T_1}, \ldots, R_{T_k}$ of arity $|\mathrm{Free}(T_1)|, \ldots, |\mathrm{Free}(T_k)|$ and fill them with facts that are exactly the solutions of

Algorithm 1 *Enumerate*(q, D)

1: Removing dangling tuples of D
2: $R_{T_1}, \ldots, R_{T_k} \leftarrow$ *Atoms for* $T_1, \ldots, T_k$ using $\mathcal{A}_1, \ldots, \mathcal{A}_1$
3: $q' \leftarrow$ copy of q
4: **for** $1 \leq i \leq k$ **do**
5: **for** $R(\vec{y}) \in T_i$ **do**
6: Remove $R(\vec{y})$ from q'
7: **end for**
8: Add $R_{T_i}(\text{Free}(T_i))$ to q'
9: **end for**
10: Do precomputation for enumerating q'
11: Enumerate q'

$T_1, \ldots, T_k$ that can be extended to a solution of q. This can be done with respect to CD $\circ$ Lin because we apply $k = O(1)$ times a CD $\circ$ Lin algorithm according to Corollary 3.6 (this algorithm will have at most $2k$ duplicates of solution of q). From the correctness of this construction, q has the same solutions as the query q' that is the same as q, but where we removed every atom of the free-trees to replace them with $R_{T_1}, \ldots, R_{T_k}$.

This CQ q' is acyclic: if it has a cycle $v_1, \ldots, v_k$ that uses edges $e_1, \ldots, e_k$, at least one edge corresponds to an R_{T_i} (otherwise the cycle is in q), but we can replace this edge by many atoms that were initially in T_i. Using this process we build a cycle for q which is supposed to be acyclic, by contradiction q' is acyclic.

q' is also free-connex: no free-path can use an R_{T_i} as an edge because they are only on free variables, hence if it has a free-path, q has the same free-path, but then it is part of a free-tree and has been removed to be part of an R_{T_i}. Hence, q' is acyclic free-connex and so is in CD $\circ$ Lin according to Theorem 2.2. Then we can use Algorithm 1 to enumerate q.

Algorithm 1 is sound assuming that R_{T_i} has been built correctly,but $\mathcal{A}_i$ are supposed to be sound and complete. We need to prove that Algorithm 1 is complete. Suppose that the algorithm misses a solution μ of q, then it can only be because there is at least one i such that $\mu_{|T_i}$ is not in R_{T_i}, but this is a solution of T_i that can be extended to a solution of q and, by hypothesis, $\mathcal{A}_i$ is complete, then it should be in R_{T_i}. Hence, the algorithm is complete.

For time complexity, there are three parts:

- On line 1, the algorithm takes $O(|D|)$ time to remove the dangling tuples according to [2].
- On line 2, the algorithm does k calls to algorithms that are constant delay. Combining a constant number of constant delay algorithms is still a constant delay algorithm, so this first step is a constant delay algorithm as $k = O(q) = O(1)$.
- The for loop of lines 3 to 8 iterates through $k = O(1)$ free-trees and for each i it iterates through each atom of T_i ($O(1)$) to do an $O(1)$ time computation,

it finally does an $O(1)$ time computation, hence we have $O(|q|^2 + |q|) = O(1)$ time for this whole loop. The pre-computation to enumerate q is linear for $|D|$ and the number of solutions printed until there (because we have added atoms to the database) according to [3]. The enumeration is with $O(1)$ delay according to [3]. Hence, the three last steps witness a constant delay algorithm for q that will enumerate all solutions of q over D.

Finally, using cheater's Lemma 2.1, we have a CD $\circ$ Lin algorithm for q. □

Corollary 3.9 (Enumeration with built relations)**.** *In the proposition Proposition 3.8, one can replace having algorithms $A_1, \ldots, A_k$ by having direct access to $R_1, \ldots, R_k$, built in the proof.*

Note that this proposition may not extend to free-pathes, in section 4 we have an open-example where all free-pathes are easy to enumerate but we don't know if q is easy. On example Fig. 2, q_2 has one free-tree (x_1, z, x_2) that is its only free-tree. The query q_2' defined in the proof of Lemma 3.1 allows to enumerate $q_{|(x_1,z,x_2)}$ and so q_2 was easy. This proposition motivates the search for a way to enumerate the solutions of $q_{|T}$, our solutions is presented in the next subsection.

3.3 Preimages and patching

Given a conjunctive query q, we need a set of queries to try to enumerate the differents q_T, motivated by Proposition 3.8. In this section we present the set that we have found to be the most interesting, especially thanks to Lemma 3.11.

Definition 3.10 (Preimage query)**.** *Consider some endomorphism h for a CQ q. Then, let $\underline{P_h(q)}$ be the CQ that has:*

- *An atom $R(h(\vec{x}))$ for every atom $R(\vec{x})$ in q*
- *Free variables $\{y \in Vars(h(q)) \mid \exists \underline{x} \in Free(q).h(\underline{x}) = y\}$.*

If we enumerate solutions of a preimage of a CQ q, for each of them we can print a unique solution to q, which is useful to have a CD $\circ$ Lin algorithm as we will have no repetition in the solutions.

Lemma 3.11 (Prehomomorphism lemma)**.** *Let q be a binary-atom acyclic CQ. Let h be an endomorphism of q, if μ_h is a solution to $P_h(q)$, then $\mu_h \circ h$ is a solution to q. Moreover, if $\mu_h^1 \neq \mu_h^2$, then $\mu_h^1 \circ h \neq \mu_h^2 \circ h$.*

Proof. Let μ_h be a solution of $P_h(q)$, it is associated to η_h an assignment of $P_h(q)$ such that $\eta_{h|\text{Free}(P_h(q))} = \mu_h$. Let x be a free variable of q, by definition of a preimage we have $h(x) \in \text{Free}(P_h(q))$, so $(\eta_h \circ h)_{|\text{Free}(q)}$ is a solution of q and it is a solution that only uses η_h over $h(\text{Free}(q)) = \text{Free}(P_h(q))$, hence $\mu_h \circ h$ is a solution to q.

Moreover, let $y \in \text{Free}(P_h(q))$. As y is free in $P_h(q)$, there exists $x \in \text{Free}(q)$ such that $h(x) = y$, hence $(\mu_h \circ h)(x) = \mu_h(y)$, i.e. the value of y propagate to the solution of q. Then, for two different solutions of $P_h(q)$, there is at least one y that has a different value in each solution, hence the solution associated is different. □

This lemma is useful because we are looking for CD $\circ$ Lin enumeration and thanks to this theorem, we can only loop on solution vectors of preimages. We will use preimage query to patch free-trees, however sometimes a preimage can contain too much information to patch a free-tree (as will be seen in Fig. 5, a bit after). Hence, we need a way to remove some non-essential information of a preimage query, as we only need the part corresponding to the free-tree:

Definition 3.12 (Deactivated query)**.** *Let q be a CQ. A deactivated query q' of q is a query with the same body as q but with $Free(q') \subseteq Free(q)$.*

We need to be prudent with deactivated query because the motivation of preimage query was Lemma 3.11. Fortunately, the following lemma can be composed with Lemma 3.11 to have the same benefits:

Lemma 3.13 (Deativated preimage query)**.** *Let q be an acyclic CQ. Let h be an endomorphism of q, let $p := P_h(q)$ and let $\tilde{p}$ be a deactivation of p. Then every solution $\tilde{\mu}$ of $\tilde{p}$ can be extended in $O(1)$ to a solution μ of p if dangling tuples have been removed. Moreover, if $\tilde{\mu}_1 \neq \tilde{\mu}_2$, then $\mu_1 \neq \mu_2$.*

Proof Sketch. Suppose that $\tilde{\mu}$ corresponds to $x_1, \ldots, x_k$ and that we have deactivate $x_{k+1}, \ldots, x_p$. For each $i \in [\![1; p-k]\!]$, one can search for a j such that the distance in the join-tree between x_j and x_i is minimal, then one can do a DFS from the entry $\tilde{\mu}(x_j)$ of x_j to one entry of x_i in the join-tree. This gives a solution μ of p.

If $\tilde{\mu_1} \neq \tilde{\mu_2}$, as extended solutions μ_1 and μ_2 verify $\tilde{\mu_1} = \mu_1$ and $\tilde{\mu_2} = \mu_2$ on $[\![x_1; x_k]\!] \neq \emptyset$, $\mu_1 \neq \mu_2$.

Definition 3.14 (Patched free-tree)**.** *Let q be a binary-atom acyclic CQ. Let q_1, q_2 be two preimages of q and let T_2 be a free-tree of q_2. T_2 is patched by q_1 if q_1 has a subquery T_1 such that:*

1. *There is a homomorphism h from q_1 to q_2 such that $h_{|T_1}$ is an isomorphism from T_1 to T_2*
2. *Every free variables of T_2 is the image of a free variable of T_1.*
3. *q_1 is in CD $\circ$ Lin_q, or q_1 has a deactivated query q_1' that does not change the free variables of T_1 that is in CD $\circ$ Lin_q.*

We will often say that a deactivated preimage q_1' is patching q_2, which means that there exists a preimage q_1 that uses q_1' to patch q_2. Motivated by Proposition 3.8, we will prove the following property:

Proposition 3.15 (Patched by a preimage)**.** *Let q be an acyclic binary-atom CQ that has two preimages q_1 and q_2. Let $\tilde{q_1}$ a deactivation of q_1 that is in CD$\circ$ Lin_q. If q_2 has a free-tree T_2 patched by $\tilde{q_1}$, then one can save $q_{|T_2}(D)$ before doing a CD$\circ$ Lin algorithm for q and still have a CD $\circ$ Lin algorithm for q.*

Proof. We can enumerate $\tilde{q_1}$ to get all solutions of the subquery T_1 of $\tilde{q_1}$ used to patch q_2. While enumerating $T_1(D)$, as every free variable of T_2 is the image of at least one free variable in T_1, we will go through each solution of T_2. We can add all these solutions to D under a fresh relation name. This strategy gives Algorithm 2.

Algorithm 2 *Edit*$(q, q_1, q_2, T_2, T_1, D)$

1: $R_{T_1} \leftarrow \emptyset$
2: Do pre-computations for $A_{\tilde{q_1}}(D)$
3: **for** $\vec{a} \in A_{\tilde{q_1}}(D)$ **do**
4: **if** $\vec{a}$ is a solution from q **then**
5: print $\vec{a}$ and get to next loop iteration.
6: **end if**
7: print a solution to q using Lemma 3.11 and Lemma 3.13
8: $\vec{b} \leftarrow$ solution of T_1 associated to $\vec{a}$
9: **if** $\vec{b}$ gives a solution to T_2 **then**
10: $\vec{c} \leftarrow$ solution of T_2 associated to $\vec{b}$
11: **if** $R_{T_1}(\vec{c}) \notin D$ **then**
12: R_{T_2}.add$(\vec{c})$
13: **end if**
14: **end if**
15: **end for**
16: return R_{T_1}

This algorithm is sound: we will iterate through each solution in $\tilde{q_1}(D)$ (because it is in CD $\circ$ Lin$_q$). Then, the variable $\vec{b}$ will go through all values of T_1 that were part of a solution of $\tilde{q_1}$, and $\vec{c}$ will go through all solutions to T_2 that is the image of a solution of T_1 that can be extended to a solution of $\tilde{q_1}$: To get $\vec{c}$ just use the isomorphism between T_1 and T_2, we have access to all needed values because by definition of deactivation, we do not deactivate variables used to patch free variables.

This algorithm is complete: let μ_2 be a solution of q_2, it is the image of a solution $\tilde{\mu}_1$ of $\tilde{q}_1$, hence $\tilde{\mu_1}_{\restriction T_1}$ will have for image $\mu_{2\restriction T_2}$.

For time complexity, we take initialize an empty list ($O(1)$), and then we do line 3. Line 3 will use a CD $\circ$ Lin$_q$ algorithm, hence the body of the loop of line 3 will alway print either directly a solution of q and wait for the next iteration, or the image of a solution of q associated to $\tilde{q_1}$.

Then the for-loop of line 3 is iterated with $O(1)$ delay (and a constant time with a linear delay) and the body of the loop consists only of $O(1)$ operations because to get solutions of T_1 (resp. T_2) given a solution of $\tilde{q_1}$ (resp. T_1) you only need to take some values in the given solution which is of length $O(|q|) = O(1)$.

Hence, the algorithm will iterate through each solution of $\tilde{q_1}$, print a unique solution of q each time (unique according to Lemma 3.11 combined with Lemma 3.13) except a constant number of times, and will edit D in the same time.

At the end, the algorithm does a constant number of linear computations, prints solutions of q with constant delay after this linear pre-processing, and end return the edited R_{T_1}. □

Consider the query q and two of its preimages as in Fig. 4. One can use $P_i(q)$ to patch $P_h(q)$ and then can use $P_h(q)$ to patch q. Here there is no use of deactivation, however, sometimes it is needed, as depicted in Fig. 5. The proof that the CQ in Fig. 5 is in CD $\circ$ Lin is in Lemma .2.

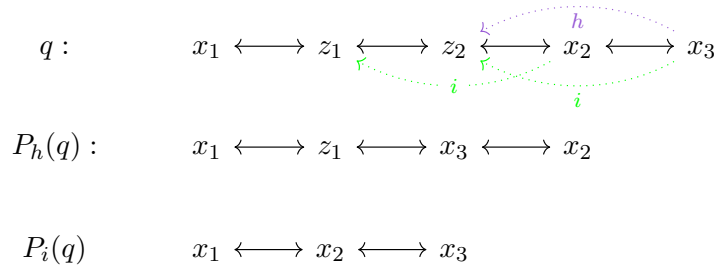

Figure 4: A query and two of its preimages. Remember that x_i are free and z_i are quantified.

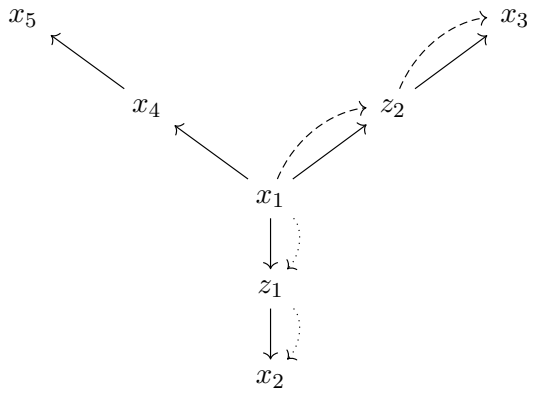

Figure 5: An easy query that needs deactivation. Dashed, dotted, and plain arrows denote three different relation symbols for atoms.

3.4 Sufficient condition based on preimage and free-trees

Now that we have Proposition 3.8 and Proposition 3.15, we would like to combine them to use preimages of a query to enumerate it with respect to CD $\circ$ Lin.

Definition 3.16 (Fully-patched query)**.** *Let q be a binary-atom acyclic CQ, q is fully-patched if every free-tree of q has been patched.*

Theorem 3.17 (Patching sufficient condition)**.** *Let q be a binary-atom acyclic CQ. If q is fully-patched, then $q \in$ CD $\circ$ Lin.*

Proof. Let p be a preimage of q, and we denote $\Psi(p)$ the set of all deactivated preimages sets that fully-patch p. Let φ a function from the deactivated preimages of q to $\mathbb{N}$ such that for p a deactivated preimage of q,

$$\varphi(p) = \begin{cases} 0 & \text{if } p \text{ is acyclic free-connex} \\ \min_{S \in \Psi(p)}(\max_{p' \in S} \varphi(p')) + 1 & \text{if it is fully-patched} \\ -\infty & \text{otherwise} \end{cases}$$

One can prove by induction over $\mathbb{N}$ that $P(n)$: "For all $k \leq n$, each deactivated preimage p such that $0 \leq \varphi(p) \leq n$ has a $\mathrm{CD} \circ \mathrm{Lin}_q$ algorithm" is true (proof of this property is in appendix Lemma .3). Once we know that P holds, if q is fully-patched then $\varphi(q) < +\infty$, then q has a $\mathrm{CD} \circ \mathrm{Lin}_q$ algorithm, i.e. q is in $\mathrm{CD} \circ \mathrm{Lin}$. □

4 Proving hardness

To prove the hardness of CQs we need to make lower-bound assumptions. We will use two assumptions in this work, each comes with its necessary conditions that capture different cases. However, we still have open cases that need a clever approach, as we will see in the last part of this section.

4.1 BMM-Hard

The BMM hypothesis states that we cannot have an algorithm that, given two boolean $n \times n$ matrices A and B, return $A \times B$ in $O(n^2)$ time. We will use the tagging technique presented in [5] to encode boolean matrix multiplication in many queries with self-join, this is the same idea as in [1] but this time for query with self-joins.

Definition 4.1 (Free variable disabled by R)**.** *Let q be an acyclic conjunctive query with atoms of arity at most 2. Let R be a relational symbol of arity one used in an atom of q and let F be a free-path of q. We say that $x \in Free(q) \setminus Vars(F)$ is disabled by R on F if:*

- $\forall y \in Vars(F).R(y) \notin q$,
- $R(x) \in q$.

Let $D(R, F)$ be the set of variables disabled by R on F.

The idea is that a variable disabled by R will not give spurious solutions because we can build a database where $R = \{\bot\}$. As R does not appear on the free-path, it is not a problem to fix it to be $\{\bot\}$.

Definition 4.2 (Unmapped to a free-path)**.** *Let q be a CQ, and let F be a free-path. Let $y \notin Vars(F)$ be a variable of q, it is unmapped to F if $\forall h \in Endo(q), h(y) \notin Vars(F)$.*

A variable y of q is unmapped to a free-path F if y is never mapped to it through any homomorphism.

Definition 4.3 (BMM-Hard)**.** *Let q be an acyclic CQ. It is BMM-Hard if $q \in CD \circ Lin$ breaks the BMM hypothesis.*

Before giving and proving the condition, we need one lemma:

Lemma 4.4. *Let q be an acyclic binary-atom CQ that we want to enumerate over D. Let q' and D' be the query and the database resulting from a tagging technique on q over D. Then, for every solution μ of $q'(D')$, the tagging map induced by μ is an automorphism of q.*

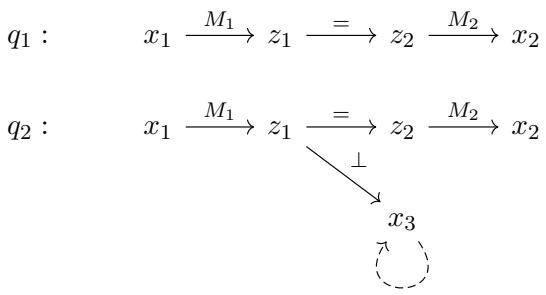

Figure 6: Two examples of BMM-hard queries. Dashed arrows are atoms of another relational symbol. Label on edges indicates the role in encoding BMM of the edge (through tagging technique).

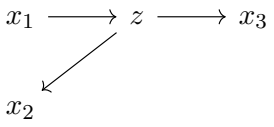

Figure 7: A CQ that is VUTD-Hard but not BMM-Hard

Proposition 4.5. *Let q be an acyclic conjunctive query with an atom of arity at most 2. If there exists a free-path F such that $\forall x \notin Vars(F), \exists R.x \in D(R, F)$ or x is unmapped to F, q is BMM-Hard.*

Proof Sketch. Let M_1 and M_2 be two boolean matrices. Let $F := x_1, z_1, \ldots, z_n, x_2$, the idea is to encode the product on F and to ensure that all other free variables can only be $\perp$ in all solutions. This can be done because free variables outside of the free-path are either in a $D(R, F)$, hence they can be set to $\perp$, or they are unmapped on F. If x is unmapped of F and receives a value tagged by a variable in F, then there is a homomorphism of q that maps x to a variable in F according to Lemma 4.4: impossible. The full proof is available in Proposition .4.

Proposition 4.5 allows to capture a lot of CQs. For an example of application one can look at the two queries in Fig. 6, the meaning of each atom is printed on the edges (the meaning is what we intend it to encode in the tagging technique).

4.2 VUTD-Hard

The Vertex Unbalanced Triangle Detection is, for any $\alpha \in [0; 1]$, given a tripartie graph V_1, V_2, V_3 such that $|V_2| = n$,$|V_1| = |V_3| = O(n^\alpha)$, to detect if there is a triangle in it. The hypothesis is that it cannot be done in $O(n^{1+\alpha})$ time.

Definition 4.6 (VUTD-Hard)**.** *Let q be an acyclic CQ. It is VUTD-Hard if $q \in CD \circ Lin$ breaks the VUTD hypothesis.*

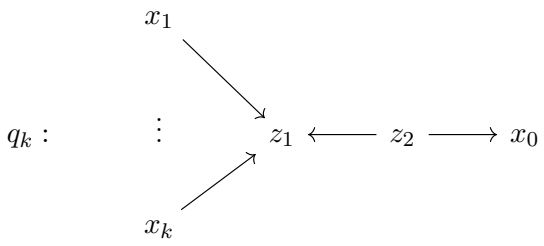

Figure 8: An in-between family of CQs

Some hard CQs do not seem to be BMM-Hard but are VUTD-Hard, an example is given in Fig. 7 (and the proof that it is hard is the core of this subsection).

Proposition 4.7. *Let q be an acyclic binary-atom CQ. If there is a quantified variable z on a free-path that is not the image of a free-variable in any body-homomorphism of q, then q is VUTD-hard.*

Proof Sketch. Let q be an acyclic binary-atom CQ. Let ζ a quantified variable that is the image of no free variable of q for any automorphism of q, i.e. $\forall h : q \to q, \forall x \in \text{Free}(q), h(x) \neq \zeta$.

Let T be the free-tree associated to ζ (it exists because there is a free-path). Fix $x_1 - ... - \zeta - ... - x_2$ one free-path in T that goes through ζ.

The idea is to do an encoding that will tag elements of V_2 only with ζ. Hence, if another free variable receives a value of V_2, from Lemma 4.4, there is a homomorphism of q that maps a free variable to ζ: impossible. The complete proof and encoding is available on Proposition .5

In Fig. 7, the quantified variable z is the image of no free variable and then the query is not in CD $\circ$ Lin according to Proposition 4.7.

4.3 Clique-based hardness

The two previous hardness techniques allow to classify many queries, but some queries seems not be classifiable using only those two. An example is (L) Fig. 8 for $k = 2$, BMM-hardness may not apply because we can not isolate a free-path, and VUTD-hardness may not work because x_0 can be mapped to z_1 and x_1 (or x_2) can be mapped to z_2.

To tackle the family of Fig. 8, we will work with k-clique, but this will work only for $k \leq 3$. From [7], we know that:

Theorem 4.8. *Let $k = 3l + i$ ($(k, i) \in \mathbb{N} \times \{0, 1, 2\}$), and ω is the optimal bound for matrix multiplication ($2 \leq \omega < 2.38$). Let G be a graph with n nodes, we can check if G has a k-clique in $O(n^{\omega l+i})$.*

In particular, for $k = 5$ the bound is $O(n^{\omega+2})$ so if we manage to solve the 5-clique problem in $O(n^4)$ using that a query q is in CD $\circ$ Lin, it is likely that $q \notin$ CD $\circ$ Lin. More generally, we do the following hypotheses:

Definition 4.9 (k-clique hypothesis for $k \leq 5$)**.** *Let $k \in \{1, 2, 3, 4, 5\}$, there is no algorithm that detects a k-clique in $O(n^{k-1})$.*

This is suitable for this lemma:

Lemma 4.10. *The k-star query is a query like $q(x_1, \ldots, x_k) \leftarrow R_1(x_1, z) \ldots, R_k(x_k, z)$ .Let $n = |dom(D)|$. Assuming k-clique hypothesis for $k \leq 5$, no algorithm can answer $q(D)$ for q a k-star query in $O(|Dom(D)|^k)$ for $k \leq 4$.*

Proof. Let $k \leq 4$, let q be the k-star query. Suppose that there is an algorithm $A(q, D)$ that returns $q(D)$ in $O(|\text{Dom}(D)|^k)$ for any D. Let $G = (V, E)$ be a graph of size n. Set $\text{Dom}(D) = V$ and $R = E$. For every solution $(a_1, \ldots, a_k) \in q(D)$ one can check that $\forall (i, j) \in [\![1; k]\!]^2 . i \neq j \Rightarrow (a_i, a_j) \in E$ holds in $O(k^2) = O(1)$. Hence we have a $O(|\text{Dom}(D)|^k) = O(n^k)$ algorithm to detect a $(k + 1)$-clique: impossible. □

Now, we can reduce k-star queries to some queries depicted in Fig. 8:

Lemma 4.11. *For $k \leq 3$, q_k depicted in Fig. 8 is hard.*

Proof. Consider a $k + 1$-star q and a database D. Construct D' that has only one relation R that contains:

- $\forall R_1(u, v).R(\langle x_1, u\rangle, \langle z_1, v\rangle)$ and $R(\langle z_2, v\rangle, \langle z_1, v\rangle)$,
- $\forall i \in [\![2; k]\!].\forall R_i(u, v).R(\langle x_i, u\rangle, \langle z_1, v\rangle)$
- $\forall R_{k+1}(u, v).R(\langle z_2, v\rangle, \langle x_0, u\rangle)$

A solution over this construction is a solution to the original query iff the tagging map is the identity on the free variable, and checking this is $O(1)$ time. We have $O(n^{k+1})$ solutions (as can be verified with a case distinction) so we can do the filtering for each solution to check if there is at least one k-star solution. □

From Lemma 4.11, we know that the queries in Fig. 8 are hard for $k \leq 3$, but we do not know for larger k yet. The only result we have yet is when the degree of the database satisfies some properties (see appendix, Proposition .11).

4.4 The need for lower-bound based on the delay

In all previous examples, we never really used enumeration as a structure that gives answers with a delay to prove hardness, we only use enumeration with a short delay to get all the answers to a query and then use the fact that we have all the answers to come to a problem. The section motivates the use of a (still to be determined) new hypothesis based on the delay and not the total time.

Consider the query q_1 depicted in Fig. 9 and q_2 the same query but without x_3. We don't know if q_1 is easy or hard, but we know that q_2 is BMM-Hard.

Lemma 4.12. *Let D be a database. We can answer $q_1(D)$ in $O(|Dom(D)|^3)$.*

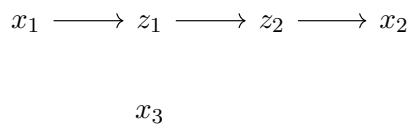

Figure 9: Motivating example. Formally, a CQ can not have a free variable that has no atom that uses it, but as we are dealing with self-joins, we can just add one unary atom to every variable with a fresh relation symbol that we fill with the active domain.

Algorithm 3 Enumerate(q_1, D)

```
1: Build the join-tree associated to q_1(D)
2: Solve (← x_1 → . → x_2)(D) using BMM in O(n^ω), let L be all the results.
3: S ← ∅
4: for (a, b) ∈ L do
5:     for c ∈ Dom(D) do
6:         if R(b, c) ∈ D then S.insert((a, c))
7:         end if
8:     end for
9: end for
10: return S
```

Proof. Consider Algorithm 3. From the join-tree one can build two $|\mathrm{Dom}(D)| \times |\mathrm{Dom}(D)|$ matrices that represent the first and the second arrow. Put a 0 in (i, j) if there is no $R(i, j)$ and 1 otherwise. The product of these two matrices will give exactly $\left\{(a, c) \in \mathrm{Dom}(D)^2 \mid \exists b \in \mathrm{Dom}(D).a \to b \to c\right\}$. This is what line 2 is doing, this process takes the time for the creation of the two matrices, $O(|\mathrm{Dom}(D)|^2)$, and the time to compute the product, $O(|\mathrm{Dom}(D)|^\omega)$. The sum of these steps fits in $O(|\mathrm{Dom}(D)|)^3$ time.

Then the loop iterates through $O(|\mathrm{Dom}(D)|^2)$ elements and then through $|\mathrm{Dom}(D)|$ elements to perform a $O(1)$ time operation (as we have constructed a matrix that indicates if $R(i, j) \in D$ for all $(i, j) \in \mathrm{Dom}(D)^2$ it is easy to check if $R(b, c) \in D$). □

Hence a hardness proof that enumerates q_1 and stores all solutions, and then uses that it has all the solutions need to be clever enough to have strictly less than $O(n^3)$ total solutions. Otherwise, we need a contradiction that uses the fact that we enumerate the solutions with a constant delay. A problem that may (or may not) be easier is the following:

Definition 4.13 (Dom-linear enumeration)**.** *We say that a CQ q is in DomLin $\circ$ Lin if there is an algorithm that, after a pre-computation that takes $O(|D|)$ times, enumerate all the solutions of q in $O(|Dom(D)|)$.*

This problem is related to CD$\circ$Lin enumeration because of the following lemma:

Lemma 4.14 (Link between CD $\circ$ Lin and DomLin $\circ$ Lin)**.** *$q_1 \in CD \circ Lin \Leftrightarrow q_2 \in DomLin \circ Lin$.*

Proof Sketch. q_1 is q_2 but with an extra free variable that allows us to repeat every solution of q_2 at maximum $O(|\mathrm{Dom}(D)|)$ times. We can use this fact in both directions to build a reduction. The complete proof is available in Lemma .6

5 Conclusion

In this work, we defined free-trees as a structure that captures the difficulty in conjunctive queries with self-joins. We defined patching as a tool to express how self-joins affect this difficulty. This definition allowed us to find a first sufficient condition to decide if a CQ with self-joins is in CD $\circ$ Lin.

We have also come to necessary conditions that catch many queries, and we have raised two kinds of CQs that are not classified with our conditions and that could lead to new work on the topic.

References

[1] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In Jacques Duparc and Thomas A. Henzinger, editors, Computer Science Logic, pages 208–222, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[2] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. J. ACM, 30(3):479–513, July 1983.

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

[4] Johann Brault-Baron. De la pertinence de l'énumération : complexité en logiques propositionnelle et du premier ordre. Theses, Université de Caen, April 2013.

[5] Nofar Carmeli and Luc Segoufin. Conjunctive queries with self-joins, towards a fine-grained complexity analysis, 2022.

[6] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration, 2019.

[7] Jaroslav Nešetřil and Svatopluk Poljak. On the complexity of the subgraph problem. Commentationes Mathematicae Universitatis Carolinae, 26(2):415–419, 1985.

Algorithm 4 Enumerate(D) for Lemma 3.1

1: q' is the query $q'(x_1, x_2, x_3) \leftarrow R(x_1, x_2), R(x_2, x3)$.
2: Do pre-computations for q'
3: $G \leftarrow$ an empty hash-table that given a value a can iterate all b such that $R(a, b)$ is in D.
4: $H \leftarrow \emptyset$ an hash-table
5: **for** $(a, b, c) \in q'(D)$ **do**
6: **if** $c \notin H[a]$ **then**
7: $H[a] \leftarrow c :: H[a]$
8: **end if**
9: print (a, b, c)
10: **end for**
11: **for** $a \in H$ **do**
12: **for** $c \in H[a]$ **do**
13: **for** $b \in G[a]$ **do**
14: print (a, b, c)
15: **end for**
16: **end for**
17: **end for**

Annexe 1: Proving Easiness

Lemma .1. *Algorithm 4 is complete, correct, and respects the constraint for q_2 to be in CD $\circ$ Lin.*

Proof. For completeness, let $(a_1, a_2, a_3) \in q_2(D)$. Then there exists $a_z \in \mathrm{Dom}(D)$ such that $\{R(a_1, a_z), R(a_z, a_2)\} \subseteq D$. Hence $(a_1, a_z, a_2) \in q'(D)$, and by completeness of the enumeration of an acyclic full CQ, (a_1, a_z, a_2) will be enumerated on line 5 of Algorithm 4 and $a_2 \in H[a_1]$. Now, by the completeness of the construction of G, when (a_1, a_2) will be enumerated on line 11,12, a_3 will be enumerated on line 13, this sequence will print (a_1, a_2, a_3).

For correctness, let (a_1, a_2, a_3) printed by the algorithm. By construction, we have $a_1 \in H$, $a_2 \in H[a_1]$ and $a_3 \in G[a_1]$. By correctness of the enumeration of an acyclic full CQ, there exists a_z such that $(a_1, a_z, a_2) \in q'(D)$. And by correctness of the construction of G, $R(a_1, a_3) \in D$, hence (a_1, a_2, a_3) is indeed in $q(D)$.

For the time restriction, lines 1 to 4 are pre-computations. Line 1 is $O(1)$ time, line 2 is linear because of Theorem 2.2. Line 3 also is linear, one can read all atoms in the database and fill G accordingly (a way of seeing it is the enumeration of the acyclic full CQ $q(x_1, x_2) : R(x_1, x_2)$ which, after a linear pre-computation, enumerate with constant delay the solutions and there is exactly $O(|D|)$ solutions). Then we enumerate with constant delay in lines 5 to 10: for each loop, we do one print and the body of the loop is $O(1)$ time. Note that there is no duplicate in this part of the algorithm because a 3-uplet (a, b, c) is enumerated at most once. Finally, the last loop chains (11-17) enumerates with constant delay and no repetition, but there may be repetitions with the solutions printed on line 9. As each solution is printed exactly once on line 14, each solution is printed at most twice and then the conditions of Lemma 2.1 are satisfied: $q_2 \in \mathrm{CD} \circ \mathrm{Lin}$.

□

Algorithm 5 Enumerate(q, D)

Remove dangling tuples on D
Solve the query that is like q where every variable is quantified except x_1 and call the set of solutions S.
$H_G, H_R \leftarrow \emptyset, \emptyset$
for $a_1 \in S$ **do**
    $a_2 \leftarrow$ A value for x_2 such that there exists a z_1 such that the (a_1, z_1, a_2) is an assignment to the bottom free-tree.
    $a_5 \leftarrow$ A value for x_5 such that there exists z_2 such that (a_1, z_2, a_5) is an assignment to the right free-tree.
    for $a_3, a_4 \in (x_4 \leftarrow x_3 \leftarrow a_1)(D)$ **do**
        print a_1, a_2, a_3, a_4, a_5
        if (a_1, a_3, a_4) is a green path **then**
            $H_G[a_1] \leftarrow (a_4)$
        end if
        if (a_1, a_3, a_4) is a red path **then**
            $H_R[a_1] \leftarrow (a_4)$
        end if
    end for
end for
for $a_1 \in S$ **do**
    for a_3, a_4 s.t. $a_1 \to a_3 \to a_4$ **do**
        for $a_2 \in H_R[a_1]$ **do**
            for $a_5 \in H_G[a_1]$ **do**
                print $(a_1, a_2, a_3, a_4, a_5)$
            end for
        end for
    end for
end for

Lemma .2. *The CQ in Fig. 5 is in CD ∘ Lin.*

Proof. The proof is an application of theorem Theorem 3.17. If one wants to see an algorithm, he is in Algorithm 5, but it will not be used here. Consider the two deacatived preimages $\tilde{P}_h(q_1)$ and $\tilde{P}_i(q_1)$ of Section 5. h is the homomorphism that maps (x_4, x_5) to (z_1, x_2) and i is the homomorphism that maps (x_4, x_5) to (z_2, x_3). If we do not deactivate the preimages, we still have free-trees in the preimages and we can not patch them. But with the deactivation depicted, both queries are easy to enumerate. Moreover, $\tilde{P}_h(q_1)$ is patching the free-tree (x_1, z_1, x_2) and $\tilde{P}_i(q_1)$ is patch the free-tree (x_1, z_2, x_3).

Hence, q_1 is fully-patched, hence $q_1 \in \mathrm{CD} \circ \mathrm{Lin}$ according to Theorem 3.17.

□

Lemma .3 (Patching sufficient condition - Induction)**.** *The induction holds of Patching sufficient condition holds.*

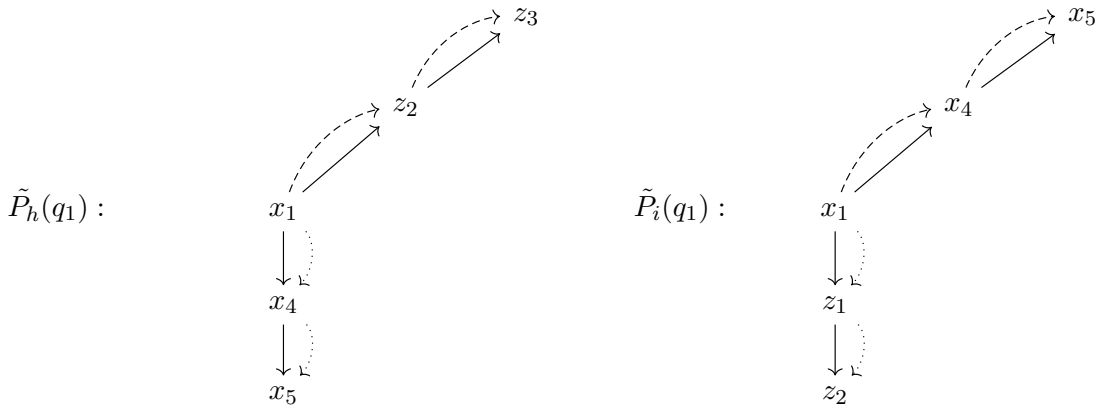

Figure 10: Two deactivated preimage of q.

Proof. For $n = 0$ there are only acyclic free-connex queries, which are in CD $\circ$ Lin so they satisfy the property.

Let $n \geq 0$ and suppose that $P(n)$ is true, let's prove that $P(n+1)$ is true. Let p be patched by $p_1, \ldots, p_k$ such that $\max_{p' \in \{p_1,\ldots,p_k\}} \varphi(p') + 1 = \varphi(p)$. As $p_1, \ldots, p_k$ fully patch p, we have that $\varphi(p) \leq \varphi(p_i)$ for all $i \in [|1; k|]$, hence we have CD $\circ$ Lin$_q$ algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_k$ for them. We can apply Proposition 3.15 to $p_1, \ldots, p_k$ to have access to $R_{T_1}, \ldots, R_{T_k}$, with $k = O(1)$ CD $\circ$ Lin$_q$ algorithms, hence in a CD $\circ$ Lin$_q$ algorithm. Using these relations we can apply Corollary 3.9 to have a CD $\circ$ Lin$_q$ algorithm for p. Combining these $k+1$ CD $\circ$ Lin$_q$ algorithm, we have that $p \in$ CD $\circ$ Lin$_q$.

We conclude from recursion principle. □

Annexe 2: Proving hardness

BMM-Hard

Proposition .4. *Let q be an acyclic conjunctive query with an atom of arity at most 2. If there exists a free-path F such that $\forall x \notin Vars(F), \exists R.x \in D(R, F)$ or x is unmapped to F, q is BMM-Hard.*

Proof. Let M_1 and M_2 be two boolean matrices. Let $F := x_1, z_1, \ldots, z_n, x_2$, we do the following reduction:

- $\forall (i,k) \in M_1$ s.t. $M_1[i][k] = 1$.
  - $\forall R(x_1, z_1) \in q.R(\langle i; x_1\rangle, \langle k; z_1\rangle)$,
  - $\forall p < n. \forall R(z_p, z_{p+1}) \in q.R(\langle i; p_i\rangle, \langle i+1; z_{p+1}\rangle)$,
  - $\forall R(x_1) \in F.R(\langle i; x_1\rangle)$,
  - $\forall R(z_p) \in F.R(\langle i; z_p\rangle)$,
  - $\forall R(y, z_p) \in q \backslash F.R(\langle \perp; y\rangle, \langle k; z_p\rangle)$. Same for $R(z_p, y)$,

  - $\forall R(y, x_1) \in q \backslash F.R(\langle \bot; x_1 \rangle, \langle k; z_p \rangle)$. Same for $R(z_p, x)$.

- $\forall (k, j) \in M_2$ s.t. $M_2[k][j] = 1$.
  - $\forall R(z_n, x_2) \in q.R(\langle k; z_n; \rangle, \langle j; x_2 \rangle)$,
  - $\forall R(x_2) \in q.R(\langle k; x_2 \rangle)$,
  - $\forall R(x_2, y) \in q \backslash F.R(\langle k; x_2 \rangle, \langle \bot; y \rangle)$. Same for $R(y, x_2)$.
- $\forall R(y_1, y_2) \in q \backslash F.R(\langle \bot; y_1 \rangle, \langle \bot; y_2 \rangle)$.
- $\forall R(y_1) \in q \backslash F.R(\langle \bot; y_1 \rangle)$.

Let's count the number of total solutions (counting spurious ones). A free-variable has two possibilities:

- It is part of the free-path and it can be any value of D, i.e. $O(|\mathrm{dom}(D)|)$ possible solutions.
- It is not in the free-path, hence either:
  - there exists $R \in q$ such that $x \in D(R, F)$. But then x can only be evaluated to $\langle \bot; x' \rangle$ with $x' \in D(R, f)$ using the last case of our construction. That is $O(|D(R, f)|) = O(1)$ possibilities.
  - x is unmapped to F. If it receives a value tagged with a variable of the free-path, then there is a morphism from x to this variable (Lemma 4.4), impossible. Hence all its values are tagged with variables not in the free-path, by construction this is only $\langle \bot; y \rangle$, i.e. $O(|q|) = O(1)$.

Hence, there is at most $O(|\mathrm{dom}(D)|^2 \times |\cup_{y \notin F} \{\langle \bot, y \rangle\}|^{|\mathrm{Free}(q)|-2}) = O(|\mathrm{dom}(D)|^2)$. But $|dom(D)| = O(n)$ hence this is $\underline{O(n^2) \text{ possible solutions}}$.

Now if we have (i, j) such that $(M_1 \times M_2)[i][j] = 1$, there exists a k such that $M_1[i][k] = M_2[k][j] = 1$ and we have a solution of the free-path that witnesses this.

Hence, we can enumerate q on the tagged database and get all the 1s of $M_1 \times M_2$. This can be done in $O(n^2)$ time because we can check if a solution of q is a valid 1 of $M_1 \times M_2$ in $O(1)$ time by checking that x_1 is tagged x_1 and x_2 is tagged x_2. □

VUTD-Hard

Proposition .5. *Let q be an acyclic binary-atom CQ. If there is a quantified variable z on a free-path that is not the image of a free-variable in any body-homomorphism of q, then q is VUTD-hard.*

Proof. Let q be an acyclic binary-atom CQ. Let ζ a quantified variable that is the image of no free variable of q for any automorphism of q, i.e. $\forall h : q \to q, \forall x, h(x) \neq z$.

Let T be the free-tree associated to ζ (it exists because there is a free-path). Fix $x_1 - ... - \zeta - ... - x_2$ one free-path in T that goes through ζ.

We split the xs of T as follows:

- $x \in X_1$ if in the only path from x to x_2 x needs to go through ζ.
- $x \in X_3$ if not.

In the same way we define Z_1 and Z_3 for zs. Now we will encode VUTD on the free-tree as this:

1. $\forall v \in V_1, \forall (z_1, z_2) \in Z_1^2$. if $R(z_1, z_2) \in q$ (resp. $R(z_2, z_1) \in q$), we add $R(\langle z_1; v\rangle, \langle z_2; v\rangle)$ (resp. $R(\langle z_2; v\rangle, \langle z_1; v\rangle)$),
2. $\forall v \in V_3, \forall (z_1, z_2) \in Z_3^2$. if $R(z_1, z_2) \in q$ (resp. $R(z_2, z_1) \in q$), we add $R(\langle z_1; v\rangle, \langle z_2; v\rangle)$ (resp. $R(\langle z_2; v\rangle, \langle z_1; v\rangle)$),
3. $\forall v \in V_1, \forall x \in X_1, \forall z \in Z_1$. if $R((x, z)) \in q$ (resp. $R((z, x)) \in q$), we add $R(\langle x; v\rangle, \langle z; v\rangle)$ (resp. $R(\langle z; v\rangle, \langle x; v\rangle)$),
4. $\forall v \in V_3, \forall x \in X_3, \forall z \in Z_3$. if $R((x, z)) \in q$ (resp. $R((z, x)) \in q$), we add $R(\langle x; v\rangle, \langle z; v\rangle)$ (resp. $R(\langle z; v\rangle, \langle x; v\rangle)$),
5. $\forall (u, v) \in E_{1,2}, \forall y \in X_1 \cup Z_1$. if $R((y, \zeta)) \in q$ (resp. $R((\zeta, y)) \in q$), we add $R(\langle y; u\rangle, \langle \zeta; v\rangle)$ (resp. $R(\langle \zeta; v\rangle, \langle y; u\rangle)$),
6. $\forall (u, v) \in E_{2,3}, \forall y \in X_3 \cup Z_3$. if $R((y, \zeta)) \in q$ (resp. $R((\zeta, y)) \in q$), we add $R(\langle y; v\rangle, \langle \zeta; u\rangle)$ (resp. $R(\langle \zeta; u\rangle, \langle y; v\rangle)$),
7. $\forall u \in V_1, \forall y_1 \in X_1 \cup Z_1, \forall y_2 \notin T$. if $R(y_1, y_2) \in q$ (resp. $R(y_2, y_1) \in q$) then we add $R(\langle y_2; u\rangle, \langle y_1; \bot\rangle)$ (resp. $R(\langle y_1; \bot\rangle, \langle y_2; u\rangle)$),
8. $\forall u \in V_3, \forall y_1 \in X_3 \cup Z_3, \forall y_2 \notin T$. if $R(y_1, y_2) \in q$ (resp. $R(y_2, y_1) \in q$) then we add $R(\langle y_2; u\rangle, \langle y_1; \bot\rangle)$ (resp. $R(\langle y_1; \bot\rangle, \langle y_2; u\rangle)$),
9. $\forall y_1, y_2 \notin T$. If $R(y_1, y_2) \in q$, $R(\langle y_1; \bot\rangle, \langle y_2; \bot\rangle)$.
10. $\forall v_2 \in V_2. \forall y \notin T$. If $R(y, \zeta) \in q$ (resp. $R(\zeta, y) \in q$), then we add $R(\langle y; \bot\rangle, \langle \zeta; v_2\rangle)$ (resp. $R(\langle \zeta; v_2\rangle, \langle y; \bot\rangle)$).

This construction gives an algorithm to detect a triangle:

- Enumerate all solutions of the query and for each:
  - Check if the tagging is the identity on the free variables, if so check if the value given to x_1 and x_2 are connected in $E_{1,3}$, if it is then we have detected a triangle, if not continue.

Let $(v_1, v_2, v_3) \in V_1 \times V_2 \times V_3$ be a triangle in G. We know that there is a path $(x_1 \to z_1^1 \to \ldots \to z_1^k \to \zeta \to z_3^1 \to \ldots \to\!\to z_3^p x_2) \in q$. From:

- Item 3., $R(\langle x_1; v_1\rangle, \langle z_1; v_1\rangle)$ (or the opposite direction, or both of them) will be in D for all $z_1 \in Z_1$,
- Item 1., we know that there will be a path of $R(\langle z_1^i; v_1\rangle, \langle z_1^{i+1}; v_1\rangle)$ (or the opposite direction, or both of them) in the database.
- Item 5., there is $R(\langle z_1^i; v_1\rangle, \langle \zeta; v_2\rangle)$ in the database (or the opposite direction, or both of them).
- Item 6., there is $R(\langle \zeta; v_2\rangle, \langle z_3^1; v_3\rangle)$ in the database (or the opposite direction, or both of them).

- Item 2., we know that there will be a path of $R(\langle z_3^i; v_3\rangle, \langle z_3^{i+1}; v_3\rangle)$ (or the opposite direction, or both of them) in the database.
- Item 4., $R(\langle x_3; v_3\rangle, \langle z_3^p; v_3\rangle)$ (or the opposite direction, or both of them) will be in D.

So the path is well-encoded (we will see after that we don't have to check how many spurious answers we have because the total number of answers is limited). However, we need to be sure that the given path can be completed in a whole solution of q. One can see that:

- Variables in Z_1 are connected to either, a variable in X_1, ζ, other variables in Z_1, variables outside of the tree, but can not be connected to variables in Z_3 or X_3 (otherwise there is a cycle). The possible connections are handled respectively by Item 3., Item 1., Item 5., Item 7.. In particular, Item 7. force variables outside of T to be $\perp$.
- Variables in X_1 are connected to either, a variable in X_1, ζ, a variable in Z_1, or variables outside. Using the same rules as for z_1, we can come to the same conclusion.
- The same goes for Z_3 and X_3 using Item 8..
- If ζ is connected to a variable in $Z_1 \cup Z_3 \cup X_1 \cup X_3$ it is already ok thanks to the previous point. If ζ is connected to a variable outside of the tree, then with Item 10. one can see that y can be set to $\perp$.
- Finally, if two variables outside of tree are connected together, they can both receive $\perp$ according to Item 9..

This query has only $O(n^{k\alpha})$ solutions with $k = |\text{free variables}|$: every elements of the database is of the form $\langle \text{variable}; \text{value}\rangle$. The only elements of the database that contain a value from V_2 are tagged with the variable ζ. Suppose that a free variable x has more than $O(n^\alpha)$ different values in all the solutions. Then it receives some value from V_2, i.e. there exists a valuation μ of q such that the tagging map sends x to ζ. This is impossible because we would have an automorphism that sends x to ζ using (4.4).

Hence for $\alpha < \frac{1}{k}$ we could find a triangle in $O(n^{1+\alpha})$ if $q \in \text{CD} \circ \text{Lin}$. □

Lemma .6 (Link between CD ∘ Lin and DomLin ∘ Lin)**.** *$q_1 \in CD \circ Lin \Leftrightarrow q_2 \in DomLin \circ Lin$.*

Proof. q_1 is q_2 but with an extra free variable that allows us to repeat every solution of q_2 at maximum $O(|\text{Dom}(D)|)$ times. We can use this fact in both directions to build a reduction. More formally:

For $\Rightarrow$, consider Algorithm 6. P starts to be enumerated after a linear pre-computation (because the hypothesis is that $q_1 \in \text{DomLin} \circ \text{Lin}$), hence we start to enumerate on the second loop after a linear pre-computation. Moreover, the second loop is printing with $O(1)$ delay, but we need to be sure that there are not too many duplicates. There exists a constant c such that every $c|\text{Dom}(D)|$, P is edited

Algorithm 6 Enumerate(q_1, q_2, D)

```
P ← ∅
Do in parallel these two for-loops:
for (a, b) ∈ q1(D) do P.append((a, b))
end for
for (a, b) ∈ P do
    while P hasn't changed do
        for c ∈ D do
            output (a, b, c)
        end for
    end while
end for
```

Algorithm 7 Enumerate(q_1, q_2, D)

```
P ← ∅
H ← ∅
c ← 0
for (a, b, c) ∈ q2(D) do
    if (a, b) ∉ H then
        H.append((a, b))
        P.append((a, b))
    end if
    c ← c + 1
    if c ≥ c2n then
        Print P.pop()
        c ← 0
    end if
end for
while P is not empty do
    Print P.pop()
end while
```

(because $q_1 \in \text{DomLin} \circ \text{Lin}$), hence P changes each $c|\text{Dom}(D)|$ and for a given $(a, b) \in P$, each solution (a, b, c) associated is printed at most $O(c|\text{Dom}(D)|)$ (the constant depends of the way we are dealing with parallelism). It just needs some fine-tuning to be sure that q_1 doesn't output two solutions without giving the hand to the second loop (but it can be forced easily with some boolean) and to be sure that the share of work is not such that we do a lot of second-lop and only a few on the first-loop (otherwise we would print too much).

For $\Leftarrow$, consider Algorithm 7, the constant c_2 is such that the delay to enumerate q_2 is bounded by b_2. From the definition of c_2, P will never be empty in the loop (each solution (a, b) is featured n times, one for all possible c) and we print solutions of q_2 with respect to $\text{DomLin} \circ \text{Lin}$. □

Algorithm 8 *Enumerate*(q_3, D)

```
1:  M ← ∅
2:  for R(x_1, z) ∈ D do
3:      for R(x_2, z) ∈ D do
4:          for R(x_3, z) ∈ D do
5:              for R(z_2, z) ∈ D do
6:                  Print (x_1, x_2, x_3, z)
7:                  M ← M@(x_1, x_2, x_3, z_2)
8:              end for
9:          end for
10:     end for
11: end for
12: for (x_1, x_2, x_3, z_2) ∈ M do
13:     for R(z_2, x_4) ∈ D do
14:         Print x_1, x_2, x_3, x_4
15:     end for
16: end for
```

Database with $O(1)$ high degrees element

Definition .7 (Degree of an element)**.** *Let D be a database over the domain $dom(D)$ with atom of arity at most 2. The in-degree (resp. out-degree) of an element $a \in dom(D)$ is $d^-(a) = |\{R(.,a) \in D\}|$ (resp. $d^+(a) = |\{R(a,.) \in D\}|$).*

Definition .8 (Number of high degree elements)**.** *Let D be a database over the domain $dom(D)$ with arity at most 2. The number of high in-degree (resp. out-degree) elements in D, denoted Δ^- (resp. Δ^+), is $\Delta^- = |\{a \in D \mid d^-(a) \neq O(1)\}|$ (resp. $\Delta^+ = |\{a \in D \mid d^+(a) \neq O(1)\}|$). Here $O(1)$ is compared to $|D|$.*

For example, if D is a clique, $\Delta^- = \Delta^+ = O(|\mathrm{dom}(D)|)$. If it is bipartite, then $\Delta^- = \Delta^+ = O(1)$. The following result, Proposition .11, might seem useless but here we only ask the in-degree, or the out-degree, to be low, not both of them, hence we can still encode some problems that need a lot of edges.

Lemma .9. *If $\Delta^-(D) = O(1)$, then there is a CD $\circ$ Lin algorithm that enumerates q on D.*

Proof. Consider Algorithm 8. Completeness and soundness are left to the reader (nothing hard in it). For the complexity, one can note that:

- Loop of line 3 to 5 are each going to at most $O(1)$ elements because z except a constant number of times where it is $O(n)$.
- Line 6 is printing a solution every $O(1)$ time, and each solution is duplicated exactly the number of times $R(z_2, z)$ appears in D, which is $O(1)$ except for a constant number of solutions where it is at most linear.

Algorithm 9 *Enumerate*(q_3, D)

```
1: M ← ∅
2: for R(z_2, x_4) ∈ D do
3:     for R(z_2, z_1) ∈ D do
4:         Print z_2, z_2, z_2, x_4
5:         M ← M@(z_1, x_4)
6:     end for
7: end for
8: for (z_1, x_4) ∈ M do
9:     for R(x_1, z_1), ..., R(x_3, z_1) ∈ D do
10:        Print x_1, x_2, x_3, x_4
11:    end for
12: end for
```

These two points make that the first chain of loop is respecting CD $\circ$ Lin (according to Cheater's Lemma 2.1). Then, the second loop prints every solution of the query with a $O(1)$ delay and any (x_1, x_2, x_3, x_4) is printed at most $d^-(x_4)$, which is $O(1)$, except for a constant number of x_4 where it is at most linear, hence this respects the condition of the cheater's Lemma 2.1.

The algorithm depicts that $q_3 \in$ CD $\circ$ Lin when the database satisfies $\Delta^-(D) = O(1)$.

□

Lemma .10. *If $\Delta^+(D) = O(1)$, then there is a CD$\circ$Lin algorithm that enumerates q on D.*

Proof. Consider Algorithm 9. Completeness and soundness are left to the reader (nothing hard in it). For the complexity, one can note that:

- For every z_2, there is only a constant number of z_1, except for a constant number of z_2 where there is at most a linear number of z_1, hence the first chain of loop is respecting the conditions of cheater's Lemma 2.1.
- On the second chain of for-loop, each z_1 gives a constant number of x_4 except a constant number of z_1 which gives at most a linear number of x_4, hence each solution is duplicated a constant number of times, except a constant number of solutions which is printed at most a linear number of times. Hence, this loop too satisfies the conditions of cheater's Lemma 2.1

Hence, if $\Delta^+(D) = O(1)$, the enumeration of q_3 is in CD $\circ$ Lin. □

Proposition .11. *If $min(\Delta^-(D), \Delta^+(D)) = O(1)$, then there is a CD$\circ$Lin algorithm that enumerates q_3 on D.*

Proof. Direct from Lemma .9 and Lemma .10. □

All the algorithms presented in this subsection can be scaled easily to any member of the family Fig. 8.