Conjunctive Queries With Self-Joins, Towards a Fine-Grained Enumeration Complexity Analysis

Nofar Carmeli
DI ENS, ENS, Université PSL, CNRS, Inria
Paris, France

Luc Segoufin
INRIA, ENS Paris, PSL
Paris, France

ABSTRACT

Even though query evaluation is a fundamental task in databases, known classifications of conjunctive queries by their fine-grained complexity only apply to queries without self-joins. We study how self-joins affect enumeration complexity, with the aim of building upon the known results to achieve general classifications. We do this by examining the extension of two known dichotomies: one with respect to linear delay, and one with respect to constant delay after linear preprocessing. As this turns out to be an intricate investigation, this paper is structured as an example-driven discussion that initiates this analysis. We show enumeration algorithms that rely on self-joins to efficiently evaluate queries that otherwise (i.e., if the relation names were replaced to eliminate self-joins) cannot be answered with the same guarantees. Due to these additional tractable cases, the hardness proofs are more complex than the self-join-free case. We show how to harness a known tagging technique to prove hardness of queries with self-joins. Our study offers sufficient conditions and necessary conditions for tractability and settles the cases of queries of low arity and queries with cyclic cores. Nevertheless, many cases remain open.

CCS CONCEPTS

• **Theory of computation → Database query processing and optimization (theory)**.

KEYWORDS

conjunctive query, self-joins, enumeration, fine-grained, complexity, constant delay

ACM Reference Format:
Nofar Carmeli and Luc Segoufin. 2023. Conjunctive Queries With Self-Joins, Towards a Fine-Grained Enumeration Complexity Analysis. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS '23), June 18–23, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3584372.3588667

1 INTRODUCTION

Query evaluation is one of the most central problems in database systems. This task asks to compute the set of solutions to a given query over a given database. In this paper we focus on joins and sometimes projections, the basic building blocks of queries, expressed as conjunctive queries. When treating both the database and the query as input, query evaluation is then NP-complete [6]. Here, we adopt the data complexity point of view: the complexity is analyzed with respect to the database size, while the query size (which is usually small) is considered constant. In this setting, query answering takes polynomial time. However, not all queries are equally hard as for some queries this polynomial is much smaller than for others. Ideally, we would like to understand, given any specific query, how fast this query can be evaluated. As the number of solutions may be much larger than the input, considering the delay between solutions and not just the total computation time can be meaningful: the user does not have to wait for the entire computation to be done before seeing some of the solutions. The ideal time guarantee we can hope to achieve is linear time before the first solution (required to read the input and determine whether a first solution exists) and constant time between successive solutions (required to print the output).

We know that any *acyclic* query with a *free-connex* form can be answered with linear preprocessing and constant delay. For acyclic queries that are not free-connex, we do not have an algorithm with such guarantees, but we do have algorithms with linear delay (and linear preprocessing). At this point, it is natural to ask: Are free-connex and acyclic queries the only ones for which such efficient algorithms exist or should we keep looking for efficient algorithms for additional queries?

To answer this question, we have conditional lower bounds showing that other queries cannot be answered within these time bounds unless there would be a significant breakthrough in well-studied problems. However, these hardness results were only proved for *self-join-free* conjunctive queries, queries in which each relation appears at most once. More specifically, if a self-join-free query is acyclic but not free-connex, then the set of all its solutions cannot be computed in time linear in the input and output sizes, assuming Boolean matrices cannot be multiplied in quadratic time [1]. Moreover, one cannot even produce a single solution to a self-join-free cyclic query in linear time, assuming hypercliques cannot be detected in linear time [3]. The hardness results described in this introduction also assume the hardness of the two aforementioned algorithmic problems.

For years, database researchers were satisfied with these hardness results despite the limitation to self-join-free queries, as self-joins are known to be problematic. We remark that self-joins are very common in some contexts (e.g., graph databases). It was believed that the above characterization would extend to queries with self-joins and that this restriction is merely a limitation of our lower-bound tools. This extension was even claimed along with a proof sketch [3]. However, Berkholz, Gerhardt, and Schweikardt [2] recently refuted this claim by showing a cyclic query with self-joins that can be evaluated efficiently. That is, we now know that queries with self-joins can be easier than self-join-free queries of the same form. Now that we have an example that shows it is possible, we would like to understand how, and in which cases, self-joins can be used to streamline query evaluation. Here, we initiate the complexity analysis for the evaluation of conjunctive queries with self-joins.

To illustrate the intricacies that arise with self-joins, consider the task of finding all occurrences of the following patterns in a node-colored directed graph (where the red circle denotes a red node). Note that these tasks can be phrased as conjunctive queries with self-joins, containing one binary relation describing the edges and one unary relation describing the red nodes.

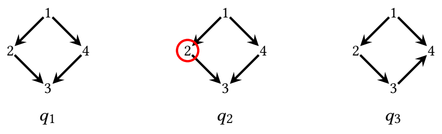


Figure 1: Variants of a 4-cycle with different complexities.

We will see that q_1, even though cyclic, can be enumerated with constant delay after a linear preprocessing time. Unlike the self-join-free case, adding a unary atom to a query may change its complexity: we will see that q_2 can be answered with linear delay but not with constant delay (after only linear preprocessing). However, for q_3 (that differs from q_1 by the direction of one edge), we cannot even produce one solution in linear time. This again differs from the self-join-free case, in which the variable order in atoms cannot affect the complexity.

In this paper, we explore the effects of self-joins on query evaluation complexity from an enumeration point of view. The aim is to use known results for the self-join-free case to reason about general queries that may contain self-joins. We show enumeration algorithms that rely on self-joins to efficiently evaluate queries that otherwise cannot be answered with the same guarantees. That is, existing conditional lower bounds suggest that similar queries with unique relation names (the same query structure but no self-joins) cannot be answered within the time guarantees obtained by our algorithms. Due to these additional tractable cases, the hardness proofs also require additional complications compared to the self-join-free case. We show how to harness a known tagging technique to prove hardness of queries with self-joins.

We discuss the difficulty in proving lower bounds for queries with self-joins in Section 3 and then define the tagging technique and explain the importance of endomorphism images for its analysis. Section 4 uses this technique to establish that self-joins do not change whether a query of arity two or less can be enumerated with constant delay. In Section 5, we show that queries with a cyclic "core" are hard in the sense that they cannot be answered with only linear preprocessing, even if they contain self-joins. As a result, we focus on queries with an acyclic core in the rest of the paper.

Next, we focus on full conjunctive queries, with no projection. In Section 6, we phrase a sufficient condition and a necessary condition for enumeration with linear delay, but we also give examples of cases that are not covered by either of these conditions. In Section 7, we identify a class of cyclic queries that can be answered with constant delay (we call these mirror queries). We then provide an example-driven discussion of how simple acyclic additions (unary atoms and spikes) to tractable queries can affect the complexity. We finish with a simple (single-cycle) query of unknown complexity. Missing details appear in the appendix.

2 PRELIMINARIES

A *database* D is a finite relational structure consisting of a relational schema σ, a finite domain V, and for each relational symbol R of σ of arity r, a subset R^D of V^r. A term $R(\bar{a})$ where $\bar{a} \in R^D$ is called a *fact* of D.

In this paper, *query* always refers to a *conjunctive query*, i.e., a conjunction of atoms $R(\bar{x})$, where $R \in \sigma$ and $\bar{x}$ is a tuple of variables of the appropriate arity, with some of the variables existentially quantified. The variables that are not quantified in q are called the *free variables* of q. The *arity* of a query is the number of its free variables. A *Boolean/unary/binary* query is a query of arity 0/1/2 respectively. A query is *full* if it contains no quantified variables, i.e., all the variables are free. A query has *self-joins* if it contains two atoms with the same relational symbol; otherwise, it is called *self-join-free*. For example, the query $R(x_1, x_2) \wedge R(x_2, x_3)$ has self-joins while the query $R(x_1, x_2) \wedge S(x_2, x_3)$ is self-join-free.

We will often specify full conjunctive queries using a colored hypergraph rather than a formula. Each node represents a variable of the query and is depicted by the number of the variable or a black dot when this number is not relevant. Each colored edge represents a binary atom, and each colored circle around a node represents a unary atom. The exact relation names are not depicted, but they are irrelevant to the complexity analysis. As an example, q_2 of Figure 1 defines the query $E(x_1, x_2) \wedge E(x_2, x_3) \wedge E(x_1, x_4) \wedge E(x_4, x_3) \wedge Red(x_2)$. We sometimes use a big circle to represent a relation of arity three or more (as in q_6 of Figure 3). When we do so, the order of the variables in this relation is not important for the example.

A query is *acyclic* if it has a join tree. A *join tree* for the query q is a tree T such that the nodes of T are the atoms of q and, for each variable x of q, the nodes of T whose atoms contain x are connected in T. An acyclic query q is said to be *free-connex* if we obtain an acyclic query by adding to q a new atom containing all the free variables of q. As an example, the acyclic query $\exists x_2 R(x_1, x_2) \wedge S(x_2, x_3)$ is not free-connex because the query $\exists x_2 R(x_1, x_2) \wedge S(x_2, x_3) \wedge T(x_1, x_3)$ is cyclic; however, the full acyclic query $R(x_1, x_2) \wedge S(x_2, x_3)$ is free-connex because the query $R(x_1, x_2) \wedge S(x_2, x_3) \wedge T(x_1, x_2, x_3)$ is acyclic. In fact, it follows from the definition of free-connexity that full acyclic queries are always free-connex. Similarly, Boolean or unary acyclic queries are free-connex as well.

The *solutions* of a query q over a database D are denoted by $q(D)$. They correspond to the set of tuples $\bar{a}$ such that: there exists an assignment of the elements of the domain of D to the variables of q making every atom of q a fact of D, and $\bar{a}$ is the assignment to the free variables of q. When we specify a solution to a query, we assume the free variables are ordered according to their numbering. The *output size* is the number of tuples in the set $q(D)$. Two queries q and q' are said to be *equivalent* if for every database D, $q(D) = q'(D)$.

An *endomorphism* of q is a mapping from the variables of q to the variables of q that preserves the atoms of q; that is, if $R(\bar{x})$ is an atom of q and v is an endomorphism of q, then $R(v(\bar{x}))$ is also an atom of q. An *automorphism* is a bijective endomorphism. A query q is said to be *minimal*[1] if any endomorphism of q that is the identity on the free variables of q is an automorphism of q. Notice that every self-join-free query is minimal as the only endomorphism in such a query is the identity mapping. Any query q has an equivalent minimal query, called the *minimal form* of q in what follows.

The problems we consider are parameterized by a query q and take as input a database D. *The evaluation problem* computes the set of solutions $q(D)$. *The testing problem* asks whether $q(D)$ is empty. *The enumeration problem* is decomposed into two phases: the preprocessing phase computes a data structure that is used during the enumeration phase for outputting the solutions of $q(D)$ one by one and with no repetition. The maximal time between two consecutive answers during enumeration is called the *delay*.

We use the RAM computation model with uniform addition and multiplication, and we adopt the data complexity point of view. That is, the constants hidden in the big O notation may depend on the query but not the database. The input size, denoted $\|D\|$, corresponds to the size of a reasonable encoding of the database D, linear in the number of database tuples. Note that an enumeration algorithm with linear preprocessing implies that the testing problem can be solved in linear time. A constant delay after linear preprocessing enumeration algorithm implies that the evaluation problem can be solved in linear input plus output time.

Many known enumeration algorithms for database queries conform to a strict definition of linear preprocessing and constant delay where the memory used during preprocessing is linear and the extra memory used after the preprocessing remains bounded by a constant [1, 7–10]. Our enumeration algorithms use the allowed time in the most general way, without memory restrictions. In particular, we assume we can test in constant time whether a given tuple is a fact of the database. It is not clear how to do this test without using perfect hashing, which requires polynomial memory in the size of the input database. The enumeration phases of our algorithms may also require a total memory of a similar size. This is required to store each produced solution (or parts of the solution), used later for producing more solutions. As this distinction is important, we will make it explicit when we use this extra memory. This is in particular the case in the proof of the Cheater's Lemma [5] that we use to eliminate duplicate answers.

Lemma 1 (Cheater's Lemma) [5]. *Let q be a conjunctive query. If there is an algorithm evaluating q with linear preprocessing time and constant/linear delay, where the number of times each output is produced is bounded by a constant, then there is an enumeration algorithm for q that runs with linear preprocessing time and constant/linear delay, producing the solutions with no repetition. This new algorithm may use during the enumeration phase a memory of size $O(\|D\|^{|q|})$ over an input database D.*

As usual in this area, our lower bounds assume algorithmic conjectures, described next. A hypergraph is $(k-1)$-uniform if every hyperedge is of size $k-1$. A k-hyperclique is a set of k nodes such that any subset of these nodes of size $k-1$ forms a hyperedge.

- sBMM conjectures that two Boolean matrices A and B, represented as lists of their non-zero entries, cannot be multiplied in time $O(m)$, where m is the number of non-zero entries in A, B, and AB.
- sHyperclique conjectures that 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 $O(m)$.
- Hyperclique conjectures that for all $k \geq 3$, it is not possible to determine the existence of a k-hyperclique in a $(k-1)$-uniform hypergraph with n nodes in time $O(n^{k-1})$.

[1]It is called minimal as there is no smaller (with strictly fewer atoms) equivalent query.

Notice that sHyperclique implies that we cannot detect a triangle in an undirected graph in linear time. Hyperclique implies the sparse hypotheses sHyperclique and sBMM.

We can now state the known results for query evaluation.

Theorem 2. *Let q be a self-join-free conjunctive query.*

- *If q is cyclic, the testing problem cannot be solved in linear time, assuming sHyperclique [3].*
- *If q is acyclic, the enumeration problem can be solved with linear delay and linear preprocessing [1].*
- *If q is acyclic but not free-connex, the evaluation problem cannot be solved in linear time in the input plus output sizes, assuming sBMM [1].*
- *If q is acyclic and free-connex, the enumeration problem can be solved with constant delay and linear preprocessing [1].*

In the above theorem, the upper bounds also hold for queries with self-joins. The enumeration algorithms obtained in [1] use the strict definition of constant delay, where the total memory used during the enumeration phase only adds a constant to the memory used during the preprocessing phase.

3 PROVING HARDNESS WITH SELF-JOINS

Since we have a good understanding of the complexity of self-join-free queries, we are interested in comparing the complexity of general conjunctive queries to self-join-free queries with the same shape. More formally, we associate a self-join-free query q' to any query q by giving distinct names to all relation symbols. As an example, the self-join-free query associated to $R(x_1,x_2) \wedge R(x_2,x_3)$ is $R_1(x_1,x_2) \wedge R_2(x_2,x_3)$. Notice that this modification does not affect properties such as acyclicity or free-connexity.

Going from a query q to its associated self-join-free query q' can only make the evaluation task harder. This folklore proposition holds since q' can be used to compute q, and it implies that the upper bounds of Theorem 2 also hold for queries with self-joins.

Proposition 3. *For any conjunctive query q, its associated self-join-free query q' is at least as hard as q for the evaluation problem, the testing problem, and the enumeration problem.*

When trying to show the other direction, reducing the self-join-free query to its initial query with self-joins, we need more care. Let us now explain why the lower bounds of Theorem 2 only apply to self-join-free queries.

Example 4. Consider the self-join-free variant q_1' of q_1 from Figure 1: $R_1(x_1,x_2) \wedge R_2(x_2,x_3) \wedge R_3(x_1,x_4) \wedge R_4(x_4,x_3)$. Theorem 2 proves that the ability to determine whether q_1' has answers in linear time would imply finding triangles in an undirected graph in linear time, which is conjectured to be impossible. The reduction works as follows: for every edge (a, b) of the input graph with $a < b$,[2] we add to the database the facts: $R_1(a, b)$, $R_2(a, b)$, $R_3(a, b)$, and $R_4(b, b)$. With this construction, every answer to q'_1 is of the form (a, b, c, c) such that (a, b, c) is a triangle. If we try to use the same technique to prove the hardness of q_1, we encounter the problem that we cannot assign different atoms with different relations. In particular, if we add to the database the facts $R(a, b)$ and $R(b, b)$ for every edge (a, b) of the graph, the query q_1 will also have solutions corresponding to 2-paths (where every atom maps to the edge relation and x_2 and x_4 map to the same value). Thus, we do not get that the query has a solution iff the graph has a triangle.

To prove the hard cases, we need a technique that allows us to assign different relations to different atoms despite the self-joins. For this reason, we will use variations of the tagging technique described below that aims to assign different variables with different domains by concatenating the variable names to the domain values.

Tagging technique.[3] Let q be a minimal conjunctive query and q' be its associated self-join-free query. Let D' be a database for q'. We construct a database D for q as follows. Let $R_1(\bar{x})$ be an atom of q' associated to the atom $R(\bar{x})$ of q, where R is of arity l. For every fact $R_1(\bar{a})$ of D', we add to D the fact $R(\langle a_1, x_1\rangle, \dots, \langle a_l, x_l\rangle)$.

With this construction, every element of the domain of D is of the form $\langle a, x\rangle$. We refer to a as the data part and to x as the tag. Let us now inspect $q(D)$. Every solution $(\langle a_1, y_1\rangle, \dots, \langle a_k, y_k\rangle)$ in $q(D)$ is witnessed by an assignment μ associating an element of D to every variable of q. As above, μ can be split into two parts: a data map associating a_i to x_i whenever $\mu(x_i) = \langle a_i, y_i\rangle$ and a tagging map associating y_i to x_i in the same cases. The data part of the solution of μ is $(a_1, \cdots, a_k)$. Notice that every solution to $q'(D')$ appears as a data part of a solution of $q(D)$ with an identity tagging map. For the converse direction, consider a solution of $q(D)$. There are two cases.

(1) If the tagging map ν is an automorphism, then the data map contains a solution of $q'(D')$ as follows. Consider the mapping μ' defined by $\mu'(x) = \mu(\nu^{-1}(x))$. It clearly witnesses a solution of $q(D)$ with an identity tagging map whose data part then belongs to $q'(D')$.
(2) If the tagging map is not an automorphism, the data part of the solution may then not correspond to a solution in $q'(D')$.

If q is minimal and the tagging map ν is the identity on the free variables, then ν must be an automorphism. Thus, when using the tagging technique, solutions with the identity tagging map are the ones we wish to return in order to answer q', and we need to ensure that there are not too many interfering solutions of the other cases. We will see that if the query is of low arity, then the number of extra solutions remains small, and so they can be ignored without incurring a high cost. For full queries, other solutions of the first case are not a problem: As we are given the assignment to all variables as part of a solution, we can compute μ' and an answer to q' given any such solution to q. Every solution to q' will be obtained a constant number of times this way, once for each automorphism, and these duplicates can be eliminated using the Cheater's Lemma. Solutions of the second case are the ones we need to beware of. For this technique to result in a useful reduction, we need to limit the number of these extra solutions, and this cannot always be achieved. We summarize the discussion above into the following lemma.

LEMMA 5. *Let q' be the self-join-free query associated with a minimal conjunctive query q, and let D be the database for q constructed using the tagging technique from a database D' for q'. Then, $q'(D')$ is precisely the data parts of those solutions to $q(D)$ in which every free variable is assigned itself as a tag. Moreover, if q is full, then each solution to $q(D)$ tagged by an automorphism can be translated in constant time to a solution of $q'(D')$ such that each solution of $q'(D')$ is obtained a constant number of times.*

The following is an example of the tagging technique and how it can fail to prove hardness due to extra solutions. Later in the paper, we will see how this difficulty can be circumvented in some cases so that the tagging technique works.

Example 6. Consider q'_1 and q_1 from Example 4. Notice that q_1 has two automorphisms: the identity and the one that switches x_2 and x_4. It also has two other endomorphisms: the one that maps x_2 and x_4 to x_2, and the one that maps them both to x_4. Consider the database D' consisting of the four facts $R_1(a, b)$, $R_2(b, c)$, $R_3(a, d)$ and $R_4(d, c)$. Then, $q'_1(D')$ consists of one solution (a, b, c, d). The database D constructed by the tagging technique consists of the facts $R(\langle a, x_1\rangle, \langle b, x_2\rangle)$, $R(\langle b, x_2\rangle, \langle c, x_3\rangle)$, $R(\langle a, x_1\rangle, \langle d, x_4\rangle)$, and $R(\langle d, x_4\rangle, \langle c, x_3\rangle)$. The solutions $q_1(D)$ contain the solutions of the first case: $(\langle a, x_1\rangle, \langle b, x_2\rangle, \langle c, x_3\rangle, \langle d, x_4\rangle)$ and $(\langle a, x_1\rangle, \langle d, x_4\rangle, \langle c, x_3\rangle, \langle b, x_2\rangle)$. In other words, these answers contain two copies of the initial solution (a, b, c, d), one per automorphism of q_1. However, $q_1(D)$ also contains the solutions $(\langle a, x_1\rangle, \langle b, x_2\rangle, \langle c, x_3\rangle, \langle b, x_2\rangle)$ and $(\langle a, x_1\rangle, \langle d, x_4\rangle, \langle c, x_3\rangle, \langle d, x_4\rangle)$, whose data parts correspond to the paths of length two (a, b, c) and (a, d, c). In this case, the tagging technique may produce too many solutions on general databases (a solution for each 2-path), and so this trick cannot be used for showing the hardness of q_1. In fact, we will see in Section 7 that q_1 can be answered with constant delay and is therefore easier than its self-join-free form.

4 QUERIES OF LOW ARITY

We use the tagging from the previous section to show that self-joins do not affect the tractable cases for constant delay enumeration of conjunctive queries with few free variables.

For Boolean and unary queries, the number of solutions is at most linear in the database size. If we could compute the entire solution set in linear time then constant delay enumeration easily follows. This situation is characterized by the following result.

THEOREM 7. *Let q be a minimal conjunctive query of arity at most one. Then, its solutions can be computed in linear time iff it is acyclic, assuming sHYPERCLIQUE.*[4]

[2] To go from an undirected graph to a binary relation we can either order the vertices of the graph as we do here or construct symmetric relations.

[3] This technique was suggested by Bagan, Durand and Grandjean [1] to show the hardness of queries with disequalities. It was later used to show the hardness of unions of conjunctive queries [5]. Brault-Baron [3, Chapter 4, Lemma 9] claimed that it can be used to show that queries with self-joins are as hard as queries without them, but this claim, which was later refuted, was not accompanied by a proof.

[4] In fact, the proof of the unary case can be applied to any minimal conjunctive query in which one atom contains all free variables.

Proof sketch. If q is cyclic then its associated self-join-free query q' is also cyclic and therefore hard by Theorem 2. We reduce the evaluation of q' to the evaluation of q using the tagging technique and extract the solutions to q' as in Lemma 5. As q is at most unary, it has a linear number of solutions, and so we can ignore the extra solutions at a low cost. □

We remark that, while Theorem 7 implies that cyclic unary queries cannot be enumerated with constant delay, it does not imply the same for linear delay. In fact, we can show that the cyclic unary query $q_4(x)$ defined as $\exists yuv E(u,y) \wedge E(y,v) \wedge E(u,x) \wedge E(x,v) \wedge P(y)$ can be enumerated with linear delay. It remains open to characterize the unary queries with respect to linear delay.

We show a similar result to that of Theorem 7 for binary queries using the non-sparse version of the hypothesis.

Theorem 8. *Let q be a minimal binary conjunctive query. If q is not acyclic free-connex, then there is no linear preprocessing and constant delay enumeration algorithm for answering q, assuming* Hyperclique.

Proof sketch. Let q be a minimal binary query and q' be its associated self-join-free query. Assume that q is cyclic. In the self-join-free case, a known construction [3] encodes the hyperclique detection problem into solving any cyclic query over a database with n domain values, where n is the number of vertices in the hypergraph. Hence, a solution to q' cannot be computed in linear time, assuming Hyperclique. Using the tagging technique, the number of extra solutions to $q(D)$ is bounded by $O(n^2)$. So, for any $k \geq 3$, the time necessary to ignore the unwanted solutions falls within the allowed time of $O(n^{k-1})$ for testing the existence of a k-hyperclique. The acyclic non-free-connex case is similar. □

5 QUERIES WITH A CYCLIC CORE

Given a conjunctive query q with free variables $\bar{x}$, let p be the minimal form of the Boolean query $\exists \bar{x} q$. We denote by the *core* of q the query constructed from p by removing all quantifications. Notice that the core of q is a full query that may not be equivalent to q. Notice also that an acyclic query has an acyclic core but the converse is not necessarily true: the core of the cyclic query q_1 of Figure 1 is acyclic. Observe that a query has a solution iff its core has a solution. Hence the following result is an immediate consequence of Theorem 7.

Theorem 9. *It is possible to test in linear time whether a conjunctive query has a solution iff it has an acyclic core, assuming* sHyperclique.

Example 10. Consider the full conjunctive query q_3 depicted by Figure 1. The core of the query is the query itself and is therefore cyclic. It follows from Theorem 9 that the query is hard, i.e., we cannot test whether it has a solution in linear time.

6 FULL CONJUNCTIVE QUERIES AND LINEAR DELAY

Assuming sHyperclique, acyclic self-join-free queries can be enumerated with linear delay (after linear preprocessing) while cyclic self-join-free queries cannot. We now ask how self-joins affect this classification. Following Proposition 3 and Theorem 9, it remains to handle cyclic queries with an acyclic core. In this section, we begin this investigation by considering only full queries of this form. We start with a simple example showing that some cyclic queries with self-joins can be enumerated with linear delay.

Example 11. Consider the full query q_2 of Figure 1. It can be enumerated with linear delay as follows. We enumerate the solutions of the core of q_2, depicted by $1 \longrightarrow \textcircled{2} \longrightarrow 3$. As the core is acyclic and free-connex, by Theorem 2, this can be done with constant delay after linear preprocessing. For each solution (a,b,c) of the core, we try all elements d of the database and test, in constant time, whether d is connected to a and c as specified by q_2. If it is, we output the solution (a,b,c,d). Remark that any solution (a,b,c) to the core gives rise to a solution (a,b,c,b) to q_2. Thus, at least one solution to q_2 is printed for each solution to the core, and so the delay is at most linear.

6.1 Sufficient condition

We now present a sufficient condition for enumeration with linear delay that builds on two key concepts: *image* and *untangling*. Consider an endomorphism ν of a full query q. The full query consisting of all atoms $R(\nu(\vec{x}))$ such that $R(\vec{x})$ is an atom of q is called an *image* of q. Figure 2 depicts examples of images. By definition, the core of q is an image. Another trivial image is the query q itself.

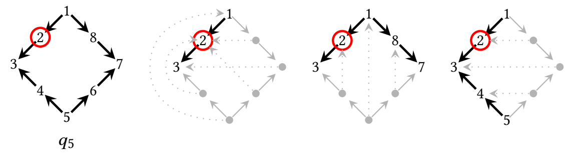


Figure 2: The query q_5 and its images. We depict images in dark and the corresponding endomorphisms in dotted gray. Self-loops are not depicted for readability.

The key property of images is that any solution to the image yields a solution to the full query: Simply instantiate any variable of the query by the value of its image. As an example, for the query q_5 depicted in Figure 2. Every solution (a,b,c) of the core yields the solution (a,b,c,b,a,b,c,b) of q_5, and every solution (a,b,c,d,e) of the rightmost depicted image yields the solution (a,b,c,d,e,d,c,b) of q_5. Hence, if the image is acyclic, we can use it to generate many "simple" solutions in linear time and constant delay, giving us some time to compute more complicated solutions.

The *restriction* of a conjunctive query q to a set of its variables S, denoted $q|_S$, is the query obtained from the self-join-free query associated to q by removing all variables not in S from its atoms. Notice that the arities of the relations may change. An *untangling* of q is a sequence of images $I_0, \ldots, I_k$ of q such that: $I_0 = q$, I_k is acyclic, and for all $0 \leq i < k$: I_{i+1} is an image of I_i and the restriction of I_i to its variables that do not appear in I_{i+1} is acyclic.

Example 12. Consider the queries q_6 and q_7 depicted in Figure 3. They have the same acyclic core, namely the ternary relation and the three binary relations forming a triangle inside of it. The figure shows an untangling for q_6 and illustrates why there is no such untangling for q_7. Another example of a query that can be untangled is given in Figure 9 in the appendix.

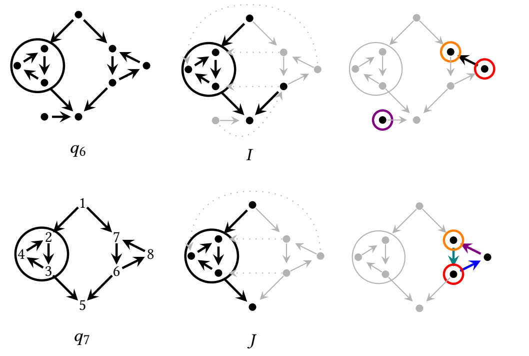


Figure 3: The query q_6 (the circle depicts a ternary relation) followed by an acyclic image I and the result of the corresponding restriction, making the sequence q_6, I an untangling of q_6. By Theorem 13, q_6 can be enumerated with linear delay. The sequence q_7, J is not an untangling of q_7 because the resulting query is cyclic. In fact, q_7 cannot be untangled and we show in Example 14 that its enumeration is hard.

Theorem 13. *Any full conjunctive query that has an untangling can be enumerated with linear delay (after linear preprocessing time).*

Proof sketch. Let $I_0, I_1, \cdots, I_k$ be an untangling of q. The proof goes by induction on k. If $k = 0$, then the query is acyclic and can be enumerated with linear delay by Theorem 2. If $k > 0$, consider the image I_1. Let q' be the restriction of q to its variables not in I_1. By construction, q' is acyclic and can be enumerated with linear delay. By induction, I_1 can be enumerated with linear delay. We show that q can then be enumerated with linear delay as well.

On an input database D, the enumeration of q works as follows. We start the enumeration for I_1. For each solution $\bar{a}$ to I_1, as witnessed by a mapping μ from the variables of I_1 to the elements of D, we compute in linear time a database D' over the schema of q' as follows. By definition, each atom $R'(\vec{x})$ of q' is computed from an atom $R(\bar{y})$ of q by removing all variables in I_1. We filter $R(\vec{y})$ to only keep the tuples that "agree" with $\vec{a}$, and then we project the result to only keep the columns corresponding to $\vec{x}$. More formally, let $\vec{u}$ be the variables of $\bar{y}$ occurring in I_1 and $\vec{v}$ those not occurring in I_1. Then, $R'(D')$ consists of all the tuples $\nu(\vec{v})$ such that ν is a mapping from $\bar{y}$ to a tuple of R with $\mu(\vec{u}) = \nu(\vec{u})$.

We then start the enumeration for q' on D'. Every solution to q' can be combined with $\bar{a}$ to form a solution to q, and the algorithm outputs all solutions with no repetitions. Since I_1 is an image, each $\bar{a}$ yields at least one solution to q', and we get linear delay. ◻

6.2 Necessary condition

We provide a condition identifying full queries that cannot be enumerated with linear delay, assuming sHyperclique. We essentially reduce sHyperclique to the computation of a constant number of solutions to the query in linear time. The construction is illustrated in the next example.

Example 14. Consider again the query q_7, depicted in Figure 3. We show that we cannot produce 3 solutions to q_7 in linear time

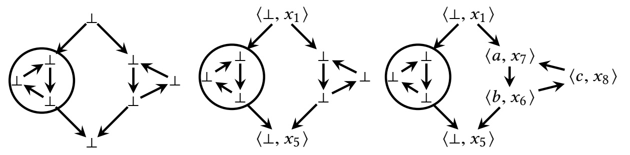

⊥
⊥
⊥
⊥
⊥
⊥
⊥
⊥
$\langle \perp, x_1 \rangle$
$\langle \perp, x_5 \rangle$
$\langle \perp, x_1 \rangle$
$\langle a, x_7 \rangle$
$\langle b, x_6 \rangle$
$\langle c, x_8 \rangle$
$\langle \perp, x_5 \rangle$

Figure 4: Illustrations of solutions to q_7 obtained by the core tagging map, the tagging that maps the right side to the left, and the identity tagging map.

unless we can test in $O(m)$ time whether a graph with m edges contains a triangle (which is not possible assuming sHyperclique).

Given an input graph G, we construct a database D as follows. For every edge (a, b) of G with $a < b$, we add to D the facts $R(\langle a, x_6 \rangle, \langle b, x_8 \rangle)$, $R(\langle b, x_8 \rangle, \langle a, x_7 \rangle)$, $R(\langle a, x_7 \rangle, \langle b, x_6 \rangle)$, $R(\langle \perp, x_1 \rangle, \langle a, x_7 \rangle)$, and $R(\langle b, x_8 \rangle, \langle \perp, x_5 \rangle)$. We also add to D the facts $R(\perp, \perp)$, $R(\langle \perp, x_1 \rangle, \perp)$, $R(\perp, \langle \perp, x_5 \rangle)$, and $S(\perp, \perp, \perp)$. The number of facts of D is linear in the number of edges of G, and this construction can be done in linear time in the size of G.

A case analysis on the tagging map, displayed in Figure 4, shows that the data part of each solution to q_7 contains either the elements with $\perp$ or we can extract from it a triangle of G. As there are only a constant number of solutions of the first kind, any linear delay algorithm will test the existence of a triangle of G in linear time.

Theorem 15 builds upon the idea outlined in Example 14, and provides a necessary condition for linear delay tractability. An endomorphism of a query is said to be *idempotent* if it is the identity on its image. As an example, in Figure 3 the endomorphism witnessing J is idempotent while the one for I is not.

Theorem 15. *Let q be a full conjunctive query. If q contains a set S of variables such that the restriction $q|_S$ is cyclic, and all its endomorphisms are idempotent and have an image containing either all the variables in S or none of them, then q cannot be enumerated with linear delay (and linear preprocessing), assuming sHyperclique.*

Proof sketch. We show that the enumeration for q is at least as hard as the enumeration for $q|_S$, using a construction that assigns the constant $\perp$ to the extra variables and a case analysis according to the endomorphism producing each answer. The idempotency and the fact that each image that contains a variable of S contains all variables of S guarantee that each answer that contains a non-constant value corresponds to an answer to $q|_S$. ◻

6.3 Other cases

The conditions from Section 6.1 and Section 6.2 do not cover all queries. We conclude this section with an example-driven illustration of the difficulty in achieving a full classification for full queries. We first give an example of a query that cannot be enumerated with linear delay but does not fulfill the condition of Theorem 15. We next give one example of a query that can be enumerated with linear delay but cannot be untangled. Finally, we specify a third query whose complexity remains unknown.

Example 16. Consider the query q_8 of Figure 5. The core of the query is the ternary atom and the triangle within it and is therefore acyclic. The query cannot be untangled because the core is the

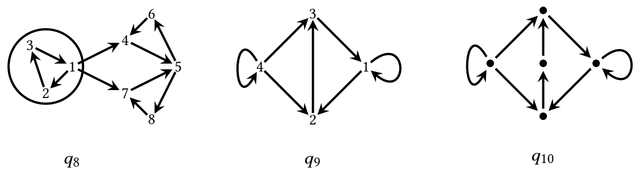


Figure 5: The hard query q_8 of Example 16, the easy query q_9 of Example 17, and the unresolved query q_{10} of Example 18.

only acyclic image and the result of the associated restriction yields a query with a cyclic core. The reader can also verify that the condition of Theorem 15 is not met. Nevertheless, q_8 cannot be enumerated with linear delay unless we can find a triangle in a graph in time linear in the number of its edges, a special case of sHyperclique. The proof idea is that we encode the triangles of the graph into both query triangles that are not covered by a ternary relation and assign constants to all other variables. Except for one solution, all solutions over this encoding identify triangles in the input graph, though they do not all originate in the same query triangle. The details can be found in the appendix.

Example 17. Consider the query q_9 depicted in Figure 5. Its core is a self-loop and is acyclic. One can verify that this query cannot be untangled. However, its solutions can be efficiently enumerated as follows. For each edge (b, c) of the database, we maintain two lists: the list of self-loops a such that (a, b) and (c, a) are edges, and a similar list of self-loops where (a, b) and (a, c) are edges. The algorithm goes through all self-loops a, and for each such self-loop, we consider all edges (b, c). If the triple (a, b, c) satisfies any of the two patterns, we add a to the corresponding list for (b, c). In addition, if (a, b, c) satisfies the first pattern, then for each a' in the second list for (b, c), we output the solution (a, b, c, a'). Similarly, if (a, b, c) satisfies the second pattern, then for each a' in the first list for (b, c), we output the solution (a', b, c, a) (unless $a' = b = c = a$ for avoiding duplicates). It can be easily verified that all solutions are printed and that no solution is printed twice. Moreover, as every self-loop a results in a solution (a, a, a, a), the algorithm exhibits linear delay. It is worth noting that, as before, the memory this algorithm uses may be linear in the output size (which may be larger than linear in the database size).

Example 18. Finally, consider the query q_{10} of Figure 5. It cannot be untangled. The ideas used in Example 17 can no longer be used as the middle part is now a path of length two and there may be more than a linear number of those. The idea from Example 16 does not work either because similarly encoding triangles yields a linear number of non-triangle solutions. Whether q_{10} can be enumerated with linear delay remains open.

7 FULL CONJUNCTIVE QUERIES AND CONSTANT DELAY

In this section, we aim to distinguish the queries that can be enumerated with constant delay after a linear time preprocessing from those where this is not possible. As in the previous section, all our queries will be full cyclic queries with an acyclic core.

7.1 Mirror queries

We start by showing that some cyclic queries can be solved with constant delay following linear preprocessing due to self-joins. This happens when half of the query can be seen as a reflection of the other half, as demonstrated in the following example.

Example 19. The cyclic query q_1 of Figure 1 can be enumerated with constant delay as we describe next. First notice that the core p of q_1, depicted by $1 \longrightarrow 2 \longrightarrow 3$, is acyclic and free-connex, and so by Theorem 2, p can be enumerated with constant delay after linear preprocessing time. We will maintain a table, initialized as empty, that on entry a pair (a, b) returns a list of elements c such that (a, c, b) is a solution to p. The enumeration of q_1 works as follows. We perform the preprocessing necessary for p, start its enumeration, and for every solution (a, c, b) we do the following. First, we output the solution (a, c, b, c). Then, for every c' that is given by the table for the entry (a, b), we output the solutions (a, c, b, c') and (a, c', b, c). Finally, we add c to the table entry (a, b). We get that the delay is constant, and every result is printed once.

Example 19 puts forward the following sufficient condition. We call a query a *mirror* if it has an acyclic image I such that the remaining atoms form a query isomorphic to I, where the isomorphism is the identity on the variables shared between the image and the remaining atoms.

Proposition 20. *Any full mirror conjunctive query can be enumerated with constant delay after a linear preprocessing time.*

As in Example 19, in the proof of Proposition 20 we use during the enumeration a memory linear in the output size (which may not be linear in the input size).

7.2 The effect of unary atoms

In contrast to the self-join-free case, we show that unary atoms can affect query complexity. Intuitively, when adding a unary atom to one of the sides of a mirror query, the solutions to this side no longer form all possible solutions to the other side, and so the algorithm we proposed for mirror queries fails. The following example differs from the previous one only by the addition of a unary atom.

Example 21. Consider again the query q_2 depicted in Figure 1. We have seen in Example 11 that it can be enumerated with linear delay. It turns out that it cannot be enumerated with linear preprocessing and constant delay unless we can test in linear time whether a graph contains a triangle, contradicting sHyperclique. Given a graph G with m edges, we construct a database D as follows. For every edge (a, b) of G with $a < b$, we add to D the facts $R(\langle a, x_1\rangle, \langle b, x_2\rangle)$, $R(\langle b, x_2\rangle, \langle b, x_3\rangle)$, $R(\langle a, x_1\rangle, \langle b, x_4\rangle)$, and $R(\langle b, x_4\rangle, \langle a, x_3\rangle)$, together with the fact $\text{Red}(\langle b, x_2\rangle)$. By considering all endomorphisms of q_2, as we explained when introducing the tagging technique, we get that a solution in $q(D)$ is either obtained by the identity automorphism and so it is of the form $(\langle a, x_1\rangle, \langle b, x_2\rangle, \langle b, x_3\rangle, \langle c, x_4\rangle)$ for a triangle (a, b, c) in G, or it is obtained by the endomorphism that maps x_4 to x_2 and so it is of the form $(\langle a, x_1\rangle, \langle b, x_2\rangle, \langle b, x_3\rangle, \langle b, x_2\rangle)$ for an edge (a, b) in G. As there are at most m solutions of the latter kind, any constant delay enumeration of q in D would be able to test in $O(m)$ whether G has a triangle.

This added difficulty is caused by any structure that breaks the symmetry between the two sides. As an example, we could replace the unary atom in q_2 by a directed path of length 2 starting in x_2 and still obtain the hardness with a similar proof.

7.3 The effect of "spikes"

While introducing unary atoms can make queries harder, we will now see that introducing "dangling" binary atoms may make queries easier. Consider the query q_5 of Figure 2 together with the three queries depicted in Figure 6. Notice that all four agree on the central loop and have the same acyclic core $1 \longrightarrow \textcircled{2} \longrightarrow 3$. They can be untangled and therefore enumerated with linear delay due to Theorem 13. We prove in the sequel that q_{11} and q_{12} can even be enumerated with constant delay, while q_5 and q_{13} cannot without a major computational breakthrough.

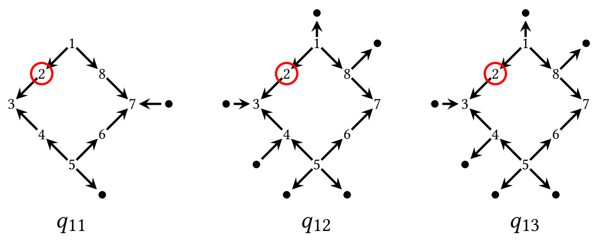


Figure 6: Queries similar to q_5 with a different status regarding constant delay enumeration.

Removing the unary atom from q_5 results in a mirror query that can be enumerated with constant delay according to Proposition 20. This unary atom renders q_5 difficult, and the proof (in the appendix) is similar to that of Example 21. The spikes of q_{11} make it so that all cycle edges appear in an acyclic image. This can be used to devise an efficient algorithm as we explain next.

Example 22. Consider q_{11} and its images of Figure 7. The top and left images can be enumerated with linear preprocessing and constant delay as they are acyclic and free-connex. We start by enumerating the solutions of the left image, and maintain a lookup table as follows. To any solution (a, b, c, d, e, f) of the left image, we add the pair (d, e) to the table at the entry (a, b, c, f). We output the solution $(a, b, c, d, e, d, c, b, f, b)$ induced by this image. Once we are done with the left image, we start the enumeration of the top image. To any solution (a, b, c, d, e, f) of the top image, we output the solution $(a, b, c, b, a, f, e, f, b, d)$ induced by this image. Moreover, for every pair (u, v) in the table for the entry (a, b, c, d), the tuple (a, b, c, u, v, d, e, f) is a solution for the main loop. We go through all edges outgoing from v and all edges incoming to e to get a solution to q_{11} in constant delay. Altogether, we get a constant delay algorithm that outputs all solutions to q_{11}, possibly 3 times. We remove the duplicates using the Cheater's Lemma.

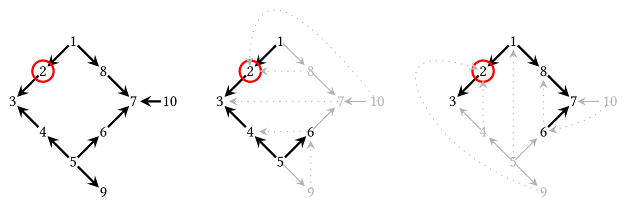


Figure 7: The query q_{11}, its left image, and its top image.

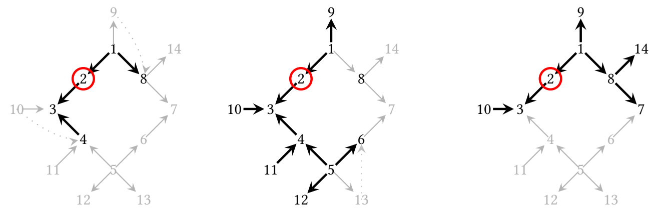


Figure 8: The small, left, and top images of q_{12}. We do not draw the entire endomorphisms for readability.

We show next that q_{12} can also be efficiently solved even though it does not have acyclic images covering all edges of the main loop.

Example 23. The query q_{12} can be enumerated with constant delay using its images depicted in Figure 8. The idea is that we enumerate the solutions to the small image, and for every such assignment, we have a way to efficiently find all solutions that extend it. We first find all of its extensions to the top image and print the solutions implied by these. While we do this, we store the possible assignments to x_7. Then, we do the same with the left image and store the possible assignments to x_6. Next, we go over all pairs of a value for x_6 and a value for x_7 and check whether there is an edge between them. If there is an edge, we have found a non-trivial assignment to the cycle, and we can enumerate all possible spike assignments to get all solutions to the full query. The crux of this proof is that we can show that the number of pairs of values we need to check is bounded by the number of "simple" solutions we find using the two images. Thus, the "simple" solutions provide enough time to perform this computation while still achieving constant delay.

The tractability that the spikes of q_{12} introduce does not stem from the mere number of spikes. We next give evidence of the difficulty of enumerating the answers to the query q_{13}, which differs from q_{12} only by the direction of one edge.

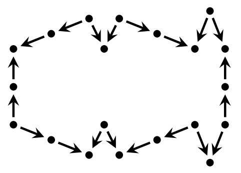
Example 24. We can show that efficient enumeration for q_{13} would imply a breakthrough in algorithms for triangle detection in unbalanced tripartite graphs, contradicting the following hypothesis. The Vertex-Unbalanced Triangle Detection (VUTD) hypothesis[5] assumes that, for every constant $\alpha \in (0, 1]$, it is not possible to test the existence of a triangle in a tripartite graph with vertex sets $|V| = n$ and $|U| = |W| = \Theta(n^{\alpha})$ in time $O(n^{1+\alpha})$. We encode triangle finding with tagging in the query like before, with two additional tricks. To handle the two spikes leaving x_5, the value tagged by x_5 corresponds to an edge between U and V. The same idea cannot be applied to x_8 as it would result in an image that contains vertices of the three parts that do not correspond to a triangle. The spike leaving x_8 makes one of the images correspond to two edges between U and W. Here, we rely on the fact that our

[5]The VUTD hypothesis [4] is less known than the previous ones used in this paper. However, it does represent a computational barrier, and relying on it gives evidence of the difficulty of the task at hand.

input graph is unbalanced to claim that there are not too many solutions of this kind.

7.4 An open problem

The techniques developed above for constant delay algorithms or hardness proofs fail on the full query q_{14} depicted below, and its classification is left for future work. Note that q_{14} is not a mirror, even though it consists of only one cycle and it has an acyclic image.



8 CONCLUSION

We have initiated the study of the fine-grained complexity of conjunctive queries with self-joins. We have seen that queries with low arity can be enumerated with constant delay iff their associated self-join-free query can. We do not know whether this property remains true with linear delay enumeration. There are several ways in which the complexity analysis is more complicated for queries with self-joins. In particular, the complexity may change with the reordering of variables inside an atom, with the addition of unary atoms, or with the addition of binary atoms where only one of their variables appears in the rest of the query (called "spikes" above). Despite showing several ways in which self-joins can be used in algorithms for queries that are otherwise hard, as well as showing ways to prove the hardness of queries with self-joins, determining the complexity of some queries (see Example 18 and Section 7.4) is left open. Most of our results concern full conjunctive queries and going beyond this would require significant additional work.

ACKNOWLEDGMENTS

We thank Louis Jachiet for his valuable input during our discussions. This work was funded by the French government under management of Agence Nationale de la Recherche as part of the "Investissements d'avenir" program, reference ANR-19-P3IA-0001 (PRAIRIE 3IA Institute), and by the ANR project EQUUS ANR-19-CE48-0019.

REFERENCES

[1] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*. Springer, 208–222.

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

[3] Johann Brault-Baron. 2013. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre.* Ph.D. Dissertation. Université de Caen.

[4] Karl Bringmann and Nofar Carmeli. 2022. Unbalanced Triangle Detection and Enumeration Hardness for Unions of Conjunctive Queries. https://doi.org/10.48550/ARXIV.2210.11996

[5] Nofar Carmeli and Markus Kröll. 2021. On the enumeration complexity of unions of conjunctive queries. *ACM Transactions on Database Systems (TODS)* 46, 2 (2021), 1–41.

[6] Ashok K Chandra and Philip M Merlin. 1977. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the ninth annual ACM symposium on Theory of computing*. 77–90.

[7] Wojciech Kazana and Luc Segoufin. 2013. Enumeration of monadic second-order queries on trees. *ACM Transactions on Computational Logic (TOCL)* 14, 4 (2013), 1–12.

[8] Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. 2022. Enumeration for FO queries over nowhere dense graphs. *ACM Journal of the ACM (JACM)* 69, 3 (2022), 1–37.

[9] Luc Segoufin and Wojciech Kazana. 2011. First-order query evaluation on structures of bounded degree. *Logical Methods in Computer Science* 7 (2011).

[10] Luc Segoufin and Wojtek Kazana. 2020. First-order queries on classes of structures with bounded expansion. *Logical Methods in Computer Science* 16 (2020).

[11] Mihalis Yannakakis. 1981. Algorithms for acyclic database schemes. In *VLDB*, Vol. 81. 82–94.

A DETAILS OMITTED FROM SECTION 3

Proposition 3. *For any conjunctive query q, its associated self-join-free query q' is at least as hard as q for the evaluation problem, the testing problem, and the enumeration problem.*

Proof. Given a database D for q we compute a database D' for q' such that the answers of q over D can be reconstructed from the answers of q' over D'. The database D' is constructed from D by duplicating each relation R as many times as its number of occurrences in q, one for each new symbol used in q' as a replacement of R. The reduction can be done in linear time, and $q(D) = q'(D')$. □

B DETAILS OMITTED FROM SECTION 4

Theorem 7. *Let q be a minimal conjunctive query of arity at most one. Then, its solutions can be computed in linear time iff it is acyclic, assuming sHyperclique.*[6]

Proof. If q is acyclic, since it is Boolean or unary, it is free-connex, and so it can be answered with constant delay after linear preprocessing time by Theorem 2. This finds all solutions in linear time. Assume now that q is cyclic.

To show that q is hard, we use a reduction based on the tagging technique. Let q' be the self-join-free query associated to q. As q' is cyclic, its set of solutions cannot be computed in linear time by Theorem 2, assuming sHyperclique. Let D' be a database for q' and let D be the database constructed from D' using the tagging technique.

If q is Boolean, since it is also minimal, every endomorphism is an automorphism, so by Lemma 5, $q(D)$ is true iff $q'(D')$ is true too.

Otherwise, q is unary, and let x be the free variable of q. From Lemma 5, we get that $q'(D') = \{a \mid \langle a, x\rangle \in q(D)\}$. As q is unary, the total size of $q(D)$ is linear in the size of D and so also in the size of D'. Hence, if we could compute all the solutions $q(D)$ in linear time, we could compute all the solutions to $q'(D')$ at the same time simply by ignoring the solutions in $q(D)$ that are not tagged by x. □

Missing details for q_4. The enumeration of q_4 in linear delay can be done as follows. We first precompute the set S of all elements a such that $P(a)$ is a fact of the database and a has incoming and outgoing edges. Notice that this is an acyclic property and therefore S can be computed in linear time. Notice also that any element of S is a solution to q. For every element a of S we do the following. First, we consider all edges $E(b, c)$. If $E(b, a)$ is an edge, then we store c in a temporary table T. Next, we start a new scan of all edges $E(b, c)$. If $E(a, c)$ is an edge and $b \in T$, then we can safely output b. For each $a \in S$, at least one solution is printed, namely a, and the delay between checking two elements of S is linear.

Theorem 8. *Let q be a minimal binary conjunctive query. If q is not acyclic free-connex, then there is no linear preprocessing and constant delay enumeration algorithm for answering q, assuming Hyperclique.*

Proof. Let q be a minimal binary query, and let q' be its associated self-join-free query. Assume q is acyclic but not free-connex. See the paper body for the proof in the cyclic case. We use the following conjecture which is implied by Hyperclique. BMM conjectures that two $n \times n$ Boolean matrices A and B cannot be multiplied in time $O(n^2)$. We show that we cannot compute $q(D)$ in time linear in the input and output sizes, assuming BMM. To see this, we need to recall how the same result was shown in the self-join-free case. Given two $n \times n$ matrices, there is a construction that encodes their multiplication task into any acyclic non-free-connex query using n domain values [1]. If the query could be solved in time linear in the input and output sizes, this construction would multiply the matrices in $O(n^2)$ time. We use the same construction to define a database D' for q'. Using the tagging technique to produce D from D', the data part of $q(D)$ contains $q'(D')$ and possibly additional answers. But, as the query is binary, the size of $q(D)$ is at most $O(n^2)$. Hence, if we could compute $q(D)$ in linear time, we would compute $q'(D')$ with an overhead of no more than $O(n^2)$ by ignoring the unwanted solutions, contradicting BMM. □

C DETAILS OMITTED FROM SECTION 5

Theorem 9. *It is possible to test in linear time whether a conjunctive query has a solution iff it has an acyclic core, assuming sHyperclique.*

Proof. Let q be a conjunctive query with free variables $\bar{x}$, and let p be the Boolean version of the core of q. Notice that p has a solution iff q has a solution. If p is acyclic, then we can test whether it has a solution in linear time using the Yannakakis algorithm [11]. Otherwise, p is a minimal cyclic Boolean query. By Theorem 7, p cannot be tested in linear time assuming sHyperclique. Hence, one cannot test in linear time whether q has a solution. □

D DETAILS OMITTED FROM SECTION 6

Theorem 15. *Let q be a full conjunctive query. If q contains a set S of variables such that the restriction $q|_S$ is cyclic, and all its endomorphisms are idempotent and have an image containing either all the variables in S or none of them, then q cannot be enumerated with linear delay (and linear preprocessing), assuming sHyperclique.*

Proof. According to Theorem 2, the testing problem for $q|_S$ cannot be solved in linear time since it is cyclic and self-join free, assuming sHyperclique. In particular, it cannot be enumerated in linear delay. According to the following Lemma 25, this implies the same for q.

[6]In fact, the proof of the unary case can be applied to any minimal conjunctive query in which one atom contains all free variables.

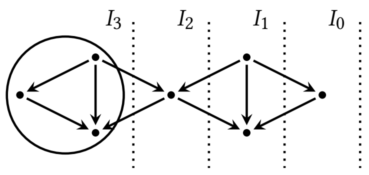


Figure 9: This query can be untangled, as witnessed by the sequence I_0, I_1, I_2 I_3, where the I_i are the images corresponding to the part to the left of each dotted vertical line.

Lemma 25. *Let q be a full conjunctive query whose endomorphisms are all idempotent, and let S be a set of variables of q such that every image of q contains either all of S or none of them. Then, q is at least as hard for enumeration as $q|_S$.*

Proof of Lemma 25. Let D' be a database for $q|_S$. We construct a database D for q as follows. Consider an atom $R(\vec{y})$ of q, and let $R'(\vec{x})$ be its restriction in $q|_S$. For every fact $R'(\vec{a})$, we construct the tuple $\vec{b}$ from $\vec{a}$ by inserting $\perp$ in all positions of $\vec{y}$ that do not contain variables of S. We then insert to D the fact $R(\bar{c})$ with $c_i = \langle b_i, y_i \rangle$. Notice that, by construction, the elements of D are pairs of the form $\langle a, x \rangle$ such that a is either $\perp$ or an element of D' and x is a variable of q. Moreover, whenever $x \in S$ then a is an element of D'.

Consider now an answer $\bar{a}$ of q on D. Let μ be the mapping witnessing the fact that $\bar{a}$ is a solution. As in the tagging technique, we perform a case analysis depending on the endomorphism ν of q induced by μ. If the image of ν does not intersect with S, then the data part of μ contains only the elements $\perp$ and, as there is only a constant number of such solutions, we can safely ignore them. Otherwise, by our assumption, the image of ν must contain S, and ν is the identity on S. This implies that for any variable $x \in S$, $\mu(x)$ is of the form $\langle b, x \rangle$ where b is an element of D'. We claim that the mapping μ' defined by $\mu'(x) = b$ whenever $x \in S$ and $\mu(x) = (b, x)$ witnesses a solution of $q|_S$ on D'. For all $R'(\vec{x})$ in $q|_S$, consider $R(\vec{y})$. Let $\vec{u}$ be the variables of y occurring in S. Since $\mu(\vec{y}) \in R$ and ν is idempotent, we have that $\mu(\vec{u})$ is a tuple of R' tagged by $\vec{x}$. Hence, our construction "filters enough", and the data part gives a solution of $q|_S$. Notice also that all solutions to $q|_S$ can be obtained this way. Hence, the enumeration of q will enumerate all the solutions to $q|_S$. □

□

Missing details in Example 16. The condition of Theorem 15 is not met because any set S of variables containing a cycle must contain the variable x_5; if it also contains x_7 and x_8 then the image $x_1, \cdots, x_6$ intersects with S in x_5 but does not contain x_7. The case where S contains x_4 and x_6 is symmetric.

The query q_8 cannot be enumerated with linear delay unless we can find a triangle in a graph in time linear in the number of its edges, a special case of sHypercliqe. To show this, we use the tagging technique as follows. Given an input graph G, we construct in linear time the following database D. To each edge (a, b) with $a < b$, we add to D the facts $R(\langle a, x_4 \rangle, \langle b, x_5 \rangle)$, $R(\langle a, x_5 \rangle, \langle b, x_6 \rangle)$, $R(\langle b, x_6 \rangle, \langle a, x_4 \rangle)$, $R(\langle a, x_5 \rangle, \langle b, x_8 \rangle)$, $R(\langle b, x_8 \rangle, \langle a, x_7 \rangle)$, $R(\langle a, x_7 \rangle, \langle b, x_5 \rangle)$. We also add to D the facts $R(\langle \perp, x_1 \rangle, \langle a, x_4 \rangle)$ and $R(\langle \perp, x_1 \rangle, \langle a, x_7 \rangle)$. Finally we also add the facts $R(\langle \perp, x_1 \rangle, \langle \perp, x_2 \rangle)$, $R(\langle \perp, x_2 \rangle, \langle \perp, x_3 \rangle)$, $R(\langle \perp, x_3 \rangle, \langle \perp, x_1 \rangle)$ and $S(\langle \perp, x_1 \rangle, \langle \perp, x_2 \rangle, \langle \perp, x_3 \rangle)$. Consider now a solution to q over D. Due to the tagging technique, the solution induces an endomorphism of q (the tagging map). Let I be the corresponding image. If I is the core, then the data part of the solution is $(\perp, \perp, \perp, \perp, \perp, \perp, \perp, \perp)$. If I is the top or bottom image, then the data part of the solution is $(\perp, \perp, \perp, a, b, c, a, c)$ such that (a, b, c) is a triangle in G with $a < b < c$. If I is q_8, then the data part of the solution is $(\perp, \perp, \perp, a_1, b, c_1, a_2, c_2)$ such that (a_1, b, c_1) and (a_2, b, c_2) are triangles in G with $a_1, a_2 < b < c_1, c_2$. Overall, there is only one solution that does not correspond to a triangle in G. Hence, if we can enumerate the solutions to the query with linear delay and linear preprocessing, then we can detect in linear time the presence of a triangle in G.

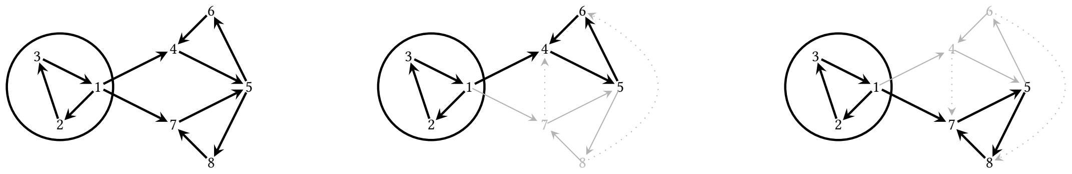


Figure 10: From left to right: query q_8, its "top" image, and its "bottom" image.

E DETAILS OMITTED FROM SECTION 7

Proposition 20. *Any full mirror conjunctive query can be enumerated with constant delay after a linear preprocessing time.*

Proof. We essentially do as in Example 19. Let I be the acyclic image witnessing the mirror, and let $\bar{x}$ be the shared variables (that is, variables that appear in I and also in atoms of the query that are not in I). Assume the free variables are ordered such that $\bar{x}$ appear first, then the other image variables, and then the remaining variables. We will maintain a table that upon entry $\bar{a}$ returns a list of tuples $\bar{b}$ such that $\bar{a}\bar{b}$ forms a solution to I. As I is full and acyclic, it can be enumerated with constant delay after a linear preprocessing time. We perform the preprocessing of I and start its enumeration. For every solution $\bar{a}\bar{b}$ to I, we do the following. First, we output the solution $\bar{a}\bar{b}\bar{b}$. Then, for every b' given by the table for $\bar{a}$, we output the solutions $\bar{a}\bar{b}\bar{b}'$ and $\bar{a}\bar{b}'\bar{b}$. Finally, we add $\bar{b}$ to the table with entry $\bar{a}$. Notice that the delay is constant, and all solutions are printed with no repetition. □

Missing details for the query q_5. The query q_5 of Figure 6 cannot be enumerated with constant delay assuming the hardness of triangle detection as conjectured in sHyperclique. Let G be a graph. We construct the following database D. For every node a of G, we add the following facts to D: $R(\langle a, x_1\rangle, \langle a, x_2\rangle)$, $R(\langle a, x_2\rangle, \langle a, x_3\rangle)$, $R(\langle a, x_4\rangle, \langle a, x_3\rangle)$, $R(\langle a, x_5\rangle, \langle a, x_4\rangle)$, $R(\langle a, x_1\rangle, \langle a, x_8\rangle)$, and $Red(\langle a, x_2\rangle)$. For every edge (a, b) of G with $a < b$, we add the following facts to D: $R(\langle a, x_5\rangle, \langle b, x_6\rangle)$, $R(\langle a, x_6\rangle, \langle b, x_7\rangle)$, $R(\langle a, x_8\rangle, \langle b, x_7\rangle)$. Consider now a solution in $q_5(D)$. It induces an endomorphism of q_5 as explained in Section 3. We do a case analysis depending on the image I of this endomorphism. If I is the core, then the data part of the solution involves only one node a of G. The same holds if I involves only the left part of q_5: the variables from x_1 to x_5. If I involves the top part of q_5, the variables x_1, x_2, x_3, x_8, x_7, then the data part of the solution corresponds to an edge of G. The data part of all the remaining images induces a triangle in G. Hence, the data part of all the solutions in $q_5(D)$ contain a triangle of G except for a number of solutions linear in the size of G. Hence, a constant delay enumeration algorithm for q_5 with linear preprocessing would induce a linear test of whether G contains a triangle.

The reader may verify that the encoding used to prove the hardness of q_5 does not work for q_{11} as the spikes make bigger images. With this encoding, there would be one solution per path of distance two in G, and so the triangles might be detected only after quadratic time, which does not contradict sHyperclique.

Missing details for the query q_{12} of Example 23. We first enumerate the solutions of the "small" image, depicted in Figure 8. For each such solution (a, b, c, d, e), we do the following.

We compute all possible assignments to x_7 while producing some solutions to q_{12} using the top image as follows. We refer to the subquery induced by x_7, x_8, x_{14} as the top complement. We enumerate the solutions to the top complement in which x_8 maps to e. For each such solution (j, e, k), we insert the value j to the set $T_7(e)$. Notice that (a, b, c, j, e, e, d, k) is a solution for the top image. We then output the solution to q_{12} derived from this solution using the mapping of the top image. This is the solution $(a, b, c, b, a, e, j, e, e, d, a, b, e, k)$.

We continue similarly with the left image as follows. We refer to the subquery induced by $x_4, x_5, x_6, x_{11}, x_{12}$ as the left complement. We enumerate the solutions to the left complement in which x_4 maps to d. For each such solution (d, f, g, h, i), we insert the value g to the set $T_6(d)$, and we insert the value f to the set $T_5(d, g)$. Notice that $(a, b, c, d, f, g, e, d, h, i)$ is a solution for the left image. We then output the solution to q_{12} derived from this solution using the mapping of the left image. This is the solution $(a, b, c, d, f, d, c, b, e, d, h, i, g, c)$.

Next, for each value u in $T_6(d)$ and each value v in $T_7(e)$ we check whether there is an edge (u, v) in our database. If there is, for each value w in $T_5(d, u)$, we have found a solution (a, b, c, d, w, u, v, e) to the query induced by the cycle. We enumerate all possible spike assignments to get all solutions to the full query. More specifically, for each s_1 with an edge (a, s_1), for each s_2 with an edge (s_2, c), for each s_3 with an edge (s_3, d), for each s_4 with an edge (w, s_4), for each s_5 with an edge (w, s_5), and for each s_6 with an edge (e, s_6), we print the solution $(a, b, c, d, w, u, v, e, s_1, s_2, s_3, s_4, s_5, s_6)$ to q_{12}.

The process described in the last paragraph finds every solution to q_{12} exactly once. The previous two paragraphs find some of the solutions to q_{12}, where the process described in each paragraph alone does not yield duplicate solutions. Overall, we find all solutions to q_{12}, and every solution is obtained at most 3 times.

Let us now discuss the time complexity. Each time we enumerate something in our algorithm, it is acyclic and can be done with linear preprocessing and constant delay. The "danger" is that we need to go over all pairs of a value in $T_6(d)$ and a value in $T_7(e)$ and check whether they share an edge. In the worst case, if this check always fails, we can get a step of quadratic delay, which is a problem if we want linear preprocessing and constant delay. However, we will show later that the number of such pairs is bounded by the number of solutions we print before these checks, using the top and left images. Overall, we get that, for all n, the time from the beginning of the execution until printing the nth solution is linear in the input size plus n. A general version of the Cheater's Lemma [5, Lemma 7] relies on this to deduce that our algorithm can be tweaked into working with linear preprocessing and constant delay with no duplicates, by withholding some of the solutions before printing them.

Denote by SolTop(e) and SolLeft(d) the number of solutions printed in the second and third paragraphs respectively. It is left to prove that $|T_6(d)| \cdot |T_7(e)| \leq \text{SolTop(e)} + \text{SolLeft(d)}$. Denote by $\text{In}(a)$ and $\text{Out}(b)$ respectively the set of all values that go into a and out of b in the input database; that is $\text{In}(a) = \{v | (v, a) \in R\}$ and $\text{Out}(b) = \{v | (b, v) \in R\}$. We have that $|T_7(e)| \leq \text{Out}(e)$ and $|T_6(d)| \leq \sum_{f \in \text{In}(d)} \text{Out}(f)$, and also $\text{SolTop(e)} = |\text{Out}(e)|^2$ and $\text{SolLeft(d)} = \sum_{f \in \text{In}(d)} |\text{In}(d)||\text{Out}(f)|^2$. Using the known inequality between the arithmetic and quadratic means, we get:

$$|T_6(d)| \cdot |T_7(e)| \leq |T_6(d)|^2 + |T_7(e)|^2 \leq \left(\sum_{f \in \text{In}(d)} \text{Out}(f) \right)^2 + (\text{Out}(e))^2 \leq |\text{In}(d)| \sum_{f \in \text{In}(d)} \text{Out}(f)^2 + \text{Out}(e)^2 = \text{SolTop(e)} + \text{SolLeft(d)}$$

This concludes the proof that q_{12} can be enumerated with linear preprocessing and constant delay.

Missing details for the query q_{13} of Example 24. Let G be an unbalanced tripartite graph. We construct the following database D. For each node $u \in U$, we add to D the facts $R(\langle u, x_1\rangle, \langle u, x_2\rangle)$, $R(\langle u, x_2\rangle, \langle u, x_3\rangle)$, $R(\langle u, x_4\rangle, \langle u, x_3\rangle)$, $R(\langle u, x_1\rangle, \langle u, x_8\rangle)$, and $Red(\langle u, x_2\rangle)$. For each edge $(u, v) \in U \times V$ of G, we add to D the facts $R(\langle (u, v), x_5\rangle, \langle u, x_4\rangle)$ and $R(\langle (u, v), x_5\rangle, \langle v, x_6\rangle)$. For each edge $(v, w) \in V \times W$ of G, we add to D the fact $R(\langle v, x_6\rangle, \langle w, x_7\rangle)$. Finally for each edge $(w, u) \in W \times U$ of G, we add to D the fact $R(\langle u, x_8\rangle, \langle w, x_7\rangle)$. Consider now a solution in $q_{13}(D)$. We perform a case analysis according to the tagging map, as explained in Section 3, see Figure 11. The only images that contain x_5 or x_6 and also x_7 come from the endomorphisms which are the identity on the main loop, and so a solution with this tagging detects a triangle of G. Images that do not contain x_5, x_6, or x_7 result in solutions whose data part is a single node $u \in U$. For images that contain x_5 and possibly also x_6 but not x_7, the data part is an edge (u, v) between U and V. For the image that contains x_7 and not x_5, the data part is a triple (w, u, w') corresponding to two edges between U and W. As U and W are small vertex sets, there cannot be too many solutions of the latter kind. Altogether the number of solutions that do not provide a triangle of G is in $O(n^{1+\alpha})$. Hence, a constant delay algorithm after linear preprocessing time will identify a triangle in G in time $O(n^{1+\alpha})$.

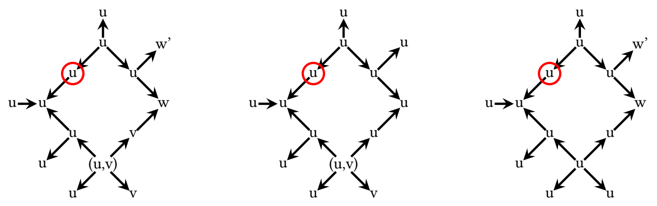


Figure 11: Data parts of solutions in Example 24 with a tagging map that: is the identity on the main loop (left), has x_5 in its image but not x_7 (center), and has x_7 in its image but not x_5 (right).

Notice that, for the construction of Example 24, it is important that the bottom-left spike points outwards. Otherwise, as in q_{12}, the left image may produce triplets of the form (u, v_1, v_2) corresponding to two edges between V and U, and there may be $n^{1+2\alpha}$ of them, which is too many for solving the VUTD problem in the desired time.