The Power of Implicit Acyclicity in the Enumeration Complexity of Database Queries

Nofar Carmeli

The Power of Implicit Acyclicity in the Enumeration Complexity of Database Queries

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Nofar Carmeli

Submitted to the Senate
of the Technion — Israel Institute of Technology Elul
5780 Haifa September 2020

This research was carried out under the supervision of Prof. Benny Kimelfeld, in the Faculty of Computer Science.

Some results in this thesis have been published as articles by the author and research collaborators in conferences and journals during the course of the author's doctoral research period, the most up-to-date versions of which being:

Nofar Carmeli and Markus Kröll. Enumeration complexity of conjunctive queries with functional dependencies. In *21st International Conference on Database Theory (ICDT 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

Nofar Carmeli and Markus Kröll. Enumeration complexity of conjunctive queries with functional dependencies. *Theory of Computing Systems*, pages 1–33, 2019.

Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 134–148, 2019.

Nofar Carmeli, Batya Kenig, and Benny Kimelfeld. Efficiently enumerating minimal triangulations. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 273–287. ACM, 2017.

Nofar Carmeli, Batya Kenig, Benny Kimelfeld, and Markus Kröll. Efficiently enumerating minimal triangulations. *Discrete Applied Mathematics*, 2020.

Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 393–409, 2020.

Acknowledgements

I owe my deepest gratitude and appreciation to my advisor, Benny Kimelfeld, for guiding me on this journey. I thank Benny for his brilliant ideas, advice, kindness, and support. I would also like to thank my co-authors for all that they taught me and to thank the final exam committee for their valuable comments. I am thankful for my office-mates and friends for being there for me every day: Markus Kröll, Yoav Nahshon, Liat Peterfreund, and Eden Saig, and for everyone else from the Computer Science Department of Technion for making it a home that I am sad to leave. Lastly, I thank my wonderful friends and family for their love and support.

The generous financial help of the Technion is gratefully acknowledged.

Contents

List of Figures

List of Algorithms

Abstract

We inspect the fine-grained complexity of answering queries over relational databases. With the ideal guarantees, linear time is required before the first answer to read the input and determine its existence, and then we need to print the answers one by one. Consequently, we wish to identify the queries that can be solved with linear preprocessing time and constant or logarithmic delay between answers. A known dichotomy classifies CQs into those that admit such enumeration and those that do not. The computationally expensive component of query answering is joining tables, which can be done efficiently if and only if the join query is acyclic. However, the join query usually does not appear in a vacuum. For example, it may be part of a larger query, or it may be applied to a database with dependencies. We inspect how the complexity changes in these settings and chart the borders of tractability within. In addition, we consider the task of enumerating query answers with a uniformly random order, and we propose to do so using a random-access structure for representing the set of answers. Our results are accompanied by conditional lower bounds showing that our algorithms capture all tractable cases for some query classes. Among our results, we show that a union of intractable conjunctive queries may be tractable w.r.t. enumeration, while on the other hand, a union of tractable conjunctive queries may be intractable w.r.t. random access. To handle cases where we cannot reach efficient enumeration after linear preprocessing time, we also suggest an algorithm for generating tree decompositions. This algorithm can be used to simplify intractable queries by extracting an acyclic structure.

Chapter 1

Introduction

Evaluating queries over databases is a fundamental and well-studied problem in data management. As data becomes bigger, and data analytics becomes more crucial to digital systems, so grows the importance of characterizing the queries that admit a highly efficient evaluation. In the effort of reducing the computational cost of answering database queries to the very least possible, recent years have seen a substantial progress in understanding the fine-grained data complexity of enumerating query answers.

When evaluating a query over a database, the number of answers may be huge, orders of magnitude larger than the size of the database itself. As we cannot hope to produce this many answers in time linear in the size of the input database, we need to use different complexity measures, and we turn to enumeration complexity. The best time guarantee we can hope for is to output all answers with a constant delay between consecutive answers after a linear preprocessing phase. This is the time it takes to read the database and then write the answers one by one. We denote the class of enumeration problems that can be solved within these time bounds as $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$. This class can be regarded as the most efficient class of nontrivial enumeration problems.

When it comes to query answering, it is common to use *data complexity*. We treat every query as fixed, and we identify it with the following enumeration problem: given a database as input, find all answers to the query over the given database. In particular, for the purpose of complexity analysis, the size of query (which is usually very small compared to the size of the database) is treated as a constant.

The very basic building blocks of common query classes are joins, which allow the combination of several relations in a single query. Joins are also usually the most computationally expensive part of answering queries. Let us first consider queries that comprise solely of joins. Providing even a first answer of such a query in linear time is impossible in general [PY99]. This does not imply, however, that no join queries are in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$. In fact, a classic algorithm by Yannakakis [Yan81b] can answer any join query with an acyclic structure efficiently. Given any cyclic join query, and in the absence of self-joins (i.e., when every relation occurs at most once in the query), Brault-Baron [BB13] showed that it is not possible to determine whether the query

has answers in linear time. This is a conditional lower bound, which holds under a generalization of the assumption that it is not possible to detect a triangle in a graph in linear time. Combining these results leads to a dichotomy for self-join-free join queries: answering such a query is in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$ if and only if the query is acyclic.

We next consider *Conjunctive Queries* (CQs) which consist of join queries followed by projection. Introducing projection may increase the difficulty of answering queries in constant delay. Two different join answers may become identical after projection. Since we do not allow to output duplicates, this reduces the total number of answers, and so we allow the algorithm less time in total to perform the join. Bagan, Durand and Grandjean [BDG07] defined a subclass of acyclic CQs called *free-connex*. An acyclic CQ is called free-connex if the query remains acyclic when adding an atom containing exactly the free-variables. They established that free-connex CQs are in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$, and that self-join-free acyclic CQs that are not free-connex cannot be solved within these bounds. The hardness results here are again conditional lower bounds, and they rely on the assumed hardness of Boolean matrix multiplication.

Combining the dichotomy for acyclic CQs with the lower bounds for cyclic CQs, we get a dichotomy for self-join-free conjunctive queries: assuming the aforementioned hypotheses, answering such a self-join-free CQ is in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$ if and only if the query is acyclic free-connex. In the years following this dichotomy, much work has been conducted to achieve similar results for other classes of queries [Seg15, Dur20].

The next natural step in considering more expressible query classes, is *Unions of Conjunctive Queries*, UCQs for short. UCQs form an important class of queries, as they capture the positive fragment of the relational algebra. Previous work that implies results on the enumeration complexity of UCQs imposes strong restrictions on the underlying database [SV17]. Here, we aim to understand the enumeration complexity of UCQs without such restrictions and based solely on their structure.

A union of tractable CQs is always tractable [Str10]. We inspect the case that a union contains an intractable CQ. We define the notion of union extensions, and show that even if a CQ is not naturally acyclic, it may admit implicit acyclicity as part of the union. Thus, we show that some unions containing intractable CQs are, in fact, tractable. Interestingly, some unions consisting of only intractable CQs are tractable too. The question of finding a full characterization of the tractability of UCQs remains open. Nevertheless, we prove that for several classes of queries, free-connex union extensions fully capture the tractable cases.

Next, we aim to obtain more when we evaluate a tractable query. We seek a structure that supports the more demanding task of a *random permutation*: enumeration in truly random order. Enumeration of this kind is required if downstream applications assume that the intermediate results are representative of the whole result set in a statistically valuable manner. An even more demanding task is that of a *random access*: the retrieval of an answer based on its position. We show how we can use random access to achieve random permutation, and study these two tasks compared to enumeration.

We consider the tractability yardstick of answer enumeration with a logarithmic delay after a linear-time preprocessing phase. We establish that the acyclic free-connex CQs are tractable in all three senses: enumeration, random permutation, and random access; and in the absence of self-joins, it follows from the hardness results we discussed that every other CQ is intractable by each of the three (under fine-grained complexity assumptions). However, the three yardsticks are separated in the case of a UCQ: while a union of acyclic free-connex CQs always admits efficient enumeration, it may not admit efficient random access. We identify a subclass of the unions of acyclic free-connex CQs that admit efficient random permutation, and devise a random-order enumeration algorithm whose delay is logarithmic *in expectation* for the rest.

The classification results we mentioned make no assumptions on the input database. Thus, the hardness results no longer hold in the common case that the database exhibits dependencies among attributes. We study the complexity of enumerating the query answers in the presence of Functional Dependencies (FDs). We show that some queries that are classified as hard are in fact tractable if dependencies are accounted for. We define the notion of FD-extensions, and show that these can convert queries of an intractable form to an acyclic free-connex form. We establish a generalization of the dichotomy to accommodate unary FDs (where one attribute implies another); hence, our classification determines which combination of a query and a set of FDs admits efficient enumeration, random permutation and random access. In addition, we generalize the the results for acyclic CQs to accommodate general FDs (where a combination of attributes imply other attributes). Our results also apply in the presence of cardinality dependencies that generalize FDs.

Our last resort, when it is not possible to use dependencies or union extensions to rewrite queries into a tractable form, is to decompose the query. Given the graph describing a query, our task is to find a tree-decomposition of high quality for this graph. With a tree-decomposition at hand, we can transform an intractable query into an acyclic form [GGS05]. However, this transformation requires a non-linear overhead, and it will not achieve the time bounds of linear preprocessing followed by a constant or logarithmic delay. Nonetheless, using a tree-decomposition can reduce the computation time significantly compared to a naive evaluation, and the quality of the decomposition can affect this time dramatically. As finding the best tree-decomposition by common measures is NP-hard [ACP87], we devise an anytime algorithm that enumerates tree-decompositions. The user can choose to stop the algorithm whenever a decomposition produced is deemed good enough.

Next, we go into more details regarding each of the goals of this thesis.

Answering UCQs with Constant Delay

Using known methods [Str10], a union of tractable enumeration problems is again tractable. As a result, any union of acyclic free-connex CQs is in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$.

However, what happens if some CQs of a union are tractable while others are not? Intuitively, one might be tempted to expect a union of enumeration problems to be harder than a single problem within the union, making such a UCQ intractable as well. As we will show, this is not necessarily the case.

Example 1.1. Let $Q = Q_1 \cup Q_2$ with

$$Q_1(x, y) \leftarrow R_1(x, y), R_2(y, z), R_3(z, x) \text{ and}$$
$$Q_2(x, y) \leftarrow R_1(x, y), R_2(y, z).$$

Even though Q_1 is hard while Q_2 is easy, a closer look shows that Q_2 contains Q_1. This means that Q_1 is redundant, and the entire union is equivalent to the easy Q_2. □

To avoid cases like these, where the UCQ can be translated to a simpler one, it makes sense to consider non-redundant unions. It was claimed that in all cases of a non-redundant union containing an intractable CQ, the UCQ is intractable too [BKS18]. The following is a counter example which refutes this claim.

Example 1.2. Let $Q = Q_1 \cup Q_2$ with

$$Q_1(x, y, w) \leftarrow R_1(x, z), R_2(z, y), R_3(y, w) \text{ and}$$
$$Q_2(x, y, w) \leftarrow R_1(x, y), R_2(y, w).$$

According to the dichotomy of Bagan et al. [BDG07], the enumeration problem for Q_2 is in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$, while Q_1 is intractable. Yet, it turns out that Q is in fact in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$. The reason is that, since Q_1 and Q_2 are evaluated over the same database I, we can use $Q_2(I)$ to find $Q_1(I)$. We can compute $Q_2(I)$ efficiently, and try to extend every such solution to solutions of Q_1 with a constant delay: for every new combination a, c of an output $(a, b, c) \in Q_2(I)$, we find all d values with $(b, d) \in R_3^I$ and then output the solution $(a, c, d) \in Q_1(I)$. Intuitively, the source of intractability for Q_1 is the join of R_1 with R_2 as we need to avoid duplicates that originate in different z values. The union is tractable since Q_2 returns exactly this join. □

As the example illustrates, to compute the answers to a UCQ in an efficient way, it is not enough to view it as a union of isolated instances of CQ enumeration. In fact, this task requires an understanding of the interaction between several queries. Example 1.2 shows that the presence of an easy query within the union may give us enough time to compute auxiliary data structures that can be added to the hard queries in order to enumerate their answers as well. In Example 1.2, we can assume that we have a ternary relation holding the result of Q_2. Then, adding the auxiliary atom $R_{Q_2}(x, z, y)$ to Q_1 results in a tractable structure. We generalize this observation and introduce the concept of *union extensions*. We then use union extensions as a central tool for evaluating the enumeration complexity of UCQs, as the structure of such queries has implications on the tractability of the UCQ.

Interestingly, this approach can be taken a step further: We show that the concept of extending the union by auxiliary atoms can even be used to efficiently enumerate

the answers of UCQs that contain *only* hard queries. We show that UCQs that admit a free-connex union extension are tractable. This results in a sufficient condition for membership in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$ beyond any classification of individual CQs.

To prove the efficiency of union extensions, we present what we refer to as the Cheater's Lemma. This lemma identifies a sufficient condition to show that a problem can be solved in linear preprocessing time and constant delay. To this aim, we define *linear partial time*: a complexity measure that requires that, if we consider the time between the beginning of the run and the production of any answer, this time is linear in the input size and the number of previously produced answers. The Cheater's lemma proves that in order to conclude that a problem is in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$, it is sufficient to show a linear partial time algorithm that produces every unique answer at most a constant number of times.

We prove that for several classes of queries, free-connex union extensions fully capture the tractable UCQs. In particular, a union of two intractable CQs that does not admit such an extension is intractable. The hardness results presented here rely on common lower-bound assumptions such as the hardness of Boolean matrix-multiplication [LG14] and finding a hyperclique in a graph [LWW18].

Finding a full characterization of UCQs with respect to $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$ remains an open problem. Nevertheless, we identify a computational hypothesis on graphs that is tightly related to our problem: unbalanced triangle detection. We show that if we assume the hardness of unbalanced triangle detection, a union of two self-join-free binary CQs is tractable iff it has a free-connex union extension. On the other hand, if such union extensions cover all tractable UCQs, unbalanced triangle detection is necessarily hard. We also discuss the connection between this hypothesis and the well-known 3SUM conjecture [GO95].

Why are lower bounds for UCQs fundamentally more challenging than for CQs? In the case of CQs, hardness results are often shown by reducing a computationally hard problem to the task of answering a query. The reduction encodes the hard problem to the relations of a self-join free CQ, such that the answers of the CQ correspond to an answer of this problem [BDG07, BB13, BKS17]. However, using such an encoding for CQs within a union does not always work. Similarly to the case of CQs with self-joins, relational symbols that appear multiple times within a query can interfere with the reduction. Indeed, when encoding a hard problem to an intractable CQ within a union, a different CQ in the union evaluates over the same relations, and may also produce answers. A large number of such supplementary answers, with constant delay per answer, accumulates to a long delay until we obtain the answers that correspond to the computationally hard problem. If this delay is larger than the lower bound we assume for the hard problem, we cannot conclude that the UCQ is intractable.

The lower bounds we present are obtained either by identifying classes of UCQs for which we can use similar reductions to the ones used for CQs, or by introducing alternative reductions.

We also inspect UCQs with disequalities (i.e., $\neq$ symbols). In the case of CQs with disequalities, the disequalities have no effect on the enumeration complexity [BDG07]. That is, one can simply ignore the disequalities, and the remaining CQ is free-connex if and only if the query is tractable (under the same assumptions). A natural question is: does this happen also with UCQs? We show that the answer is negative: a tractable UCQ may become intractable when adding disequalities. We then show how to identify easy UCQs with disequalities using the same techniques we introduce for UCQs without them.

The final modification we consider is that of restricted space. We discuss the space consumption used in our approach, and demonstrate that in some cases it can be reduced to constant additional space. The question of when exactly this can be done remains open, but we claim in favor of using the measure of linear partial time as an intermediate step for future research.

Enumerating Query Answers in Random Order

As a query-evaluation paradigm, the enumeration approach has the important guarantee that the number of produced results is proportional to the elapsed processing time. This guarantee is useful when the query is a part of a larger analytics pipeline where the answers are fed into downstream processing such as machine learning, summarisation, and search. The intermediate results can be used to save time by invoking the next-step processing (e.g., as in streaming learning algorithms [SK01]), computing approximate summaries that improve in time (e.g., as in online aggregation [HH99, LWYZ19]), and presenting the first pages of search results (e.g., as in keyword search over structured data [HP02, GKS08]). Yet, at least the latter two applications make the implicit assumption that the collection of intermediate results is a representative of the entire space of answers. In contrast, the aforementioned constant-delay algorithms enumerate in an order that is a merely an artifact of the tree selected to utilize free-connexity, and hence, intermediate answers may feature an extreme bias. There has been a considerable recent progress in understanding the ability to enumerate the answers not just efficiently, but also in a ranked manner [DK19, TAG$^+$19]. Yet, to be a statistically meaningful representation of the space of answers, the enumeration order needs to be random.

We investigate the task of enumerating answers in a uniformly random order. To be more precise, the goal is to enumerate the answers without repetitions, and the output induces a uniform distribution over the space of permutations of the answer set. We refer to this task as *random permutation*. Similarly to the recent work on ranked enumeration [DK19, TAG$^+$19], our focus here is on achieving a *logarithmic delay* after a linear preprocessing time. Hence, more technically, the goal we seek is to construct in linear-time a data structure that allows to sample query answers *without replacement*, with a logarithmic-time per sample. Note that sampling *with* replacement has been studied in the past [AGPR99, CMN99] and recently gained a

renewed attention [ZCL+18].

One way of achieving a random permutation is via *random access*—a structure that is tied to some enumeration order and, given a position i, returns the ith answer in the order. Random access, in general, can be seen as an efficient way of accessing the query answers as if they are already computed and stored in an array. One could imagine additional uses for an efficient random-access algorithm. For example, a server implementing a random-access algorithm can provide answers to concurrent users in a stateless manner: the users ask for a range of indices, and the server does not need to keep track of the answers already sent to each user. To satisfy our target of an efficient permutation, we need a random-access structure that can be constructed in linear time (preprocessing) and supports answer retrieval (given i) in logarithmic time. We show that, having this structure at hand, we can use the Fisher-Yates shuffle [Dur64] to design a random permutation with a negligible additive overhead over the preprocessing and enumeration phases.

So far, we have mentioned three tasks of an increasing demand: *(a)* enumeration, *(b)* random permutation, and *(c)* random access. We show that all three tasks can be performed efficiently (i.e., linear preprocessing time and evaluation with logarithmic time per answer) over the class of free-connex CQs. We conclude that within the class of CQs without self-joins, it is the same precise set of queries where these tasks are tractable—the free-connex CQs. (We remind the reader that all mentioned lower bounds are under assumptions in fine-grained complexity.) The existence of a random access for free-connex CQs has been established by Brault-Baron [BB13]. Here, we devise our own random-access algorithm for free-connex CQs that is simpler and better lends itself to a practical implementation. Moreover, we design our algorithm in such a way that it is accompanied by an *inverted access* that is needed for our later results on UCQs.

Note that an alternative approach to our algorithm for random permutation would be to repeatedly sample tuples uniformly with replacement (using known techniques, e.g., [ZCL+18]) and reject tuples that have already been produced. In expectation, this alternative would have the same *total* time as our algorithm, namely $O(M \log M)$ where M is the number of answers, due to the *coupon collector* argument and the fact that the delay of our algorithm is $O(\log M)$. However, this alternative approach would *not* have the strict (and deterministic) guarantee that we provide on the delay, and would not even be counted as an enumeration algorithm with a sublinear delay.

The tractability of enumeration generalizes from free-connex CQs to *unions* of free-connex CQs [CK19b, BKS18]. Interestingly, this is no longer the case for random access. The reason is as follows. Efficient random access, which identifies when a given index is out of bound, allows to count the answers; while counting can be done in linear time for free-connex CQs, we show the existence of a union of free-connex CQs where linear-time counting can be used for linear-time triangle detection in a graph, which is assumed not to be possible. At this point, we ask: Can we get an efficient random permutation for unions of free-connex CQs, without requiring a random access? We

show that the answer is positive under the following weakening of the delay guarantee: there is a random permutation where *each delay* is a geometric random variable with a logarithmic mean. In particular, each delay is logarithmic in expectation. We also show a random-permutation algorithm with a logarithmic bound of the delay that can be used for some classes of UCQs.

Finally, we inspect the space used by our random-permutation solutions, and show that they can be implemented with the minimal space required for random permutation.

Exploiting Functional Dependencies for Query Answering

The characterizations discussed so far only hold when applied to databases with no additional assumptions, but oftentimes this is not the case. In practice, there is usually a connection between different attributes, and *Functional Dependencies* (FDs) and *Cardinality Dependencies* (CDs) are widely used to model situations where some attributes imply others. As the following example shows, these constraints also have an immediate effect on the complexity of enumerating query answers over such a schema.

Example 1.3. For a list of actors and the years in which a movie featuring them was released, consider the query

$$Q(actor, year) \leftarrow \text{Cast}(movie, actor), \text{Release}(movie, year).$$

At first glance, it appears as though this query is not in $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$, as it is acyclic but not free-connex. Nevertheless, if we take the fact that a movie has only one release date into account, we have the FD Release : $1 \rightarrow 2$, and the enumeration problem becomes easy: we only need to iterate over all tuples of Cast and replace the *movie* value with the single *year* value that the relation Release assigns to it. This can be done in linear time by first building a lookup table from Release in linear time. □

Example 1.3 shows that the dichotomy by Bagan et al. [BDG07] does not hold in the presence of FDs. In fact, we believe that dependencies between attributes are so common in real life that ignoring them in such dichotomies can lead to missing a significant portion of the tractable cases. Therefore, to get a more realistic picture of the enumeration complexity of CQs, we have to take dependencies into account. Our goal is to generalize the dichotomy to fully accommodate FDs.

Towards this goal, we introduce an extension of a query Q according to the FDs. This is called an FD-extension and denoted Q^+. In this extension, each atom, as well as the head of the query, contains all variables that can be implied by its variables according to the FDs. This way, instead of classifying every combination of CQ and FDs directly, we encode the dependencies within the extended query, and use the classification of Q^+ to gain insight regarding Q. This approach draws inspiration from the proof of a dichotomy in the complexity of *deletion propagation*, in the presence of FDs [Kim12]. However, the problem and consequently the proof techniques are fundamentally different.

The FD-extension is defined in such a way that if Q is satisfied by an assignment, then the same assignment also satisfies the extension Q^+, as the underlying instance is bound by the FDs. In fact, we can show that enumerating the solutions of Q under FDs can be reduced to enumerating the solutions of Q^+. Therefore, tractability of Q^+ ensures that Q can be efficiently solved as well. By using the positive result in the known dichotomy, Q^+ is tractable w.r.t. enumeration if it is free-connex. Moreover, it can be shown that the structural restrictions of acyclicity and free-connex are closed under FD-extensions. Hence, the class of all queries Q such that Q^+ is free-connex is a proper extension of the class of free-connex queries. We denote the classes of queries Q such that Q^+ is acyclic or free-connex as FD-acyclic or FD-free-connex, respectively.

To reach a dichotomy, we now need to answer the following question: Is it possible that Q can be enumerated efficiently even if it is not FD-free-connex? The reduction by Bagan et al. [BDG07] that shows hardness of CQs over general schemas fails when dependencies are imposed on the data, as the constructed database instance does not necessarily satisfy the underlying dependencies. As it turns out, the structure of the FD-extended query Q^+ allows us to extend this reduction to our setting. We establish a dichotomy by carefully modifying the reduction such that the dependencies hold, while the construction can still be done within linear time. That is, we show that the tractability of enumerating the answers of a self-join-free query Q in the presence of FDs is exactly characterized by the structure of Q^+: Given an FD-acyclic query Q, we can enumerate the answers to Q within the class $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$ iff Q is FD-free-connex. The same results apply also with respect to random permutation and random access if we consider logarithmic delay.

The resulting extended dichotomy, as well as the original one, brings insight to the case of acyclic queries. Regarding cyclic CQs, again the hardness proof by Brault-Baron [BB13] no longer applies in the presence of FDs. Moreover, it is possible for Q to be cyclic and Q^+ acyclic. In fact, Q^+ may even be free-connex, and therefore tractable in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$. We show that, under the same assumptions used by Brault-Baron [BB13], the evaluation problem for a self-join-free CQ in the presence of unary FDs where Q^+ is cyclic cannot be solved in linear time. As linear time preprocessing is not enough to achieve the first answer, a consequence is that enumeration within $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$ is impossible in that case. Therefore, random permutation and random access are impossible as well. This covers all types of self-join-free CQs and shows a full dichotomy for the case of unary FDs.

The results we present here are not limited to CQs and FDs. CQs with disequalities are an extension of CQs, allowing to restrict the satisfying assignments and demand that some pairs of variables map to different values. We prove that our results apply to the more general query class: CQs with disequalities.

Another way of generalizing our results is not to extend the class of queries, but the class of dependencies. Cardinality Dependencies (CDs) [CFWY14, AFG16] are a generalization of FDs, denoted $(R_i : A \to B, c)$. Here, the right-hand side does not have

to be unique for every assignment to the left-hand side, but there can be at most c different values to the variables of B for every value of the variables of A. FDs are in fact a special case of CDs where $c = 1$. Constraints of that form appear naturally in many applications. For example: a movie has only a handful of directors, there are at most 200 countries, and a person is typically limited to at most 5000 friends in (some) social networks. We show that our results also apply to CDs.

Finally, we show that the utility of FD-extensions does not stop in CQs. Our enumeration and random permutation algorithms for UCQs can also be applied based on the structure of the FD-extensions of the CQs in the union.

Enumerating Tree Decompositions

When the queries are neither naturally acyclic nor acyclic via extensions based on dependencies or unions, linear preprocessing time will not suffice. However, this does not mean that we must compute the intractable query as is. Consider the graph associated with our query: a node for every variable and an edge for every atom. Given an input database and a decomposition of this graph, we can create a database and a matching acyclic query with the same answers after a non-linear overhead [TR15, GGS05, KEK16]. This approach of compiling the problem into a tractable form and evaluating the new problem can lead to an order-of magnitude improvement in the time required to evaluate the query. As the translation overhead depends exponentially on the quality of the decomposition, we study the task of finding a good decomposition with the query as input. (In particular, we no longer use data complexity, since the data is not part of the problem we address.)

More specifically, a *tree decomposition* extracts a tree structure from a graph by grouping nodes into *bags* (each treated as a single node). The corresponding operation on hypergraphs is that of a *generalized hypertree decomposition* [GGS05] that consists of a tree decomposition of the *primal* graph (which has the same set of nodes, and an edge between every two hyperedge neighbors), and an assignment of hyperedge labels (edge covers) to the tree nodes [GGM$^+$05]. Tree decompositions and generalized hypertree decompositions have a plethora of applications. In addition to the optimization of join queries in databases, these include containment of database queries, constraint satisfaction problems [KV00], prediction of RNA secondary structure [ZMC06], computation of Nash equilibria in games [GGS05], inference in probabilistic graphical models [LS88], and weighted model counting [KG15].

Past research has focused on obtaining a "good" tree decomposition, where goodness is typically defined as having low *tree width* [RS84]—the maximal cardinality of a bag (minus one). Finding a tree decomposition of the minimal tree width is NP-hard [ACP87], as is the case for other common measures of goodness for tree decompositions such as *fill* [Yan81a], and in the case of hypergraphs *hypertree width* [GLS02], *generalized hypertree width* [GMS09], and *fractional hypertree width* [Mar10]. Therefore, heuristic

algorithms are often applied [BBH02, BBH$^+$06]. The different measures of goodness are motivated by the fact that the needs of different applications are often different from (though related to) the width. Additional examples are the complexity of weighted model counting, induced by a parameter associated with the "CNF-tree" of the formula [KG15, GGM$^+$05], and the effectiveness of *adhesions* (parent-child intersection) for caching in terms of dimension and skew [KEK16]. In fact, Kalinsky et al. [KEK16] have illustrated how, in real-life scenarios, isomorphic tree decompositions of a minimal width may result in an orders-of-magnitude difference in join performance.

The common approach is to devise a decomposition algorithm (exact, approximate or heuristic) to capture the desired measure of goodness per application. However, this is a nontrivial challenge that (to the least) requires high expertise in algorithms and tree decompositions. We propose an alternative approach—produce a large number of different tree decompositions, using a baseline decomposition method, and allow the application at hand to choose the best according to its internal measure function. Our approach brings together results and techniques from the areas of chordal graphs and enumeration theory in order to establish a practical tool for enhancing decomposition algorithms and, by implication, the performance of various inference and optimization algorithms. Specifically, we explore the task of enumerating all (or a subset of) the tree decompositions. Such algorithms have been proposed in the past for small graphs (representing database queries), without complexity guarantees [TR15]. Our main result is an enumeration algorithm that runs in *incremental polynomial time* [JPY88], that is, the time between producing the Nth result and the $(N+1)$st result is polynomial in N and in the size of the input.

We first need to define which tree decompositions should be enumerated, as many of them are effectively useless. For example, if we take a graph that is already a tree, we do not wish to enumerate the tree decompositions that group nodes with no reason; in fact, the tree itself is the only reasonable decomposition in this case. Therefore, we consider only tree decompositions that cannot be "improved" by removing or splitting a bag, and we call such tree decompositions *proper*. As it turns out, the proper tree decompositions are in a bijective (and efficiently computable) correspondence to the *minimal triangulations* of the graph at hand. A *triangulation* of a graph g is a graph g' that is obtained from g by adding edges so that g' is *chordal*, that is, g' does not have any induced simple cycle of more than three nodes. A triangulation is *minimal* if no triangulation can be obtained using only a strict subset of the added edges.

So, the problem is reduced to the task of enumerating all of the minimal triangulations of a graph. The enumeration complexity of this task was an open problem, and we resolve it in this thesis. We devise an algorithm for performing this task in incremental polynomial time. Our approach is as follows. Parra and Scheffler [PS97] have shown that there is a one-to-one correspondence between the minimal triangulations of a graph g and the maximal independent sets of a special graph $\mathcal{G}$. The nodes of $\mathcal{G}$ are the so called *minimal separators of* g, and the edges are between *crossing* minimal separators.

So, enumerating the minimal triangulations of a graph boils down to enumerating these maximal sets. It is well known that all the maximal independent sets of a graph can be enumerated with polynomial delay [JPY88, CKS08]. However, this is insufficient for us, since the graph $\mathcal{G}$ is not given as input, and in fact, its number of nodes can be exponential in the size of the original graph g. Therefore, we cannot construct this graph ahead of time, and cannot directly use existing algorithms to establish incremental polynomial time.

We address this problem by defining an abstraction of the graph $\mathcal{G}$ of minimal separators by means of a *Succinct Graph Representation* (SGR), which is represented compactly by two algorithms: one for enumerating the nodes and one for testing whether a given pair of nodes forms an edge. In particular, we can access the nodes of $\mathcal{G}$ through a polynomial-delay iterator, due to a result by Berry et al. [BBC99] (who show how to enumerate the minimal separators of a graph). Applying previous results, we prove that the SGR for the minimal separator graph (i.e., $\mathcal{G}$) meets certain tractability conditions termed *tractable expansion*, which enable the enumeration of its maximal independent sets (i.e., g's minimal triangulations) in incremental polynomial time in the size of the representation (which can be logarithmic in the size of the graph itself).

In summary, we reduce the problem of enumerating the proper tree decompositions to that of enumerating the minimal triangulations, which we reduce to the problem of enumerating the maximal independent sets of an SGR with tractability properties, and we devise an algorithm for the latter task. An important feature of the algorithm is that it can incorporate any black-box procedure for expanding a given independent set into a maximal one. When applied to enumerating the proper tree decompositions, such a procedure can be any off-the-shelf algorithm for minimal triangulation or tree decomposition (e.g., Maximum Cardinality Search [BBH02] and LB-Triang [BBH$^+$06]). However, our algorithm executes this procedure on different versions of the original graph, each time with some new edges added. Hence, our algorithm has the potential of using a high-quality decomposition algorithm for producing *many* high-quality decompositions, enabling the user to choose the best one generated according to the specific measures of her use case (may it be width or anything else).

After establishing our algorithm, we describe an experimental study where we have tested the ability of the algorithm to utilize the aforementioned triangulation algorithms. The experimental study covers graphs of a wide range of domains (where tree decomposition is needed for efficient analysis): join queries (from the TPC-H collection), Bayesian networks, Markov Random Fields, grids, and random graphs. We tested the execution time (delay) of the algorithm, its ability to reduce the width or fill (number of edges added to establish chordality), and the number of decompositions of the same or better quality (width/fill) compared to that of the original off-the-shelf algorithm. The results show that, indeed, our algorithm can effectively enhance the quality of the corresponding decomposition algorithm.

Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 provides the required background and describes the main definitions and notation used throughout the thesis. In Chapter 3, we study which UCQs can be answered in linear preprocessing time and constant delay. We then inspect also the tasks of random access and random permutation and compare them to enumeration in Chapter 4. Chapter 5 studies the impact of functional dependencies on the results of the previous chapters. In Chapter 6, we address the task of decomposing queries and propose an algorithm enumerating tree decompositions. Finally, we conclude this thesis and discuss open problems and directions for future work in Chapter 7.

Chapter 2

Preliminaries

This section defines the main concepts used throughout this thesis and presents known results that we build upon.

2.1 Schemas, Databases and Queries

Databases. A (relational) *schema* $\mathbf{S}$ is a collection $\mathcal{R}$ of *relation symbols* R, each with an associated arity denoted $\text{arity}(R)$. Sometimes the schema also contains a set Δ of *Functional Dependencies* (FDs), as defined next. A *relation* is a set of tuples of *constants*, where each tuple has the same arity (length). A *database* (instance) I over the schema $\mathbf{S}$ associates with each relation symbol R a finite relation, which we denote by R^I, such that $\text{arity}(R) = \text{arity}(R^I)$ and all FDs in Δ are *satisfied*.

Functional Dependencies. An FD $\delta \in \Delta$ has the form $R\colon A \to B$, where $R \in \mathcal{R}$ and $A, B \subseteq \{1, \ldots, \text{arity}(R)\}$. An FD $\delta = R\colon A \to B$ is said to be *satisfied* in a database I if, for all tuples $u, v \in R^I$ that are equal on the indices of A, u and v are equal on the indices of B. In this thesis, we assume that all FDs are of the form $R\colon A \to b$, where $b \in \{1, \ldots, \text{arity}(R)\}$. This is assumed without loss of generality as we can replace an FD of the form $R\colon A \to B$ where $|B| \neq 1$ by the set of FDs $\{R\colon A \to b \mid b \in B\}$. If $|A| = 1$, we say that δ is a *unary* FD.

Conjunctive Queries. A *Conjunctive Query* (CQ) Q over a schema $\mathbf{S}$ is defined by an expression of the form $Q(\vec{u}) \leftarrow R_1(\vec{v}_1), \ldots, R_n(\vec{v}_n)$, where each R_i is a relation symbol of $\mathbf{S}$, each $\vec{v}_i$ is a tuple of variables and constants with the same arity as R_i, and $\vec{u}$ is a tuple of variables from $\vec{v}_1, \ldots, \vec{v}_n$. We usually omit the explicit specification of the schema $\mathbf{S}$, and simply assume that it contains the relation symbols that occur in the query at hand. Each $R_i(\vec{v}_i)$ is an *atom* of Q, and the set of all atoms of Q is denoted $\text{atoms}(Q)$. A *homomorphism* μ from a CQ Q to a database I is a mapping of the variables in Q to the constants of I, such that for every atom $R_i(\vec{v}_i)$ of the CQ, it holds that $\mu(\vec{v}_i) \in R^I$. Each such homomorphism μ yields an *answer* $\mu(\vec{u})$ to Q. Given

a mapping $\mu : A \to B$ and a set $S \subseteq A$, $\mu|_S$ denotes the restriction (or projection) of μ to the variables S. To ease notation, we often identify the answer $\mu(\vec{u})$ with the mapping $\mu|_{\vec{u}}$. We denote by $Q(I)$ the set of all answers to Q on I.

CQ Notation. We call $Q(\vec{u})$ the *head* of Q and $R_1(\vec{v}_1), \ldots, R_n(\vec{v}_n)$ the *body*. We use $\text{var}(Q)$ to denote the set of all variables in Q. The variables in the head are called the *free variables* and denoted $\text{free}(Q)$, while the variables in the body but not the head are called *existential variables*. A CQ with no existential variables is a *full join query*. Denote by $\text{full}(Q)$ the full version of a CQ Q obtained by adding all variables to the head of Q. We say that a CQ Q is *self-join-free* if every relation symbol occurs at most once. When a CQ is self-join-free, we use $\text{var}(R_i)$ to denote the set of variables that occur in the atom containing R_i.

CQs and Database Instances. Let $R(\vec{v})$ be an atom of a CQ, and let $x \in \vec{v}$. We say that a tuple $\vec{t} \in R^I$ *assigns* x with the value c if for every index i such that $\vec{v}[i] = x$ we have that $\vec{t}[i] = c$. Here, $\vec{v}[i]$ is the ith value of $\vec{v}$. We say that two tuples $\vec{t}_a, \vec{t}_b \in R^I$ *agree* on the value of x if they assign x with the same value. We say that a tuple $\vec{t} \in R^I$ *agrees* with an answer $\mu|_{\text{free}(Q)} \in Q(I)$ if $\mu(\vec{v}) = \vec{t}$. A tuple that does not agree with any answer in $Q(I)$ is called a *dangling tuple*. A database I is *globally consistent* with respect to Q if it contains no dangling tuples [AHV95, Chapter 6.4].

Unions of CQs. A *Union of Conjunctive Queries (UCQ)* Q is a finite set of CQs, denoted $Q = \bigcup_{i=1}^{\ell} Q_i$, where $\text{free}(Q_i)$ is the same for all $1 \leq i \leq \ell$. The set of answers to Q over a database I is the union $Q(I) = \bigcup_{i=1}^{\ell} Q_i(I)$. Let Q_1, Q_2 be CQs. A *body-homomorphism* from Q_2 to Q_1 is a mapping $h : \text{var}(Q_2) \to \text{var}(Q_1)$ such that for every atom $R(\vec{v})$ of Q_2 we have that $R(h(\vec{v})) \in Q_1$. If there exists a body-homomorphism h from Q_2 to Q_1 and vice versa, we say that Q_1 and Q_2 are *body-homomorphically equivalent*. We say that two CQs are *body-isomorphic* is they have the same body up to a renaming of the variables. As easy observation is that if two CQs are self-join-free and body-homomorphically equivalent, then they are also body-isomorphic. In this case, a body-homomorphism between them is also called a *body-isomorphism*. A *homomorphism* from Q_2 to Q_1 is a body-homomorphism h such that $h(\text{free}(Q_2)) = \text{free}(Q_1)$. It is well known that $Q_1 \subseteq Q_2$ iff there exists a homomorphism from Q_2 to Q_1 [CM77]. We say that a UCQ is *non-redundant* if it does not contain two different CQs Q_1 and Q_2 such that there is a homomorphism from Q_2 to Q_1. We assume that UCQs are non-redundant; otherwise, an equivalent non-redundant UCQ can be obtained by removing the redundant CQs.

Query Evaluation *Evaluating* a query Q means computing the answers $Q(I)$ given a database I. We denote by $\text{Enum}_{\Delta}\langle Q \rangle$ the problem of evaluating Q over a schema with the FDs Δ. The problem $\text{Decide}_{\Delta}\langle Q \rangle$ is that of determining whether Q has answers.

Sometimes it is clear from the context that there are no FDs, and we write $\textsc{Decide}\langle Q\rangle$ and $\textsc{Enum}\langle Q\rangle$ to denote $\textsc{Decide}_{\emptyset}\langle Q\rangle$ and $\textsc{Enum}_{\emptyset}\langle Q\rangle$, respectively.

2.2 Hypergraphs

Hypergraphs and Join Trees. A *hypergraph* $\mathcal{H} = (V, E)$ is a set V of *vertices* and a set E of non-empty subsets of V called *hyperedges* (sometimes *edges*). A *join tree* of a hypergraph $\mathcal{H} = (V, E)$ is a tree T where the nodes are the hyperedges of $\mathcal{H}$, and the *running intersection* property holds, namely: for all $u \in V$ the set $\{e \in E \mid u \in e\}$ forms a (connected) subtree in T. A hypergraph $\mathcal{H}$ is *acyclic* if there exists a join tree for $\mathcal{H}$ (this is known as α-acyclicity [Fag83]). A hypergraph that is not acyclic is called *cyclic*. A hypergraph $\mathcal{H}'$ is an *inclusive extension* of $\mathcal{H}$ if every edge of $\mathcal{H}$ appears in $\mathcal{H}'$, and every edge of $\mathcal{H}'$ is a subset of some edge in $\mathcal{H}$. A tree T is an *ext-S-connex tree* for a hypergraph $\mathcal{H}$ if: (1) T is a join tree of an inclusive extension of $\mathcal{H}$, and (2) there is a subtree T' of T that contains exactly the vertices S [BDG07].

Paths and Cycles. Two vertices in a hypergraph are *neighbors* if they appear in the same edge. A *path* of $\mathcal{H}$ is a sequence of vertices such that every two succeeding variables are neighbors. The *length* of a path $v_1, \ldots, v_n$ is $n - 1$. A *simple path* of $\mathcal{H}$ is a path where every vertex appears at most once. A *chordless path* is a simple path in which no two non-succeeding vertices are neighbors. A *cycle* is a path that starts and ends in the same vertex. A *simple cycle* is a cycle of length 3 or more where every vertex appears at most once (except for the first and last vertex). A *chordless cycle* is a simple cycle such that no two non-succeeding vertices are neighbors and no edge contains all cycle vertices.

Hypercliques and Tetras. A *clique* of a hypergraph is a set of vertices that are pairwise neighbors in $\mathcal{H}$. If every edge in $\mathcal{H}$ has k vertices, then we call $\mathcal{H}$ *k-uniform*. An *l-hyperclique* in a k-uniform hypergraph $\mathcal{H}$ is a set V' of $l > k$ vertices, such that every subset of V' of size k forms a hyperedge. A hypergraph $\mathcal{H}$ is said to be *conformal* if every clique of $\mathcal{H}$ is contained in some edge of $\mathcal{H}$. A hypergraph is acyclic iff it is conformal and contains no chordless cycles [BFMY83]. A *tetra* of size k, denoted k-tetra, is a set of k vertices such that every $k - 1$ of them are contained in an edge, and no edge contains all k vertices. A hypergraph is cyclic if and only if it contains a chordless cycle or a tetra [BB13].

2.3 Query Structure

CQ Hypergraph. We associate a hypergraph $\mathcal{H}(Q) = (V, E)$ to a CQ Q where the vertices are the variables of Q, and every hyperedge is a set of variables occurring in

a single atom of Q. That is, $E = \{\{v_1, \ldots, v_n\} \mid R_i(v_1, \ldots, v_n) \in \text{atoms}(Q)\}$. With a slight abuse of notation, we identify atoms of Q with edges of $\mathcal{H}(Q)$.

Acyclicity and Free-Connexity. A CQ Q is said to be *acyclic* if $\mathcal{H}(Q)$ is acyclic, and it is *S-connex* if $\mathcal{H}(Q)$ has an ext-S-connex tree [BDG07]. Given a CQ Q and a set $S \subseteq \text{var}(Q)$, an *$S$-path* is a chordless path $(x, z_1, \ldots, z_k, y)$ in $\mathcal{H}(Q)$ with $k \geq 1$, such that $x, y \in S$, and $z_1, \ldots, z_k \notin S$. An acyclic CQ has an S-path iff it is not S-connex [BDG07]. Given a CQ Q, *free-path* stands for free(Q)-path and *free-connex* stands for free(Q)-connex. A CQ Q is free-connex iff both Q and $(V, E \cup \{\text{free}(Q)\})$ are acyclic [BB13].

2.4 Computational Model

Input. Using *data complexity* for most of our problems, the input is measured only by the size of the database instance I (the query and the schema are treated as fixed). Let I be a database over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$. We denote by $||o||$ the size of an object o (i.e., the number of integers required to store it), whereas $|o|$ is its cardinality. We assume the input database is given by the reasonable encoding suggested by Flum et al. [FFG02]. Thus, the input is of size $||I|| = 1 + |dom| + |\mathcal{R}| + \sum_{R \in \mathcal{R}} \text{arity}(R)|R^I|$ over integers bounded by $\max\{|dom|, max_{R \in \mathcal{R}}|R^I|\}$. When we say *linear time*, we mean that the number of operations is $\mathcal{O}(||I||)$.

Cost Measure. We adopt the *Random Access Machine* (RAM) model with uniform-cost measure and word length $\Theta(\log(n))$ on input of size n [Gra96]. Operations such as addition of the values of two registers or concatenation can be performed in constant time. In contrast to the Turing model of computation, the RAM model with uniform-cost measure can retrieve the content of any register via its unique address in constant time. As we assume that the word length is $\Theta(\log(||I||))$, the values stored in registers are at most $||I||^c$ for some fixed integer c. As a consequence, and since we assume that the values may correspond to addresses, the amount of available memory is polynomial in $||I||$.

Lookup Tables. The RAM model enables the construction of large lookup tables that can be queried within constant time. In particular, it is possible to compute the semi-join of two relations (filtering one relation according to the other) in linear time. If the available memory is as large as the number of different possible keys, the lookup table can be as simple as initializing the memory to False, and setting the addresses matching the keys to be True. Note that the time for initialization is not an issue due to constant time initialization techniques [MS91]. Solutions such as binary search trees and hash tables can reduce the memory consumption significantly. For example, a lookup table based on a binary search tree only requires linear memory in its content

at the cost of a logarithmic factor to the time. In this thesis, we generally (other than Sections 3.4.2, 4.4 and 5.4 discussing the space consumption) wish to identify what can and cannot be done within certain time bounds without restrictions on the memory consumption. Thus, we assume in our analysis that the available memory is very large and that we can access lookup tables of polynomial size in constant time.

Sorting. Grandjean [Gra96] proved that sorting strings can be done in time $\mathcal{O}(n/\log n)$, where n is the size of the input containing strings encoded in some fixed alphabet and separated by some special symbol, even in the more restrictive DLINRAM model. This method can be used to sort relations. To construct the input to the sorting algorithm, we first translate the values from dom to a possibly smaller domain dom_R, containing only the values that appear in R^I. Note that $|dom_R| \leq ||R^I||$. Then, we translate these values to binary (since we are required to use a fixed alphabet), where each value takes $\log(dom_R)$ bits. The size of the input to the sorting problem is $n = ||R^I|| \cdot \log(|dom_R|) + (|R^I| - 1)$. Therefore, we can sort the tuples of a relation in time $\mathcal{O}(n/\log n) = \mathcal{O}(||R^I||)$. That is, it is possible to sort relations within linear time [SV17].

2.5 Enumeration Complexity

Classic Enumeration Complexity. An *enumeration problem* P is a collection of pairs (I, Y) where I is an *input* and Y is a finite set of *answers* for I, denoted by $P(I)$. An *enumeration algorithm* for P is an algorithm that, when given an input I, produces (or *prints*) a sequence of answers such that every answer in $P(I)$ is printed precisely once. Johnson, Papadimitriou and Yannakakis [JPY88] introduced several different notions of *efficiency* for enumeration algorithms, and we recall these now. Let $\mathcal{A}$ be an enumeration algorithm for a problem P. We say that $\mathcal{A}$ runs in: *polynomial total time* if the total execution time of $\mathcal{A}$ is polynomial in $(|I| + |P(I)|)$; *polynomial delay* if the time between printing every two consecutive answers is polynomial in $|I|$; *incremental polynomial time* if, after printing N answers, the time to print the next answer is polynomial in $(|I| + N)$.[1] The requirements between answers apply also for the time before the first answer and the time between the last answer and termination. Observe that polynomial delay implies incremental polynomial time, which, in turn, implies polynomial total time.

Fine-Grained Enumeration Complexity. Since databases are often very large, we aim to achieve a stricter time guarantee than that of polynomial delay when enumerating query answers. Computing the first answer requires at least linear time (to read the input

[1] The definition of Johnson [JPY88] requires the delay to be polynomial in the size of the input and the *size* of the previously produced results (not just their *number* N as we define here). However, the definitions are equivalent when the size of each answer is polynomial in that of the input, as in our case.

and decide whether an answer exists), but sometimes we can achieve a smaller delay between the subsequent answers. For this reason, we separate the requirement regarding the time before the first answer from that of the following answers. An enumeration algorithm $\mathcal{A}$ for an enumeration problem P may consist of two phases: *preprocessing* and *enumeration*. During preprocessing, $\mathcal{A}$ is given an input I, and it may build data structures. During the enumeration phase, $\mathcal{A}$ can access the data structures built during preprocessing, and it emits the answers $P(I)$, one by one, without repetitions. We say that $\mathcal{A}$ enumerates with *delay* $t_d(|I|)$ if the time between any two consecutive outputs, as well as the time between the beginning of the enumeration phase and the first output and the time between the last output and termination are each bounded by $t_d(|I|)$. We denote the running time of the preprocessing phase by $t_p(|I|)$. The class $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$ is defined to contain all enumeration problems that have an enumeration algorithm with preprocessing $t_p(|I|) \in O(|I|)$ and delay $t_d(|I|) \in O(1)$. Note that we do not impose a restriction on the memory used. In particular, such an algorithm may use additional constant memory for writing between two consecutive answers.

Exact Reductions. Let P_1 and P_2 be enumeration problems, and denote by $\mathcal{I}_1$ and $\mathcal{I}_2$ their sets of possible inputs, respectively. Then, $P_1(\mathcal{I}_1)$ and $P_2(\mathcal{I}_2)$ denote their sets of possible outputs. There is an *exact reduction* from P_1 to P_2, denoted $P_1 \leq_e P_2$, if there exist mappings $\sigma : \mathcal{I}_1 \to \mathcal{I}_2$ and $\tau : P_2(\mathcal{I}_2) \to P_1(\mathcal{I}_1)$ such that: (1) σ is computable in time linear in the size of its input; (2) the time to compute τ is constant with respect to the input to the enumeration problems; and (3) Given any $I \in \mathcal{I}_1$, τ acts as a bijection from $P_2(\sigma(I))$ to $P_1(I)$. The notation $P_1 \equiv_e P_2$ means that $P_1 \leq_e P_2$ and $P_2 \leq_e P_1$. An enumeration class $\mathcal{C}$ is said to be *closed under exact reduction* if for every P_1 and P_2 with $P_2 \in \mathcal{C}$ and $P_1 \leq_e P_2$, we have that $P_1 \in \mathcal{C}$. Bagan et al. [BDG07] proved that $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$ is closed under exact reduction. The same proof holds for any meaningful enumeration complexity class that guarantees generating all unique answers with at least linear preprocessing time and at least constant delay between answers.

2.6 Complexity Hypotheses

We define here the known hypotheses used in this thesis. BMM and Hyperclique were previously used to show the hardness of CQ evaluation.

BMM. BMM states that two Boolean $n \times n$ matrices cannot be multiplied in time $O(n^2)$. Boolean matrix multiplication is equivalent to the evaluation of the query $\Pi(x, y) \leftarrow A(x, z), B(z, y)$ over the schema $\{A, B\}$ where $A, B \subseteq \{1, \ldots, n\}^2$. The matrix multiplication exponent ω is the smallest number such that for any $\varepsilon > 0$ there is an algorithm that multiplies two rational $n \times n$ matrices with at most $O(n^{\omega+\varepsilon})$ (arithmetic) operations. Currently, the best bound on ω is $\omega < 2.373$ [LG14, AW14].

sparseBMM SPARSEBMM assumes that two Boolean matrices A and B, represented as lists of their non-zero entries, cannot be multiplied in time $m^{1+o(1)}$, where m is the number of non-zero entries in A, B, and AB. The best known running time for this problem is $O(m^{4/3})$ [AP09].

4-clique. 4-CLIQUE states that it is not possible to determine the existence of a 4-clique in a graph with n nodes in time $O(n^3)$. This is a special case of the k-Clique Hypothesis [LWW18], which states that detecting a clique in a graph with n nodes requires $n^{\frac{\omega k}{3}-o(1)}$ time, where ω is again the matrix multiplication exponent (note that $\omega \geq 2$ by its definition).

Hyperclique. HYPERCLIQUE states that for all $k \geq 3$, it is not possible to determine the existence of a k-hyperclique in a $(k-1)$-uniform graph with n nodes in time $O(n^{k-1})$. When $k = 3$, this is the assumption that we cannot detect a triangle in a graph in $O(n^2)$ time [AYZ97]. This assumption is stronger than BMM, as triangle finding can be reduced to the matrix multiplication problem [WW10]. When $k > 3$, this is a special case of the $(\ell, k)-$ Hyperclique Hypothesis [LWW18], which states that, in a k-uniform hypergraph of n vertices, $n^{k-o(1)}$ time is required to find a set of ℓ vertices such that each of it subsets of size k forms a hyperedge. The HYPERCLIQUE hypothesis is sometimes called Tetra$\langle k \rangle$ [BB13].

2.7 Complexity of CQ Evaluation

Bagan, Durand and Grandjean [BDG07] showed that the answers to free-connex CQs can be efficiently enumerated. This result is complemented by conditional lower bounds that show that other CQs are intractable over general schemas.

Theorem 2.1 ([BDG07, BB13])**.** *Let Q be a self-join-free CQ.*

1. *If Q is free-connex, then* $\text{ENUM}_\emptyset\langle Q \rangle \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$.
2. *If Q is acyclic and not free-connex, then* $\text{ENUM}_\emptyset\langle Q \rangle \notin \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$*, assuming* BMM.
3. *If Q is cyclic, then* $\text{ENUM}_\emptyset\langle Q \rangle \notin \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$*, as* $\text{DECIDE}_\emptyset\langle Q \rangle$ *cannot be solved in linear time, assuming* HYPERCLIQUE.

Tractability. The positive case of this dichotomy can be shown using the *Constant Delay Yannakakis* (CDY) algorithm [IUV17], and it is also explained in detail in Section 4.2 as part of a different proof. Given a database I and an ext-S-connex tree for a CQ Q, we can compute all projections into S of the homomorphisms from Q to I with linear preprocessing and constant delay. During preprocessing, we construct a relation for each node in the tree and apply the classical full reduction by Yannakakis [Yan81b]. The reduction removes all dangling tuples and results in a globally consistent database with respect to Q. Then, we consider only the subtree of T containing S, and join

the relations corresponding to this subtree along the tree with constant delay. When $S = \text{free}(Q)$, this computes exactly $Q(I)$. This solution can be applied to any free-connex CQ, as such a CQ has an ext-free(Q)-connex tree.

Difficult Structures. Every CQ is one of the following: (1) free-connex; (2) acyclic and not free-connex, and therefore contains a free-path [BDG07]; (3) cyclic, and therefore contains a chordless cycle or a tetra [BB13]. We call free-paths, chordless cycles and tetras *difficult structures*. Hence, every CQ that is not free-connex contains a difficult structure. The restriction of considering only self-joins-free queries in Theorem 2.1 is in fact required only for the negative side, and it allows to assign different atoms with different relations independently to prove hardness. Boolean matrix multiplication can be encoded in free-paths, and so self-join-free acyclic CQs are intractable assuming BMM [BDG07]. The detection of hypercliques can be encoded in tetras and chordless cycles, and so the first answer to self-join-free cyclic CQs cannot be found in linear time assuming HYPERCLIQUE [BB13]. We call a CQ *difficult* if is self-join-free and it is cyclic or acyclic not free-connex. As we rely heavily on the proof of hardness for acyclic non-free-connex CQs, we explain the proof next.

Hardness of Acyclic CQs. The hardness proof [BDG07, Lemma 26] can be seen as an exact reduction $\text{ENUM}_{\emptyset}\langle\Pi\rangle \leq_e \text{ENUM}_{\emptyset}\langle Q\rangle$, where Π is the Boolean matrix multiplication query given in Section 2.6. Since Q is acyclic but not free-connex, it contains a free-path $(x, z_1, \ldots, z_k, y)$. For a given an instance of the matrix multiplication problem, an instance of $\text{ENUM}_{\emptyset}\langle Q\rangle$ is constructed, where the variables x,y and $z_1, \ldots, z_k$ of the free-path encode the variables x, y and z of Π, respectively. All other variables of Q are assigned constants. This way, A is encoded by an atom containing x and z_1, and B is encoded by an atom containing z_k and y. Atoms containing some z_i and z_{i+1} propagate the value of z. Since x and y are free, but z_i are not, the answers to Q correspond to those of Π. As no atom of Q contains both x and y, the instance can be constructed in linear time. Constant delay enumeration for Q following a linear time preprocessing would result in the computation of the answers of Π in $\mathcal{O}(n^2)$ time, contradicting BMM.

Example 2.2. Assume we want to compute the multiplication $\left(\begin{smallmatrix}1&1\\0&1\end{smallmatrix}\right)\left(\begin{smallmatrix}0&1\\0&1\end{smallmatrix}\right) = \left(\begin{smallmatrix}0&1\\0&1\end{smallmatrix}\right)$. We can do that via the matrix multiplication query $\Pi(x, y) \leftarrow A(x, z), B(z, y)$ with input $A^I = \{(1, 1), (1, 2), (2, 2)\}$ and $B^I = \{(1, 1), (1, 2)\}$. The result $\Pi(I) = \{(1, 1), (1, 2)\}$ is exactly the indices of the non-zero entries in the result. This computation can be encoded into any acyclic non-free-connex CQ. For example, given the CQ $Q(x, y, t) \leftarrow R_1(x, z_1), R_2(z_1, z_2), R_3(z_2, y), R_4(y, t)$, we can set $R_1^I = \{(1, 1), (1, 2), (2, 2)\}$, $R_2^I = \{(1, 1), (2, 2)\}$, $R_3^I = \{(1, 1), (1, 2)\}$, and $R_4^I = \{(1, \perp), (2, \perp)\}$. Then, the answers $Q(I) = \{(1, 1, \perp), (1, 2, \perp)\}$ represent the mutiplication result. □

Chapter 3

Answering UCQs with Constant Delay

In this chapter, we inspect which UCQs can be answered efficiently, with linear preprocessing time and constant delay, in the general case that we can make no assumptions on the data. In particular, we assume in this chapter that the schema does not contain dependencies. We make no requirements here on the order in which the answers are enumerated. We define the notion of union extensions, show that all UCQs with free-connex union extensions are tractable, and that these also include some unions of intractable CQs. We also prove lower bounds showing that for several classes of UCQs, free-connex union extensions captures all tractable queries. To address other classes of UCQs, we define the problem of Unbalanced Triangle Detection, and show a tight connection between the complexity of this problem and the enumeration hardness for UCQs that do not admit a free-connex union extension.

This chapter contains joint work with Markus Kröl and Karl Bringmann. Some of the results presented in this chapter were published in the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems [CK19b], in a paper that received the *best student paper award* and was invited for a special issue of the ACM Transaction on Database Systems (TODS) journal on selected publications from PODS 2019.

Organization. In Section 3.1, we formalize how CQs within a union can make each other easier by providing variables to one another, and show that UCQs with free-connex union extensions are in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$. This implies that UCQs are not required to contain tractable CQs in order to be tractable themselves. We then set off to inspect whether our positive results cover all tractable cases. In Section 3.2, we build upon the known characterization for CQs, and show hardness results for UCQs without tractable extensions. These results are conditional lower bounds, based on known hypotheses, and they include the proof that a union of two difficult CQs is not tractable if it does not have a free-connex union extension. In Section 3.3, we discuss where the previously used

hypotheses are no longer enough for proving the hardness of UCQs, and we phrase a new hypothesis on graphs, called Unbalanced Triangle Detection, which is tightly related to the hardness of UCQs. We show that, assuming this hypothesis alone, all self-join-free unions of two binary CQs are intractable if they do not admit a free-connex union extension. On the other direction, if free-connex union extensions capture all tractable UCQs, this hypothesis necessarily holds. We conclude this chapter by discussing variants of the problem. In Section 3.4.1, we show that, unlike the case of CQs, allowing for disequalities in the query affects the enumeration complexity, and we prove tractability results for UCQs with disequalities. In Section 3.4.2, we provide a short discussion regarding the setting with restricted space.

3.1 Tractable Cases

In this section, we identify tractable UCQs. Section 3.1.1 discusses unions that contain only tractable CQs. Section 3.1.2 inspects the requirements of $\mathsf{Enum}\langle \text{lin}, \text{const} \rangle$ and proves the Cheater's Lemma: a tool that allows us to compile several enumeration algorithms into one. In Section 3.1.3, we introduce the concepts of *union extensions* and variable sets that a CQ can *supply* in order to help the evaluation of another CQ in the union. We show that UCQs that admit free-connex union extensions are in $\mathsf{Enum}\langle \text{lin}, \text{const} \rangle$ in Section 3.1.4.

3.1.1 Unions of Tractable CQs

Using known techniques [Str10, Proposition 2.38], a union of tractable CQs is also tractable.

Theorem 3.1. *Let $Q = Q_1 \cup \ldots \cup Q_n$ be a UCQ for some fixed $n \geq 1$. If all CQs in Q are free-connex, then $\textsc{Enum}\langle Q \rangle \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$.*

Proof. Algorithm 3.1 evaluates a union of two CQs. In case of a union $Q = \bigcup_{i=1}^{\ell} Q_i$ of more CQs, we can use this recursively by treating the second query as $Q_2 \cup \ldots \cup Q_\ell$.

Algorithm 3.1 Enumerating a union of two tractable CQs [Str10]

```
1: while a ← Q1(I).next() do
2:    if a ∉ Q2(I) then
3:       print a
4:    else
5:       print Q2(I).next()
6: while a ← Q2(I).next() do
7:    print a
```

By the end of the run, the algorithm prints $Q_1(I) \setminus Q_2(I)$ over all iterations of line 3, and it prints $Q_2(I)$ in lines 5 and 7. Line 5 is called $Q_1(I) \cap Q_2(I)$ times, so the command $Q_2(I).next()$ always succeeds there. For free-connex CQs after linear preprocessing

time, the answers can be enumerated in constant delay, and testing whether a given mapping is an answer can be done in constant time [BGS20]. Thus, this algorithm runs within the required time bounds. □

The technique presented in the proof of Theorem 3.1 has the advantage that it does not require more than constant memory available for writing in the enumeration phase. Alternatively, this theorem is a consequence of Lemma 3.4, which we prove next and gives us a general approach to compile several enumeration algorithms into one.

3.1.2 The Cheater's Lemma

In this section, we prove a lemma that is useful to show upper bounds for UCQs even in cases not covered by Theorem 3.1. To get a clearer notion of what it means for a problem to be in the class $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$, we first define linear partial time.

Definition 3.2 (Linear Partial Time)**.** An algorithm runs in *linear partial time* if, for every input x, the time before the nth output is $O(|x| + n)$.

We claim next that if we relax the requirement of linear preprocessing and constant delay to allow a constant number of linear delay steps, we get linear partial time.

Proposition 3.3. *Let $\mathcal{A}$ be an algorithm. If there exist constants a, b and c such that, on any input x, the time between two successive outputs of $\mathcal{A}$ is bounded by $a(|x| + n)$ at most b times, where n is the number of answers printed up to that point, and bounded by c otherwise, then $\mathcal{A}$ runs in linear partial time.*

Proof. Before the nth answer, there are at most b steps with delay larger than constant, and in these cases the delay is bounded by $a(|x| + n)$. Thus, the time in which the nth answer is produced is bounded by $ba(|x| + n) + cn \leq (ab + c)(|x| + n)$ for every n. □

If space is not restricted (as in our case), this relaxation in the phrasing of the requirements does not change the requirements themselves, as we show that any algorithm that runs in linear partial time can be modified to achieve linear preprocessing and constant delay. This can be done using the known technique [CS18, Proposition 12] of delaying the results to regularize the delay. In fact, we can further relax the phrasing by allowing a constant number of duplicates per answer.

Lemma 3.4 (The Cheater's Lemma)**.** *Let P be an enumeration problem. The following are equivalent:*

1. *$P \in \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$.*
2. *There exist an algorithm $\mathcal{A}$ and constants c and d such that: $\mathcal{A}$ outputs the solutions to P, the time before the nth answer is bounded by $c(|x| + n)$, and every result is produced at most d times.*

Proof. As any algorithm that runs in linear preprocessing and constant delay also runs in linear partial time, the direction 1⇒2 is trivial. We now show the opposite direction.

We describe an algorithm $\mathcal{A}'$ that simulates $\mathcal{A}$, stores all generated results to prevent duplicates and holds back generated results to regularize the delay. $\mathcal{A}'$ maintains a lookup table containing the results that $\mathcal{A}$ generated and a queue containing those that were not yet printed. Both are initialized as empty. $\mathcal{A}'$ calls $\mathcal{A}$. When $\mathcal{A}$ returns a result, $\mathcal{A}'$ checks the lookup table to determine whether it was found before. If it was not, the result is added to both the lookup table and the queue. Otherwise, it is ignored. Since $\mathcal{A}$ runs in linear partial time, there exists a constant c such that the nth answer to $\mathcal{A}$ is obtained after $c(|x| + n)$ operations. $\mathcal{A}'$ first performs $c|x|$ computation steps, and then after every cd computation steps, it outputs a result from the queue. The queue is never empty when used: $\mathcal{A}'$ returns its ith result after $c|x| + (cd)i = c(|x| + di)$ computation steps; At this time, $\mathcal{A}$ produced at least di results, which contain at least i unique results. When it is done simulating $\mathcal{A}'$, $\mathcal{A}$ outputs all remaining results in the queue. By definition, $\mathcal{A}'$ operates with linear preprocessing time and constant delay. It outputs all results of $\mathcal{A}$ with no duplicates since, due to the lookup table, every result enters the queue exactly once. □

The Cheater's Lemma formalizes how much we are allowed to "cheat" in order to show that a problem can be solved with linear preprocessing and constant delay by only showing an algorithm with relaxed requirements. To show that a problem is in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$, it suffices to find an algorithm for this problem where the delay is usually constant, but it may be linear a constant number of times, and the number of times every result is produced is bounded by a constant. The allowed linear delay is not only with respect to the input, but also with respect to the already produced answers.

3.1.3 Union Extensions

As Example 1.2 shows, Theorem 3.1 does not cover all tractable UCQs. We now define union extensions and address the other cases. We first formalize the way that one CQ can help with evaluating another CQ in the same union by supplying variables.

Definition 3.5. We say that a CQ Q *supplies* a set V of variables if there exists S such that $V \subseteq S \subseteq \mathrm{free}(Q)$ and Q is S-connex.

Recall that we define a *body-homomorphism* between CQs to have the standard meaning of a homomorphism, but without the restriction on the heads of the queries (see Section 2.1). The following lemma shows why body-homomorphisms and supplying variables play an important role in UCQ enumeration. In case a body-homomorphism from a supplying CQ Q_2 to another CQ Q_1, we can produce an auxiliary relation that contains all possible value combinations of the matching variables in Q_1. This can be done efficiently while producing some answers to Q_2.

Lemma 3.6. *Let Q_2 be a CQ that supplies the variables $\vec{v}_2$. Given an instance I, one can compute with linear time preprocessing and constant delay a set of mappings M from* $\text{free}(Q_2)$ *to the domain such that:*

- $M \subseteq Q_2(I)$
- *M can be translated in time $O(|M|)$ to a relation R^M such that: for every CQ Q_1 with a body-homomorphism h from Q_2 to Q_1 and for every answer $\mu_1 \in \text{full}(Q_1)(I)$, there is the tuple $\mu_1(h(\vec{v}_2)) \in R^M$.*

Proof. According to Definition 3.5, there exists $\vec{v}_2 \subseteq S \subseteq \text{free}(Q_2)$ such that Q_2 is S-connex. Take an ext-S-connex tree T for Q_2, and perform the CDY algorithm [IUV17] on Q_2 while treating S as the free-variables. This results in a set N of mappings from the variables of S to the domain such that $N = Q_2(I)|_S$. Note that, as mentioned in Section 2.7, the CDY algorithm has a preprocessing stage that removes dangling tuples and guarantees that, at its end, there is a relation for each vertex of the tree such that each tuple of such a relation agrees with some answer.

For every mapping $\mu \in N$, extend it once to obtain a mapping from all variables of Q_2 as follows. Go over all vertices of T starting from the connected part containing S and treating a neighbor of an already treated vertex at every step. Consider a step where in its beginning μ is a homomorphism from a set S_1, and we are treating an atom $R(\vec{v}, \vec{u})$ where $\vec{v} \subseteq S_1$ and $\vec{u} \cap S_1 = \emptyset$. We take some tuple in R of the form $(\mu(\vec{v}), \vec{t})$ and extend μ into μ^+ that also maps $\vec{u}$ to $\vec{t}$. Such a tuple exists since the dangling tuples were removed. This extension takes constant time, and in its end we have that $\mu^+|_{\text{free}(Q_2)} \in Q_2(I)$. Using these extensions, we set $M = \{\mu^+|_{\text{free}(Q_2)} \mid \mu^+ \text{ is an extension of } \mu \in N\}$. We have that $M \subseteq Q_2(I)$. We also have that $M|_S = N = Q_2(I)|_S$, and since $\vec{v}_2 \subseteq S$, this means that $M|_{\vec{v}_2} = Q_2(I)|_{\vec{v}_2}$. The mappings M are computed with linear preprocessing and constant delay as this is the complexity of the CDY algorithm and the manipulations we describe only require constant time per answer.

We define $R^M = \{\mu^+(\vec{v}_2) \mid \mu^+ \in M\}$. Since $M|_{\vec{v}_2} = Q_2(I)|_{\vec{v}_2}$, this is the same as $\{\mu_2(\vec{v}_2) \mid \mu_2 \in Q_2(I)\}$. Let Q_1 be a CQ such that there is a body-homomorphism h from Q_2 to Q_1, and let $\mu_1 \in \text{full}(Q_1)(I)$. Since h is a body-homomorphism, for every atom $R(\vec{v})$ in Q_2, $R(h(\vec{v}))$ is an atom in Q_1. Since μ_1 forms an answer to the full Q_1, for every such atom $\mu_1(h(\vec{v})) \in R^I$. This means that $\mu_1 \circ h|_{\text{free}(Q_2)}$ is an answer to Q_2, so there exists $\mu_2 \in Q_2(I)$ such that $\mu_1 \circ h|_{\text{free}(Q_2)} = \mu_2$. By construction, $\mu_1(h(\vec{v}_2)) = \mu_2(\vec{v}_2) \in R^M$. □

Note that if the mapping h is not a body-homomorphism, Lemma 3.6 does not hold in general. Here is an example.

Example 3.7. Consider the following slight modification of the UCQ from Example 1.2:

$$Q_1(x, y, w) \leftarrow R_1(x, z), R_2(z, y), R_3(y, w) \text{ and}$$
$$Q_2(x, y, w) \leftarrow R_1(x, y), R_2(y, w), R_4(y).$$

Since R_4 is not a relational symbol in Q_1, there is no body-homomorphism from Q_2 to Q_1. If $R_4^I = dom$, then we can take the same approach as in Example 1.2, as the answers of Q_2 form $Q_1(I)|_{\{x,z,y\}}$. However, if R_4^I is smaller, this extra atom may filter the answers to Q_2, and we do not obtain all of $Q_1(I)|_{\{x,z,y\}}$ in general. □

During evaluation, a set of supplied variables can form an auxiliary relation, accessible by an auxiliary atom. We call the query with its auxiliary atoms a *union extension*.

Definition 3.8. Let $Q = Q_1 \cup \ldots \cup Q_n$ be a UCQ. An *extension sequence* for Q is a sequence $Q^1, \ldots, Q^N$ where $Q = Q^1$ and for all $1 < j \leq N$, Q^j is a UCQ of the form $Q_1^j \cup \ldots \cup Q_n^j$ such that the following holds. For some relational symbol R_j that does not appear in Q_{j-1} and a sequence $\vec{v}_j$ of variables supplied by a previous $Q^\ell_{p(j)}$ (i.e., $\ell \leq j$ and $1 \leq p(j) \leq n$) we have the following for all $i = 1, \ldots, n$:

- $Q_i^j = Q_i^{j-1}$; or
- there is a body-homomorphism $h_{p(j),i}$ from $Q^1_{p(j)}$ to Q_i^1, and Q_i^j is obtained by adding the atom $R_j(h_{p(j),i}(\vec{v}_j))$ to Q_i^{j-1}.

If such an extension sequence exists, we call Q^N a *union extension* of Q^1. Atoms that appear in Q^N but not in Q^1 are called *virtual atoms*.

3.1.4 Extension-Based Tractability

We now claim that if a CQ can be extended to a tractable form via a union extension, then it can be evaluated efficiently as part of the union.

Theorem 3.9. *If Q is a UCQ with a free-connex union extension, then* $\textsc{Enum}\langle Q\rangle$ *is in* $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$.

Proof. We begin by sketching the proof. For answering Q, we describe an algorithm $\mathcal{A}$ which is comprised of two phases: a provision phase and a final results phase. In the provision phase, a free-connex union extension is instantiated. In the final results phase, the answers of every CQ in the union are enumerated via the free-connex union extension. These answers can be enumerated efficiently using the CDY algorithm (see Theorem 2.1). We will use Lemma 3.6 to generate some answers to Q during the provision phase; this permits us to use more than linear time before the final results phase while still achieving an enumeration algorithm with only linear preprocessing time. We will use the Cheater's Lemma (Lemma 3.4) to remove duplicates and obtain the time bounds we want.

Initial Algorithm. Let $Q = Q_1 \cup \ldots \cup Q_n$, and take an extension sequence $Q^1, \ldots, Q^N$ where $Q^1 = Q$ and Q^N comprises of free-connex CQs. The provision phase consists of $N - 1$ provision steps. Let $I = I^1$ be the input database instance. During the jth provision step (with $1 < j \leq N$), we extend the database instance I_{j-1} into an instance I_j that matches Q^j. We use Lemma 3.6 to generate a set of answers $M_j \subseteq Q^{j-1}_{p(j)}(I_{j-1})$

while also computing a relation R^{M_j}. We set $(R_j)^{I_j} := R^{M_j}$. The other relations remain as they were; that is, $R^{I_j} = R^{I_{j-1}}$ for every relational symbol in the UCQ except for R_j. By the end of the provision phase, the algorithm computes an instance I_N that matches Q^N. Finally, we perform the final results phase where we compute $Q_i^N(I_N)$ for every Q_i in the union using the CDY algorithm.

Correctness. We prove that for all $1 \leq i \leq n$ and $1 \leq j \leq N$ we have that $Q_i(I) = Q_i^j(I_j)$ by induction on j. The base case trivially holds as $Q_i(I) = Q_i^1(I_1)$ by definition. Now consider the jth provision step. Intuitively, answers to an extended CQ are by definition all answers to its previous version that agree with some tuple in the new atom. Since the new atom contains a projection of the answers, the extension has exactly the same answers as its previous version. More formally, let $Q_{e(j)}$ be a CQ extended in step j. For every mapping μ, by definition of the extension we have that $\mu \in Q_{e(j)}^j(I_j)$ if and only if both $\mu \in Q_{e(j)}^{j-1}(I_{j-1})$ and $\mu(h(\vec{v}_j)) \in R_j^{I_j}$, and these two conditions hold if and only if $\mu \in Q_{e(j)}^{j-1}(I_{j-1})$ since for all $\mu \in Q_{e(j)}^{j-1}(I_{j-1})$ we have that $\mu(h(\vec{v}_j)) \in R_j^{I_j}$. This proves that $Q_{e(j)}^j(I_j) = Q_{e(j)}^{j-1}(I_{j-1})$. This shows that for every Q_i in the union, $Q_i^j(I_j) = Q_i^{j-1}(I_{j-1})$. By the induction hypothesis, $Q_i^{j-1}(I_{j-1}) = Q_i(I)$. This concludes the proof that $Q_i^j(I_j) = Q_i(I)$ for all $j \leq N$, and in particular, $Q^N(I_N) = Q(I)$. Since $Q^N(I_N) = Q(I)$, the final results phase computes all answers to Q. Since $Q_i(I) = Q_i^j(I_j)$ for all i and j, the mappings generated by Lemma 3.6 during the provision phase are also answers to Q.

The algorithm $\mathcal{A}$ we presented so far produces the results we want, but it is not a constant delay enumeration algorithm. It is left to show that $\mathcal{A}$ conforms to the conditions of the Cheater's lemma. This would mean that we can apply the lemma, and conclude that $\textsc{Enum}\langle Q \rangle \in \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$.

Duplicates. Overall the algorithm produces answers in N provision steps during the provision phase and in n CQ evaluation steps during the final results phase. The answers produced at each individual step contain no duplicates. Therefore, every result appears at most $N + n$ times. Since N and n are constants, this is a constant number of duplicates per answer.

Delay. Consider the jth provision step. It starts with a preprocessing of time $\mathcal{O}(|I_{j-1}|)$ followed by a constant delay enumeration of a set M_j of answers. At its end, $\mathcal{O}(|M_j|)$ time is required to compute the new relation. This means that the delay before the first answer of the jth provision step is $O(|I_{j-1}| + |M_{j-1}|)$, and the delay before answers in the provision phase that are not first in their step is constant. During the final results phase, we have n steps in which we apply the CDY algorithm on an extended CQ. Each such step starts with $O(|I_N|)$ preprocessing time followed by constant delay. Thus, during the final results phase, there are n times where the delay is $O(|I_N|)$, and in other

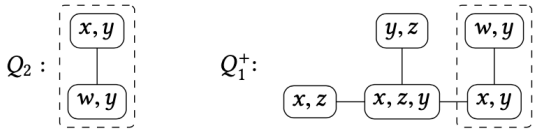


Figure 3.1: Ext-$\{x, y, w\}$-connex trees for Example 1.2.

times the delay is constant.

We have that $|I_j| = |I_{j-1}| + |R_j^{M_j}| \leq |I_{j-1}| + |M_j|$ for all $1 < j \leq N$, and $|I_1| = |I|$. By induction, this shows that $|I_j| \leq |I| + \sum_{i=2}^{j} |M_i|$. This means that whenever the delay is not constant, it is linear in the input size plus the number of (not necessarily unique) answers produces thus far. Since there are $n + N$ such times where the delay is not constant, according to Proposition 3.3, the algorithm $\mathcal{A}$ runs in linear partial time, and the requirements of the Cheater's Lemma on the delay are met. □

We can now revisit Example 1.2 and explain its tractability using the terminology and results introduced in this section. The query Q_2 supplies $\{x, y, w\} \subseteq \text{free}(Q_2)$ as Q_2 is $\{x, y, w\}$-connex. There is a body-homomorphism $h : \text{var}(Q_2) \to \text{var}(Q_1)$ with $h((x, y, w)) = (x, z, y)$. As illustrated in Figure 3.1, adding $R'(x, z, y)$ to Q_1 results in a free-connex union extension $Q_1^+(x, y, w) \leftarrow R_1(x, z), R_2(z, y), R_3(y, w), R'(x, z, y)$. By Theorem 3.9, we have that $\textsc{Enum}\langle Q \rangle \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$.

Remark. Example 1.2 is a counter example to a past made claim [BKS18, Theorem 4.2b]. The claim is that if a non-redundant UCQ contains an intractable CQ, then the union is intractable. In contrast, none of the CQs in Example 1.2 is redundant, Q_1 is intractable, and yet the UCQ is tractable.

The intuition behind the proof of the past claim is reducing the hard CQ Q_1 to Q. This can be done by assigning each variable of Q_1 with a different and disjoint domain (e.g., by concatenating the variable names to the values in the relations corresponding to the atoms), and leaving the relations that do not appear in the atoms of Q_1 empty. It is well known that $Q_1 \subseteq Q_2$ if and only if there exists a homomorphism from Q_2 to Q_1. The claim is that since there is no homomorphism from another CQ in the union to Q_1, then there are no answers to the other CQs with this reduction. However, it is possible that there is a body-homomorphism from another CQ to Q_1 even if it is not a full homomorphism (the free variables do not map to each other). Therefore, in cases of a body-homomorphism, the reduction from Q_1 to Q does not work. In such cases, the union may be tractable, as we show in Theorem 3.9. In Lemma 3.12, we use the same proof described here, but restrict it to UCQs where there is no body-homomorphism from other CQs to Q_1. □

The tractability result in Theorem 3.9 is based on the structure of the union extensions. This means that the intractability of any query within a UCQ can be resolved as long as another query can supply the right variables. The following example

shows that this can even be the case for a UCQ only consisting of non-free-connex CQs. It also illustrates why the definition of union extensions needs to be recursive.

Example 3.10. Let $Q = Q_1 \cup Q_2 \cup Q_3$ with

$$Q_1(x, y, v, u) \leftarrow R_1(x, z_1), R_2(z_1, z_2), R_3(z_2, z_3), R_4(z_3, y), R_5(y, v, u),$$
$$Q_2(x, y, v, u) \leftarrow R_1(x, y), R_2(y, v), R_3(v, z_1), R_4(z_1, u), R_5(u, t_1, t_2),$$
$$Q_3(x, y, v, u) \leftarrow R_1(x, z_1), R_2(z_1, y), R_3(y, v), R_4(v, u), R_5(u, t_1, t_2).$$

Each of three CQs is difficult on its own: Q_1 has the free-path (x, z_1, z_2, z_3, y), while Q_2 has the free-path (v, z_1, u), and Q_3 has the free-path (x, z_1, y). The CQ Q_2 supplies $\{x, y, v\}$ as Q_2 is $\{x, y, v\}$-connex and $\{x, y, v\}$ are free in Q_2. Since there is a body-homomorphism $h_{2,3}$ from Q_2 to Q_3 with $h_{2,3}((x, y, v)) = (x, z_1, y)$, we can extend the body of Q_3 by the virtual atom $R'(x, z_1, y)$, which yields a free-connex extension Q_3^+ of Q_3. Similarly, we have that Q_3 supplies $\{y, v, u\}$, and there is a body-homomorphism $h_{3,2}$ from Q_3 to Q_2 with $h_{3,2}((y, v, u)) = (v, z_1, u)$. Extending Q_2 by $R''(v, z_1, u)$ yields the free-connex extension Q_2^+. Since Q_2^+ and Q_3^+ each supply $\{x, y, v, u\}$, we can add virtual atoms with the variables (x, z_1, z_2, y) and (x, z_2, z_3, y) to Q_1. This results in a free-connex extension Q_1^+. The UCQ $Q_1^+ \cup Q_2^+ \cup Q_3^+$ is a free-connex union extension of Q. By Theorem 3.9, $\textsc{Enum}\langle Q \rangle \in \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$. □

3.2 Hardness under Traditional Assumptions

In this section, we prove lower bounds for evaluating UCQs within the time bounds of $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$. Since for CQs we currently only have hardness result when they are self-join free, we focus on unions of self-join-free CQs. We begin with some general observations regarding cases where a UCQ is at least as hard as a single CQ it contains, and then continue to handle other cases. In Section 3.2.1 we discuss unions containing only difficult CQs, and in Section 3.2.2 we discuss unions containing two body-isomorphic CQs. In both cases such UCQs may be tractable, and in case of such a union of size two, we show that our results from Section 3.1 capture all tractable unions.

In order to provide some intuition for the choices we make throughout this section, we first explain where the approach used for proving the hardness of single CQs fails for UCQs. Consider the union of $Q_1(x, y, w) \leftarrow R_1(x, z), R_2(z, y), R_3(y, w)$ and $Q_2(x, y, w) \leftarrow R_1(x, y), R_2(y, w)$ from Example 1.2. Recall that the original proof that shows that Q_1 is hard describes a reduction from Boolean matrix multiplication. Given binary representations A and B of Boolean $n \times n$ matrices, the reduction defines a database instance I as $R_1^I = A$, $R_2^I = B$, and $R_3^I = \{1, \ldots, n\} \times \{\perp\}$. Then, $Q_1(I)$ corresponds to the answers of AB. If $\textsc{Enum}\langle Q_1 \rangle \in \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$, we can solve matrix multiplication in time $\mathcal{O}(n^2)$, in contradiction to BMM. Since Q_2 evaluates over the same relations, Q_2 also produces answers over this construction. Since the number of answers for Q_2 might reach up to n^3, evaluating Q in constant delay does not necessarily compute the answers to Q_1 in $O(n^2)$ time and does not contradict our assumption.

In general, whenever we show a lower bound to a UCQ by computing a hard problem through answering one CQ in the union, we need to ensure that the other CQs cannot have too many answers over this construction. The following lemma formalizes the idea that by assigning variables of a CQ with different domains, we can restrict the answers obtained by other CQs.

Lemma 3.11. *Given a CQ Q_1 over a schema $\mathcal{S}$, there exist mappings σ and τ such that for every database instance I over $\mathcal{S}$:*

- *τ is a bijection from $Q_1(\sigma(I))$ to $Q_1(I)$.*
- *$\sigma(I)$ can be computed in linear time.*
- *For every CQ Q_i over $\mathcal{S}$:*
  - *$\tau(\mu)$ can be computed in constant time for every answer $\mu \in Q_i(\sigma(I))$.*
  - *If there is no body-homomorphism from Q_i to Q_1, $Q_i(\sigma(I)) = \emptyset$.*
  - *If there is no homomorphism from Q_i to Q_1, given $\mu \in Q_1(\sigma(I)) \cup Q_i(\sigma(I))$, it is possible to determine in constant time whether $\mu \in Q_1(\sigma(I))$.*

Proof. We define σ to assign each variable of Q_1 with a different and disjoint domain by concatenating the variable names to the values in their corresponding relations. For every atom $R(v_1, \cdots, v_m)$ in Q_1 and tuple $(c_1, \cdots, c_m) \in R^I$, we add the tuple $((c_1, v_1), \ldots, (c_m, v_m))$ to $R^{\sigma(I)}$. All relations that do not appear in Q_1 are left empty. We claim that the results of Q_1 over the original instance are exactly the same as over our construction if we omit the variable names. That is, we define $\tau : \mathit{dom} \times \mathrm{var}(Q_1) \to \mathit{dom}$ as $\tau((c, v)) = c$, and show that $Q_1(I) = \tau(Q(\sigma(I)))$. Note the σ and τ can be computed in linear and constant time respectively.

We first prove that $Q_1(I) = \tau(Q_1(\sigma(I)))$. The first direction is trivial: if $\nu|_{\mathrm{free}(Q_1)} \in Q_1(\sigma(I))$, then for every atom $R(\vec{v})$ in Q_1, $\nu(\vec{v}) \in R^{\sigma(I)}$. By construction, $\tau(\nu(\vec{v})) \in R^I$, and therefore $\tau \circ \nu|_{\mathrm{free}(Q_1)} \in Q_1(I)$. We now show the opposite direction. If $\mu|_{\mathrm{free}(Q_1)} \in Q_1(I)$, then for every atom $R(v_1, \cdots, v_m)$ in Q_1, $(\mu(v_1), \ldots, \mu(v_m)) \in R^I$. By construction, $((\mu(v_1), v_1), \ldots, (\mu(v_m), v_m)) \in R^{\sigma(I)}$. By defining $f_\mu : \mathrm{var}(Q_1) \to \mathit{dom} \times \mathrm{var}(Q_1)$ as $f_\mu(u) = (\mu(u), u)$, we have $f_\mu \in Q_1(\sigma(I))$. Since $\tau \circ f_\mu = \mu$, we have that $\mu|_{\mathrm{free}(Q_1)} \in \tau(Q_1(\sigma(I)))$, and this concludes that $Q_1(I) \subseteq \tau(Q_1(\sigma(I)))$.

We now show that $Q_i(\sigma(I)) = \emptyset$ if there is no body-homomorphism from Q_i to Q_1. Assume by contradiction that there exists such $\mu|_{\mathrm{free}(Q_i)} \in Q_i(\sigma(I))$. This means that for every atom $R(\vec{v})$ in Q_i, $\mu(\vec{v}) \in R^{\sigma(I)}$. By construction, $\mu(\vec{v})$, like all tuples in $R^{\sigma(I)}$, is of the form $((c_1, v_1), \ldots, (c_m, v_m))$ such that $R(v_1, \ldots, v_m)$ is an atom in Q_1. Define $\eta : \mathit{dom} \times \mathrm{var}(Q_1) \to \mathrm{var}(Q_1)$ as $\eta(c, v) = v$. We have that for every atom $R(\vec{v})$ in Q_i, $R(\eta(\mu(\vec{v})))$ is an atom in Q_1. This means that $\eta \circ \mu$ is a body-homomorphism from Q_i to Q_1, which is a contradiction.

It is left to show that, if there is no homomorphism from $Q_i(\vec{u})$ to $Q_1(\vec{u})$, given an answer to $Q_1(\sigma(I)) \cup Q_i(\sigma(I))$, it is possible to determine in constant time whether it is in $Q_1(\sigma(I))$. Here, we need to be more careful, so we use the definition of an answer as a tuple (rather than a mapping). As we showed, Q_i has answers over $\sigma(I)$ only if

there is a body-homomorphism h from it to Q_1. Since this is not a full homomorphism, $h(\vec{u}) \neq \vec{u}$. If $\vec{a}$ is an answer to Q_1, then $\eta(\vec{a}) = \vec{u}$. Otherwise, if $\vec{a}$ is an answer to Q_i, then $\eta(\vec{a}) = h(\vec{u})$. Since $h(\vec{u}) \neq \vec{u}$, this helps us distinguish the answers: apply η on the answer; if this results in $\vec{u}$, then it is an answer to Q_1; otherwise, μ is an answer to Q_i. □

The following lemma identifies cases where we can encode any arbitrary instance of a CQ to an instance of the union containing it, such that no other CQ in the union returns answers.

Lemma 3.12. *Let Q be a UCQ, and let $Q_1 \in Q$ such that for all $Q_i \in Q \setminus \{Q_1\}$ there is no body-homomorphism from Q_i to Q_1. Then,* $\textsc{Enum}\langle Q_1 \rangle \leq_e \textsc{Enum}\langle Q \rangle$.

Proof. This reduction can be performed using the mappings σ and τ defined in Lemma 3.11. For every $Q_i \in Q \setminus \{Q_1\}$, there is no body-homomorphism from Q_i to Q_1, and so $Q_i(\sigma(I)) = \emptyset$. We now have that $\tau(Q(\sigma(I))) = \bigcup_{Q_i \in Q} \tau(Q_i(\sigma(I))) = \tau(Q_1(\sigma(I)))$. Since also $\tau(Q_1(\sigma(I))) = Q_1(I)$, this concludes our reduction. □

The lemma above implies that if there is a difficult CQ in a union where no other CQ maps to it via a body-homomorphism, then the entire union is intractable. This also captures cases such as a union of CQs where one of them is hard, and the others contain a relation that does not appear in the hard CQ.

Using the same reduction, a similar statement with relaxed requirements can be made in case it is sufficient to consider the decision problem, denoted $\textsc{Decide}\langle Q \rangle$, that determines whether the answer is non-empty (see definition in Section 2.1).

Lemma 3.13. *Let Q be a UCQ, and let $Q_1 \in Q$ such that for all $Q_i \in Q$, either there exists no body-homomorphism from Q_i to Q_1, or Q_1 and Q_i are body-isomorphic. Then,* $\textsc{Decide}\langle Q_1 \rangle \leq \textsc{Decide}\langle Q \rangle$ *via a linear-time many-one reduction.*

Proof. We use the encoding from Lemma 3.11. We know that $Q_i(\sigma(I)) = \emptyset$ for every CQ Q_i with no body-homomorphism to Q_1. We claim now that a CQ Q_j which is body-isomorphic to Q_1 has an answer iff Q_1 has an answer. Therefore $Q(\sigma(I)) \neq \emptyset$ iff $Q_1(\sigma(I)) \neq \emptyset$. We know that Q_1 retains the same answers under this encoding, so in particular $Q_1(\sigma(I)) = \emptyset$ if and only if $Q_1(I) = \emptyset$. Overall this shows that $Q(\sigma(I)) \neq \emptyset$ if and only if $Q_1(I) \neq \emptyset$.

We now show formally that for body-isomorphic CQs Q_1 and Q_2 and database I, $Q_1(I) \neq \emptyset$ iff $Q_2(I) \neq \emptyset$. Let h be a body-homomorphism from Q_2 to Q_1. That is, for every atom $R(\vec{v}) \in Q_2$, we have $R(h(\vec{v})) \in Q_1$. If $Q_1(I) \neq \emptyset$, then there exists $\mu|_{\text{free}(Q_1)} \in Q_1(I)$, and for every atom $R(h(\vec{v})) \in Q_1$, we have $\mu(h(\vec{v})) \in R^I$. This means that $\mu \circ h$ is a homomorphism from Q_2 to I, and $\mu \circ h|_{\text{free}(Q_2)} \in Q_2(I)$. So, $Q_2(I) \neq \emptyset$. Since there is also a body-homomorphism from Q_1 to Q_2, we can show in the same way that if $Q_2(I) \neq \emptyset$ then $Q_1(I) \neq \emptyset$. □

Theorem 2.1 states that deciding whether a cyclic CQ has any answers cannot be done in linear time (assuming HYPERCLIQUE). Following Lemma 3.13, if a UCQ Q contains a cyclic Q_1, and the conditions of Lemma 3.13 are satisfied with respect to this CQ, then the entire union cannot be decided in linear time, and thus $\text{ENUM}\langle Q\rangle \notin \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$.

3.2.1 Unions of Difficult CQs

We now discuss unions containing only CQs classified as hard according to Theorem 2.1. Recall that these are called *difficult CQs*, and they are self-join-free CQs that are not free-connex. The following lemma identifies a CQ on which we can apply Lemma 3.12 or Lemma 3.13.

Lemma 3.14. *Let Q be a UCQ. There exists a query $Q_1 \in Q$ such that for all $Q_i \in Q$ either there is no body-homomorphism from Q_i to Q_1 or Q_1 and Q_i are body-homomorphically equivalent.*

Proof. Consider a longest sequence $(Q^1, \ldots, Q^m)$ of CQs from Q such that for every $2 \leq j \leq m$ there is a body-homomorphism from Q^j to Q^{j-1}, but no body-homomorphism in the opposite direction. We claim that $Q_1 = Q^m$ satisfies the conditions of the lemma.

First we show that such a sequence exists. We denote the body-homomorphism from Q^j to Q^{j-1} by h^j. It is not possible that the same query appears twice in the sequence: if $Q^i = Q^j$ where $j > i$, then there is a mapping $h^{i+2} \circ \ldots \circ h^j$ from $Q^j = Q^i$ to Q^{i+1}, in contradiction to the definition of the sequence. Therefore, $m \leq |Q|$, and a longest sequence exists. We now show that Q^m satisfies the requirements. First consider some $Q^j \in \{Q^1, \ldots, Q^{m-1}\}$. There is a body-homomorphism from Q^m to Q^j which is the composition $h^{j+1} \circ \ldots \circ h^m$. Therefore, either there is no body-homomorphism from Q^j to Q^m, or Q^m and Q^j are body-homomorphically equivalent. In addition, Q^m is body-homomorphically equivalent to itself with the identity mapping serving as the body-homomorphisms. It is left to consider CQs that are not on the sequence. Let $Q_i \in Q \setminus \{Q^1, \ldots, Q^m\}$. If there is no body-homomorphism from Q_i to Q^m, then we are done. Otherwise, if there is also no body-homomorphism from Q_1 to Q^m, then $(Q^1, \ldots, Q^m, Q_i)$ is a longer sequence, contradicting the maximality of the sequence we started with. Therefore, in this case, Q^m and Q_i are body-homomorphically equivalent. Overall we showed that for each query in Q either there is no body-homomorphism from this query to Q^m or these two CQs are body-homomorphically equivalent. □

Using the results obtained so far, we deduce a characterization of all cases of a union of difficult CQs, except those that contain a pair of body-isomorphic acyclic CQs.

Theorem 3.15. *Let Q be a union of difficult CQs not containing two body-isomorphic acyclic CQs. Then, $Q \notin \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$, assuming* BMM *and* HYPERCLIQUE.

Proof. Let Q_1 be a CQ in Q given by Lemma 3.14. By definition, difficult CQs are self-join-free, and so if two CQs in Q are body-homomorphically equivalent, they are

also body-isomorphic. We treat the two possible cases of the structure of Q_1. In case Q_1 is acyclic, since we know that Q does not contain body-isomorphic acyclic CQs, then for all $Q_i \in Q \setminus \{Q_1\}$ there is no body-homomorphism from Q_i to Q_1. According to Lemma 3.12, $\textsc{Enum}\langle Q_1 \rangle \leq_e \textsc{Enum}\langle Q \rangle$. Since Q_1 is self-join-free acyclic non-free-connex, we have that $\textsc{Enum}\langle Q_1 \rangle \notin \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$ assuming BMM. Therefore $\textsc{Enum}\langle Q \rangle$ is not in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$ either. In case Q_1 is cyclic, we use Lemma 3.13 to conclude that $\textsc{Decide}\langle Q_1 \rangle \leq \textsc{Decide}\langle Q \rangle$. According to Theorem 2.1, since Q_1 is self-join-free cyclic, $\textsc{Decide}\langle Q_1 \rangle$ cannot be solved in linear time assuming Hyperclique. Therefore $\textsc{Decide}\langle Q \rangle$ cannot be solved in linear time either, and $\textsc{Enum}\langle Q \rangle \notin \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$.□

In the next example, we demonstrate how the reductions described in Lemma 3.13 and Theorem 2.1 combine in Theorem 3.15.

Example 3.16. Consider the UCQ $Q = Q_1 \cup Q_2 \cup Q_3$ with

$$Q_1(x, y) \leftarrow R_1(x, y), R_2(y, u), R_3(x, u),$$
$$Q_2(x, y) \leftarrow R_1(y, v), R_2(v, x), R_3(y, x),$$
$$Q_3(x, y) \leftarrow R_1(x, z), R_2(y, z).$$

The queries Q_1 and Q_2 are cyclic, and Q_3 is acyclic but not free-connex. This union is intractable according to Theorem 3.15. Note that Q_1 and Q_2 are body-isomorphic, but there is no body-homomorphism from Q_3 to Q_1. The proof of Theorem 2.1 states the following: if $\textsc{Enum}\langle Q_1 \rangle \in \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$, then given an input graph G, we can use Q_1 to decide the existence of triangles in G in time $O(n^2)$, in contradiction to Hyperclique. The same holds true for $\textsc{Enum}\langle Q \rangle$. For every edge (u, v) in G with $u < v$ we add $((u, x), (v, y))$ to R_1^I, $((u, y), (v, z))$ to R_2^I and $((u, x), (v, z))$ to R_3^I. The query detects triangles: for every triangle a, b, c in G with $a < b < c$, the query Q_1 returns $((a, x), (b, y))$. The union only returns answers corresponding to triangles:

- For every answer $((d, x), (e, y))$ to Q_1, there exists some f such that d, e, f is a triangle in G with $d < e < f$.
- For every answer $((g, z), (h, x))$ to Q_2, there exists some i such that g, h, i is a triangle in G with $h < i < g$.
- The query Q_3 returns no answers over I. □

Theorem 3.15 does not cover the case of a UCQ containing acyclic non-free-connex queries with isomorphic bodies. We handle such queries next. Since this requires a more intricate analysis, we restrict ourselves to such unions of size two.

3.2.2 Unions of Two Body-Isomorphic CQs

To complete our hardness results from the previous section, we now consider a set of self-join-free body-isomorphic CQs. As all of the CQs in such a set have the same structure, either every CQ in this set is cyclic, or every CQ is acyclic. In the case of a union of two cyclic CQs, the UCQ is intractable according to Theorem 3.15. So in

this section, we discuss the union of body-isomorphic acyclic CQs. Unlike the previous section, we do not restrict ourselves in the following discussion to difficult CQs. We first introduce a new notation for body-isomorphic UCQs that we use hereafter.

Consider a union of self-join-free CQs of the form $Q_1 \cup Q_2$, where there exists a body-isomorphism h from Q_2 to Q_1. That is, the CQs have the structure:

$$Q_1(\vec{v}_1) \leftarrow R_1(h(\vec{w}_1)), \ldots, R_n(h(\vec{w}_n)),$$
$$Q_2(\vec{v}_2) \leftarrow R_1(\vec{w}_1), \ldots, R_n(\vec{w}_n).$$

Applying h^{-1} to the variables of Q_1 does not affect evaluation, so we can rewrite Q_1 as $Q_1(h^{-1}(\vec{v}_1)) \leftarrow R_1(\vec{w}_1), \ldots, R_n(\vec{w}_n)$. Since now the two CQs have exactly the same body, we can treat the UCQ as a query with one body and two heads:

$$Q_1(h^{-1}(\vec{v}_1)), Q_2(\vec{v}_2) \leftarrow R_1(\vec{w}_1), \ldots, R_n(\vec{w}_n)$$

We use this notation from now on for UCQs containing only body-isomorphic CQs. Note that when treating a UCQ as one CQ with several heads, we can use the notation atoms(Q), as the atoms are the same for all CQs in the union, and the notation free(Q_i), as the free variables may differ between different queries Q_i in the union. With this notation at hand, we now inspect some examples of two body-isomorphic acyclic CQs.

Example 3.17. Consider $Q = Q_1 \cup Q_2$ with

$$Q_1(x, y, v) \leftarrow R_1(x, z), R_2(z, y), R_3(y, v), R_4(v, w) \text{ and}$$
$$Q_2(x, y, v) \leftarrow R_1(w, v), R_2(v, y), R_3(y, z), R_4(z, x).$$

Since Q_1 and Q_2 are body-isomorphic, Q can be rewritten as

$$Q_1(w, y, z), Q_2(x, y, v) \leftarrow R_1(w, v), R_2(v, y), R_3(y, z), R_4(z, x)$$

In this case we can use the same approach used for single CQs in Theorem 2.1, and show that this UCQ is not in $\mathsf{Enum}\langle \text{lin}, \text{const} \rangle$ assuming BMM. Let A and B be binary representations of Boolean $n \times n$ matrices. Define a database instance I with $R_1^I = A$, $R_2^I = B$, $R_3^I = \{1, \ldots, n\} \times \{\bot\}$ and $R_4^I = \{(\bot, \bot)\}$. $Q_1(I)$ corresponds to the answers of AB, and $|Q_2(I)| = \mathcal{O}(n^2)$. Assume by contradiction that $\textsc{Enum}\langle Q \rangle \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$. Then we find $Q(I)$ in $\mathcal{O}(n^2)$ time. We can distinguish the answers of Q_1 from those of Q_2 since the possible assignments to the variables in different queries are from disjoint domains: given an answer $\mu \in Q(I)$, we have that $\mu \in Q_1(I)$ if and only if $\mu(v) = \bot$ (or, in tuple notation, the answer is in Q_1 iff its last element is $\bot$). This means we can solve matrix multiplication in time $\mathcal{O}(n^2)$, which contradicts BMM. □

A union of two difficult body-isomorphic acyclic CQs may also be tractable. In fact, by adding a single variable to the heads in Example 3.17, it becomes tractable.

Example 3.18. Let Q be the UCQ

$$Q_1(w, y, x, z), Q_2(x, y, w, v) \leftarrow R_1(w, v), R_2(v, y), R_3(y, z), R_4(z, x)$$

Both CQs are acyclic non-free-connex. As Q_2 supplies the variables $\{v, w, y\}$ and Q_1 supplies $\{x, y, z\}$, both CQs have free-connex union extensions:

$$Q_1^+(w, y, x, z) \leftarrow R_1(w, v), R_2(v, y), R_3(y, z), R_4(z, x), P_1(v, w, y),$$
$$Q_2^+(x, y, w, v) \leftarrow R_1(w, v), R_2(v, y), R_3(y, z), R_4(z, x), P_2(x, y, z).$$

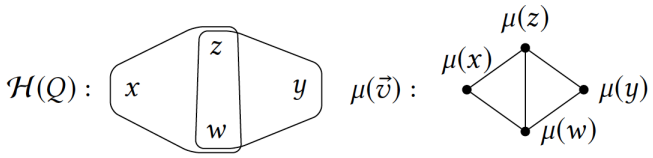


Figure 3.2: In Example 3.19, $\mu(w, x, y, z) \in Q(I)$ forms a 4-clique where one edge might be missing.

By Theorem 3.9, $\textsc{Enum}\langle Q \rangle \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$. □

Intuitively, the reason the reduction of Example 3.17 fails in Example 3.18 is the fact that all the variables of the free-paths in one CQ, which are used to encode matrix multiplication, are free in the other CQ. Indeed, if we encode matrices A and B to the relations of the free path w, v, y in Q_1, there can be n^3 answers to Q_2. The answer set in this case is too large to contradict the assumed lower bound for matrix multiplication. As it turns out, there are cases where we cannot reduce matrix multiplication to a union in this manner, and yet we can show that it is intractable using an alternative problem.

Example 3.19. Let Q be the UCQ

$$Q_1(x, y, u), Q_2(x, y, z) \leftarrow R_1(x, u, z), R_2(y, u, z).$$

This union is intractable under the 4-clique assumption. For a given graph $G = (V, E)$ with $|V| = n$, we compute the set T of all triangles in G in time $O(n^3)$. Define a database instance I as $R_1^I = R_2^I = T$. For every output $\mu|_{\text{free}(Q_i)}$ in $Q(I)$ with $i \in \{1, 2\}$, we know that $(\mu(x), \mu(u), \mu(z))$ and $(\mu(y), \mu(u), \mu(z))$ are triangles. If $\mu(x) \neq \mu(y)$, this means that $\mu((x, y)) \in E$ if and only if $(\mu(x), \mu(y), \mu(u), \mu(z))$ forms a 4-clique (see Figure 3.2). Moreover, for every 4-clique (a, b, c, d) in the graph, we have $(a, b, c) \in Q_1(I)$, so this will detect all 4-cliques. Since there are $O(n^3)$ answers to Q, if $\textsc{Enum}\langle Q \rangle \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$, we can check whether $\mu((x, y)) \in E$ for every answer in $Q(I)$ (in tuple notation, for every answer we check whether its first two elements share an edge), and determine whether a 4-clique appears in G in time $\mathcal{O}(n^3)$. □

Note that we can use the 4-clique assumption in Example 3.19, since, in addition to the free-path variables, there is another variable in both free-path relations. We now generalize the observations from the examples.

Definition 3.20. Let $Q = Q_1 \cup Q_2$ be a UCQ where Q_1 and Q_2 are body-isomorphic.

- Q_1 is said to be *free-path guarded* if for every free-path P in Q_1, we have that $\text{var}(P) \subseteq \text{free}(Q_2)$.
- Consider a path $P = (u_1, \dots u_k)$ in Q_1. Two atoms $R_1(\vec{v}_1)$ and $R_2(\vec{v}_2)$ of Q_1 are called *subsequent P-atoms* if $\{u_{i-1}, u_i\} \subseteq \vec{v}_1$ and $\{u_i, u_{i+1}\} \subseteq \vec{v}_2$ for some $1 < i < k$.
- Q_1 is said to be *bypass guarded* if for every free-path P in Q_1 and variable u that appears in two subsequent P-atoms, we have that $u \in \text{free}(Q_2)$.

Note that every free-connex CQ is trivially free-path guarded and bypass guarded.

Let us demonstrate the definitions on the recent examples. The CQ Q_1 of Example 3.18 is both free-path guarded and bypass guarded: the only free-path it contains is $P = (w, v, y)$, and Q_1 is free-path guarded since $\{w, v, y\} \subseteq \text{free}(Q_2)$; the only subsequent P-atoms are $R_1(w, v)$ and $R_2(v, y)$, and Q_1 is bypass guarded since $v \in \text{free}(Q_2)$. The query Q_1 of Example 3.17 is not free-path guarded as the variables of the free-path $P' = (w, v, y)$ of Q_1 are not contained in $\text{free}(Q_2)$. The query Q_1 of Example 3.19 is not bypass guarded. Consider the free-path $P'' = (x, z, y)$ of Q_1. The atoms $R_1(x, u, z)$ and $R_2(y, u, z)$ are subsequent P'' atoms. Since u is contained in both atoms but not in $\text{free}(Q_2)$, we have that Q_1 is not bypass guarded.

In the following two lemmas, we prove that if some CQ in a union is either not free-path guarded or not bypass guarded, then the UCQ is intractable. As in the characterization for CQs in Theorem 2.1, our lower bounds apply only when the CQs are self-join-free; with this restriction, we can assign different atoms with different relations. The next lemma shows that the reduction in Example 3.17, where we can use the fact that Q_1 is not free-path guarded to compute matrix multiplication, can be constructed in the general case as well.

Lemma 3.21. *Let $Q = Q_1 \cup Q_2$ be a non-redundant union of self-join-free body-isomorphic acyclic CQs. Assuming* BMM*, if Q_1 is not free-path guarded, then* $\textsc{Enum}\langle Q \rangle$ *is not in* $\mathsf{Enum}\langle \text{lin}, \text{const} \rangle$.

Proof. Let A and B be Boolean $n \times n$ matrices represented as binary relations, i.e. $A \subseteq \{1, \ldots, n\}^2$, where $(a, b) \in A$ means that the entry in the ath row and bth column is 1. Further let $P = (z_0, \ldots, z_{k+1})$ be a free-path in Q_1 that is not guarded, and let i be the minimal index such that $z_i \notin \text{free}(Q_2)$. We assign the path variables to three roles as follows. If $0 < i < k + 1$, we define $V_x = \{z_0, \ldots, z_{i-1}\}$, $V_z = \{z_i\}$ and $V_y = \{z_{i+1} \ldots, z_{k+1}\}$. Otherwise (if i is 0 or $k+1$), we define $V_x = \{z_0\}$, $V_z = \{z_1, \ldots, z_k\}$ and $V_y = \{z_{k+1}\}$. Note that, in both cases, there exists some $\alpha \in \{x, y, z\}$ such that $V_\alpha \cap \text{free}(Q_2) = \emptyset$. Intuitively that means that some role is not guarded. Note also that since P is chordless and $k \geq 1$, there is no atom in Q that contains both a variable from V_x and a variable from V_y. Thus we can partition the atoms into nonempty sets $\mathcal{R}_A = \{R(\vec{v}) \in \text{atoms}(Q) \mid V_y \cap \vec{v} = \emptyset\}$ and $\mathcal{R}_B = \text{atoms}(Q) \setminus \mathcal{R}_A$, and we have that the atoms of $\mathcal{R}_A$ do not contain variables of V_y, and the atoms of $\mathcal{R}_B$ do not contain variables of V_x.

Given three values (a, b, c) we define a function $\tau_{(a,b,c)} : \text{var}(Q) \to \{a, b, c, \bot\}$ that assigns every variable with the value corresponding to its role:

$$\tau_{(a,b,c)}(v) = \begin{cases} a & \text{if } v \in V_x, \\ b & \text{if } v \in V_z, \\ c & \text{if } v \in V_y, \\ \bot & \text{otherwise,} \end{cases}$$

For a vector $\vec{v}$, we denote by $\tau_{(a,b,c)}(\vec{v})$ the vector obtained by element-wise application of $\tau_{(a,b,c)}$. We define a database instance I over Q as follows: For every atom $R(\vec{v})$, if $R(\vec{v}) \in \mathcal{R}_A$ we set $R^I = \{\tau_{(a,b,\perp)}(\vec{v}) \mid (a,b) \in A\}$, and if $R(\vec{v}) \in \mathcal{R}_B$ we set $R^I = \{\tau_{(\perp,b,c)}(\vec{v}) \mid (b,c) \in B\}$. Note that every relation is defined only once since $\mathcal{R}_A$ and $\mathcal{R}_B$ are disjoint and Q is self-join-free.

Consider an answer $\mu \in Q(I)$. In the case that $0 < i < k+1$, we have that $\mu(z_0) = \cdots = \mu(z_{i-1}) = a$, $\mu(z_i) = b$ and $\mu(z_{i+1}) = \cdots = \mu(z_{k+1}) = c$ for some $(a,b) \in A$ and $(b,c) \in B$, and in case that $i \in \{0, k+1\}$, we have that $\mu(z_0) = a$, $\mu(z_1) = \cdots = \mu(z_k) = b$ and $\mu(z_{k+1}) = c$ for some $(a,b) \in A$ and $(b,c) \in B$. This is since the variables z_i are connected via the path in both CQs. In either case, $\mu(\text{free}(Q_1))$ is a tuple only containing the values a, c and $\perp$. So the answers to Q_1 represent the answer to the matrix multiplication task we started with. We now need to verify that the answers to Q_2 do not interfere with the reduction. If $0 < i < k+1$, $\mu(\text{free}(Q_2))$ is a tuple only containing the values $\{a, c, \perp\}$; if $i = 0$, it only contains $\{b, c, \perp\}$; and if $i = k+1$, it only contains $\{a, b, \perp\}$. Thus the number of answers to Q is at most of size $2n^2$. Assume by contradiction that we can enumerate the solutions of $Q(I)$ with linear preprocessing and constant delay. To distinguish the answers of Q_1 from those of Q_2, we can concatenate the variable names to the values, as described in Lemma 3.11. That way, we can ignore the answers that correspond to the values (a,b) or (b,c), and use the (a,c) pairs as the answers to matrix multiplication. This solves matrix multiplication in $O(n^2)$ time, in contradiction to BMM. □

In Example 3.19, we encounter a UCQ where both CQs are free-path guarded, but Q_1 is not bypass guarded. We can encode 4-clique in every UCQ with this property.

Lemma 3.22. *Let $Q = Q_1 \cup Q_2$ be a non-redundant union of self-join-free body-isomorphic acyclic CQs. If Q_1 and Q_2 are free-path guarded and Q_1 is not bypass guarded, then* Enum$\langle Q \rangle$ *is not in* Enum$\langle \text{lin}, \text{const} \rangle$*, assuming 4-*clique.

Proof. Let $G = (V, E)$ be a graph with $|V| = n$. We show how to solve the 4-clique problem on G in time $\mathcal{O}(n^3)$ if Enum$\langle Q \rangle$ is in Enum$\langle \text{lin}, \text{const} \rangle$. Let P be a free-path in Q_1 and let $u \notin \text{free}(Q_2)$ such that u appears in two subsequent P-atoms.

We first claim that, under the conditions of this lemma, P is of length 2. Let $P = (z_0, \ldots, z_{k+1})$ and $1 \leq i \leq k$ such that $\{u, z_{i-1}, z_i\}$ and $\{u, z_i, z_{i+1}\}$ are contained in edges of $\mathcal{H}(Q)$. As P is chordless, there is no edge containing $\{z_{i-1}, z_{i+1}\}$, thus the path (z_{i-1}, u, z_{i+1}) is a chordless path. As Q_1 is free-path guarded, $z_{i-1}, z_{i+1} \in \text{free}(Q_2)$ and since $u \notin \text{free}(Q_2)$, this is a free-path of Q_2. Since Q_2 is free-path guarded we have that $z_{i-1}, z_{i+1} \in \text{free}(Q_1)$. Since P is a free-path in Q_1, the only variables of P that are free in Q_1 appear in the edges of the path, and so $i = k = 1$, and P is of the form (z_0, z_1, z_2). Therefore, there exist atoms R_1 and R_2 with $\{z_0, z_1, u\} \subseteq \text{var}(R_1)$ and $\{z_1, z_2, u\} \subseteq \text{var}(R_2)$.

Given three values (a, b, c) we define a function $\tau_{(a,b,c)} : \text{var}(Q) \to \{a, b, c\}$ as follows:

$$\tau_{a,b,c}(v) = \begin{cases} a & \text{if } v \in \{z_0, z_2\}, \\ b & \text{if } v = z_1, \\ c & \text{if } v = u, \\ \perp & \text{otherwise.} \end{cases}$$

For every atom $R(\vec{v}) \in \text{atoms}(Q)$, we define $R^I = \{\tau_{a,b,c}(\vec{v}) \mid (a, b, c) \text{ a triangle in } G\}$. Every relation is defined only once since Q is self-join-free. Note that $|R^I| \in \mathcal{O}(n^3)$, as there are at most n^3 triangles in G, and that we can construct I with $\mathcal{O}(n^3)$ time.

Consider an answer $\mu|_{\text{free}(Q_1)} \in Q_1(I)$. Since R_1 and R_2 are atoms in Q_1, we are guaranteed that $(\mu(z_0), \mu(z_1), \mu(u))$ and $(\mu(z_1), \mu(z_2), \mu(u))$ form triangles in G. Therefore, the graph contains a 4-clique if and only if there is an edge $(\mu(z_0), \mu(z_2))$. As $z_0, z_2 \in \text{free}(Q_1)$ it suffices to check every $\mu|_{\text{free}(Q_1)} \in Q_1(I)$ for this property. Since Q_1 is free-path guarded, we know that z_1 is existential in Q_1 but free in Q_2. This means that $\text{free}(Q_1) \neq \text{free}(Q_2)$ and so there is no homomorphism from Q_2 to Q_1. We can therefore use the mappings from Lemma 3.11 in order to distinguish the answers of Q_1 from those of Q_2. We have that $\{z_0, z_1, z_2, u\}$ is neither contained in $\text{free}(Q_1)$ nor in $\text{free}(Q_2)$. Thus, $|Q(I)| \in \mathcal{O}(n^3)$. If $\text{Enum}\langle Q \rangle$ is in $\mathsf{Enum}\langle \text{lin}, \text{const} \rangle$, we can construct the database instance, compute Q, and check every answer to Q_1 for an edge of the form $(\mu(z_0), \mu(z_2))$ in total time $\mathcal{O}(n^3)$, which contradicts 4-clique. □

In the next section, we show that Lemma 3.21 and Lemma 3.22 cover all intractable cases of the UCQs discussed in this section.

3.2.3 Complete Classification of Fragments of UCQs

We show next that the hardness proofs given above complete our tractability results into a dichotomy for the fragments of UCQs we examined. We first establish this for unions of two body-isomorphic acyclic CQs, and then conclude the same for unions of two difficult CQs.

To prove that any union of two body-isomorphic acyclic CQs that is not covered by Lemma 3.21 and Lemma 3.22 has a free-connex union extension, we need some observations regarding the place of appearance of relevant variables in join-trees. Recall that we call $(v_1, \ldots, v_n)$ a path of variables in a query Q if for all $1 \leq i < n$, there exists an atom $R(\vec{u_i})$ in Q such that $\{v_i, v_{i+1}\} \subseteq \vec{v}$.

Proposition 3.23. *Let $(v_1, \ldots, v_n)$ be a path of variables in an acyclic query Q, and let A_s and A_t be atoms containing $\{v_1, v_2\}$ and $\{v_{n-1}, v_n\}$ respectively. For all $1 \leq i < n$, the simple path between A_s and A_t on a join-tree of Q contains an atom $R_i(\vec{u_i})$ such that $\{v_i, v_{i+1}\} \subseteq \vec{u_i}$.*

Proof. We prove this by induction on n. If $n = 3$, this trivially holds as the endpoint, A_s and A_t, contain the required variables. We now assume this proposition holds for paths of length $n - 1$ and show that it also holds for paths of length n where $n \geq 4$. Consider

the simple path between A_s and a node containing $\{v_{n-2}, v_{n-1}\}$. Let A_m be the first node on that path that contains $\{v_{n-2}, v_{n-1}\}$. Denote by P_s the simple path between A_s and A_m. Note that A_m is the only node on P_s that contains $\{v_{n-2}, v_{n-1}\}$. By the induction assumption, P_s contains $\{v_i, v_{i+1}\}$ for all $1 \leq i < n-1$, and in particular it contains $\{v_{n-3}, v_{n-2}\}$. Denote by P_t the simple path between A_m and A_t, and denote by P the concatenation of P_s and P_t. The path P goes between A_s and A_t and contains $\{v_i, v_{i+1}\}$ for all $1 \leq i < n$. We claim that P is a simple path. Assume by contradiction that P is not simple. This means that there is a node A_v that P_s and P_t share other than A_m. Due to the running intersection property, since A_v is on the simple path between A_m and A_t, A_v contains v_{n-1}. Denote by A_{m-1} the node preceding A_m on P_s. Due to the running intersection property, since A_{m-1} is on the simple path between A_v and A_m, A_{m-1} contains v_{n-1} too. Since a node containing $\{v_{n-3}, v_{n-2}\}$ is on P_s, and since A_{m-1} is on the simple path between this node and A_m, A_{m-1} contains v_{n-2}. Overall, we have that A_{m-1} contains $\{v_{n-2}, v_{n-1}\}$, which is a contradiction to the fact that A_m is the only such node on P_s. □

Proposition 3.24. *Let $(v_1, \ldots, v_n)$ be a path of variables in an acyclic query Q, and let A_s and A_t be atoms containing $\{v_1, v_2\}$ and $\{v_{n-1}, v_n\}$ respectively. If A_1 and A_2 are two subsequent nodes on the simple path P between A_s and A_t on a join-tree of Q, then there exists some $1 \leq i \leq n$ such that $v_i \in A_1 \cap A_2$.*

Proof. Denote the subpath of P between A_s and A_1 by P_s, and the subpath between A_2 and A_t by P_t. Denote by i the maximal index such that $v_i \in \text{var}(P_s)$. If $i = n$, note that an atom in P_s and an atom in P_t both contain v_i. Otherwise, $i < n$. According to Proposition 3.23 and since all atoms of P appear in either P_s or P_t, an atom containing the variables $\{v_i, v_{i+1}\}$ must appear in P_s or P_t. It does not appear in P_s because of the maximality of i, so it appears in P_t. In this case too, an atom in P_s and an atom in P_t both contain v_i. Since A_1 and A_2 are on the path between these atoms, due to the running intersection property, $v_i \in A_1 \cap A_2$. □

Using this proposition, we can prove the structural property that we need.

Lemma 3.25. *Let $Q = Q_1 \cup Q_2$ be a union of body-isomorphic acyclic CQs, where Q_1 and Q_2 are free-path guarded, and Q_1 is bypass guarded, and let $P = (z_0, \ldots, z_{k+1})$ be a free-path in Q_1. There exists a join-tree T for Q_1 with a subtree T_P such that:*

- $\text{var}(P) \subseteq \text{var}(T_P)$.
- *For every variable u that appears in two different atoms of T_P:*
  - $u \in \text{free}(Q_2)$.
  - *There is an atom $R(\vec{v})$ in T_P such that $u, z_i \in \vec{v}$ for some $0 < i < k+1$.*

Proof. Recall according to our notation, we assume that body-isomorphic CQs have exactly the same body. Thus, T is also a join-tree for Q_2. In the following, we refer to the body of Q when we make statements that apply to the bodies of both Q_1 and Q_2.

Consider a path $P_A = (A_1, \ldots, A_s)$ between two atoms on a join tree. We define a *contraction step* for a path of length 2 or more: if there exists j such that $A_j \cap A_{j+1} \subseteq A_1 \cap A_s$, then remove the edge (A_j, A_{j+1}) and add the edge (A_1, A_s). A path on a join-tree is said to be *fully-contracted* if none of its subpaths can be contracted. Given any two atoms on a join-tree, it is possible to fully contract the path between them by performing any arbitrary sequence of contraction steps until it is no longer possible: the process will end as every contraction step reduces the length of the path.

We now claim that the graph T' obtained from such a contraction step of a path P_A on a join-tree T, remains a join-tree. It is still a tree since it remains connected and with the same number of edges as before. It is left to show the running intersection property. We start with some observations regarding T. Since the running intersection property holds in T, for all $1 \leq i \leq s$, $A_1 \cap A_s \subseteq A_i$. Since $A_j \cap A_{j+1} \subseteq A_1 \cap A_s$, we also have that $A_j \cap A_{j+1} \subseteq A_i$. Now consider two nodes B and C. We need to show that every node on the simple path between them in T' contains $B \cap C$. If it is the same path as in T, then we are done. Otherwise, a path between B and C in T' can be obtained by using the simple path between them in T and replacing the edge (A_j, A_{j+1}) with $(A_{j+1}, \ldots, A_s, A_1, \ldots, A_j)$. The simple path between B and C is contained in this path. This means that atoms on the simple path between B and C in T' are either: (1) on the simple path between B and C in T, and therefore contain $B \cap C$; (2) on the path P_A and therefore contain $A_j \cap A_{j+1}$. Since A_j and A_{j+1} are on the path between B and C in T, we have that $B \cap C \subseteq A_j \cap A_{j+1}$, so in this case too, the atoms contain $B \cap C$. This proves that the contracted graph is indeed a join-tree. By induction, if we fully contract a path on a join-tree, we still have a join-tree.

Now let T be a join-tree of Q, and denote $P = (z_0, \ldots, z_{k+1})$. We consider some path in T between an atom containing $\{z_0, z_1\}$ and an atom containing $\{z_k, z_{k+1}\}$. Take the unique subpath of it containing only one atom with $\{z_0, z_1\}$ and one atom with $\{z_k, z_{k+1}\}$, and fully contract it. We denote this fully contracted subpath as $T_P = (A_1, \ldots, A_s)$. Due to Proposition 3.23, $\text{var}(P) \subseteq \text{var}(T_P)$.

First, we claim that every variable u that appears in two or more atoms of T_P is part of a chordless path from z_0 to z_{k+1}. We first show a chordless path from u to z_{k+1}. Denote the last atom on T_P containing u by A_i. If A_i contains z_{k+1}, we have found the chordless path (u, z_{k+1}), and we are done. Otherwise, A_i is not the last atom on T_P. It is also not the first atom on T_P, as another atom contains u. Consider the subpath A_{i-1}, A_i, A_{i+1}. Since it is fully contracted, $A_i \cap A_{i+1} \not\subseteq A_{i-1} \cap A_{i+1}$. This means that there is a variable u_{+1} in A_i and in A_{i+1} that does not appear in A_{i-1}. Now consider the last atom containing u_{+1}, and continue with the same process iteratively until reaching z_{k+1}. This results in a chordless path $u, u_{+1}, \ldots, u_{+m} = z_{k+1}$ with $m \geq 1$. Do the same symmetrically to find a chordless path $z_0 = u_{-n}, \ldots, u_{-1}, u$ with $n \geq 1$. We now claim that the concatenation $P_u = (u_{-n}, \ldots, u_{-1}, u, u_{+1}, \ldots, u_{+m})$ is chordless. We prove that u_{-t} and $u_{+\ell}$ are not neighbors for all $1 \leq t \leq n$ and $1 \leq \ell \leq m$ by induction on $\ell + t$. Out of the atoms containing u, the variable u_{+1} only appears in

the last atom by construction: if $u_{+1} = z_{k+1}$ this is true since z_{k+1} appears only in one atom, and otherwise it is true because this is how we chose u_{+1}. Similarly u_{-t} only appears in the first. Since there are at least two atoms of the path containing u, we have that u_{-1} and u_{+1} are not neighbors, and this proves the induction base. Next, assume by way of contradiction that u_{-t} and $u_{+\ell}$ are neighbors. By using the induction assumption, we have that $u_{-t}, \ldots, u_{-1}, u, u_{+1}, \ldots, u_{+\ell}, u_{-t}$ is a chordless cycle of length four or more, contradicting the fact that Q is acyclic and therefore chordal. This proves that u is part of the chordless path P_u from z_0 to z_{k+1}.

Assume by contradiction that a variable $u \notin \text{free}(Q_2)$ appears in two distinct atoms of T_P. There is a chordless path from z_0 to z_{k+1} that contains u. Denote this path P_u. Take a subpath of P_u starting with the last variable on P_u before u that is in $\text{free}(Q_2)$, and ending with the first variable on P_u after u that is in $\text{free}(Q_2)$. Such variables exist on this path because $z_0, z_{k+1} \in \text{free}(Q_2)$. This subpath is a free-path in Q_2, and since Q_2 is free-path guarded, $u \in \text{free}(Q_1)$. Next consider two neighboring atoms on T_P that contain u. According to Proposition 3.24, there exists some z_i that appears in both atoms. Note that $i > 0$ and $i < k + 1$ since the path only contains one atom with z_0 and one atom with z_{k+1}. Since Q_1 is bypass guarded and $u \notin \text{free}(Q_2)$, it is not possible that there is both an atom with $\{z_{i-1}, z_i, u\}$ and an atom with $\{z_i, z_{i+1}, u\}$ in Q. Without loss of generality, assume there is no atom with $\{z_i, z_{i+1}, u\}$. Since the query is acyclic, this means that u and z_{i+1} are not neighbors (if the three variables z_i, z_{i+1}, u are pairwise neighbors in an acyclic graph, then necessarily they all appear in the same hyperedge). Then, there is a chordless path $(u, z_i, z_{i+1}, \ldots, z_{k+1})$. Since $u \in \text{free}(Q_1)$, it is a free-path. This contradicts the fact that Q_1 is free-path guarded since $u \notin \text{free}(Q_2)$.

It is left to show that there is an atom $R(\vec{v})$ in T_P such that $u, z_i \in \vec{v}$ for some $0 < i < k + 1$. Since u appears in two distinct atoms of T_P, and since T_P is a join-tree, u also appears in two adjacent atoms of T_P. According to Proposition 3.24, there exists $0 \leq i \leq k + 1$ such that z_i is in those atoms. This cannot be 0 or $k + 1$ because they appear only in one atom in T_P. □

We now get that the properties of free-path guarded and bypass guarded imply the existence of a free-connex union extension.

Lemma 3.26. *Let $Q = Q_1 \cup Q_2$ be a union of body-isomorphic acyclic CQs. If Q_1 and Q_2 are both free-path guarded and bypass guarded, then Q has a free-connex union extension.*

Proof. We describe how to iteratively build a union extension for each CQ. In every step, we take one free-path among the queries in Q and add a virtual atom in order to eliminate this free-path. More specifically, let $P = (z_0, \ldots, z_{k+1})$ be a free-path in Q_1. Take T_P according to Lemma 3.25, and denote by V_P the variables of P and all variables that appear in more than one atom of T_P. We add the atom $R(V_P)$ to

both Q_1 and Q_2 and obtain Q_1^+ and Q_2^+ respectively. We will show that repeatedly applying such steps eventually leads to a free-connex union extension.

First, we claim that Q_2 supplies V_P. It is guaranteed that $V_P \subseteq \text{free}(Q_2)$ since P is guarded and due to Lemma 3.25. We now show that Q_2 is acyclic V_P-connex. We know that T_P is a subtree of T that contains V_P, but it may contain additional variables, each appearing in only atom, so we need to modify the tree. For every vertex A_i in T_P, add another vertex with $A'_i = \text{var}(A_i) \cap V_P$ and an edge (A_i, A'_i). Then, for every edge (A_i, A_j) in T_P, replace it with the edge (A'_i, A'_j). The new graph is a tree since it is connected and the number of added vertices is equal to the number of added edges (hence, after the modification, the number of edges remains equal to the number of vertices minus one). We next show that the running intersection property is maintained. For every edge (A_i, A_j) removed, $\text{var}(A_i) \cap \text{var}(A_j) \subseteq V_P$, and so by definition of the new vertices, $\text{var}(A_i) \cap \text{var}(A_j) = \text{var}(A'_i) \cap \text{var}(A'_j)$. Given two nodes B and C of the join-tree, the path between them in the modified join-tree is similar to the path in the original join-tree, except it may contain the node (A'_i, A'_j) if it contained the node (A_i, A_j) before. Since the running intersection property holds in T, every node on the path between B and C contains $\text{var}(B) \cap \text{var}(C)$. Since $\text{var}(A_i) \cap \text{var}(A_j) = \text{var}(A'_i) \cap \text{var}(A'_j)$, the running intersection property also holds in the modified tree. As the variables of the subtree that consists of the new vertices are exactly V_P, this concludes the proof that Q_2 supplies V_P.

After the extension step we described, there are no free-paths that start in z_0 and end in z_{k+1} since these variables are now neighbors. If both of the CQs are now free-connex, then we are done. Otherwise, we use the extension recursively. We can apply the extension again as we show next that the UCQ $Q_1^+ \cup Q_2^+$ conforms to the conditions of this lemma. Note that after a free-path from z_0 to z_{k+1} is treated, and even after future extension, there will be no free-path from z_0 to z_{k+1} since they are now neighbors. Since there is a finite number of variable pairs, after a finite number of such steps, all pairs that have a free-path between them are resolved. At this point, we have an acyclic extension with no free-paths, so it is free-connex. It is left to prove that the conditions of this lemma are maintained after the extension steps as long as at least one of the extended CQs is not free-connex. We show that in the following three claims.

Claim 3.27. *Q_1^+ and Q_2^+ are body-isomorphic acyclic.*

We now prove Claim 3.27. We show a join-tree T^+ for the extension. Take the join-tree T according to Lemma 3.25, and add the vertex $R(V_P)$. Remove all edges in T_P and add an edge between $R(V_P)$ and every atom in T_P. This construction results in a connected graph with no cycles, and so it is a tree. We claim that the running intersection property is preserved. Let B and C be two nodes in T^+. We first handle the case that none of B and C are $R(V_P)$. We need to show that every node on the path between them in T^+ contains $B \cap C$. Since this property holds in T, every node on the path between them in T^+ that also appears on the path between them in T preserves this property. By

construction, the only new node that may appear on this path is $R(V_P)$. In this case, the two nodes before and after $R(V_P)$ on this path contain $B \cap C$. By definition, V_P contains all variables that appear in more that one atom in T_P, so $R(V_P)$ contains every variable that appears in more than one of its neighbors, and it also contains $B \cap C$. It is left to handle the second case. Assume without loss of generality that $C = R(V_P)$. We need to handle the path between $R(V_P)$ and B. Let $v \in V_P \cap B$. Since $v \in V_P$, there exists some node A_v in T_P that contains it. Consider the simple path in T between A_v and B, and let A'_v be the last node on this path which is in T_P. Due to the running intersection property, every node on this path contains $A_v \cap B$, and so it also contains v. The edge from V_P to A'_v and the simple path from A'_v to B is therefore a simple path in T^+ from V_P to B that contains v. This concludes the claim proof.

Claim 3.28. *Q_1^+ and Q_2^+ are free-path guarded.*

We now prove Claim 3.28. Since the original CQs are free-path guarded, every free-path in the extension that is also a free-path in the original query is guarded. According to our construction, the only atom that was added in the extension contains exactly V_P. Thus, a new free-path $(v_0, \ldots, v_{m+1})$ contains $v_j, v_{j+1} \in V_P \subseteq \text{free}(Q_2) = \text{free}(Q_2^+)$. In particular, since the variables in the center of a free-path are existential, Q_2^+ does not contain new free-paths. It is left to handle free-paths that appear in Q_1^+ but not in Q_1.

Let $P' = (v_0, \ldots, v_{m+1})$ be a free-path in Q_1^+ but not in Q_1, and let $v_j, v_{j+1} \in V_P$. We need to show that $v_i \in \text{free}(Q_2^+)$ for all $0 \leq i \leq m+1$. First note that $v_j, v_{j+1} \in V_P \subseteq \text{free}(Q_2) = \text{free}(Q_2^+)$. We next prove the same for v_0 and v_{m+1}.

We first claim that there is a path P_{mid} between v_j and v_{j+1} that goes only through existential variables in Q_1. We prove that every variable in V_P is either of the form z_i such that $0 < i < k+1$ (recall that these are the variables in the center of the eliminated free-path) or it has a neighbor of that form. Let $v \in V_P$. By definition of V_P, either there exists $0 \leq i \leq k+1$ such that $v = z_i$ or v appear in two atoms or more in T_P. In the first case, if $0 < i < k+1$, then it is of the required form. Otherwise, if $i = 0$, it has the neighbor z_1, and if $i = k+1$, it has the neighbor z_k. In the second case (if v appear in two atoms or more in T_P), according to Lemma 3.25, v has a neighbor of the required form. Since $v_j, v_{j+1} \in V_P$, this proves that each of v_j and v_{j+1} is either of the form z_i such that $0 < i < k+1$ or it has a neighbor of that form. Since all z_i such that $0 < i < k+1$ are connected through P, there is a path P_{mid} between v_j and v_{j+1} that goes only through existential variables in Q_1.

Since P' is chordless, no variable in P' other than v_j and v_{j+1} is in V_P. One consequence of this is that $P_s = (v_0, \ldots, v_j)$ and $P_t = (v_{j+1}, \ldots, v_{m+1})$ are also paths in Q_1. Another consequence is that v_0 and v_{m+1} are not both in V_P. Since P' is chordless and $m \geq 1$, we have that v_0 and v_{m+1} are not neighbors in Q^+. Since they are not both on the new atom, v_0 and v_{m+1} are not neighbors in Q either. By concatenating P_s, P_{mid} and P_t, we obtain a path in Q from v_0 to v_{m+1} that goes only through existential variables in Q_1. Take a simple chordless path P_ℓ contained in it. Since v_0 and v_{m+1} are

not neighbors, the path P_ℓ is of length two at least, and so it is a free-path in Q_1. Since Q_1 is free-path guarded, we conclude that $v_0, v_{m+1} \in \text{free}(Q_2)$.

So far we have that $v_0, v_j, v_{j+1}, v_{m+1} \in \text{free}(Q_2)$. Assume by contradiction that there exists $0 < i < j$ such that $v_i \notin \text{free}(Q_2)$, and consider the subpath of P_s beginning with the last variable before v_i on P_s that is in $\text{free}(Q_2)$ and ending with the first after v_i on P_s that is in $\text{free}(Q_2)$. This is a free-path in Q_2 containing v_i. We know that $v_i \notin \text{free}(Q_1)$ as it is in the center of the free-path P', but this contradicts the fact that Q_2 is free-path guarded. In the same way, we can show that $v_i \in \text{free}(Q_2)$ for all $j+1 < i < m+1$. Overall we have seen that $v_i \in \text{free}(Q_2) = \text{free}(Q_2^+)$ for all $0 \leq i \leq m+1$ as required. This concludes the claim proof.

Claim 3.29. *Q_1^+ and Q_2^+ are bypass guarded.*

We now prove Claim 3.29. First let $P' = (t_0, \ldots, t_{m+1})$ be a free-path in Q_2^+, and assume by contradiction that there exists some $u \notin \text{free}(Q_1^+)$ that appears in two subsequent P'-atoms. This means that there exists i such that Q_2^+ has an atom containing $\{t_{i-1}, t_i, u\}$ and an atom containing $\{t_i, t_{i+1}, u\}$. As explained in the previous claim, Q_2^+ has no new free-paths, so P' is a free-path in Q_2 as well. Since Q_2 is bypass guarded, u does not appear in two subsequent P'-atoms in Q_2, so one of these atoms is new in Q_2^+. Assume without loss of generality that it is $\{t_i, t_{i+1}, u\}$. Then, $\{t_i, t_{i+1}, u\} \subseteq V_P \subseteq \text{free}(Q_2)$. This contradicts the fact that $t_i \notin \text{free}(Q_2)$ since P' is a free-path.

Now let $P' = (t_0, \ldots, t_{m+1})$ be a free-path in Q_1^+. Assume by contradiction that there exists some $u \notin \text{free}(Q_2^+)$, that appears in two subsequent P'-atoms. This means that Q^+ has an atom containing $\{t_{i-1}, t_i, u\}$ and an atom containing $\{t_i, t_{i+1}, u\}$. Since P' is chordless, t_{i-1} and t_{i+1} are not neighbors, and so (t_{i-1}, u, t_{i+1}) is a chordless path as well. According to Claim 3.28, $\text{var}(P') \subseteq \text{free}(Q_2^+)$, and so (t_{i-1}, u, t_{i+1}) is a free-path in Q_2^+. According to Claim 3.28 again, $\{t_{i-1}, u, t_{i+1}\} \subseteq \text{free}(Q_1^+)$. Since the only free variables on a free-path are at its ends, this means that P' is of length two, and $i = m = 1$. Since $u \notin \text{free}(Q_2)$ and $V_P \subseteq \text{free}(Q_2)$, we have that the atoms containing $\{t_0, t_1, u\}$ and $\{t_1, t_2, u\}$ are not the added atom, and so they appear also in Q_2 and in Q_1. This means that P' is a free-path in Q_1, and $u \notin \text{free}(Q_2)$ appears in two subsequent atoms of P' in Q_1. Thus, Q_1 is not bypass guarded, in contradiction to the conditions of this lemma. This concludes the claim proof.

This proves that the lemma can be applied iteratively. □

Since Lemma 3.21, Lemma 3.22 and Lemma 3.26 cover all cases of a union of two self-join-free body-isomorphic acyclic CQs, we have a dichotomy that characterizes the fragment of UCQs we discuss.

Theorem 3.30. *Let $Q = Q_1 \cup Q_2$ be a non-redundant union of self-join-free body-isomorphic CQs.*

- *If Q_1 and Q_2 are acyclic, free-path guarded and bypass guarded, then Q has a free-connex union extension and* $\text{ENUM}\langle Q \rangle \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$.

- *Otherwise, Q does not have a free-connex union extension and* $\textsc{Enum}\langle Q\rangle$ *is not in* $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$*, assuming* Hyperclique, BMM *and* 4-clique.

Proof. Since the CQs are body-isomorphic, they are either both acyclic or both cyclic. First assume that the CQs are acyclic. By Lemma 3.26, if Q_1 and Q_2 are both free-path guarded and bypass guarded, then Q has a free-connex union extension, and $\textsc{Enum}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$ by Theorem 3.9. Otherwise, either one of Q_1 and Q_2 is not free-path guarded, or they both are but one of them is not bypass guarded. In these cases, by Lemma 3.21 and Lemma 3.22, $\textsc{Enum}\langle Q\rangle \notin \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$ assuming BMM and 4-clique. By Theorem 3.15, if the CQs are cyclic, $\textsc{Enum}\langle Q\rangle$ is not in $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$ assuming Hyperclique. In all cases where $\textsc{Enum}\langle Q\rangle \notin \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$, we know that Q does not have a free-connex union extension by Theorem 3.9. □

By combining Theorem 3.30 with Theorem 3.15, we have the following dichotomy for the case of unions containing exactly two difficult CQs.

Theorem 3.31. *Let $Q = Q_1 \cup Q_2$ be a non-redundant union of difficult CQs.*

- *If Q has a free-connex union extension, then* $\textsc{Enum}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$*.*
- *Otherwise, Q does not have a free-connex union extension and* $\textsc{Enum}\langle Q\rangle$ *is not in* $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$*, assuming* BMM*,* Hyperclique *and* 4-clique*.*

Proof. If Q has a free-connex union extension, Theorem 3.1 proves that it is tractable. If Q_1 and Q_2 are acyclic and body-isomorphic but Q does not have a free-connex union extension, then Theorem 3.30 proves that Q is intractable. Otherwise, Q is a union of difficult CQs that does not contain body-isomorphic acyclic CQs, and Theorem 3.15 proves that it is intractable. □

3.3 The Unbalanced Triangle Detection Hypothesis

In this section, we discuss the problem of evaluating unions of two CQs that are not covered by the previous sections. These are UCQs that contain a tractable CQ and a difficult CQ that are not body-isomorphic. We define the Unbalanced Triangle Detection (UTD) hypothesis, and discuss the connections between the two problems. UTD is a simple hypothesis on graphs, and we show that in some cases, it exactly captures the hardness of UCQs. We also show that, assuming this hypothesis alone, all self-join-free unions of two binary CQs are intractable if they do not admit a free-connex union extension.

We start with discussing a similar assumption. The unbalanced triangle listing hypothesis states that listing any number of triangles of an unbalanced tripartite graph requires super-linear time in terms of input and output size.

Definition 3.32 (Unbalanced Triangle Listing)**.** We denote by UTL the following hypothesis: For any constant $\alpha \in (0, 1]$, listing t triangles in a tripartite graph with $|V_1| = |V_2| = n^\alpha$ and $|V_3| = n$ cannot be done in time $O(n^{1+\alpha} + t)$.

The 3SUM conjecture [GO95, KPP16] is a well-known hypothesis that states that, given a set A of integers, the time required to decide whether there is a triple $(a, b, c) \in A^3$ of distinct elements such that $a + b = c$ is $\Omega(n^{2-o(1)})$. UTL is a consequence of 3SUM.

Proposition 3.33. *If* 3SUM *holds, then* UTL *holds too.*

Proof. Fix a constant $\alpha \in (0, 1]$ and set $\delta = \gamma = \min\{\frac{\alpha}{3}, \frac{1}{6}\}$. Starting from a 3SUM instance of size n, a construction by Kopelowitz, Pettie and Porat [KPP16, Proof of Theorem 1.8] generates an unbalanced triangle listing instance with the following parameters: $|V_1| = |V_2| = n^{\frac{1+\delta+\gamma}{2}} = n^{\frac{1}{2}+\delta}$, $|V_3| = n^{1+\delta-\gamma} = n$ and the number of triangles is at most $n^{2-\delta}$. Listing the triangles over this construction solves the 3SUM instance, so we assume this cannot be done in subquadratic time.

As a first step, we split V_1 and V_2 each into n^δ sets of size $n^{\frac{1}{2}}$. This yields $n^{2\delta}$ subproblems, each with $|V_1| = |V_2| = n^{\frac{1}{2}}$ and $|V_3| = n$, and their total number of triangles is at most $n^{2-\delta}$. Listing the triangles of all subproblems yields the same result as listing the triangles of the original construction. Assume by contradiction that it is possible to list t triangles in a tripartite graph with $|V_1| = |V_2| = |V_3|^\alpha$ in time $O(|V_3|^{1+\alpha} + t)$.

In case $\alpha = \frac{1}{2}$ we are done now. Indeed, each subproblem has $|V_1| = |V_2| = |V_3|^\alpha$, and if each subproblem could be solved in time $O(n^{1+\alpha} + t)$ then the total running time to solve all subproblems would be $O(n^{2\delta+1+\alpha} + n^{2-\delta})$ since there are $n^{2\delta}$ subproblems and their total number of triangles is at most $n^{2-\delta}$. We can simplify this time bound to $O(n^{\frac{3}{2}+2\delta} + n^{2-\delta}) = O(n^{2-\frac{1}{6}})$ since $\alpha = \frac{1}{2}$ and $\delta = \frac{\alpha}{3} = \frac{1}{6}$. This running time is subquadratic, contradicting 3SUM.

In case $\alpha < \frac{1}{2}$, we further split V_1 and V_2 each into $n^{\frac{1}{2}-\alpha}$ sets of size n^α. Together with the first splitting step (where we split into $n^{2\delta}$ subproblems), this yields $n^{2\delta+1-2a}$ subproblems, each with $|V_1| = |V_2| = n^\alpha$ and $|V_3| = n$, and their total number of triangles is at most $n^{2-\delta}$. If each subproblem could be solved in time $O(n^{1+\alpha} + t)$, then all subproblems in total could be solved in time $O(n^{2\delta+1-2\alpha} n^{1+\alpha} + n^{2-\delta})$ since there are $n^{2\delta+1-2\alpha}$ subproblems and their total number of triangles is at most $n^{2-\delta}$. We can simplify this time bound to $O(n^{2-\alpha+2\delta} + n^{2-\delta}) = O(n^{2-\frac{\alpha}{3}})$ since $\delta = \frac{\alpha}{3}$. This running time is subquadratic for any fixed constant $\alpha \in (0, \frac{1}{2})$, contradicting 3SUM.

In case $\alpha > \frac{1}{2}$, we split V_3 into $n^{1-\frac{1}{2\alpha}}$ sets of size $n^{\frac{1}{2\alpha}}$. Together with the first splitting step (where we split into $n^{2\delta}$ subproblems), this yields $n^{2\delta+1-\frac{1}{2\alpha}}$ subproblems, each with $|V_1| = |V_2| = n^{\frac{1}{2}}$ and $|V_3| = n^{\frac{1}{2\alpha}}$, so $|V_1| = |V_2| = |V_3|^\alpha$. If each subproblem could be solved in time $O(|V_3|^{1+\alpha} + t)$, then all subproblems in total could be solved in time $O(n^{2\delta+1-\frac{1}{2\alpha}} |V_3|^{1+\alpha} + n^{2-\delta})$ since there are $n^{2\delta+1-\frac{1}{2\alpha}}$ subproblems and their total number of triangles is at most $n^{2-\delta}$. Plugging in $|V_3| = n^{\frac{1}{2\alpha}}$ yields time $O(n^{2\delta+1-\frac{1}{2\alpha}+\frac{1+\alpha}{2\alpha}} + n^{2-\delta}) = O(n^{\frac{3}{2}+2\delta} + n^{2-\delta}) = O(n^{2-\frac{1}{6}})$ since $\delta = \frac{1}{6}$, again contradicting 3SUM. □

UTL (or 3SUM) can be used to show that some UCQs, for which we did not have a prior characterization, are intractable. In the following, we say that a CQ Q_2 *provides*

a set V_1 of variables to Q_1 if there is a body homomorphism h from Q_2 to Q_1 and Q_1 supplies V_2 such that $h(V_2) = V_1$. Note that if Q_2 is free-connex, this requirement boils down to having $V_2 \subseteq \text{free}(Q_2)$ and $h(V_2) = V_1$.

Example 3.34. [CK19b, Example 37] Let $Q = Q_1 \cup Q_2$ with

$$Q_1(x, z, y, v) \leftarrow R_1(x, z, v), R_2(z, y, v), R_3(y, x, v) \text{ and}$$
$$Q_2(x, z, y, v) \leftarrow R_1(x, z, v), R_2(y, t_1, v), R_3(t_2, x, v).$$

The CQ Q_2 is free-connex. Hence, it is not difficult, and Q is not covered by our results in Theorem 3.31. The only difficult structure in Q_1 is the cycle x, y, z. If all cycle variables were provided by Q_2, we would be able to eliminate the cycle by adding a virtual atom with its variables, and extend the CQ to a tractable form. However, y is not provided. The existing approach to show the difficulty of a CQ with a cycle is to encode the triangle finding problem to this cycle. In our case, this encoding may result in n^3 answers to Q_2. Thus, if the input graph has triangles, we are not guaranteed to find one in $O(n^2)$ time by evaluating the union efficiently. By using *unbalanced* tripartite graphs, we make use of the fact that y is not provided to show hardness. We assign the heavy node set to y, and we assign to x and z node sets of size n^α. Thus, when Q_1 finds the triangles in the graph, Q_2 has at most $n^{3\alpha}$ answers. If $\textsc{Enum}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$, we can compute all answers over this construction in $O(n^{1+\alpha} + t)$ time, contradicting UTL. □

In Example 3.34, we are able to use a triangle listing assumption because the variables of the cycle in Q_1 are free. However, there exist similar cases where some of these variables are projected. In these cases, we can use a similar argument, but we must use a detection assumption rather than a listing one.

Definition 3.35 (Unbalanced Triangle Detection)**.** We denote by UTD the following hypothesis: For any constant $\alpha \in (0, 1]$, determining whether there exists a triangle in a tripartite graph with $|V_1| = |V_2| = n^\alpha$ and $|V_3| = n$ cannot be done in time $O(n^{1+\alpha})$.

The UTD hypothesis can be used not only when a CQ in the union contains a cycle, but also when it contains a free-path. The following example is also not covered by the previous sections since Q_2 is tractable and the two CQs in the union are not body-isomorphic.

Example 3.36. [CK19b, Example 29] Let $Q = Q_1 \cup Q_2$ with

$$Q_1(x, y, w) \leftarrow R_1(x, z), R_2(z, y), R_3(y, w) \text{ and}$$
$$Q_2(x, y, w) \leftarrow R_1(x, t_1), R_2(t_2, y), R_3(w, t_3).$$

The only difficult structure in Q_1 is the free-path x, z, y. If all free-path variables were provided by Q_2, we would be able to eliminate the free-path by adding a virtual atom with its variables and extend the CQ to a tractable form. However, here, z is not provided. The existing approach to show the difficulty of a CQ with a free-path is to encode the Boolean matrix multiplication problem to this free-path. In our case, such

an encoding may result in n^3 answers to Q_2. This means that we are not guaranteed to find all non-zero entries in the multiplication result in $O(n^2)$ time by evaluating the union efficiently. By using *unbalanced* tripartite graphs, we can rely on the fact that z is not provided to show hardness. We assign the heavy node set to z, and we assign to x and y node sets of size n^α. The variable w is assigned some constant $\perp$. Under this construction, Q_1 returns tuples $(a, b, \perp)$ such that there exists a node c which is a neighbor to both a and b. For every such answer, we check whether a and b are neighbors. If they are, a cycle exists. When Q_1 finds all candidates for triangles in the graph, Q_1 and Q_2 have at most $n^{3\alpha}$ answers each. If $\textsc{Enum}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$, we can compute all answers over such a construction in $O(n^{1+\alpha})$ time, contradicting UTD. □

Example 3.36 demonstrates that if we assume UTD, we can prove the hardness of previously unclassified UCQs. However, unlike the similar UTL, we are not aware of a well-established complexity hypothesis that implies UTD. We offer next another reason to use UTD to reason about the hardness of UCQs that do not admit a free-connex union extension: in some cases, UTD exactly captures the hardness of such UCQs.

3.3.1 UCQ Hardness Implies UTD

In this section, we show a tight connection between unbalanced triangle detection and the evaluation of UCQs: in some cases UTD exactly captures the hardness of queries for which the currently known efficient algorithms cannot be applied. In particular, if free-connex union extensions capture all tractable UCQs, then UTD necessarily holds. We prove the following theorem.

Theorem 3.37. *There exists a family of UCQs with no free-connex union extensions such that* UTD *holds iff no query of the family is in* $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$.

To prove Theorem 3.37, we need to be more specific about the values of α for which we assume that UTD holds. For this reason, we define the following hypothesis.

Definition 3.38. We denote by α-UTD the following hypothesis: Determining whether there exists a triangle in a tripartite graph with $|V_1| = |V_2| = n^\alpha$ and $|V_3| = n$ cannot be done in time $O(n^{1+\alpha})$.

Then, the UTD hypothesis is that α-UTD holds for every constant $\alpha \in (0, 1]$. We next show that α-UTD is "monotone" in the sense that it implies β-UTD for larger values of β.

Proposition 3.39. *If* α-UTD *holds, then* β-UTD *holds for all* $\beta \geq \alpha$.

Proof. We show a self-reduction that splits the set V_3. Let $0 < \alpha < \beta \leq 1$, and assume that determining whether there exists a triangle in a tripartite graph with

$|V_1| = |V_2| = n^\beta$ and $|V_3| = n$ can be done in time $O(n^{1+\beta})$. Let $G = (V_1 \cup V_2 \cup V_3, E)$ be a tripartite graph with $|V_1| = |V_2| = n^\alpha$ and $|V_3| = n$. Split V_3 into $n^{1-\frac{\alpha}{\beta}}$ subsets of size $n^{\frac{\alpha}{\beta}}$. This splits G into $n^{1-\frac{\alpha}{\beta}}$ subgraphs $G_1, \ldots, G_t$. Each subgraph is tripartite with parts V_1, V_2, V_3' with $|V_1| = |V_2| = n^\alpha = |V_3'|^\beta$. Therefore, the assumed algorithm determines whether G_i has a triangle in time $O(|V_3'|^{1+\beta})$. Running this algorithm on each graph G_i takes total time $O(n^{1-\frac{\alpha}{\beta}}|V_3'|^{1+\beta}) = O(n^{1-\frac{\alpha}{\beta}+\frac{\alpha}{\beta}+\alpha}) = O(n^{1+\alpha})$. Thus, we can solve the given α-UTD instance G in time $O(n^{1+\alpha})$. □

We prove Theorem 3.37 with the following example.

Example 3.40. Let $Q_{[c]}$ be the union of the following CQs, where atoms that contain i are replaced with c atoms with all integers $1 \leq i \leq c$.

$$Q_1(v_1 \ldots, v_{2c}) \leftarrow R_1(x, y), R_2(y, z), R_3(x, z), R_4(v_1 \ldots, v_{2c}), R_{x,i}(x), R_{y,i}(y)$$
$$Q_2(v_1 \ldots, v_{2c}) \leftarrow R_{x,i}(v_i), R_{y,i}(v_{c+i})$$
$$Q_3(v_1 \ldots, v_{2c}) \leftarrow R_1(v_1, t_1), R_2(t_2, v_2), R_4(t_3, t_4, v_3, \ldots, v_{2c})$$
$$Q_4(v_1 \ldots, v_{2c}) \leftarrow R_1(t_1, v_1), R_2(t_2, v_2), R_4(t_3, t_4, v_3, \ldots, v_{2c})$$

Note that $Q_{[c]}$ does not have a free-connex union extension. In particular, Q_1 contains a cycle x, y, z. Since y is not provided by any of the other CQs in the union, any union extension of Q_1 preserves this cycle.

Claim 3.41. *If* UTD *does not hold, then* $Q_{[c]} \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$ *for* c *large enough.*

Proof. If UTD does not hold, then β-UTD does not hold for some $\beta \in (0, 1)$. According to Proposition 3.39, this also means that for all $\alpha < \beta$, we have that α-UTD does not hold. That is, for all $\alpha \in (0, \beta)$, determining whether there exists a triangle in a tripartite graph with $|V_1| = |V_2| = n^\alpha$ and $|V_3| = n$ can be done in time $O(n^{1+\alpha})$. Let $c \geq \frac{1}{\beta} + 1$. We show how, given a database I, we can enumerate $Q_{[c]}(I)$ with linear preprocessing and constant delay.

We can first compute $Q_2(I)$, $Q_3(I)$ and $Q_4(I)$ with linear preprocessing and constant delay each, as these CQs are free-connex. In the following, we show how to find $Q_1(I)$ with constant delay after $O(|I| + |Q_2(I)| + |Q_3(I)| + |Q_4(I)|)$ preprocessing time. This means that by interleaving the computation of the preprocessing of Q_1 with the evaluation of the other CQs, we can enumerate the answers to Q_1 with constant delay directly after the end of the enumeration of the other CQs. According to the Cheater's Lemma (Lemma 3.4), since the delay between answers is constant except for at most three times where the delay is linear, and since there are at most four duplicates per answer, the algorithm we present here can be modified to work with linear preprocessing time and constant delay with no duplicates.

Note that if one of the relations of Q are empty, than $Q_1(I) = \emptyset$, and we can finish the evaluation of $Q_1(I)$ immediately. In the following we assume that no relation is empty. Consider the Boolean query $Q_1'() \leftarrow R_1(x, y), R_2(y, z), R_3(x, z), R_{x,i}(x), R_{y,i}(y)$. Note that $Q_1(I)$ is exactly R_4^I if Q_1' is evaluated to true, and it is empty otherwise. To

evaluate Q_1', we can first filter the relations R_1, R_2 and R_3 by performing semi-joins with $R_{x,i}$ and $R_{y,i}$ for all i. Formally, we set

$$E_{12} = \{(a,b) \mid R_1(a,b) \wedge \forall i \in [c] : R_{x,i}(a) \wedge R_{y,i}(b)\},$$
$$E_{23} = \{(b,c) \mid R_2(b,c) \wedge \forall i \in [c] : R_{y,i}(b)\}, \text{ and}$$
$$E_{13} = \{(a,c) \mid R_3(a,c) \wedge \forall i \in [c] : R_{x,i}(a)\}.$$

Now, it is left to evaluate $Q_1''() \leftarrow E_{12}(x,y), E_{23}(y,z), E_{13}(x,z)$ as $Q_1'() = Q_1''()$. Denote $V_1 = \{a \mid \exists b : E_{1,2}(a,b)\}$, $V_2 = \{b \mid \exists a : E_{1,2}(a,b)\}$, and $V_3 = \{c \mid \exists b : E_{2,3}(b,c)\}$. If $|V_3| \leq \max\{|V_1|, |V_2|\}^{c-1}$, then we evaluate Q_1'' in $O(|V_1||V_2||V_3|)$ time by checking all possible assignments to x, y and z. Since $|V_1||V_2||V_3| \leq (|V_1||V_2|)^c \leq |Q_2(I)|$, this takes $O(|Q_2(I)|)$ time. The second case is that $|V_3| > \max\{|V_1|, |V_2|\}^{c-1}$. We set $n = |V_3|$ and $\alpha = \log_n \max\{|V_1|, |V_2|\}$; note that $\alpha < \frac{1}{c-1} \leq \beta$. If we can solve triangle detection in time $O(n^{1+\alpha})$, then we can answer Q_1'' within this time. Note that $|Q_3(I)| \geq |V_1||V_3|$ and $|Q_4(I)| \geq |V_2||V_3|$. Therefore, $\max\{|Q_3(I)|, |Q_4(I)|\} \geq |V_3| \max\{|V_1|, |V_2|\} = n^{1+\alpha}$. Therefore, this check takes $O(\max\{|Q_3(I)|, |Q_4(I)|\})$ time. If Q_1'' evaluates to false, Q_4 returns no answers and we are done; otherwise, we output R_4^I in constant delay. □

Note that as part of the claim proof, we showed that $Q_{[c]}$ is tractable in case $|V_3| \leq \max\{|V_1|, |V_2|\}^{c-1}$ without relying on any assumptions. This demonstrates that z must have a large domain for this query to be intractable. That is, $Q_{[c]}$ is intractable assuming UTD only when we can make no additional assumptions on the instance; if the domain of z is limited, the query may become tractable. This also shows that in any construction that proves a lower bound for $Q_{[c]}$, we must assign z with a larger domain than that of the other variables. We do this in the following claim.

Claim 3.42. *If* UTD *holds, then* $Q_{[c]} \notin \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$ *for all* c.

Proof. Assume by contradiction that $Q_{[c]} \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$ for some c. We start with a tripartite graph G with V_1,V_2,V_3, E_{12}, E_{23} and E_{13}, where $|V_1| = |V_2| = n^\alpha$ and $|V_3| = n$ for some $n \in \mathbb{N}$ and $\alpha \leq \frac{1}{2c-1}$. We construct a database instance I as follows: we assign $R_1 = E_{12}$, $R_2 = E_{23}$, $R_3 = E_{13}$, and $R_4 = \{(\perp, \ldots, \perp)\}$. For all $i \in [c]$, we assign $R_{x,i} = V_1$ and $R_{y,i} = V_2$. The answers $Q_1(I)$ consist of $(\perp, \ldots, \perp)$ if there is a cycle in G and no answers otherwise. As for the other CQs, $|Q_2(I)| = (|V_1||V_2|)^c$, $|Q_3(I)| = |V_1||V_3|$, and $|Q_4(I)| = |V_2||V_3|$. The tuple $(\perp, \ldots, \perp)$ is not an answer to CQs other than Q_1, so $(\perp, \ldots, \perp) \in Q_{[c]}(I)$ if and only if there is a triangle in G. If $Q_{[c]} \in \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$, then we can compute all of $Q_{[c]}(I)$ in time $O((|V_1||V_2|)^c + |V_1||V_3| + |V_2||V_3|)$ and determine the existence of a triangle in G within this time. Since $(|V_1||V_2|)^c + |V_1||V_3| + |V_2||V_3| = n^{2\alpha c} + 2n^{1+\alpha} = O(n^{1+\alpha})$, this contradicts UTD. □

In this section we showed that if free-connex union extensions capture all tractable UCQs, then UTD holds. The next section inspects the opposite direction: assuming UTD, we prove the hardness of UCQs that do not admit free-connex union extensions.

3.3.2 UTD Implies UCQ Hardness

In this section, we prove the hardness of UCQs that do not admit a free-connex union extension assuming UTD. We consider unions of a tractable CQ and an difficult binary CQ. Then, we show how UTD can be used instead of the hypotheses previously used to show the hardness of UCQs.

The Reduction

The following lemma identifies cases in which we can perform a reduction from unbalanced triangle detection to UCQ evaluation. The reduction requires identifying variable sets in the UCQ that conform to certain conditions. We encode the tripartite graph in the relations of the query by assigning variables from the same set with the same values. The first three conditions ensure that we can construct the relations of Q_1 in a way that it detect triangles in the graph. The first condition requires that no atom contains variables of all sets, which restricts the size of the relations and allows for efficient construction. The second condition requires that each set is connected, which ensures that in every answer, variables from the same set are assigned the same values. The third condition ensures that the atoms can encode all three edge sets. The fourth condition restricts the free variables of the other CQ in the union, which ensures that it does not have too many answers, and the enumeration of the answers of the entire union does not take too long. Given a function $h : X \to Y$ and a set $S \subseteq Y$, we denote $h^{-1}(S) = \{x \in X \mid h(x) \in S\}$.

Lemma 3.43. *Let $Q = Q_1 \cup Q_2$ non-redundant where Q_1 is self-join-free and there exist non-empty and disjoint sets $X_1, ..., X_\ell \subseteq \text{var}(Q_1)$ with $\ell \geq 3$ such that:*

1. *For every atom $R(V)$ in Q_1, there exists X_i s.t. $V \cap X_i = \emptyset$.*
2. *$\mathcal{H}(Q_1)[X_i]$ is connected for all i.*
3. *Define $\text{connectors}(Q_1) = \{V \mid R(V) \in \text{atoms}(Q_1)\}$; if there exists X_i such that $\text{free}(Q_1) \cap X_i = \emptyset$, also add $\text{free}(Q_1)$ to $\text{connectors}(Q_1)$.*
   For every $S \in \{\{1,2\}, \{1,3,\ldots,\ell\}, \{2,3,\ldots,\ell\}\}$, there exists $V \in \text{connectors}(Q_1)$ such that $V \cap X_i \neq \emptyset$ for all $i \in S$.
4. *For every body-homomorphism h from Q_2 to Q_1, if $\text{free}(Q_2) \cap h^{-1}(X_\ell) \neq \emptyset$, then $|\text{free}(Q_2) \cap h^{-1}(X_\ell)| = 1$ and $|\text{free}(Q_2) \cap h^{-1}(\bigcup_{1 \leq i \leq \ell-1} X_i)| \leq \ell - 2$.*

Then, $Q \notin \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$ assuming UTD.

Note that the second condition trivially holds when $|X_i| = 1$.

Proof. We set $\alpha = \max\{|\text{free}(Q_2)|, \ell - 2\}^{-1}$. Assume we are given a tripartite graph with node sets V_1,V_2,V_3 and edges sets $E_{1,2}$, $E_{2,3}$, $E_{1,3}$ where $|V_1| = |V_2| = n^\alpha$ and $|V_3| = n$. We set $U_1 = V_1$, $U_2 = V_2$, and we encode the vertices of V_3 as $U_3 \times \cdots \times U_\ell$ such that $|U_3| = \ldots = |U_{\ell-1}| = n^\alpha$ and $|U_\ell| = n^{1-(\ell-3)\alpha}$.

We now construct a database instance. We leave every relation that does not appear in Q_1 empty. We next discuss the atoms of Q_1. Denote by $\mathcal{R}_{1,2}$ the atoms

that contain a variable of X_1 and a variable of X_2; denote by $\mathcal{R}_{1,3}$ the atoms that contain at least one variable of each X_i for $i \in \{1, 3, \ldots, \ell\}$; and similarly for $\mathcal{R}_{2,3}$ and $\{2, 3, \ldots, \ell\}$. According to condition 1, these three sets are disjoint. We encode the edge sets $E_{1,2}, E_{1,3}, E_{2,3}$ to these relations respectively. Specifically, given an atom $R(\vec{v})$ in $\mathcal{R}_{1,3}$, for every edge $(v_1, v_3) \in E_{1,3}$, add a tuple $\tau(\vec{v})$ to R as follows: denote by $u_3, \ldots, u_\ell$ the representation of v_3 and set $u_1 = v_1$; the mapping τ replaces every variable of the set X_i with the value u_i; every variable that does not appear in such a set X_i is replaced with the constant $\perp$. The construction of relations in $\mathcal{R}_{2,3}$ follows along the same lines. For atoms in $\mathcal{R}_{1,2}$ we have a similar construction, except if they contain a variable of X_i for $i > 2$, we duplicate each edge and insert it with all possible values in U_i. If there are variables of several such sets, we apply all combinations of possible values. For example, if Q_1 contains the atom $R_1(v_1, v_2, v_4, v_5, t)$ with $v_i \in X_i$ and $t \notin X_i$ for all i, then the relation R_1 will be assigned $E_{1,2} \times U_4 \times U_5 \times \{\perp\}$. Similarly, for atoms that do not belong to these sets, we assign variables of X_i with all values of U_i and other variables with $\perp$. For example, the atom $R_2(v_4, v_4', v_5, t)$ with $v_i, v_i' \in X_i$ and $t \notin X_i$ for all i, will result in $R_2 = \{(u_4, u_4, u_5, \perp) \mid u_4 \in U_4 \wedge u_5 \in U_5\}$. Note that whenever two variables from the same set X_i appear together in the same atom, we always assign both with the same value. Note also that each relation is defined only once since Q_1 is self-join-free and since $\mathcal{R}_{1,3}$, $\mathcal{R}_{2,3}$ and $\mathcal{R}_{1,2}$ are disjoint.

We first claim that the answers to Q_1 detect triangles in the graph. Since any given tuple in our construction assigns different variables of the same set X_i with the same value, if two variables of the same set appear together in an atom, then they are mapped to the same value in any such answer. Condition 2 asserts that all variables in a set are connected through atoms in Q_1, and so, for every set X_i, all variables of the set are mapped to the same value in any answer to Q_1. Our construction also ensures that, if Q_1 contains an atom with a variable of X_1 and a variable of X_2, then any answer to Q_1 will map X_1 and X_2 to vertices that are neighbors in $E_{1,2}$. The same holds for $X_1, X_3, \ldots, X_\ell$ and $E_{1,3}$, and it also holds for $X_2, X_3, \ldots, X_\ell$ and $E_{2,3}$. Thus, if we do not use $\text{free}(Q_1)$ as a connector, condition 3 ensures that the answers are filtered by at least one atom that corresponds to each edge set, and so answers correspond to triangles. That is, Q has an answer if and only if the graph has a triangle. If we do use $\text{free}(Q_1)$ as a connector, some edge is not verified. This means that the answers to Q_1 are candidates for triangles, and we need to check every answer for the missing edge to determine if it corresponds to a triangle. In this case, by the definition of the connectors set, there exists i such that $\text{free}(Q_1) \cap X_i = \emptyset$. If $i = \ell$, the number of answers to Q_1 is at most $n^{1-(\ell-3)\alpha}(n^\alpha)^{\ell-2} = n^{1+\alpha}$. If $i < \ell$, it is at most $(n^\alpha)^{\ell-1} \leq n^{1+\alpha}$. Therefore, performing a check that takes constant time for each answer takes $O(n^{1+\alpha})$ time overall.

We now show that the answers to Q_2 do not interfere with detecting the triangles efficiently. First note that we can distinguish the answers of Q_1 from those of Q_2. Since we assume that Q is non-redundant, we have that Q_1 is not contained in Q_2, and so there is no homomorphism from Q_2 to Q_1. Thus, we can apply the construction from

Lemma 3.11 (which assigns different domains to different variables) to distinguish the answers of the two CQs, and we can ignore the answers of Q_2. We show next that Q_2 does not have too many answers. If v is a free variable in Q_2, it also appears in the body of Q_2 in some atom $R(\vec{v})$ with $\vec{v}[i] = v$ for some i. According to Lemma 3.11, if Q_2 has answers, then there exists a body-homomorphism h from Q_2 to Q_1. Thus, $R(h(\vec{v}))$ appears in the body of Q_1. If our construction assigns $h(v)$ with c different values across the different tuples in R, then there are at most c different values to which v can map across the different answers of Q_2. Thus, if no free variable of Q_2 maps via a body-homomorphism to a variable of X_ℓ, then the domain of any free variable in Q_2 is at most of size n^α. Since there are at most $\frac{1}{\alpha}$ free variables, Q_2 has at most n answers. Otherwise, due to condition 4, exactly one free variables of Q_2 map to a variable of X_ℓ and at most $\ell - 2$ free variables of Q_2 map to variables of the other sets. In this case, the number of answers to Q_2 is at most $n^{1-(\ell-3)\alpha}(n^\alpha)^{\ell-2} = n^{1+\alpha}$.

If $Q \in \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$, this construction detects triangles in the given graph in time $O(n^{1+\alpha})$, in contradiction to UTD. □

We want to use this reduction to show the hardness of non-free connex UCQs that contain a tractable CQ and a difficult CQ. As the difficult CQ is self-join-free, we first notice that there is at most one body-homomorphism mapping to a self-join-free CQ.

Lemma 3.44. *Let $Q = Q_1 \cup Q_2$ where Q_1 is self-join-free. There is at most one body homomorphism from Q_2 to Q_1.*

Proof. Let h_1 and h_2 be body-homomorphisms from Q_2 to Q_1. If $h_1 \neq h_2$, there exists a variable $v \in \mathrm{var}(Q_2)$ such that $h_1(v) \neq h_2(v)$. Consider an atom $R(\vec{v})$ in Q_2 such that $v \in \vec{v}$. Since they are body homomorphisms, $R(h_1(\vec{v}))$ and $R(h_2(\vec{v}))$ are in Q_1. This is a contradiction to the fact that Q_1 is self-join-free. □

We can show that the reduction can be applied whenever the tractable CQ does not provide all variables of some difficult structure in the difficult CQ.

Lemma 3.45. *Consider a UCQ $Q = Q_1 \cup Q_2$ where Q_1 is difficult and Q_2 is tractable. If Q_2 does not provide all variables of a difficult structure in Q_1, then the conditions of Lemma 3.43 hold.*

Proof. We separate to cases according to the type of difficult structure. In all cases we show how to select the sets x_i such that the first three conditions hold and X_ℓ contains a single unprovided variable v. Since v is not provided and Q_2 is free-connex, either there is no body-homomorphism h from Q_2 to Q_1, or $v \notin h(\mathrm{free}(Q_2))$. In both cases, Condition 4 holds.

In case of a tetra, denote its variables by $\{x_1, \ldots, x_k\}$ such that x_k is not provided, and set $X_i = \{x_i\}$. Since no edge contains all tetra variables, Condition 1 holds. Condition 2 trivially holds since the sets X_i are of size one. Condition 3 holds since the tetra edges form the connectors of $\{x_1, x_2\}$, $\{x_2, \ldots, x_k\}$, and $\{x_1, x_3, \ldots, x_k\}$.

In case of a chordless cycle, denote it as $x_1, \ldots, x_k, x_1$ such that x_k is not provided. Set $X_1 = \{x_1, .., x_{k-2}\}$, $X_2 = \{x_{k-1}\}$, and $X_3 = \{x_k\}$. As the cycle is chordless, Condition 1 holds. Condition 2 holds due to the path $x_1, .., x_{k-2}$ that lies on the cycle. Condition 3 holds due to the three edges containing $\{x_{k-2}, x_{k-1}\}$, $\{x_{k-1}, x_k\}$ and $\{x_k, x_1\}$ on the cycle.

In case of a free-path, we further split into two cases. If an end variable of the path is not provided, denote the path by $x, z_1, \ldots, z_k, y$ such that y is not provided. We set $X_1 = \{x\}$, $X_2 = \{z_1, .., z_k\}$ and $X_3 = \{y\}$. Otherwise, if both end variables are provided, a middle variable is not provided. Denote this variable by z, and the path by $x_1, \ldots, x_k, z, y_1, \ldots, y_m$. We set $X_1 = \{x_1, \ldots, x_k\}$, $X_2 = \{y_1, \ldots, y_m\}$ and $X_3 = \{z\}$. In both cases, Condition 1 holds since the path is chordless and so no atom contains both a variable containing x in the name and a variable containing y. Condition 2 holds due to the relevant subpaths. For Condition 3, the connection between the sets containing the end variables is done through the connector free(Q_1); this is possible since the remaining set contains only existential variables. The other two connectors appear on the path. □

Note that Lemma 3.43 and Lemma 3.45 do not require the tractable CQ to be self-join-free. As an example, consider the following modification of Example 3.36.

$$Q_1(x, y, w) \leftarrow R_1(x, z), R_2(z, y), R_3(y, w) \text{ and}$$
$$Q_2(x, y, w) \leftarrow R_1(x, t_1), R_3(y, t_2), R_3(w, t_3).$$

The reduction can be applied here with $X_1 = \{x\}$, $X_2 = \{y\}$, and $X_3 = \{z\}$.

Completing the Characterization

We next show that, in case the difficult CQ is binary, if the UCQ is not covered by Lemma 3.45, then the union is necessarily in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$. Recall that a UCQ is tractable if it has a free-connex union extension, and that a difficult CQ has difficult structures (a free-path, a chordless cycle or a tetra). We define a process of generating a union extension of a difficult CQ by repeatedly adding virtual atoms that correspond to difficult structures (thus eliminating the difficult structures). This is similar to the approach we took in proving Lemma 3.26.

Definition 3.46. Let Q_1 be a CQ in a union. We define a *resolution step* over Q_1: if there is a difficult structure in Q_1 with the variables V, and V is provided by a CQ $Q_2 \in Q$, extend Q_1 with a virtual atom with the variables V. *Resolving* a CQ in a union is applying resolutions steps until it is no longer possible. Similarly, a resolved UCQ is one where each CQ is resolved. When a query Q has been resolved, we denote it Q^+.

We aspire to show that whenever the resolution process does not result in a free-connex union extension, the reduction from UTD can be applied. First, we give two observations regarding the relationship between the original and the resolved queries.

Lemma 3.47. *Let $Q = Q_1 \cup Q_2$, and let Q_1^+ be a union extension of Q_1 due to a body-homomorphism h from Q_2 to Q_1. If there is a path P^+ between v_1 and v_k in Q_1^+, then there is a simple chordless path between v_1 and v_k in Q_1 that goes only through variables of* $\text{var}(P^+) \cup h(\text{free}(Q_2))$.

Proof. Every edge in Q_1^+ either: (1) is an edge of Q_1; (2) contains the variables of a difficult structure with variables contained in $h(\text{free}(Q_2))$. Note that all difficult structures are connected. First obtain a path in Q_1 that starts and ends in the same variables as P^+: replace every new edge of Q_1^+ in P^+ with a corresponding path through the difficult structure that it covers. Then, take a simple chordless path contained in this path. □

Lemma 3.48. *Let $Q = Q_1 \cup Q_2$, and let Q_1^+ be a union extension of Q_1 due to a body-homomorphism h from Q_2 to Q_1. If there is a path P^+ in Q_1^+ from a variable v to a variable in* $\text{free}(Q_1)$*, then there is a simple chordless path P in Q_1 from v to a variable $u \in \text{free}(Q_1)$ such that* $\text{var}(P) \cap \text{free}(Q_1) = \{u\}$*, and* $\text{var}(P) \subseteq \text{var}(P^+) \cup h(\text{free}(Q_2))$.

Proof. First, take the simple chordless path P' in Q_1 that is obtained from P^+ using Lemma 3.47. Then, take the subpath of P' between v and the first variable in $\text{free}(Q_1)$. Such a variable exists because P' ends in a free variable. □

We also need to conclude the existence of chordless cycles from that of simple cycles. Lemma 3.49 may seem trivial for graphs, but it does not hold for hypergraphs. In fact, this difference between graphs and hypergraphs is the main reason why we cannot show Lemma 3.50 for UCQs with general relations (of arity larger than 2).

Lemma 3.49. *If a vertex v appears in a simple cycle in a graph, then v also appears in a simple chordless cycle.*

Proof. Denote the cycle by $v, v_2, \ldots, v_m, v$. Take a chordless path contained in $v_2, \ldots, v_m$, and denote it $v_2 = u_1, \ldots, u_k = v_m$. Let $t > 1$ be the smallest such that u_t is a neighbor of v. Such t exists since u_m is a neighbor of v. Then, the cycle $v, u_1, \ldots, u_t, v$ is chordless. □

We now show that in the restricted case of binary relations, when all variables that participate in difficult structures are provided, the resolution process given in Definition 3.46 results in a free-connex CQ.

Lemma 3.50. *Let $Q = Q_1 \cup Q_2$ where Q_1 is difficult and contains only unary or binary relations, Q_2 is tractable, and Q_2 provides all variables of the difficult structures in Q_1. Then, the resolved Q^+ is free-connex.*

Proof. Let Q_1^+ be the resolved Q_1, and assume by contradiction that Q_1^+ is not free-connex. Thus, it contains a difficult structure. Since Q_2 provides all variables of the

difficult structures, by construction, Q_1^+ has no difficult structures that also appear in Q_1. By Lemma 3.44, there is a single body-homomorphism h from Q_2 to Q_1. By the definition of the resolution process, $h(\text{free}(Q_2))$ does not contain all variables of any difficult structure in Q_1^+.

We first show that Q_1^+ is acyclic. Assume by contradiction that it is cyclic, then it either contains a chordless cycle or a tetra of size $k > 3$. The first case is that it contains a tetra of size k. Since Q_1^+ is resolved, some variable of the tetra is not in $h(\text{free}(Q_2))$. Thus, all atoms of Q_1^+ that contain this variable appear in Q_1. These atoms are therefore binary, and $k = 3$. We now treat the case that the new structure is a simple chordless cycle. Denote the cycle by $x_1, \ldots, x_k$ such that $x_k \notin h(\text{free}(Q_2))$. Note that x_{k-1}, x_k, x_1 are distinct variables. Since x_k is not provided, we know that the edges containing $\{x_{k-1}, x_k\}$ and $\{x_k, x_1\}$ are original. According to Lemma 3.47, due to the path $x_1, \ldots, x_{k-1}$ and since $x_k \notin h(\text{free}(Q_2))$, there is a simple path between x_1 and x_{k-1} in Q_1 that does not go through x_k. This, together with the two edges $\{x_{k-1}, x_k\}$ and $\{x_k, x_1\}$, results in a simple cycle $x_1, \ldots, x_{k-1}, x_k$ in Q_1. According to Lemma 3.49, x_k appears in a chordless cycle in Q_1. Since $x_k \notin h(\text{free}(Q_2))$, this contradicts our assumptions. Therefore Q_1^+ is acyclic.

Since Q_1^+ is acyclic but not free-connex, it contains a free-path which we denote by $P^+ = x_1, \ldots, x_k$. We have that $x_j \notin h(\text{free}(Q_2))$ for some $1 \leq j \leq k$. We now prove that x_j appears in a difficult structure in Q_1. This would mean that $x_j \in h(\text{free}(Q_2))$, which is a contradiction.

First assume that x_j is at an end of the path; without loss of generality, $j = 1$. Since $x_j \notin h(\text{free}(Q_2))$, every edge containing x_j in Q_1^+ also appears in Q_1, and so there is an edge $\{x_1, x_2\}$ in Q_1. Take the path $x_2 = t_1, \ldots, t_m$ that is obtained from the subpath $x_2, \ldots, x_k$ of P^+ using Lemma 3.48. Note that this path does not contain x_1, and it is a simple chordless path of length 1 or more that ends with a free variable, and all other variables are not free. If there is a neighbor t_i of x_1 with $i > 1$, take i to be the minimal such index, and $x_1, t_1, \ldots, t_i, x_1$ is a chordless cycle. Otherwise, $x_1, t_1, \ldots, t_m$ is a chordless path, and it is a free-path.

We now address the case that $1 < j < k$. Apply Lemma 3.48 as before on both sides of P^+ to obtain chordless simple paths $x_{j+1} = t_1, \ldots, t_m$ and $x_{j-1} = v_1, \ldots, v_n$ that do not contain x_j, where v_n and t_m are free, and the other variables are projected. Note that since x_{j-1}, x_j, x_{j+1} is part of a simple chordless path in Q_1, these three variables are distinct. If the two paths share a variable or neighbors, x_j is part of a simple chordless cycle. Otherwise, $v_n, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, t_m$ is a free-path. □

We can now show that free-connex union extensions capture all tractable UCQs containing one tractable CQ and one difficult CQ when the relations are binary.

Theorem 3.51. *Let $Q = Q_1 \cup Q_2$ be a non-redundant UCQ where Q_1 is difficult and contains only unary or binary relations and Q_2 is tractable. If Q does not admit a free-connex union extension, then $Q \notin \mathsf{Enum}\langle \text{lin}, \text{const} \rangle$, assuming* UTD.

Proof. According to Lemma 3.50, since Q does not admit a free-connex union extension, Q_2 does not provides all variables of the difficult structures in Q_1. According to Lemma 3.45, Lemma 3.43 can be applied, and $Q \notin \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$ assuming UTD.□

Beyond Binary CQs

Lemma 3.50 does not hold when we allow general arities. However, this does not mean that Theorem 3.51 does not hold. Here is an example for when Lemma 3.50 does not apply, but we can still show that the UCQ is hard assuming UTD.

Example 3.52. [CK19b, Example 38] Consider the union of the following:

$$Q_1(x_2, \ldots, x_k) \leftarrow \{R_i(x_1, ..., x_{i-1}, x_{i+1}, ..., x_k) \mid 1 \leq i \leq k-1\}$$
$$Q_2(x_2, \ldots, x_k) \leftarrow R_1(x_2, \ldots, x_{k-1}, x_1), R_2(x_k, x_3, \ldots, x_{k-1}, v).$$

The query Q_1 is cyclic and Q_2 is free-connex. Even though Q_2 provides $\{x_1, \ldots, x_{k-1}\}$, adding a virtual atom with these variables does not result in a free-connex extension, as this extension is exactly a tetra.

Lemma 3.43 can be applied here by setting $X_i = \{x_i\}$. Condition 1 holds since no edge contains $\{x_1 \ldots, x_k\}$. Condition 2 holds trivially since the sets are of size one. Condition 3 holds due to the edges $\{x_1 \ldots, x_k\} \setminus \{x_1\}$, $\{x_1 \ldots, x_k\} \setminus \{x_2\}$, and $\{x_1 \ldots, x_k\} \setminus \{x_3\}$. Condition 4 holds since $x_k \notin h(\text{free}(Q_2))$.

It is left for future work to try and prove Theorem 3.51 for general arity.

Replacing Previous Assumptions

In this section we show that, if we assume UTD, we can conclude the previously known hardness results without making additional assumptions. In Section 3.2 we showed that if a union of two difficult CQs does not admit a free-connex union extension, then it is intractable assuming BMM, Hyperclique and 4-clique. We show that BMM and Hyperclique can always be replaced with assuming UTD, and we show an alternative reduction for the cases that rely on 4-clique. Thus, we prove that this holds independently of additional assumptions if we assume UTD.

Proposition 3.53. BMM *is a consequence of* UTD.

Proof. Boolean matrix multiplication can be used to detect triangles in a tripartite graph: Consider the multiplication of the adjacency matrix of V_1 and V_2 with the adjacency matrix of V_2 and V_3. Every result is a path of length two, and we can check in constant time whether its end-points are neighbors. Therefore, we can find all triangles in the same time it takes to multiply the matrices. If BMM does not hold, it is possible to multiply two Boolean $n \times n$ matrices in $O(n^2)$ time, and so it is possible to find triangles in a tripartite graph of n vertices in each set in time $O(n^2)$. This contradicts UTD with $\alpha = 1$. □

Proposition 3.54. HYPERCLIQUE *is a consequence of* UTD.

Proof. Assume by contradiction that UTD holds but HYPERCLIQUE does not. If HYPERCLIQUE does not hold, there exists $k \geq 3$ such that it is possible to determine the existence of a k-hyperclique in a $(k-1)$-uniform hypergraph with m nodes in time $O(m^{k-1})$. Set $\alpha = \frac{1}{k-2}$. Assume we are given a tripartite graph g with node sets V_1,V_2,V_3 and edges sets $E_{1,2}$, $E_{2,3}$, $E_{1,3}$ where $|V_1| = |V_2| = n^\alpha$ and $|V_3| = n$. We now construct a hyperclique instance g'.

We encode the vertices of V_3 as $U_3 \times \cdots \times U_k$ such that $|U_3| = \ldots = |U_k| = n^\alpha$. For every edge $(v_1, v_3) \in E_{1,3}$, add an edge $\{v_1, u_3, \ldots, u_k\}$ where $u_3, \ldots, u_k$ is the representation of v_3. For every edge $(v_2, v_3) \in E_{2,3}$, add an edge $\{v_2, u_3, \ldots, u_k\}$ where $u_3, \ldots, u_k$ is the representation of v_3. For every edge $(v_2, v_3) \in E_{1,2}$, add an edge containing v_2, v_3 and every combination of $k-3$ nodes from distinct sets in $U_3, \cdots, U_k$. This results in a $(k-1)$-uniform hypergraph with kn^α nodes, and k-hypercliques in this construction correspond exactly to triangles in the tripartite graph. Then, we can detect a k-hyperclique in g' and a triangle in g in time $O((kn^\alpha)^{k-1}) = O(n^{1+\alpha})$ in contradiction to UTD. □

Let us now prove the equivalent of Lemma 3.22 based on UTD.

Lemma 3.55. *Let $Q = Q_1 \cup Q_2$ be a union of self-join-free body-isomorphic acyclic CQs. If Q_1 and Q_2 are free-path guarded and Q_1 is not bypass guarded, then* ENUM$\langle Q \rangle$ *is not in* $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$*, assuming* UTD.

Proof. We saw in Lemma 3.22 that there exist variables z_0, z_1, z_2, u such that the following holds: $z_0, z_2 \in \text{free}(Q_1)$, $z_1 \notin \text{free}(Q_1)$, $u \notin \text{free}(Q_2)$, there are two atoms containing $\{z_0, z_1, u\}$ and $\{z_1, z_2, u\}$, and there is no atom containing $\{z_0, z_2\}$. Use Lemma 3.43 with $X_1 = \{z_0\}$, $X_2 = \{z_2\}$, $X_3 = \{z_1\}$, and $X_4 = \{u\}$. Since there is no atom containing both z_0 and z_2, Condition 1 holds. Condition 2 trivially holds since the sets X_i are of size one. Condition 3 holds due to the atoms containing $\{z_0, z_1, u\}$ and $\{z_1, z_2, u\}$, and since $z_0, z_2 \in \text{free}(Q_1)$. The free variables form a valid connector since $z_1 \notin \text{free}(Q_1)$. Since $u \notin \text{free}(Q_2)$, Condition 4 holds. Therefore, ENUM$\langle Q \rangle$ is not in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$. □

Since all hardness results in Section 3.2 other than Lemma 3.22 rely on BMM or HYPERCLIQUE, we have proved that all of the results of that section also apply when replacing the assumptions with UTD. In particular, the following holds.

Theorem 3.56. *Let $Q = Q_1 \cup Q_2$ be a non-redundant union of difficult CQs. If Q does not have a free-connex union extension, then* ENUM$\langle Q \rangle \notin \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$*, assuming* UTD.

By combining Theorem 3.56 with Theorem 3.51 and Theorem 3.9, we get a full characterization based on UTD for a union of two binary CQs.

Corollary 3.57. *Let $Q = Q_1 \cup Q_2$ be a non-redundant union of binary CQs.*

- *If Q has a free-connex union extension, then* $\text{ENUM}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$.
- *Otherwise, Q does not have a free-connex union extension and* $\text{ENUM}\langle Q\rangle$ *is not in* $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$, *assuming* UTD.

3.4 Additional Notes

We devote this section to discuss some implications of the work presented in this chapter when the settings slightly differ from the ones assumed so far. In Section 3.4.1 we discuss unions of conjunctive queries with disequalities, and in Section 3.4.2 we discuss the case where we want to restrict the space requirements of our algorithm.

3.4.1 Unions of CQs with Disequalities

In this section we discuss the enumeration complexity of unions of CQs with disequalities. In the case of individual CQs with disequalities, the disequalities have no effect on the enumeration complexity. That is, one can simply ignore the disequalities, and the remaining CQ is tractable if and only if it is free-connex (under the same assumptions as in Theorem 2.1) [BDG07]. A natural question is whether this happens also with UCQs. We next show that it does not, and some tractable UCQs become intractable with the addition of disequalities.

Example 3.58. Let $Q = Q_1 \cup Q_2$ with

$$Q_1(x, y, z, w) \leftarrow R_1(x, y), R_2(y, z), R_3(z, z), R_4(x, z), R_5(w)$$
$$Q_2(x, y, z, w) \leftarrow R_1(x, y), R_3(w, z), z \neq w$$

We can show this union is hard by encoding triangle detection to the cycle in Q_1 as in the proof of Theorem 2.1. This construction assigns R_3 with only pairs of the same vertex. Due to the disequality, Q_2 has no answers over this construction, so finding an answer to the union in linear time detects a triangle in linear time. Therefore, Q is intractable assuming HYPERCLIQUE. If there was no disequality, Q_2 would provide the cycle variables to Q_1 and the UCQ would become tractable. □

This example shows that, unlike the known dichotomy for CQs, the positive results for UCQs presented in this chapter do not carry over to UCQs with disequalities by simply ignoring the disequalities. That is, a union extension, as defined for UCQs without inequalities, does not suffice to ensure tractability for UCQs with inequalities. Nevertheless, a simple adjustment of the definitions and proofs given in Section 3.1 reveals cases where a UCQ with disequalities containing hard CQs is easy.

Definition 3.59. Let Q_1, Q_2 be CQs with disequalities.

- A *body-homomorphism* from Q_2 to Q_1 is a mapping $h : \text{var}(Q_2) \to \text{var}(Q_1)$ where:
  - for every atom $R(\vec{v})$ of Q_2, we have $R(h(\vec{v}))$ in Q_1.
  - for every disequality $v \neq u$ of Q_2, we have $h(v) \neq h(u)$ in Q_1.

- We say that Q_2 *supplies* a set V of variables if Q_2 is free-connex and $V \subseteq \text{free}(Q_2)$.

Note that the definition of a body-homomorphism now takes the disequalities into account. Note also that the definition of supplying variables here is simpler than before as we require that the supplying CQ will be free-connex[1]. We set the definition of a union extension as in Section 3.1 (but using the new definition of supplied variables and body-homomorphisms). Using these definitions, we can get a result similar to that of Theorem 3.9. We first prove the equivalent of Lemma 3.6 in our settings.

Lemma 3.60. *Let Q_2 be a CQ with disequalities that supplies the variables $\vec{v}_2$. Given an instance I, one can compute with linear time preprocessing and constant delay $M = Q_2(I)$, and M can be translated in time $O(|M|)$ to a relation R^M such that: for every CQ Q_1 with a body-homomorphism h from Q_2 to Q_1 and for every answer $\mu_1 \in \text{full}(Q_1)(I)$, there is the tuple $\mu_1(h(\vec{v}_2)) \in R^M$.*

Proof. Let Q_1, Q_2 be CQs with disequalities such that Q_2 provides V_1 to Q_1. Since Q_2 is free-connex, it can be answered with linear preprocessing and constant delay [BDG07]. We define $R^M = \{\mu_2(\vec{v}_2) \mid \mu_2 \in Q_2(I)\}$. Let Q_1 be a CQ such that there is a body-homomorphism h from Q_2 to Q_1, and let $\mu_1 \in \text{full}(Q_1)(I)$. Since h is a body-homomorphism, for every atom $R(\vec{v})$ in Q_2, $R(h(\vec{v}))$ is an atom in Q_1, and for every disequality $v \neq u$ of Q_2, $h(v) \neq h(u)$ is in Q_1. Since μ_1 is an answer to $\text{full}(Q_1)$, for such atoms and disequalities $\mu_1(h(\vec{v})) \in R^I$ and $\mu_1(h(v)) \neq \mu_1(h(u))$. This means that $\mu_1 \circ h|_{\text{free}(Q_2)}$ is an answer to Q_2, so there exists $\mu_2 \in Q_2(I)$ such that $\mu_1 \circ h|_{\text{free}(Q_2)} = \mu_2$. By construction, $\mu_1(h(\vec{v}_2)) = \mu_2(\vec{v}_2) \in R^M$. □

With Lemma 3.60 at hand, we can apply the proof of Theorem 3.9 as is. We only need to use Lemma 3.60 instead of Lemma 3.6. This proves the following result.

Theorem 3.61. *Let Q be a union of CQ with disequalities. If Q has a free-connex union extension, then* $\text{ENUM}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$.

Example 3.62. Let $Q = Q_1 \cup Q_2$ with

$$Q_1(x, y, w) \leftarrow R_1(x, z), R_2(z, y), R_3(y, w), x \neq z \text{ and}$$
$$Q_2(x, y, w) \leftarrow R_1(x, y), R_2(y, w), x \neq y$$

This is the same as Example 1.2 except for the added disequalities. Without the disequalities this union is easy as Q_2 supplies $\{x, y, w\}$, and adding a virtual atom with a union extension to Q_1 results in a free-connex form. Since Q_2 is free-connex and the disequality of Q_2 maps to that of Q_1 using the body-homomorphism, this union is easy even with the disequalities according to Theorem 3.61. □

[1]This is a restriction we impose for simplicity, and it is possible that a more refined condition may lead to additional tractable cases. However, this seems to require going into the details of the algorithm for CQs with disequalities, and it is left for future work.

Theorem 3.61 can be used also to show the tractability of UCQs that are not naturally of the form defined above. For example, in the case of the last example, the union is easy even without the disequality in Q_1.

Example 3.63. Let $Q = Q_1 \cup Q_2$ with

$$Q_1(x, y, w) \leftarrow R_1(x, z), R_2(z, y), R_3(y, w) \text{ and}$$
$$Q_2(x, y, w) \leftarrow R_1(x, y), R_2(y, w), x \neq y$$

We can partition the answers to Q_1 to those that assign the same values to x and z and those that do not. Q is equivalent to $Q' = Q_1^{\neq} \cup Q_1^{=} \cup Q_2$ with

$$Q_1^{\neq}(x, y, w) \leftarrow R_1(x, z), R_2(z, y), R_3(y, w), x \neq z \text{ and}$$
$$Q_1^{=}(x, y, w) \leftarrow R_1(x, x), R_2(x, y), R_3(y, w) \text{ and}$$
$$Q_2(x, y, w) \leftarrow R_1(x, y), R_2(y, w), x \neq y$$

Here, $Q_1^{\neq} \cup Q_2$ can be enumerated efficiently as in Example 3.62. Since $Q_1^{=}$ is free-connex, it can also be answered efficiently. Using the Cheater's Lemma, these answers can be combined to conclude that $\textsc{Enum}\langle Q \rangle \in \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$. □

In conclusion, unlike individual CQs, the complexity of answering unions of CQs changes with the addition of disequalities. We proved the tractability of some unions with disequalities that can be easily concluded from the results of this chapter, but identifying all of the tractable cases is left for future work.

3.4.2 Note on Space Usage

In this chapter, we only considered time bounds. The class $\mathsf{CD}\circ\mathsf{Lin}$ describes the problems that can be solved with the same time bounds, but with the additional restriction that the available space for writing during the enumeration phase is constant. Evaluating free-connex CQs is in $\mathsf{CD}\circ\mathsf{Lin}$, and Kazana offers a comparison between $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$ and $\mathsf{CD}\circ\mathsf{Lin}$ [Kaz13, Section 8.1.2]. All lower bounds presented in this chapter naturally hold for $\mathsf{CD}\circ\mathsf{Lin}$ as $\mathsf{CD}\circ\mathsf{Lin} \subseteq \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$ by definition. The tractability of unions containing only free-connex CQs, as explained in Theorem 3.1, also holds for this class. However, the memory we used in our techniques for the tractable UCQs that do not contain only tractable CQs may increase in size by a constant with every new answer. An interesting question is whether we can achieve the same time bounds when restricting the memory according to $\mathsf{CD}\circ\mathsf{Lin}$.

The first reason we use polynomial space is to regularize the delay and avoid duplicates. This is achieved in the Cheater's Lemma (Lemma 3.4) by storing all results we produce. We should mention in that context that regularizing the delay is arguably not of practical importance. In practice, once we have an answer, we will probably want to use it immediately. For this reason, one might be satisfied with simply using a relaxed complexity measure which captures the fact that an algorithm performs "just as good" as linear preprocessing and constant delay when given enough space: linear partial time (see Definition 3.2). The Cheater's Lemma shows that, whenever we have

an algorithm $\mathcal{A}$ that runs in linear partial time, there is a linear preprocessing and constant delay algorithm $\mathcal{A}'$ that uses additional space and computes the same results, where every result in $\mathcal{A}$ is returned no later than the same result in $\mathcal{A}'$. In fact, if we denote by $\mathsf{Lin}_{\mathsf{Partial}}$ the class of enumeration problems that can be solved by a linear partial time algorithm, $\mathsf{Lin}_{\mathsf{Partial}} = \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$. Please note that this does not imply $\mathsf{Lin}_{\mathsf{Partial}} = \mathsf{CD}\circ\mathsf{Lin}$, as we currently do not know whether $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle = \mathsf{CD}\circ\mathsf{Lin}$.

The other reason that we use large amounts of space is that we need to store the generated relations that correspond to the virtual atoms in order to evaluate the union extension as specified in Theorem 3.9. The discussion above implies that it makes sense to focus on eliminating this fundamental reason. This seems to require a different approach than the one presented in this chapter.

Example 3.64.

$$Q_1(x, y, z, w) \leftarrow R_1(x, y), R_2(y, z), R_3(z, x), R_4(x, w)$$
$$Q_2(x, y, z, w) \leftarrow R_1(x, t_1), R_2(y, t_2), R_3(z, t_3), R_4(w, t_4)$$

Q_1 is cyclic, so it is not possible to find an answer to it in linear time. However, since Q_2 is free-connex and provides the cycle variables $\{x, y, z\}$, and since Q_1 becomes free connex when adding an atom with these variables, the union is solvable in linear preprocessing time and constant delay by saving the results of Q_2 as a relation as suggested in Section 3.1. In this case, there is also a way of solving it with no additional space: use CDY to solve Q_2 efficiently; for every answer (a, b, c, d) to Q_2, check if $a = d$, $(a, b) \in R_1$, $(b, c) \in R_2$ and $(c, a) \in R_3$; if all these conditions are met, go over all tuples $(a, e) \in R_4$ and output (a, b, c, e). This finds all answers to $Q_1 \cup Q_2$ with constant time per answer. However, if an answer belongs to both CQs, we print it twice. The duplicates can easily be avoided by replacing R_4 in Q_1 with a modified version, lacking the tuples that cause duplicates. During preprocessing, compute $R'_4 = R_4 \setminus \{(a, e) \mid \exists t \text{ s.t. } (e, t) \in R_4\}$. During enumeration, for every qualifying answer (a, b, c, d) to Q_2, go over all tuples $(a, e) \in R'_4$ and output (a, b, c, e). □

Example 3.64 shows that reducing the space requirements is sometimes possible. We do not know however whether this is the case for all UCQs with a free-connex union extension. Consider Example 1.2. The same approach would mean that for every answer (a, b, c) to Q_2 we would go over all tuples $(c, d) \in R_3$ and print (a, c, d). The problem here is that we would print many duplicates: every answer (a, c, d) would be produced as many times as there are different b values with $(a, b, c) \in Q_2(I)$.

In conclusion, UCQs that contain intractable CQs can sometimes be made easy by using virtual atoms containing provided variables. In this chapter, we proposed instantiating a relation that corresponds to each virtual atom during enumeration. This instantiation can sometimes be avoided, but it requires a different approach, and it is left for future work to find when this can be done. The discussion above also gives rise to a possible relaxation that can be considered a first step: can we evaluate such UCQs efficiently with only constant writing memory available during enumeration, when we

allow for linear partial time and a constant number of duplicates per answer? Note that in example 3.64 we do not need this relaxation, so another interesting question is whether this relaxation is useful for our task or can all UCQs that can be answered with this relaxed measure also be answered with linear preprocessing and constant delay without extra space.

Chapter 4

Enumerating Query Answers in Random Order

In the previous chapter, we discussed the question of when UCQs can be answered efficiently, but we did not address an important aspect: the order in which the answers are emitted. Measuring the complexity in terms of delay rather than total time stems from the assumption that partial results are useful, but if we make no requirements on the order, the first results may be very similar to one another and give very little information on what we can expect from the results to come. In this chapter, we require the intermediate results to be representative of the entire solution space. To this aim, we consider the task of enumerating answers in provably random order. We show how a uniformly random permutation of the answers can be achieved assuming we can perform random access to the query answers, and we compare the complexity of these two tasks with that of enumeration with no order requirements in the cases of CQs and UCQs. As in the previous chapter, we again assume a general schema with no dependencies.

This chapter includes joint work with Shai Zeevi, Christoph Berkholz, Alessio Conte, Benny Kimelfeld, and Nicole Schweikardt. Some of the findings of this chapter were published in the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems [CZB+20].

Organization In Section 4.1, we define the three tasks related to our goal: (1) Enumeration, as covered in the previous chapter; (2) Random Permutation, with the additional order requirement; (3) Random Access, which simulates the ability to read the answers from a precomputed array. We discuss the relations between the three tasks, and show how random access can be used to achieve random permutation. In Section 4.2, we show which CQs admit efficient algorithms for these three tasks, with logarithmic time per answer after linear preprocessing. Free-connex CQs admit efficient random access, and so they are tractable with respect to all three tasks. Since self-join-free non-free-connex CQs are intractable with respect to enumeration, they are also intractable with respect to the other two tasks. In Section 4.3, we address UCQs and

show that, unlike for CQs, the three tasks do not behave similarly when inspecting which queries admit efficient algorithms. A union of free-connex CQs is always tractable with respect to enumeration, but it may not be tractable with respect to random access. We then show two solutions for unions of free-connex CQs: an efficient random-permutation algorithm that can only be applied for unions with a tractable intersection, and an algorithm that can always be applied but guarantees a relaxed measure of logarithmic delay in expectation.

4.1 The Three Tasks

In this section, we define classes of enumeration problems and discuss the relationship between them.

4.1.1 Definitions

We write d to denote a function from the positive integers $\mathbb{N}_{\geq 1}$ to the non-negative reals $\mathbb{R}_{\geq 0}$, and $d = \text{const}$, $d = \text{lin}$, $d = \log^c$ (for $c \geq 1$) mean $d(n) = 1$, $d(n) = n$, $d(n) = \log^c(n)$, respectively. We define the following generalization of $\mathsf{Enum}\langle \text{lin}, \text{const} \rangle$.

Definition 4.1. Let d be a function from $\mathbb{N}_{\geq 1}$ to $\mathbb{R}_{\geq 0}$. We define $\mathsf{Enum}\langle \text{lin}, d \rangle$ to be the class of enumeration problems for which there exists an enumeration algorithm $\mathcal{A}$ such that for every input I it holds that $t_p \in O(|I|)$ and $t_d \in O(d(|I|))$. Furthermore, $\mathsf{Enum}\langle \text{lin}, \text{polylog} \rangle$ is the union of $\mathsf{Enum}\langle \text{lin}, \log^c \rangle$ for all $c \geq 1$.

A *random-permutation algorithm* for an enumeration problem P is an enumeration algorithm where every emission is done uniformly at random. That is, at every emission, every not yet emitted answer has equal probability of being emitted. As a result, if $|P(I)| = n$, every ordering of the answers $P(I)$ has the probability $\frac{1}{n!}$ of representing the order of printed answers.

Definition 4.2. Let d be a function from $\mathbb{N}_{\geq 1}$ to $\mathbb{R}_{\geq 0}$. We define $\mathsf{REnum}\langle \text{lin}, d \rangle$ to be the class of enumeration problems for which there exists a random-permutation algorithm $\mathcal{A}$ such that for every input I it holds that $t_p \in O(|I|)$ and $t_d \in O(d(|I|))$. Furthermore, $\mathsf{REnum}\langle \text{lin}, \text{polylog} \rangle$ is the union of $\mathsf{REnum}\langle \text{lin}, \log^c \rangle$ for all $c \geq 1$.

Fact 4.3. By definition, $\mathsf{REnum}\langle \text{lin}, d \rangle \subseteq \mathsf{Enum}\langle \text{lin}, d \rangle$ for all d.

A *random-access algorithm* for an enumeration problem P is an algorithm $\mathcal{A}$ consisting of a preprocessing phase and an access routine. The preprocessing phase builds a data structure based on the input I. Afterwards, the access routine may be called any number of times, and it may use the data structure built during preprocessing. There exists an order of $P(I)$, denoted $a_1, ..., a_n$ and called *the enumeration order of* $\mathcal{A}$ such that, when the access routine is called with parameter i, it returns a_i if $1 \leq i \leq n$, and

an error message otherwise. Note that there are no constraints on the order as long as the routine consistently uses the same order in all calls. Using the access routine with parameter i is called *accessing* a_i; the time it takes to access an answer is called *access time* and denoted t_a.

Definition 4.4. Let d be a function from $\mathbb{N}_{\geq 1}$ to $\mathbb{R}_{\geq 0}$. We define $\mathsf{RAccess}\langle \text{lin}, d\rangle$ to be the class of enumeration problems for which there exists a random-access algorithm $\mathcal{A}$ such that for every input I the preprocessing phase takes time $t_p \in O(|I|)$ and the access time is $t_a \in O(d(|I|))$. Furthermore, $\mathsf{RAccess}\langle \text{lin}, \text{polylog}\rangle$ is the union of $\mathsf{RAccess}\langle \text{lin}, \log^c\rangle$ for all $c \geq 1$.

Successively calling the access routine for $i = 1, 2, 3, \ldots$ leads to:

Fact 4.5. By definition, $\mathsf{RAccess}\langle \text{lin}, d\rangle \subseteq \mathsf{Enum}\langle \text{lin}, d\rangle$ for all d.

A *two-way-access algorithm* for an enumeration problem P is an enhancement of a random-access algorithm with the inverse operation: given an element a, the inverted-access operation returns i such that the ith answer in the random-access is a. If the given element is not an answer, then the algorithm indicates so by returning "not-an-answer." The time it takes to perform the inverted-access is denoted t_r.

Definition 4.6. Let d be a function from $\mathbb{N}_{\geq 1}$ to $\mathbb{R}_{\geq 0}$. We define $\mathsf{TWAccess}\langle \text{lin}, d\rangle$ to be the class of enumeration problems for which there exists a two-way-access algorithm $\mathcal{A}$ such that for every input I the preprocessing phase takes time $t_p \in O(|I|)$ and the access time and inverted-access time are $t_a, t_r \in O(d(|I|))$. Furthermore, $\mathsf{TWAccess}\langle \text{lin}, \text{polylog}\rangle$ is the union of $\mathsf{TWAccess}\langle \text{lin}, \log^c\rangle$ for all $c \geq 1$.

Fact 4.7. By definition, $\mathsf{TWAccess}\langle \text{lin}, d\rangle \subseteq \mathsf{RAccess}\langle \text{lin}, d\rangle$ for all d.

Next, we discuss the connection between the classes $\mathsf{RAccess}\langle \text{lin}, d\rangle$ and $\mathsf{REnum}\langle \text{lin}, d\rangle$.

4.1.2 Random-Access and Random-Permutation

We now show that, under certain conditions, it suffices to devise a random-access algorithm in order to obtain a random-permutation algorithm. To achieve this, we need to produce a random permutation of the indices of the answers.

Note that the trivial approach of producing the permutation upfront will not work: the length of the permutation is the number of answers, which can be super linear in the size of the input; however, we need to produce the first answer after linear time in the size of the input.

Instead, we adapt a known random-permutation algorithm, the *Fisher-Yates Shuffle* [Dur64], so that it works with constant delay after constant preprocessing time. The original version of the Fisher-Yates Shuffle (also known as *Knuth Shuffle*) [Dur64] generates a random permutation in time linear in the number of items in the permutation, which in our setting is polynomial in the size of the input. It initializes an array

Algorithm 4.1 Random permutation of the indices $0, \ldots, n-1$

```
1: assume a[0], ..., a[n−1] are uninitialized
2: for i in 0, ..., n−1 do
3:     choose j uniformly from i, ..., n−1
4:     if a[i] is uninitialized then
5:         a[i] = i
6:     if a[j] is uninitialized then
7:         a[j] = j
8:     swap a[i] and a[j] ; output a[i]
```

containing the numbers $0, \ldots, n-1$. Then, at each step i, it chooses a random index, j, greater than or equal to i and swaps the chosen cell with the ith cell. At the end of this procedure, the array contains a random permutation. Proposition 4.8 describes an adaptation of this procedure that runs with constant delay and constant preprocessing time in the RAM model.

Proposition 4.8. *A random permutation of* $0, \ldots, n-1$ *can be generated with constant delay and constant preprocessing time.*

Proof. Algorithm 4.1 generates a random permutation with the required time constraints by simulating the Fisher-Yates Shuffle. Conceptually, it uses an array a where at first all values are marked as "uninitialized", and an uninitialized cell $a[k]$ is considered to contain the value k. At every iteration, the algorithm prints the next value in the permutation.

Denote by a_j the value $a[j]$ if it is initialized, or j otherwise. We claim that in the beginning of the ith iteration, the values $a_i, \cdots, a_{n-1}$ are exactly those that the procedure did not print yet. This can be shown by induction: at the beginning of the first iteration, $a_0, \ldots, a_{n-1}$ represent $0, \ldots, n-1$, and no numbers were printed; at iteration $i-1$, the procedure stores in $a[i-1]$ the value that it prints, and moves the value that was there to a higher index.

At iteration i, the algorithm chooses to print uniformly at random a value between $a_i, \cdots, a_{n-1}$, so the printed answer at every iteration has equal probability among all the values that have not yet been printed. Therefore, Algorithm 4.1 correctly generates a random permutation.

The array a can be simulated using a lookup table that is empty at first and is assigned with the required values when the array changes. In the RAM model with uniform cost measure, accessing such a table takes constant time. Overall, Algorithm 4.1 runs with constant delay (and constant preprocessing time). Note that $O(n)$ space is used to generate a permutation of n numbers. □

With the ability to efficiently generate a random permutation of $\{0, \ldots, n-1\}$, whenever we have available a random-access algorithm for an enumeration problem and

if we can also tell the number of answers, then we can build a random-permutation algorithm as follows: we can produce, on the fly, a random permutation of the indices of the answers and output each answer using the access routine.

We say that an enumeration problem has *polynomially many answers* if the number of answers per input I is bounded by a polynomial in the size of I. In particular, if P is the evaluation of a CQ or a UCQ, then P has polynomially many answers.

Theorem 4.9. *If* $P \in \mathsf{RAccess}\langle \mathrm{lin}, \log^c \rangle$ *and* P *has polynomially many answers, then* $P \in \mathsf{REnum}\langle \mathrm{lin}, \log^c \rangle$*, for all* $c \geq 1$*.*

Proof. Let P be an enumeration problem in $\mathsf{RAccess}\langle \mathrm{lin}, \log^c \rangle$, and let $\mathcal{A}$ be the associated random-access algorithm for P. Given an input I, our random-permutation algorithm performs the preprocessing phase of $\mathcal{A}$ and then, still during its preprocessing phase, computes the number of answers $|P(I)|$ as follows. We can tell whether $|P(I)| < k$ for any fixed k by trying to access the kth answer and checking if we get an out of bound error. We can use this to do a binary search for the number of answers using $O(\log(|P(I)|))$ calls to $\mathcal{A}$'s access procedure. Since $|P(I)|$ is polynomial in the size of the input, $\log(|P(I)|) = O(\log(|I|))$. Each access costs time $O(\log^c(|I|))$. In total, the number $|P(I)|$ is thus computed in time $O(\log^{c+1}(|I|))$, which still is in $O(|I|)$.

During the enumeration phase, we use Proposition 4.8 to generate a random permutation of $0, \ldots, |P(I)|-1$ with constant delay. Whenever we get the next element i of the random permutation, we use the access routine of $\mathcal{A}$ to access the $(i+1)$th answer to our problem. This procedure results in a random permutation of all the answers with linear preprocessing time and delay $O(\log^c)$. □

4.2 CQs

In this section, we discuss random access for CQs. For enumeration, the proof of Theorem 2.1 can also be used to reason about polylogarithmic delay. Theorem 4.10 is similar to Theorem 2.1, except we use sparseBMM instead of BMM and we get the lower bound for $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{polylog} \rangle$ instead of $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$.

Theorem 4.10 ([BDG07, BB13])**.** *Let* Q *be a CQ. If* Q *is free-connex, then* Enum$\langle Q \rangle$ *is in* $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$*. Otherwise, if it is also self-join-free, then* Enum$\langle Q \rangle$ *is not in* $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{polylog} \rangle$ *assuming* sparseBMM *and* Hyperclique*.*

Due to Theorem 2.1, we know that any free-connex CQ is in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$ and that the first answer to self-join-free cyclic CQs cannot be found in linear time assuming Hyperclique. Regarding self-join-free acyclic non-free-connex CQs, we can perform the same reduction as in Theorem 2.1, but use a different variant of the complexity hypothesis to account for the logarithmic delay. Using the reduction defined there, if such a CQ is in $\mathsf{Enum}\langle \mathrm{lin}, \log^c \rangle$, then any two Boolean matrices of size $n \times n$ can be multiplied in $O(m_1 + m_2 + m_3 \cdot \log^c(n))$ time, where m_1, m_2, and m_3 are the

number of non-zero entries in A, B and AB, respectively. sparseBMM states that this multiplication cannot be done in $(m_1 + m_2 + m_3)^{1+o(1)}$ time, so this is a contradiction and it proves Theorem 4.10.

An implication of Theorem 4.10 is that free-connex CQs can be answered with logarithmic delay. Since Brault-Baron [BB13] proved that there exists a random-access algorithm that works with linear preprocessing and logarithmic access time, we get a strengthening of that fact: free-connex CQs belong to $\mathsf{RAccess}\langle \mathrm{lin}, \log \rangle$. According to Theorem 4.9, this also shows the tractability of a random-order enumeration, that is, membership in $\mathsf{REnum}\langle \mathrm{lin}, \log \rangle$.

4.2.1 Two-Way-Access Algorithm

We next present a two-way-access algorithm for free-connex CQs. Compared to Brault-Baron [BB13], the following random-access algorithm is simpler and better lends itself to a practical implementation. In addition, the inverted-access routine that we introduce is needed for our results on UCQs in Section 4.3.

To proceed, we use the following folklore result.

Proposition 4.11. *For any free-connex CQ Q over a database D, one can compute in linear time a full acyclic join query Q' and a database D' such that $Q(D) = Q'(D')$ and D' is globally consistent w.r.t. Q'.*

This reduction was implicitly used in the past as part of CQ answering algorithms (cf., e.g., [IUV17, OZ15]). To prove it, the first step is performing a full reduction to remove dangling tuples (tuples that do not agree with any answer) from the database. This can be done in linear time as proposed by Yannakakis [Yan81b] for acyclic join queries. Then, we utilize the fact that Q is *free-connex*, which enables us to drop some atoms and attributes that correspond to quantified variables and be left with an acyclic CQ that contains exactly the free-variables. This leaves us with a full acyclic join that has the same answers as the original free-connex CQ.

So, it is left to design a random-access algorithm for full acyclic CQs. We do so in the remainder of this section. Algorithm 4.2 describes the preprocessing phase that builds the data structure and computes the count (i.e., the number of answers). Then, Algorithm 4.3 provides random-access, and Algorithm 4.4 provides inverted-access.

Given a relation R, denote by $\mathrm{pAtts}_\mathrm{R}$ the attributes that appear both in R and in its parent. If R is the root, then $\mathrm{pAtts}_\mathrm{R} = \emptyset$. Given a relation R and an answer a, we denote by $\mathrm{bucket}[S, a]$ all tuples in S that agree with a over the attributes that S and a share. We use the notation $\mathrm{bucket}[S, a]$ in the same way also when a is a tuple.

The preprocessing starts by partitioning every relation to buckets according to the different assignments to the attributes shared with the parent relation. This can be done in linear time in the RAM model. Then, we compute a weight $\mathrm{w}(t)$ for each tuple t. This weight represents the number of different answers this tuple agrees with when

Algorithm 4.2 Preprocessing for a globally consistent full acyclic join query

```
1: procedure PREPROCESSING(D, Q)
2:   for R in leaf-to-root order do
3:     Partition R to buckets according to pAtts_R
4:     for bucket B in R do
5:       for tuple t in B do
6:         if R is a leaf then
7:           w(t) = 1
8:         else
9:           let C be the children of R
10:          w(t) = ∏_{S∈C} w(bucket[S, t])
11:        let P be the tuples preceding t in B
12:        startIndex(t) = Σ_{s∈P} w(s)
13:      w(B) = Σ_{t∈B} w(t)
```

Algorithm 4.3 Random access for a globally consistent full acyclic join query

```
1: procedure ACCESS(j)
2:   if j ≥ w(bucket[root, ∅]) then
3:     return out-of-bound
4:   else
5:     answer = ∅
6:     SUBTREEACCESS(bucket[root, ∅], j)
7:     return answer
8: procedure SUBTREEACCESS(B ⊆ R, j)
9:   find t ∈ B s.t. startIndex(t) ≤ j < startIndex(t+1)
10:  answer = answer ∪ {Atts_R → Atts_R(t)}
11:  let R_1, ..., R_m be the children of R
12:  j_1, ..., j_m = SPLITINDEX(j − startIndex(t),
13:                    w(bucket[R_1, t]), ..., w(bucket[R_m, t]))
14:  for i in 1, ..., m do
15:    SUBTREEACCESS(bucket[R_i, t], j_i)
```

only joining the relations of the subtree rooted in the current relation. The weight is computed in a leaf-to-root order, where tuples of a leaf relation have weight 1. The weight of a tuple t in a non-leaf relation R is determined by the product of the weights of the corresponding tuples in the children's relations. These corresponding tuples are the ones that agree with t on the attributes that R shares with its child. The weight of each bucket is the sum of the weights of the tuples it contains. In addition, we assign each tuple t with an index range that starts with startIndex(t) and ends with the startIndex of the following tuple in the bucket (or the total weight of the bucket if this is the last tuple). This represents a partition of the indices from 0 to the bucket weight, such that the length of the range of each tuple is equal to its weight. At the end of preprocessing, the root relation has one bucket (since $\text{pAtts}_{root} = \emptyset$), and the weight of this bucket represents the number of answers to the query.

The random-access is done recursively in a root-to-leaf order: we start from the

Algorithm 4.4 Inverted access for a globally consistent full acyclic join query

```
1: procedure InvertedAccess(answer)
2:   return InvertedSubtreeAccess(root, answer)
3: procedure InvertedSubtreeAccess(R, answer)
4:   find t ∈ R s.t. Atts_R(t) = Atts_R(answer)
5:   if t was not found then
6:     return not-an-answer
7:   let R_1, ..., R_m be the children of R
8:   for i in 1, ..., m do
9:     j_i = InvertedSubtreeAccess(R_i, answer)
10:    if j_i = not-an-answer then
11:      return not-an-answer
12:  offset = CombineIndex(w(bucket[R_1, answer]), j_1, ...,
13:                         w(bucket[R_m, answer]), j_m)
14:  return startIndex(t) + offset
```

single bucket at the root. At each step we find the tuple t in the current relation that holds the required index in its range (we denote by $t{+}1$ the tuple that follows t in the bucket). Then, we assign the rest of the search to the children of the current relation, restricted to the bucket that corresponds to t. Finding t can be done in logarithmic time using binary search. The remaining index $j' = j - \text{startIndex}(t)$ is split into search tasks for the children using the method SplitIndex. The split can be seen as representing j' in a mixed radix numeral system where the units are the bucket weights. In other words, it is done in the same way as an index is split in standard multidimensional arrays: if the last bucket is of weight m, its index is j' mod m, and the other buckets recursively split between them the index $\lfloor \frac{j'}{m} \rfloor$.

Algorithm 4.4 works similarly to Algorithm 4.3. But while the search down the tree in Algorithm 4.3 is guided by the index and the answer is the assignment, in Algorithm 4.4 the search is guided by the assignment and the answer is the index. The function CombineIndex is the reverse of SplitIndex, used in line 13 of Algorithm 4.3. Recursively, CombineIndex$(w_1, j_1, \ldots, w_m, j_m)$ is given by $j_m + w_m \cdot$ CombineIndex$(w_1, j_1, \ldots, w_{m-1}, j_{m-1})$ with CombineIndex$()=0$.

Line 4 can be supported in constant time after an appropriate indexing of the buckets at preprocessing (in our RAM model). Since Algorithm 4.4 has a constant number of operations (in data complexity), inverted-access can be done in constant time (after the linear preprocessing provided by Algorithm 4.2).

The next theorem, parts of which are already given in [BB13], summarizes the algorithms presented so far.

Theorem 4.12. *Given a free-connex CQ Q and a database D, it is possible to build in linear time a data structure that allows to output the count $|Q(D)|$ in constant time and provides random-access in logarithmic time, and inverted-access in constant time.*

Example 4.13. Consider the CQ $Q(v,w,x,y,z) \leftarrow R_1(x,v,w), R_2(v,y), R_3(w,z)$ with the join-tree with R_1 as root and R_2 and R_3 as its children. The following is an example of an input for Q and the computed information available at the end of preprocessing.

R_1			w	startIndex
a_1	b_1	c_1	6	0
a_1	b_1	c_2	2	6
a_2	b_2	c_1	6	8
a_2	b_2	c_2	2	14

R_2		w	startIndex
b_1	d_1	1	0
b_1	d_2	1	1
b_2	d_2	1	0
b_2	d_3	1	1

R_3		w	startIndex
c_1	e_1	1	0
c_1	e_2	1	1
c_1	e_3	1	2
c_2	e_4	1	0

Calling Access(13) finds $(a_2, b_2, c_1) \in R_1$. Then, the remaining $13 - 8 = 5$ is split to 5 mod $3 = 2$ in the top bucket of R_3 and $\lfloor \frac{5}{3} \rfloor = 1$ in the bottom bucket of R_2. These in turn find $(b_2, d_3) \in R_2$ and $(c_1, e_3) \in R_3$. Overall, the result is $(a_2, b_2, c_1, d_3, e_3)$.

Calling InvertedAccess$(a_2, b_2, c_1, d_3, e_3)$ finds $(a_2, b_2, c_1) \in R_1$ with startIndex $= 8$. Then, calling InvertedSubtreeAccess on R_2 returns startIndex$(b_2, d_3) = 1$ from a bucket of weight 2, and calling InvertedSubtreeAccess on R_3 returns startIndex$(c_1, e_3) = 2$ from a bucket of weight 3. The call for CombineIndex$(2, 1, 3, 2)$ returns $2 + 3 \cdot 1 = 5$, and the result is $8 + 5 = 13$. □

4.2.2 Correctness

We prove the correctness of Algorithms 4.2, 4.3 and 4.4. Let T be a join tree of a full CQ Q over a globally consistent database D, and let R be a relation in Q. We denote by T_R the subtree of T rooted in R, and by join$(\mathrm{T_R})$ the join of the relations in T_R.

Claim 4.14. *Let $t \in R$. The number of answers in* join$(\mathrm{T_R})$ *that agree with t is* w(t).

Proof. We prove that $\mathrm{w}(t) = |\{a \in \mathrm{join}(\mathrm{T_R}) \mid a \text{ agrees with } t\}|$ by induction on the join tree. If R is a leaf, then join$(\mathrm{T_R}) = R$, and every tuple agrees only with itself. Therefore, $|\{a \in \mathrm{join}(\mathrm{T_R}) \mid a \text{ agrees with } t\}| = 1$. By definition of w on leaves, $\mathrm{w}(t) = 1$. This proves the induction base. We now prove the induction step. In this case, R is not a leaf. Denote its children by $R_1, \ldots, R_m$. Consider two children R_i and R_j. Since T is a join tree, it is not possible that R_i and R_j contain a variable that does not appear in R. Therefore, for every pair of answers $a_i \in \mathrm{join}(\mathrm{T_{R_i}})$ and $a_j \in \mathrm{join}(\mathrm{T_{R_j}})$, if both of them agree with t, then they also agree with each other. This means that the answers in join$(\mathrm{T_R})$ that agree with t can be obtained by independently selecting answers of the

subtrees rooted in the children of R that agree with t and combining them.

$$
\begin{aligned}
&|\{a \in \text{join}(\mathrm{T_R}) \mid a \text{ agrees with } t\}| \\
&\quad \overset{(1)}{=} \prod_{i=1}^{m} |\{a \in \text{join}(\mathrm{T_{R_i}}) \mid a \text{ agrees with } t\}| \\
&\quad \overset{(2)}{=} \prod_{i=1}^{m} \sum_{\{s \in R_i \mid s \text{ agrees with } t\}} |\{a \in \text{join}(\mathrm{T_{R_i}}) \mid a \text{ agrees with } s\}| \\
&\quad \overset{(3)}{=} \prod_{i=1}^{m} \sum_{\{s \in R_i \mid s \text{ agrees with } t\}} \mathrm{w}(s) \\
&\quad \overset{(4)}{=} \prod_{i=1}^{m} \sum_{s \in \text{bucket}[R_i,t]} \mathrm{w}(s) \overset{(5)}{=} \prod_{i=1}^{m} \mathrm{w}(\text{bucket}[R_i,t]) \overset{(6)}{=} \mathrm{w}(t)
\end{aligned}
$$

Explanations: (1) all answers of the subtrees can be combined since T is a join tree; (2) partitioning the answers by the tuple used in R_i; (3) induction assumption; (4) bucket definition; (5) bucket weight definition; (6) tuple weight definition; □

As a result of Claim 4.14, $\mathrm{w}(\text{bucket}[root, \emptyset]) = |Q(D)|$.

Let T be a join-tree. Let $R_1, \ldots, R_n$ be the DFS ordering of the nodes of T, where the order of visiting the children of each node is the same as in Algorithm 4.3 and Algorithm 4.4. We denote by parent(i) an index j such that R_j is the parent of R_i in T. Algorithm 4.5 computes $Q(D)$ with constant delay. Note that we assume that the order of tuples within each bucket is consistent between the different algorithms. That is, the for loops in Algorithm 4.5 go over the tuples in each bucket in the same order that was used to set startIndex during preprocessing.

Algorithm 4.5 Enumeration for a globally consistent full acyclic join query

```
1: procedure ENUMERATE(D, T)
2:   for t_1 in bucket[R_1, ()] do
3:     for t_2 in bucket[R_2, t_parent(2)] do
4:       ...
5:       for t_n in bucket[R_n, t_parent(n)] do
6:         output ⋃_{i=1}^{n} Atts_{R_i} → Atts_{R_i}(t_i)
```

Claim 4.15. *Algorithm 4.5 outputs $Q(D)$.*

Proof. The correctness follows from the correctness of the classic Yannakakis algorithm for acyclic join queries [Yan81b]. The semi-join reductions were performed at preprocessing (before applying Algorithm 4.2), and when Algorithm 4.5 is performed, we are left with a full acyclic join over a globally consistent database instance. Therefore, Algorithm 4.5 only needs to join the relations along the join tree. □

We consider the ordering of $Q(D)$ produced by Algorithm 4.5, and prove that Algorithm 4.3 is correct with respect to that order.

Claim 4.16. SUBTREEACCESS*(B, j) returns the jth answer of* ENUMERATE*(D, T_B).*

Proof. We prove the claim by induction on the join tree. If B is a bucket in a leaf relation R, then every tuple in B has weight 1, and startIndex corresponds to its line number within the bucket. In this case, SUBTREEACCESS(B, j) returns the tuple in line j of B, since for this tuple, startIndex $= j$. When B is a leaf, ENUMERATE(D, T_R) behaves as follows:

procedure ENUMERATE(D, T_B)
  for t in B **do**
    output $(\text{Atts}_\text{R} \to \text{Atts}_\text{R}(t))$

Therefore, the jth answer of ENUMERATE(D, T_R) is the tuple in line j of B. This concludes the induction base.

Let B be a bucket in a relation R with children $R_1, \ldots, R_m$, and let a be the answer returned by SUBTREEACCESS(B, j). We prove that a is the jth answer of ENUMERATE(D, T_B). Since the loops in Algorithm 4.5 are ordered by a DFS, the algorithm behaves as follows:

procedure ENUMERATE(D, T_B)
  for t in B **do**
    for $\{a_1 \in \text{join}(\text{T}_{\text{R}_1}) \mid a_1 \text{ agrees with } t\}$ **do**
      ...
      for $\{a_m \in \text{join}(\text{T}_{\text{R}_\text{m}}) \mid a_m \text{ agrees with } t\}$ **do**
        output $(\text{Atts}_\text{R} \to \text{Atts}_\text{R}(t)) \cup \bigcup_{i=1}^{m} a_i$

According to Claim 4.14, an iteration of the outermost loop of ENUMERATE(D, T_B) with tuple $t \in B$ prints $\text{w}(t)$ answers. Consider the iteration in which a is printed. Prior to this iteration, $\sum_{s \in P} \text{w}(s)$ answers were printed, where P is the set of tuples preceding t in B. By definition, $\text{startIndex}(t) = \sum_{s \in P} \text{w}(s)$. It is left to show that a is answer number $j - \text{startIndex}(t)$ within the outermost iteration in which it is printed. Note that line 9 of Algorithm 4.3 chooses the same t as line 2 of ENUMERATE(D, T_B) since $\text{startIndex}(t) \leq j < \text{startIndex}(t{+}1)$.

Let $j_1, \ldots, j_n$ obtained from SPLITINDEX in line 13 of Algorithm 4.3. By the induction hypothesis, SUBTREEACCESS(bucket$[R_i, t]$, j_i) returns the j_ith answer of ENUMERATE(D, $T_{\text{bucket}[R_i,t]}$). This is answer number j_i of $\{a_i \in \text{join}(\text{T}_{\text{R}_\text{i}}) \mid a_i \text{ agrees with } t\}$. Since SPLITINDEX captures the behavior of indices in nested for-loops, the returned answer is number $j - \text{startIndex}(t)$ in the current iteration of the outermost loop. □

According to Claim 4.14, $\text{w}(\text{bucket}[root, \emptyset]) = |Q(D)|$. Therefore, ACCESS($j$) recognizes out-of-bound correctly. Due to Claim 4.16, when j is not out-of-bound, ACCESS(j) returns the jth answer of ENUMERATE(D, T). Since the answers of ENUMERATE(D, T) are exactly $Q(D)$ (according to Claim 4.15), this proves that Algorithm 4.3 is a random-access algorithm for $Q(D)$. It is left to prove the correctness of the inverted-access.

Claim 4.17. *Algorithm 4.4 is an inverted-access algorithm with respect to Algorithm 4.3.*

Proof. Let a be a mapping from the variables of the query to the domain. If an answer agreeing with a cannot be obtained by a call to SubtreeAccess(B, j) with any $B \in R$ and natural number j, then due to the correctness of SubtreeAccess, this means that a does not agree with any answer in join($\mathrm{T_R}$). Therefore, for some relation $R' \in T_R$, no tuple in R' agrees with a. In this case, InvertedSubtreeAccess(R, a) will eventually call InvertedSubtreeAccess(R', a) which correctly returns not-an-answer.

Let B be a bucket in a relation R, and let j be a natural number. We now claim that if SubtreeAccess(B, j) returns a, then InvertedSubtreeAccess(R, a) returns j. We prove this claim by induction on the join-tree. If R is a leaf, then every tuple in B has weight 1, and if we denote by $t+1$ the tuple succeeding t in B, then startIndex(t+1) = startIndex(t)+w(t+1) = startIndex(t)+1 for every $t \in B$. Denote by t the tuple agreeing with a in B. Since a is the answer obtained by SubtreeAccess(B, j), then due to line 9, startIndex(t) $\leq j <$ startIndex(t+1) = startIndex(t) + 1, and so startIndex(t) is exactly j. When InvertedSubtreeAccess(R, a) is called, *offset* = 0 as R is a leaf, and so startIndex(t) = j is returned. This concludes the induction base.

Let $R_1, \ldots, R_m$ be the children of R. If SubtreeAccess(B, j) returns a, then a is the result of combining a tuple $t \in B$ with the answers $a_1, \ldots, a_m$ obtained from applying SubtreeAccess on buckets in $R_1, \ldots, R_m$ respectively. Note that line 4 of Algorithm 4.4 finds the tuple t used in Algorithm 4.3. Let $j_1, \ldots, j_m$ be the indices obtained in line 13 of Algorithm 4.3. According to the induction hypothesis, InvertedSubtreeAccess(R_i, a_i) returns j_i. Since CombineIndex is the reverse of SplitIndex, line 13 of Algorithm 4.4 sets *offset* to be $j -$ startIndex(t). InvertedSubtreeAccess(R, a) then returns $j -$ startIndex(t) + startIndex(t) = j. □

4.2.3 Dichotomy for CQs

Theorem 4.12 and Theorem 4.9 imply that the dichotomy of Theorem 4.10 extends to the problems of random permutation and random access. This also means that for self-join-free CQs, the classes of efficient two-way-access, random-access, random-permutation and enumeration collapse. This is summarized by the next corollary.

Corollary 4.18. *For every CQ Q, the following holds:*

- *If Q is free-connex, then* Enum$\langle Q \rangle$ *is in* TWAccess$\langle$lin, log$\rangle$, RAccess$\langle$lin, log$\rangle$, REnum$\langle$lin, log$\rangle$ *and* Enum$\langle$lin, const$\rangle$.
- *Otherwise, if Q is also self-join-free, then* Enum$\langle Q \rangle$ *is not in* TWAccess$\langle$lin, polylog$\rangle$, RAccess$\langle$lin, polylog$\rangle$, REnum$\langle$lin, polylog$\rangle$ *or* Enum$\langle$lin, polylog$\rangle$, *assuming the* sparseBMM *and* Hyperclique *hypotheses.*

Proof. Due to Theorem 4.12, every free-connex CQ Q is in the class TWAccess$\langle$lin, log$\rangle$, and also the number of answers to the query can be computed during the linear time preprocessing phase. By definition, this also means $Q \in$ RAccess$\langle$lin, log$\rangle$. From Theorem 4.9 we obtain that Q also is in REnum$\langle$lin, log$\rangle$. Free-connex CQs are in Enum$\langle$lin, log$\rangle$ according to Theorem 4.10 or by using Algorithm 4.5.

Due to Theorem 4.10, self-join-free non-free-connex CQs are not in $\mathsf{Enum}\langle \text{lin}, \log \rangle$ assuming sparseBMM and Hyperclique; since $\mathsf{REnum}\langle \text{lin}, \log \rangle \subseteq \mathsf{Enum}\langle \text{lin}, \log \rangle$, they are also not in $\mathsf{REnum}\langle \text{lin}, \log \rangle$. According to Theorem 4.9, these CQs are also not in $\mathsf{RAccess}\langle \text{lin}, \log \rangle$, and therefore they are not in $\mathsf{TWAccess}\langle \text{lin}, \log \rangle$. □

4.3 UCQs

In this section, we discuss the availability of random-order enumeration and random-access in UCQs. We first show that not all UCQs that admit efficient enumeration also admit efficient random-access. Then, we identify a subclass of UCQs that do allow for efficient random-permutation. In addition, we relax the delay requirements and provide an algorithm for the enumeration in random order of a union of sets, and show that the algorithm can be applied for all unions of free-connex CQs.

If several CQs are in $\mathsf{Enum}\langle \text{lin}, d \rangle$, for some d, then their union can also be enumerated within the same time bounds (see Theorem 3.1). Since our goal is to answer queries in random order, a natural question arises: does the same apply to queries in $\mathsf{RAccess}\langle \text{lin}, d \rangle$ and $\mathsf{REnum}\langle \text{lin}, d \rangle$? We show that it does not apply to CQs in $\mathsf{RAccess}\langle \text{lin}, d \rangle$. This means that for UCQs we cannot always rely on random-access to achieve an efficient random-permutation algorithm as we did for CQs. The following is an example of two free-connex CQs (therefore, each one admits efficient counting, enumeration, random-order enumeration and random-access), but we show that their union is not in $\mathsf{RAccess}\langle \text{lin}, \text{lin} \rangle$ under the hypothesis that there is no $O(m)$ time algorithm that detects whether a graph with m edges contains a triangle.

Example 4.19. Consider the UCQ $Q_\cup = Q_1 \cup Q_2$ with $Q_1(x, y, z) \leftarrow R(x, y), S(y, z)$ and $Q_2(x, y, z) \leftarrow S(y, z), T(x, z)$. Since Q_1 and Q_2 are both free-connex, we can find $|Q_1(D)|$ and $|Q_2(D)|$ in linear time by Theorem 4.12. Note that $|Q_\cup(D)| = |Q_1(D)| + |Q_2(D)| - |Q_1(D) \cap Q_2(D)|$. Therefore, $|Q_1(D) \cap Q_2(D)| > 0$ iff $|Q_\cup(D)| < |Q_1(D)| + |Q_2(D)|$.

Now let us assume that $\textsc{Enum}\langle Q_\cup \rangle \in \mathsf{RAccess}\langle \text{lin}, \text{lin} \rangle$. We can then ask the random-access algorithm for $Q_\cup$ to retrieve index number $|Q_1(D)| + |Q_2(D)|$. The algorithm will raise an out-of-bound error exactly if $|Q_\cup(D)| < |Q_1(D)| + |Q_2(D)|$. Therefore, we can check whether $Q_1(D) \cap Q_2(D) = \emptyset$ in linear time. But consider the "triangle query" $Q_\cap(x, y, z) \leftarrow R(x, y), S(y, z), T(x, z)$ and note that $Q_\cap(D) = Q_1(D) \cap Q_2(D)$ for all D. We can hence determine if the query $Q_\cap$ has answers in linear time, which contradicts the hypothesis that there is no $O(m)$ time algorithm that detects whether a graph with m edges contains a triangle. Thus, under Hyperclique, $\textsc{Enum}\langle Q_\cup \rangle \notin \mathsf{RAccess}\langle \text{lin}, \text{lin} \rangle$. □

Example 4.19 shows that (assuming Hyperclique) $\mathsf{RAccess}\langle \text{lin}, \log \rangle$ is not closed under union. It also shows that, when considering UCQs, we have that $\mathsf{Enum}\langle \text{lin}, \text{const} \rangle \not\subseteq \mathsf{RAccess}\langle \text{lin}, \text{lin} \rangle$. In particular, this means that $\mathsf{Enum}\langle \text{lin}, \log \rangle \neq \mathsf{RAccess}\langle \text{lin}, \log \rangle$, which is not the case when only considering CQs.

The source of hardness in Example 4.19 is that we cannot count the intersection efficiently. In Section 4.3.2, we show that we can obtain efficient random permutation in cases where we do not have this problem. Then, in Section 4.3.3, we show that if we relax the bound to logarithmic time *in expectation*, we can enumerate in a random-order any union comprised of free-connex CQs. Our algorithms are phrased in general as random-permutation algorithms for unions of sets. The sets are assumed to admit efficient counting, uniform sampling, membership testing, and deletion. In Section 4.3.1, we show that answers to CQs support such operations.

4.3.1 Supporting Deletion of CQ Answers

In order to use our suggested algorithms for the random-permutation of a union of sets, the sets must support counting, membership testing, sampling and deletion. We next show how to support these operations using the shuffle mechanism provided in Algorithm 4.1, assuming that the sets support efficient counting, random-access and inverted-access. We first show how to support these operations on a set of consecutive indices.

Proposition 4.20. *It is possible to support counting, testing, sampling and deletion of a set initialized as* $\{0, \ldots, n-1\}$ *with constant time per operation.*

Proof. Algorithm 4.6 describes a data structure supporting the required operations. As in Algorithm 4.1, our data structure contains an array a of length n and an integer i. Here, i corresponds to the number of elements deleted. The values $a[0], \ldots, a[i-1]$ represent the deleted elements, while $a[i], \ldots, a[n-1]$ hold the elements that remain in the set. We also use a reverse index b: whenever we set $a[i] = j$, we also set $b[j] = i$. Conceptually, a and b start initialized with $a[j] = b[j] = j$ and $i = 0$. Practically, the arrays can be implemented as lookup tables as in Algorithm 4.1. When *counting*, we return $n-i$. When *sampling*, we return $a[j]$ for a random $j \geq i$. When *deleting* k, we find the index j such that $a[j] = k$, swap $a[j]$ with $a[i]$, and increase i by one. In order to efficiently find j, we use the reverse index b. Note that all operations run with constant time and that $O(n)$ space is used.

The correctness of these procedures follows along the same lines of that of Algorithm 4.1. Denote by a_j the value $a[j]$ if it is initialized, or j otherwise. We claim that the values $a_i, \cdots, a_{n-1}$ are exactly those that were not deleted. This can be shown by induction: after initialisation, $a_0, \ldots, a_{n-1}$ represent $0, \ldots, n-1$, and no elements were deleted; when the ith element is deleted, the procedure stores in $a[i]$ the deleted value, moves the value that was there to a higher index, and increases i by one. This implies the correctness of the other operations. Counting returns the number of non-deleted elements, and testing checks whether the element is in $a_i, \cdots, a_{n-1}$. When sampling, the algorithm chooses to print uniformly at random a value between $a_i, \cdots, a_{n-1}$, so the printed answer has equal probability among all non-deleted values. □

Algorithm 4.6 Counting, testing, sampling and deletion for $0, \ldots, n-1$

```
1:  procedure INITIALIZE(n)
2:      assume a[0], ..., a[n−1] are uninitialized
3:      assume b[0], ..., b[n−1] are uninitialized
4:      i = 0
5:  procedure COUNT()
6:      output n − i
7:  procedure SAMPLE()
8:      choose j uniformly from i, ..., n−1
9:      if a[j] is uninitialized then
10:         a[j] = j; b[j] = j
11:     output a[j]
12: procedure TEST(k)
13:     if b[k] is uninitialized then
14:         b[k] = k
15:     output b[k] ≥ i
16: procedure DELETE(k)
17:     if b[k] is uninitialized then
18:         b[k] = k
19:     j = b[k]
20:     if a[i] is uninitialized then
21:         a[i] = i
22:     a[j] = a[i]
23:     a[i] = k
24:     b[a[i]] = i; b[a[j]] = j
25:     i = i + 1
```

If we have counting, random-access and inverted-access procedures for some enumeration problem, we can use Algorithm 4.6 on the indices of the answers in order to support counting, testing, sampling and deletion of the answers. This is describes in Algorithm 4.7. During initialization, we count the number of answers to our problem P, and initialize Algorithm 4.6 accordingly to obtain a data structure which we denote D. When *sampling*, we generate a non-deleted index uniformly at random from D. We then return the answer with that index using the random-access routine. When *testing*, we call the inverted-access routine and return "True" iff we obtain a non-deleted valid index. When *deleting*, we use the inverted-access routine to find the index k of the item to be deleted and then delete it from D. This proves the following lemma.

Lemma 4.21. *If an enumeration problem admits counting, random-access and inverted-access in time t, then the set of its answers also supports sampling, testing, deletion and counting in time $O(t)$.*

Since free-connex CQs admit efficient algorithms for counting, random-access and inverted-access (Theorem 4.12), we can apply Lemma 4.21 to free-connex CQs and conclude that they support sampling, testing, deletion and counting in logarithmic time

Algorithm 4.7 counting, testing, sampling and deletion for P

```
1: procedure Initialize()
2:    D.Initialize(P.Count())
3: procedure Count()
4:    output D.Count()
5: procedure Sample()
6:    output P.Access(D.Sample())
7: procedure Test(a)
8:    k =P.InvertedAccess(a)
9:    output k ≠ not-an-answer ∧ D.Test(k)
10: procedure Delete(k)
11:   k =P.InvertedAccess(a)
12:   if k ≠ not-an-answer then
13:      D.Delete(k)
```

(after linear preprocessing).

4.3.2 UCQs that Allow for Random-Permutation

We describe a random-permutation algorithm for a union of sets where each set supports operations efficiently and the size of the intersection is known. This algorithm can be used for unions of tractable CQs with tractable intersections and guarantees a logarithmic bound on the delay. We first show an algorithm for a union of two sets.

Lemma 4.22. *Let S_1 and S_2 be sets, each supports sampling, testing, deletion and counting in time t. If we can also count $S_1 \cap S_2$ in time t, then it is possible to enumerate $S_1 \cup S_2$ in uniformly random order with $O(t)$ delay.*

Algorithm 4.8 enumerates the union of two sets in uniformly random order. It iteratively samples an element from the union uniformly at random, then deletes the element and repeats. Lines 3 and 4 choose a random set and a random element it contains, where the choice of set is weighted by the number of elements it contains. In line 4, an element that appears in the two sets has twice the probability of being chosen compared to an element that appears in only one set. The rest of the algorithm corrects this bias by sampling twice and then randomly choosing between the two sampled elements. If one of the sampled elements appears in both sets and the other does not, the probabilities are set such that we are more likely to choose the latter. Note that the size of the union can easily be computed since $|S_1 \cup S_2| = |S_1| + |S_2| - |S_1 \cap S_2|$.

We now prove that Algorithm 4.8 prints the results in a uniformly random order. First note that for every $i \in \{1, 2\}$ and $e \in S_i$, we have that $(\mathit{chosen}, \mathit{element}) = (e, i)$ with probability $\frac{1}{|S_1|+|S_2|} = \frac{|S_1|}{|S_1|+|S_2|}\frac{1}{|S_1|} = \frac{|S_2|}{|S_1|+|S_2|}\frac{1}{|S_2|}$. Thus, for every $e \notin S_1 \cap S_2$, the probability of $\mathit{element} = e$ is $\frac{1}{|S_1|+|S_2|}$, while for $e \in S_1 \cap S_2$ this probability is $\frac{2}{|S_1|+|S_2|}$.

We first show that for every element $e \notin S_1 \cap S_2$, the probability of printing e in a given iteration is $\frac{1}{|S_1 \cup S_2|}$. There are three cases that can lead to printing e. The

Algorithm 4.8 Random-order enumeration of $S_1 \cup S_2$ given the intersection size

while $|S_1| + |S_2| > 0$ **do**
  for $k \in \{1, 2\}$ **do**
    $chosen_k$ = choose $i \in \{1, 2\}$ with probability $\frac{|S_i|}{|S_1|+|S_2|}$
    $element_k = S_{chosen}$.SAMPLE()
  if $element_1 \in S_1 \cap S_2$ and $element_2 \notin S_1 \cap S_2$ **then**
    $p_1 = \frac{|S_1 \cup S_2| - |S_1 \cap S_2|}{4|S_1 \cup S_2|}$; $p_2 = 1 - p_1$
  else if $element_1 \notin S_1 \cap S_2$ and $element_2 \in S_1 \cap S_2$ **then**
    $p_2 = \frac{|S_1 \cup S_2| - |S_1 \cap S_2|}{4|S_1 \cup S_2|}$; $p_1 = 1 - p_2$
  else
    $p_1 = 1$; $p_2 = 0$
  $element$ = choose $element_i$ with probability p_i
  S_1.DELETE($element$); S_2.DELETE($element$); **output** $element$

first case is that $e = element_1$ and $element_2 \notin S_1 \cap S_2$, and then e is printed with probability 1. The probability of this case is $\frac{1}{|S_1|+|S_2|}$ for selecting e as the first element multiplied by $(|S_1 \cup S_2| - |S_1 \cap S_2|)\frac{1}{|S_1|+|S_2|}$ for selecting the second element not from the intersection. The second case is that $e = element_1$ again, but $element_2 \in S_1 \cap S_2$. In this case, e is printed with probability $\frac{3|S_1 \cup S_2| + |S_1 \cap S_2|}{4|S_1 \cup S_2|}$. The probability of the second case is $\frac{1}{|S_1|+|S_2|}$ for selecting e as the first element multiplied by $|S_1 \cap S_2|\frac{2}{|S_1|+|S_2|}$ for selecting the second element from the intersection. The second case is that $e = element_2$, and $element_1 \in S_1 \cap S_2$. This case has the same probability as the second case. Overall, the probability of choosing e is:

$$\begin{aligned}
&\frac{1}{|S_1| + |S_2|}(|S_1 \cup S_2| - |S_1 \cap S_2|)\frac{1}{|S_1| + |S_2|} \\
&\qquad + 2\frac{1}{|S_1| + |S_2|}|S_1 \cap S_2|\frac{2}{|S_1| + |S_2|}\frac{3|S_1 \cup S_2| + |S_1 \cap S_2|}{4|S_1 \cup S_2|} \\
&= \frac{1}{(|S_1| + |S_2|)^2}(|S_1 \cup S_2| - |S_1 \cap S_2| + |S_1 \cap S_2|\frac{3|S_1 \cup S_2| + |S_1 \cap S_2|}{|S_1 \cup S_2|}) \\
&= \frac{1}{(|S_1| + |S_2|)^2}\frac{(|S_1 \cup S_2| + |S_1 \cap S_2|)^2}{|S_1 \cup S_2|} = \frac{1}{|S_1 \cup S_2|}
\end{aligned}$$

In total, the probability of printing an element that is not in the intersection is $\frac{|S_1 \cup S_2| - |S_1 \cap S_2|}{|S_1| \cup |S_2|}$. This means that the probability of printing an element from the intersection is $1 - \frac{|S_1 \cup S_2| - |S_1 \cap S_2|}{|S_1| \cup |S_2|} = \frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}$. Since all elements in the intersection are treated equally in Algorithm 4.8, we have that, given $e \in S_1 \cap S_2$, the probability of choosing e is $\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|}\frac{1}{|S_1 \cap S_2|} = \frac{1}{|S_1 \cup S_2|}$.

Overall, we showed that, in every iteration, all elements have probability $\frac{1}{|S_1 \cup S_2|}$ of being printed. A printed element is deleted from all sets containing it, so it will not be printed twice. Thus, Algorithm 4.8 generates the elements in uniformly random order. Regarding time complexity, as every iteration prints after a constant number of operations, each taking $O(t)$ time, the delay is bounded by $O(t)$. This proves Lemma 4.22.

Next, we generalize Lemma 4.22 to a union of an arbitrary number of sets $S_1, \ldots, S_k$.

Lemma 4.23. *Let $S_1, \ldots, S_k$ be sets, each supports sampling, testing, deletion and counting in time t. If for every $I \subseteq [1, m]$ we can also count $\bigcap_{i \in I} S_i$ in time $|I|t$, then it is possible to enumerate $\bigcup_{i \in [k]} S_i$ in uniformly random order with $O(2^k t)$ delay.*

We prove that we can sample a union of k of these sets uniformly in time $O(2^k t)$ by induction on k. Then, we can repeatedly sample and delete the sampled element to obtain a random permutation. When $k = 1$, we can sample S_1 in time $O(t)$. When $k > 1$, we use one iteration of Algorithm 4.8 for the induction step. We apply this algorithm with the union of the first $\lfloor \frac{k}{2} \rfloor$ sets as one set and the union of the other sets as the second. We proved that this samples the union uniformly at random correctly. We now discuss the time complexity. The two sampling operations take $O(2^k t)$ time together by the induction assumption. Testing membership in the intersection requires testing all sets, which takes $O(kt)$ time. Computing the sizes of the sets and their intersections can be done via the inclusion-exclusion principle in time $O(2^k t)$. Overall, an iteration of Algorithm 4.8 contains a constant number of operations, each taking $O(2^k t)$ time. This proves Lemma 4.23.

Since free-connex CQs admit sampling, testing, deletion and counting in logarithmic time, we can apply Lemma 4.23 to UCQs and prove the following theorem.

Theorem 4.24. *If Q is a union of free-connex CQs where the intersection of every subset of the CQs is also free-connex, then* $\textsc{Enum}\langle Q \rangle \in \mathsf{REnum}\langle \mathrm{lin}, \log \rangle$.

Theorem 4.12 and Lemma 4.21 show that the individual free-connex CQs support sampling, testing, deletion and counting in logarithmic time. The sizes of the free-connex intersections can be efficiently computed and stored during preprocessing and updated upon deletion. Thus, we can use Lemma 4.23 to prove Theorem 4.24.

Remark. In this section, we proved that a union of tractable CQs with tractable intersections admits efficient random permutation. If we add the requirement that the individual CQs support random access in compatible orders, we obtain the class of mc-UCQs that also admits efficient random access. This discussion is beyond the scope of this thesis, and additional details can be found in the conference paper [CZB+20].

4.3.3 Random-Permutation with Expected Logarithmic Delay

Theorem 4.24 does not apply to all unions of free-connex CQs. In this section, we show that if we relax the bound to logarithmic time *in expectation*, we can enumerate in a random-order the answers to any such UCQ. Here again we use the abstraction of a union of sets that support sampling, testing, deletion and counting. If the number of sets in the union is constant, the algorithm carries the guarantees of expected and amortized constant number of such operations between every pair of successively printed answers. The algorithm is an adaptation of the sampling algorithm by Karp and Luby [KLM89]

Algorithm 4.9 Random-order enumeration of $S_1 \cup \cdots \cup S_k$

1: **while** $\sum_{j=1}^{k} |S_j| > 0$ **do**
2: $\quad$ *chosen* = choose i with probability $\frac{|S_i|}{\sum_{j=1}^{k} |S_j|}$
3: $\quad$ *element* = S_{chosen}.SAMPLE()
4: $\quad$ *providers* = $\{S_j \mid S_j.\text{TEST}(element) = \mathsf{true}\}$
5: $\quad$ *owner* = $\min\{j \mid S_j \in providers\}$
6: $\quad$ **for** $S_j \in providers \setminus \{S_{owner}\}$ **do**
7: $\quad\quad$ S_j.DELETE(*element*)
8: $\quad$ **if** $S_{owner} = S_{chosen}$ **then**
9: $\quad\quad$ S_{chosen}.DELETE(*element*) ; **output** *element*

extended by tuple deletions that allow for sampling without repetitions. We prove the following lemma.

Lemma 4.25. *Let $S_1, \ldots, S_k$ be sets, each supports sampling, testing, deletion and counting in time t. Then, it is possible to enumerate $\bigcup_{j=1}^{k} S_j$ in uniformly random order with expected $O(kt)$ delay.*

Similarly to Algorithm 4.8, Algorithm 4.9 chooses a random set and a random element it contains. Here again, if the algorithm printed the element at that stage (after line 3), then an element that appears in two sets would have twice the probability of being chosen compared to an element that appears in only one set. The following lines correct this bias differently than Algorithm 4.8. We denote by *providers* all sets that contain the chosen element. Then, the algorithm assigns one owner to this element out of its providers (as the choice of the owner is not important, we arbitrarily choose to take the provider with the minimum index). The element is then deleted from non-owners, and is printed only if the algorithm chooses its owner in line 2. If the element was reached through a non-owner, then the current iteration "rejects" by printing nothing.

Algorithm 4.9 prints the results in a uniformly random order since, in every iteration, every answer remaining in the union has equal probability of being printed. Denote by *Choices* the set of all possible (*chosen*, *element*) pairs that the algorithm may choose in lines 2 and 3. The probability of such a pair is $\frac{|S_{chosen}|}{\sum_{j=1}^{k} |S_j|} \frac{1}{|S_{chosen}|} = \frac{1}{\sum_{j=1}^{k} |S_j|}$, which is the same for all pairs in *Choices*. Denote by *AccChoices* $\subseteq$ *Choices* the pairs for which S_{chosen} is the owner of *element*. Line 8 guarantees that an element is printed only when the selections the algorithm makes are in *AccChoices*. Since every possible answer only appears once as an element in *AccChoices*, the probability of each element to be printed is $\frac{1}{\sum_{j=1}^{k} |S_j|}$. Therefore, all answers have the same probability of being printed. Note, however, that the sum of these probabilities does not necessarily add up to one, and with probability $\frac{|\bigcup_{j=1}^{k} S_j|}{\sum_{j=1}^{k} |S_j|}$ the iteration does not print any answer. A printed answer is deleted from all sets containing it, so it will not be printed twice.

We now discuss the time complexity. If some iteration rejects an answer, this iteration also deletes it from all non-owner sets. This guarantees that each unique

answer will only be rejected once, as it only has one provider in the second time it is seen. This means that the total number of iterations Algorithm 4.9 performs is bounded by twice the number of answers. The number of operations between successive answers is therefore amortized constant. In addition, since by definition $|Choices| \leq k|AccChoices|$, in every iteration the probability that an answer will be printed is $\frac{|AccChoices|}{|Choices|} \geq \frac{1}{k}$. The delay between two successive answers therefore comprises of a constant number of operations both in expectation and in amortized complexity. This proves Lemma 4.25.

By combining Theorem 4.12 with Lemma 4.25 and Lemma 4.21, we can answer unions of free-connex CQs with random order. We get the following result.

Theorem 4.26. *Let Q be a union of free-connex CQs. There is a random-permutation algorithm for answering Q that uses linear preprocessing and expected logarithmic delay.*

4.4 Note on Space Usage

Our random permutation solution for CQs consists of two components: shuffle and random access. The random access solution we propose uses only a constant amount of registers in the access phase. However, the shuffle solution we propose has higher space requirements. If there are n answers, Algorithm 4.1 uses $O(n)$ registers. Recall that n may be larger than the input. Next, we inspect whether there exist solutions that use less space.

Proposition 4.27. *Any random-permutation algorithm for a problem with n answers must use at least n bits.*

Proof. For any subset of the answers, a correct random-permutation algorithm may reach a state where exactly these answers were printed. The algorithm must distinguish the 2^n possible states corresponding to the printed answers in order to guarantee printing every answer exactly once. Distinguishing 2^n states requires at least n bits. □

The $O(n)$ registers required by our solution correspond to $O(n \log n)$ bits in the model we use where every register is assumed to have $\Theta(\log n)$ bits. Proposition 4.27 means that, despite using a substantial amount of space, the space complexity of our solution is only a log factor away from optimal. We note that to store a permutation of n elements, we need $\log(n!) = \Theta(n \log n)$ bits. Therefore, any solution that uses only $O(n)$ bits must avoid keeping the information of the order in which the answers were printed, and instead store only which answers were printed. We next propose an alternative to Algorithm 4.1 that requires no more than the optimal amount of space.

We propose using a bitset: $\frac{n}{\log n}$ registers where each bit is on iff the answer of this index was printed. In order to search these registers efficiently, we store them as leaves in a complete binary search tree where each inner node stores the amount of off bits in the leaves of its subtree. Given such a tree, testing and deletion can be done by directly accessing the desired bit, counting corresponds to the number of off bits stored in the

root, and sampling can be done as follows. Draw a random index i bounded by the count, and follow a path from the root to the relevant leaf: go right iff i is larger than the amount of off bits in the left child. If you go right, subtract this number of off bits from i. When reaching the leaf, return the index of the ith off bit. The number of nodes in such a tree is $O(\frac{n}{\log n})$, and searching for the ith off bit can be done in time proportional to the depth of the tree, $O(\log n)$ time. If we construct this tree lazily and initialize nodes only as they are used, the initialization can take constant time. We conclude the following result.

Lemma 4.28. *The set* $0, \ldots, n-1$ *supports sampling, testing, deletion and counting in* $O(\log n)$ *delay and* $O(n)$ *bits.*

The solution presented here is arguably less elegant than the $O(1)$ delay solution based on the Fisher-Yates shuffle. Nevertheless, in our problem, the delay is logarithmic in both cases due to the random-access, and the bit-set solution enjoys reduced space requirements. Repeatedly performing sampling and deletion results in random permutation. By replacing *Algorithm* 4.1 with the solution we propose here, we obtain a random permutation algorithm for free-connex CQs with the same time guarantees (linear preprocessing and logarithmic delay) and optimal space consumption. By applying this same solution in Lemma 4.21, we also obtain our results for UCQs with optimal space consumption.

Theorem 4.29. *Algorithms 4.9 and 4.8 can be realized with optimal space consumption.*

Chapter 5

Exploiting Functional Dependencies for Query Answering

In the previous chapters, we discussed the enumeration complexity of UCQs when we have no information on the input data other than the relations and their arities. In this chapter, we inspect how the characterizations of the previous chapters change in the common case that the possible values in the relations are restricted. We show that queries that are intractable over general schemas may become tractable in the presence of dependencies, and we strive for lower bounds that show that our definitions cover all tractable cases. Throughout most of this chapter, the queries we discuss are CQs and the constraints are functional dependencies. At the end of this chapter, we discuss how to extend these results to cardinality dependencies, CQs with disequalities, and UCQs.

This chapter contains joint work with Markus Kröll. The findings of this chapter were published in the International Conference on Database Theory [CK18], and invited for a special issue of Theory of Computing Systems (TOCS) Journal on selected papers from ICDT 2018 [CK19a].

Organization. In Section 5.1, we define an extension of a CQ based on the FDs, and show that if the extension is free-connex, the original CQ can be efficiently solved. In Section 5.2, we show results of the opposite direction: if the extension is not free-connex, than the original CQ is intractable. This is shown for acyclic CQs in general, and for cyclic CQs only when the FDs are unary. Section 5.3 extends the results beyond CQs and functional dependencies.

5.1 FD-Extensions

We define an extension of a CQ based on FDs in Section 5.1.1. In Section 5.1.2, We discuss the equivalence between Q and Q^+ with respect to enumeration. As a result,

we obtain in Section 5.1.3 that if Q^+ is in a class of queries that allows for tractable enumeration, then Q is tractable as well.

5.1.1 Definition and Structure

In this section, we formally define the FD-extension Q^+ of a CQ Q. Then, we discuss the possible structural differences between Q and Q^+. Intuitively, the extension is based on treating the FDs as dependencies between variables. Assume that a variable x implies a variable y via an FD. Given an assignment for x in some answer to the query, we know the unique assignment of y according to R. Therefore, we can add the y value to every tuple that has an x value. Note however that when we extend the relations with these known values, the data still conforms to the FDs. Thus, the process of extension does not remove the FDs. Instead, it ensures that the extended query holds in its structure the added value that the FDs provide.

The *extension* of an atom $R(\vec{v})$ according to an FD $S\colon A \to b$ and an atom $S(\vec{u})$ is possible if $\vec{u}[A] \subseteq \vec{v}$ but $\vec{u}[b] \notin \vec{v}$. In this case, $\vec{u}[b]$ is added to the variables of R. The *FD-extension* of a query is defined by iteratively extending all atoms as well as the head according to every possible dependency in the schema, until a fixpoint is reached. The schema extends accordingly: the arities of the relations increase as their corresponding atoms extend, and the FDs apply in every relation that contains all relevant variables. Dummy variables are added to adjust to the change in arity in case of self-joins.

Definition 5.1 (FD-Extension)**.** Let $Q(\vec{p}) \leftarrow R_1(\vec{v_1}), \ldots, R_m(\vec{v_m})$ be a CQ over a schema $\mathcal{S} = (\mathcal{R}, \Delta)$. We define two types of extension steps:

- The extension of an atom $R_i(\vec{v_i})$ according to an FD $R_j\colon A \to b$.
  Prerequisites: $\vec{v_j}[A] \subseteq \vec{v_i}$ and $\vec{v_j}[b] \notin \vec{v_i}$.
  Effect: The arity of R_i increases by one, $R_i(\vec{v_i})$ is replaced by $R_i(\vec{v_i}, \vec{v_j}[b])$. In addition, every $R_k(\vec{v_k})$ such that $R_k{=}R_i$ and $k \neq i$ is replaced with $R_k(\vec{v_k}, t_k)$, where t_k is a fresh variable in every such step.
- The extension of the head $Q(\vec{p})$ according to an FD $R_j\colon A \to b$.
  Prerequisites: $\vec{v_j}[A] \subseteq \vec{p}$ and $\vec{v_j}[b] \notin \vec{p}$.
  Effect: The head is replaced by $Q(\vec{p}, \vec{v_j}[b])$.

The *FD-extension* of Q is the query $Q^+(\vec{q}) \leftarrow R_1^+(\vec{u_m}), \ldots, R_m^+(\vec{u_m})$, obtained by performing all possible extension steps on Q according to FDs of Δ until a fixpoint is reached. The extension is defined over the schema $\mathcal{S}^+ = (\mathcal{R}^+, \Delta_{Q^+})$, where $\mathcal{R}^+$ is $\mathcal{R}$ with the extended arities, and Δ_{Q^+} is:

$$\{R_i^+\colon C \to d \mid \exists R_i^+(\vec{u_i}) \in \text{atoms}(Q^+), \exists R_j(\vec{v_j}) \in \text{atoms}(Q), \exists (R_j\colon A \to b) \in \Delta,$$
$$\text{s.t. } \vec{u_i}[C] = \vec{v_j}[A] \text{ and } \vec{u_i}[d] = \vec{v_j}[b]\}.$$

Given a query, its FD-extension is unique up to a permutation of the added variables and renaming of the new variables. As the order of the variables and the naming make no difference w.r.t. enumeration, we can treat the FD-extension as unique.

Since our method treats the FDs as between variables, for simplicity, we sometimes denote the FDs accordingly. If an FD δ has the form $R : A \to b$ for $A = \{a_1, \ldots, a_{|A|}\}$ and the query has an atom $R(\vec{v})$, we sometimes denote δ by $R : \{\vec{v}[a_1], \ldots, \vec{v}[a_{|A|}]\} \to \vec{v}[b]$. To help distinguish these two representations, we usually denote integers by lower case letters from the beginning of the alphabet, while variables take letters from the end of the alphabet. Sets are usually denoted by capital letters.

Example 5.2. Consider a schema with $\Delta = \{R_1 \colon 1 \to 2, R_3 \colon 2, 3 \to 1\}$, and the query $Q(x) \leftarrow R_1(x, y), R_2(x, z), R_2(u, z), R_3(w, y, z)$. As the FDs are $x \to y$ and $yz \to w$, the FD-extension is $Q^+(x, y) \leftarrow R_1^+(x, y), R_2^+(x, z, y, w), R_2^+(u, z, t_1, t_2), R_3^+(w, y, z)$. We first apply $x \to y$ on the head, and then $x \to y$ and consequently $yz \to w$ on $R_2(x, z)$. These two FDs are now in the schema also for R_2, and the FDs of the extension are $\Delta_{Q^+} = \{R_1^+ \colon 1 \to 2, R_2^+ \colon 1 \to 3, R_2^+ \colon 3, 2 \to 4, R_3^+ \colon 2, 3 \to 1\}$. □

We later show that the enumeration complexity of a CQ Q over a schema with FDs only depends on the structure of Q^+, which is implicitly given by Q and its schema. Therefore, we introduce the notions of acyclic and free-connex queries for FD-extensions:

Definition 5.3. Let Q be a CQ over a schema $(\mathcal{R}, \Delta)$, and let Q^+ be its FD-extension.

- We say that Q is *FD-acyclic* if Q^+ is acyclic.
- We say that Q is *FD-free-connex* if Q^+ is free-connex.
- We say that Q is *FD-cyclic* if Q^+ is cyclic.

The following proposition shows that the classes of acyclic queries and free-connex queries are both closed under constructing FD-extensions.

Proposition 5.4. *Let Q be a CQ over a schema $(\mathcal{R}, \Delta)$.*

- *If the query Q is acyclic, then it is FD-acyclic.*
- *If the query Q is free-connex, then it is FD-free-connex.*

Proof. We prove that if Q is acyclic, then Q^+ is also acyclic, by constructing a join-tree of $\mathcal{H}(Q^+)$ given one of $\mathcal{H}(Q)$. The same proof can be applied to a join tree containing the head to show that if Q is free-connex, then so is Q^+. Denote by $Q = Q_0, Q_1, \ldots, Q_n = Q^+$ a sequence of queries such that Q_{i+1} is the result of extending all possible relations of Q_i according to a single FD $\delta \in \Delta$. By induction, it suffices to show that if $\mathcal{H}(Q_i)$ has a join tree, then $\mathcal{H}(Q_{i+1})$ has one too. So consider an acyclic query $Q_i(\vec{p}) \leftarrow R_1(\vec{v}_1), \ldots, R_m(\vec{v}_m)$ extended to the query $Q_{i+1}(\vec{q}) \leftarrow R_1(\vec{u}_1), \ldots, R_m(\vec{u}_m)$ according to the FD $\delta = R_j \colon \vec{x} \to y$, and let $T_i = (V_i, E_i)$ be a join tree of $\mathcal{H}(Q_i)$. We claim that the same tree (but with the extended atoms), is a join tree for Q_{i+1}. Formally, define $T_{i+1} = (V_{i+1}, E_{i+1})$ such that $V_{i+1} = \{R_k(\vec{u}_k) \mid 1 \leq k \leq m\}$ and $E_{i+1} = \{(R_k(\vec{u}_k), R_l(\vec{u}_l)) \mid (R_k(\vec{v}_k), R_l(\vec{v}_l)) \in E_i\}$. Next we show that the running intersection property holds in T_{i+1}, and therefore it is a join tree of $\mathcal{H}(Q_{i+1})$.

Every new variable introduced in the extension appears only in one atom, so the subtree of T_{i+1} containing such a variable contains one node and is trivially connected.

For any other variable $w \neq y$, the attribute w appears in the same atoms in Q and Q^+. Therefore, the subgraph of T_{i+1} containing w is isomorphic to the subgraph of T_i containing w, and since T_i is a join tree, it is connected. It is left to show that the subtree of T_{i+1} containing y is connected. Since R_j is an atom in Q containing δ, it corresponds to vertices in T_i and T_{i+1} containing $\vec{x} \cup \{y\}$. Let R_k be some vertex in T_{i+1} containing y. We will show that all vertices $S_1, \ldots, S_r$ on the path between R_k and R_j contain y. If y appears in the vertex R_k in T_i, then it also appears in $S_1, \ldots, S_r$ since T_i is a join tree. Since the extension does not remove occurrences of variables, y appears in these vertices in T_{i+1} as well. Otherwise, y was added to R_k via δ, so R_k contains $\vec{x}$. Since T_i is a join tree, the vertices $S_1, \ldots, S_r$ all contain the variables $\vec{x}$. Thus by the definition of Q_{i+1}, y is added to each of $S_1, \ldots, S_r$ (if it was not already there) in T_{i+1}. Thus also the subtree of T_{i+1} containing y is connected. Therefore T_{i+1} is indeed a join tree. □

Example 1.3 shows that the converse of the proposition above does not hold. This means that, by Theorem 2.1, there are queries Q such that evaluating Q^+ is in $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const} \rangle$, but evaluating Q cannot be done with the same complexity if we do not assume the FDs. The next section shows that, when relying on the FDs, evaluating Q^+ is equally hard to evaluating Q.

5.1.2 Enumeration Complexity

In this section, we show the equivalence between Q and Q^+ with respect to complexity classes that are closed under exact reductions. We prove the following theorem.

Theorem 5.5. *Let Q be a CQ over a schema $(\mathcal{R}, \Delta)$, and let Q^+ be its FD-extension. Then,* $\textsc{Enum}_{\Delta}\langle Q \rangle \equiv_e \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+ \rangle$.

Proof. Consider a query $Q(\vec{p}) \leftarrow R_1(\vec{v}_1), \ldots, R_m(\vec{v}_m)$ and its FD-extension $Q^+(\vec{q}) \leftarrow R_1^+(\vec{u}_1), \ldots, R_m^+(\vec{u}_m)$. We show the two parts of the equivalence.

Claim 5.6. $\textsc{Enum}_{\Delta}\langle Q \rangle \leq_e \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+ \rangle$.

Construction. Given an instance I for $\textsc{Enum}_{\Delta}\langle Q \rangle$, we construct an instance $\sigma(I)$ for $\textsc{Enum}_{\Delta_{Q^+}}\langle Q^+ \rangle$ with two phases: cleaning and extension. In the cleaning phase, we remove tuples that interfere with the extended dependencies. For every dependency $\delta = R_j \colon X \to y$ and every atom $R_k(\vec{v}_k)$ that contains the corresponding variables (i.e., $X \cup \{y\} \subseteq \vec{v}_k$), we correct R_k according to δ: we only keep tuples of R_k^I that agree with some tuple of R_j^I over the values of $X \cup \{y\}$. We denote the cleaned instance by I_0. The cleaning phase can be done in linear time by first sorting both R_j^I and R_k^I according to $X \cup \{y\}$, and then performing one scan over both of them. Next, we perform the extension phase. We follow the extension of the schema as described in Definition 5.1 and extend the instance accordingly. This phase results in a sequence of instances

$I_0, I_1, \ldots, I_n = \sigma(I)$ that correspond to a sequence of queries $Q = Q_0, Q_1, \ldots, Q_n = Q^+$ such that each query is the result of extending an atom or the head of the previous query according to an FD. If in step i the head was extended, we set $I_{i+1} = I_i$. Now assume some relation R_k is extended according to some FD $R_j \colon X \to y$. For each tuple $t \in R_k^{I_i}$, if there is no tuple $s \in R_j^{I_i}$ that agrees with t over the values of X, then we remove t altogether. Otherwise, we copy t to $R_k^{I_{i+1}}$ and assign y with the same value that s assigns it. The extension phase takes linear time for each step. Since the number of FDs is constant in data complexity, the overall construction takes linear time. Note that this construction ensures that the extended dependencies hold in $\sigma(I)$. Given an answer $\mu|_{\text{free}(Q^+)} \in Q^+(\sigma(I))$, we set $\tau(\mu) = \mu|_{\text{free}(Q)}$. This projection only requires constant time.

Correctness. We now show that $Q(I) = \{\mu|_{\text{free}(Q)} : \mu|_{\text{free}(Q^+)} \in Q^+(I^+)\}$. First, if $\mu|_{\text{free}(Q^+)}$ is an answer of $Q^+(I^+)$, then μ is a homomorphism from Q^+ to I^+. Since all tuples of I^+ appear (perhaps projected) in I, then μ is also a homomorphism from Q to I, and $\mu|_{\text{free}(Q)} \in Q(I)$. It is left to show the opposite direction: if $\mu|_{\text{free}(Q)} \in Q(I)$ then $\mu|_{\text{free}(Q^+)} \in Q^+(I^+)$. We show by induction on $Q = Q_0, Q_1, \ldots, Q_n = Q^+$ that $\mu|_{\text{free}(Q_i)} \in Q_i(I_i)$. The induction base holds since in the cleaning phase we did not remove "useful" tuples. Since $\mu|_{\text{free}(Q)} \in Q(I)$, there exist tuples, one of each relation of the query, that agree on the values of $X \cup \{y\}$ (they all assign them with the values μ assigns them). These tuples were not removed in the cleaning phase, and therefore $\mu|_{\text{free}(Q)} \in Q(I_0)$. Next assume that $\mu|_{\text{free}(Q_i)} \in Q_i(I_i)$, and we want to show that $\mu|_{\text{free}(Q_{i+1})} \in Q_{i+1}(I_{i+1})$. This claim is trivial in case the head was extended. Note also that there cannot be two distinct answers $\mu|_{\text{free}(Q_{i+1})}$ and $\mu'|_{\text{free}(Q_{i+1})}$ in $Q_{i+1}(I_{i+1})$ such that $\mu|_{\text{free}(Q_i)} = \mu'|_{\text{free}(Q_i)}$, as the added variable is bound by the FD to have only one possible value. Now consider the case where an atom $R_k(\vec{v}_k)$ was extended according to an FD $R_j \colon X \to y$ since $X \subseteq \vec{v}_k$. The tuple $\mu(\vec{v}_k) \in R_k^{I_i}$ was extended with the value $\mu(y)$ due to the tuple $\mu(\vec{v}_j) \in R_j^{I_i}$ that agrees with it on the values of X, and so $\mu(\vec{v}_k, y) \in R_k^{I_{i+1}}$. In case of self-joins, other atoms with the relation R_k are extended with a new and distinct variable. Such variables will be mapped to this value $\mu(y)$ as well. Overall, we have that μ (extended by mappings of the fresh variables) is also a homomorphism in $Q_{i+1}(I_{i+1})$.

Claim 5.7. $\textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle \leq_e \textsc{Enum}_{\Delta}\langle Q\rangle$.

Construction. Given an instance I^+ for $\textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$, we construct an instance $\sigma(I^+)$ for $\textsc{Enum}_{\Delta}\langle Q\rangle$ with three phases: cleaning, building a lookup table and projection. In the cleaning phase, we remove tuples that do not contribute to the answer set of Q^+ in order to prevent additional answers from appearing in Q after the projection. This can be seen as unifying the restrictions of different FDs in Δ_{Q^+} that originate in the same FD in Δ. For every FD $R_j \colon X \to y$ in Δ and every atom $R_k^+(\vec{u}_k)$ such that $X \cup \{y\} \subseteq \vec{u}_k$, we remove all tuples $t \in R_k^{+I^+}$ that agree with some tuple $s \in R_j^{+I^+}$ over

X but disagree with s over y. The cleaning phase can be done in linear time by first sorting both $R_k^{+I^+}$ and $R_j^{+I^+}$ according to X. Next, we construct a lookup table T to later reconstruct the assignments to $\text{free}(Q^+) \setminus \text{free}(Q)$. For every $y \in \text{free}(Q^+) \setminus \text{free}(Q)$ added to the head due to an FD $R_j \colon X \to y$, denote by $\vec{x}$ a vector containing the variables of X in lexicographic order. For every tuple in $R_j^{+I^+}$ that assigns y and $\vec{x}$ with the values y_0 and $\vec{x}_0$ respectively, we set $T(\vec{x}, \vec{x}_0, y) = y_0$. Note that due to the FD, a key cannot map to two different values. We conclude the construction by projecting the relations of I^+ according to the schema of Q. These steps result in the construction of an instance $\sigma(I^+)$ and a lookup table T in linear time. Note that Δ hold in $\sigma(I^+)$ since Δ_{Q^+} contains them.

Given $\mu|_{\text{free}(Q)} \in Q(I)$, we define $\tau(\mu|_{\text{free}(Q)}) = \mu|_{\text{free}(Q)} \cup \nu_\mu$, where the mapping $\nu_\mu : \text{free}(Q^+) \setminus \text{free}(Q) \to \mathit{dom}$ uses the lookup table: For every $y \in \text{free}(Q^+) \setminus \text{free}(Q)$ added due to some FD $R_j \colon X \to y$, we set $\nu_\mu(y) = T[(\vec{x}, \mu(\vec{x}), y)]$. Note that τ is computable in constant time since we use the lookup table $|\text{free}(Q^+) \setminus \text{free}(Q)|$ times, and each access takes constant time.

Correctness. We first claim that the lookup table succeeds in reconstructing the values for the missing head variables: if $\mu|_{\text{free}(Q)} \in Q(\sigma(I^+))$, then $\tau(\mu|_{\text{free}(Q)}) = \mu|_{\text{free}(Q^+)}$. By definition, for every $y \in \text{free}(Q)$, $\tau(\mu(y)) = \mu(y)$. We need to show the same for $y \in \text{free}(Q^+) \setminus \text{free}(Q)$. In this case, y was added to the head due to some FD $R_j \colon X \to y$, and $\tau(\mu(y))$ is defined to be $\nu_\mu(y) = T[(\vec{x}, \mu(\vec{x}), y)]$. Since μ is a homomorphism into $\sigma(I^+)$, there exists some tuple in $R_j^{\sigma(I^+)}$ that assigns $\vec{x}$ and y with $\mu(\vec{x})$ and respectively $\mu(y)$. This tuple is a projection of a tuple in $R_j^{I^+}$ that assigns $\vec{x}$ and y with the same values. Due to this tuple, when constructing the lookup table, we set $T[(\vec{x}, \mu(\vec{x}), y)] = \mu(y)$.

We now show that $Q^+(I^+) = \{\mu|_{\text{free}(Q^+)} : \mu|_{\text{free}(Q)} \in Q(\sigma(I^+))\}$. We start by showing that if $\mu|_{\text{free}(Q^+)} \in Q^+(I^+)$, then $\mu|_{\text{free}(Q)} \in Q(\sigma(I^+))$. Let μ be a homomorphism from the variables of Q^+ to I^+. Then, for every atom $R_i^+(\vec{u}_i)$ of Q^+, we have that $R_i^{+I^+}$ contains the tuple $\mu(\vec{u}_i)$, and these tuples all agree on all variables of Q^+. In particular, for every FD $X \to y$ in Δ_{Q^+}, these tuples agree on the y value. Therefore, none of these tuples are removed in the cleaning phase when constructing $\sigma(I^+)$. After projecting the cleaned relations of I^+ into those of $\sigma(I^+)$, the projections of these tuples appear in $\sigma(I^+)$. Thus, for every $R_i(\vec{v}_i)$ in Q, we have that $R_i^{\sigma(I^+)}$ contains the tuple $\mu(\vec{v}_i)$, and so μ is a homomorphism from Q to $\sigma(I^+)$. In other words, $\mu|_{\text{free}(Q)} \in Q(\sigma(I^+))$.

For the second direction we need to show that if $\mu|_{\text{free}(Q)} \in Q(\sigma(I^+))$, then $\mu|_{\text{free}(Q^+)} \in Q^+(I^+)$. Let μ be a homomorphism from Q to $\sigma(I^+)$. Thus, for every $R_i(\vec{v}_i)$ in Q, we have that $R_i^{\sigma(I^+)}$ contains the tuple $\mu(\vec{v}_i)$. By construction of $\sigma(I^+)$, these tuples are projections of tuples in the cleaned I^+. Consider an atom $R_i^+(\vec{u}_i)$ of Q^+. Since $R_i^{\sigma(I^+)}$ contains $\mu(\vec{v}_i)$, we have that the cleaned $R_i^{+I^+}$ contains a tuple s_i whose projection into $\vec{v}_i$ is $\mu(\vec{v}_i)$. We claim that $s_i = \mu(\vec{u}_i)$. That is, s_i agrees with μ also on the new attributes. Indeed, if R_i was extended due to an FD $R_j \colon X \to y$, then we know that $\mu(\vec{v}_j) \in R_j^{\sigma(I^+)}$, and that $\mu(\vec{v}_j)$ and s_i must agree on y (in addition

to agreeing on X), otherwise this s_i would have been deleted in the cleaning phase. Therefore, s_i assigns y with $\mu(y)$. Then, for every atom $R_i^+(\vec{u}_i)$ of Q^+, we have that $R_i^{+I^+}$ contains the tuple $\mu(\vec{u}_i)$, and we conclude that $\mu|_{\text{free}(Q^+)} \in Q^+(I^+)$.

5.1.3 Tractability

As described in Corollary 5.8, a direct consequence of Theorem 5.5 is that FD-extensions can be used to expand tractable enumeration classes. This is due to the fact that $\text{ENUM}_{\Delta}\langle Q\rangle \leq_e \text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$. The opposite direction is used in Section 5.2 to show the lower bounds required for a dichotomy.

Corollary 5.8. *Let $\mathcal{C}$ be an enumeration class that is closed under exact reduction. Let Q be a CQ and let Q^+ be its FD-extension. If $\text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle \in \mathcal{C}$, then $\text{ENUM}_{\Delta}\langle Q\rangle \in \mathcal{C}$.*

$\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$, $\mathsf{REnum}\langle\text{lin}, \log\rangle$, $\mathsf{RAccess}\langle\text{lin}, \log\rangle$ and $\mathsf{TWAccess}\langle\text{lin}, \log\rangle$ are all closed under exact reduction. Since free-connex queries are in these classes (Corollary 4.18), we get the following corollary.

Corollary 5.9. *Let Q be a CQ over a schema $(\mathcal{R}, \Delta)$. If Q is FD-free-connex, then $\text{ENUM}_{\Delta}\langle Q\rangle$ is in each of $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$, $\mathsf{REnum}\langle\text{lin}, \log\rangle$, $\mathsf{RAccess}\langle\text{lin}, \log\rangle$ and $\mathsf{TWAccess}\langle\text{lin}, \log\rangle$.*

Proof. Since Q^+ is free-connex, and due to Corollary 4.18, we have $\text{ENUM}_{\emptyset}\langle Q^+\rangle$ is in $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$, $\mathsf{REnum}\langle\text{lin}, \log\rangle$, $\mathsf{RAccess}\langle\text{lin}, \log\rangle$ and $\mathsf{TWAccess}\langle\text{lin}, \log\rangle$. Any instance over the schema $(\mathcal{R}, \Delta_{Q^+})$ is also valid over $(\mathcal{R}, \emptyset)$, so using the identity mapping shows that $\text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle \leq_e \text{ENUM}_{\emptyset}\langle Q^+\rangle$. Thus, $\text{ENUM}_{\Delta_{Q^+}}\langle Q\rangle$ in is these four classes, and from Corollary 5.8 we get that $\text{ENUM}_{\Delta}\langle Q\rangle$ is in these classes as well. □

We can now revisit Example 1.3. The query $Q(x, y) \leftarrow R_1(z, x), R_2(z, y)$ is not free-connex. Therefore, if we ignore the FDs, enumerating Q is not in $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$ according to Theorem 2.1. However, given the dependency $R_2\colon z \to y$, the FD-extension is $Q^+(x, y) \leftarrow R_1^+(z, y, x), R_2^+(z, y)$. As it is free-connex, evaluating Q is in $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$ by Corollary 5.9.

5.2 Hardness Results

We now prove lower bounds for CQs in the presence of FDs. Section 5.2.1 shows the hardness of FD-acyclic CQs that are not FD-free-connex, and Section 5.2.2 handles FD-cyclic CQs.

5.2.1 FD-Acyclic CQs

In this section, we characterize which self-join-free FD-acyclic CQs are in the class $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$. We use the notion of FD-extensions defined in the previous section to

establish a dichotomy stating that enumerating the answers to an acyclic query is in $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$ iff the query is FD-free-connex. The positive case for the dichotomy is described in Corollary 5.9, and this section concludes the negative case. We prove the following theorem.

Theorem 5.10. *Let Q be a self-join-free FD-acyclic CQ over the schema $(\mathcal{R}, \Delta)$. If Q is not FD-free-connex, then* $\textsc{Enum}_{\Delta}\langle Q\rangle \notin \mathsf{Enum}\langle \text{lin}, \text{polylog}\rangle$*, assuming* sparseBMM.

Recall that the proof of Theorem 2.1 describes an exact reduction $\textsc{Enum}_{\emptyset}\langle \Pi\rangle \leq_e \textsc{Enum}_{\emptyset}\langle Q\rangle$ from the matrix multiplication query $\Pi(x, y) \leftarrow A(x, z), B(z, y)$ to any self-join-free acyclic non-free-connex CQ Q. It relies on a free-path $(x, z_1, \ldots, z_k, y)$ in Q to encode any instance of the matrix multiplication problem in Q: the variables x,y and $z_1, \ldots, z_k$ of the free-path encode the variables x, y and z of Π, respectively. This way, A is encoded by an atom containing x and z_1, and B is encoded by an atom containing z_k and y. Atoms containing some z_i and z_{i+1} propagate the value of z. FDs restrict the relations that can be assigned to atoms. This means that the reduction cannot be freely performed on databases with FDs, and the hardness proof no longer holds. The following example illustrates where the reduction fails in the presence of FDs.

Example 5.11. The CQ from Example 1.3 has the form $Q(x, y) \leftarrow R_1(z, x), R_2(z, y)$ with the single FD $\Delta = \{R_2\colon z \to y\}$. In the previous section, we show that it is in $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$, so the reduction should fail. Indeed, it would assign R_2 with the same relation as B of the matrix multiplication problem, but this may have two tuples with the same z value and different y values. Therefore, the construction does not yield a valid instance of $\textsc{Enum}_{\Delta}\langle Q\rangle$. □

We now provide a modification of this construction to show an exact reduction from $\textsc{Enum}_{\emptyset}\langle \Pi\rangle$ to $\textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$. Any violations of the FDs are fixed by carefully picking more variables other than those of the free-path to take the roles of x,y and z of the matrix multiplication problem. This is done by introducing the sets V_x,V_y and V_z which are subsets of $\text{var}(Q)$. We say that a variable β *plays the role* of α, if $\beta \in V_\alpha$. To clarify the reduction, we start by describing a restricted case, where all FDs are unary. The basic idea in the case of general FDs remains the same, but it requires a more involved construction of the sets V_α.

Unary FDs

For the unary case, we define the sets V_x, V_y and V_z to hold the variables that iteratively imply x, y and some z_i, respectively. That is, for $\alpha \in \{x, y, z_1, \ldots, z_k\}$ we set $V_\alpha := \{\alpha\}$ and apply $V_\alpha := V_\alpha \cup \{\gamma \in \text{var}(Q) \mid \gamma \to \beta \in \Delta_{Q^+} \wedge \beta \in V_\alpha\}$ until a fixpoint is reached. We then define $V_z := V_{z_1} \cup \cdots \cup V_{z_k}$.

The Reduction. Let $I = (A^I, B^I)$ be an instance of $\text{ENUM}_\emptyset\langle\Pi\rangle$. We define $\sigma(I)$ by describing the relation R^I for every atom $R(\vec{v}) \in \text{atoms}(Q^+)$. If $\text{var}(R) \cap V_y = \emptyset$, then every tuple $(a, c) \in A^I$ is copied to a tuple in R^I. Variables in V_x get the value a, variables in V_z get the value c, and variables that play no role are assigned a constant $\perp$. That is, we define $R^{\sigma(I)}$ to be $\{(f(v_1, a, c), \ldots, f(v_k, a, c)) \mid (a, c) \in A^I\}$, where:

$$f(v_i, a, c) = \begin{cases} a & \text{if } v_i \in V_x \setminus V_z, \\ c & \text{if } v_i \in V_z \setminus V_x, \\ (a, c) & \text{if } v_i \in V_x \cap V_z, \\ \perp & \text{otherwise.} \end{cases}$$

If $\text{var}(R) \cap V_y \neq \emptyset$, we show that $\text{var}(R) \cap V_x = \emptyset$ (see "well-defined reduction" below). In this case we define the relation similarly with B^I. Given a tuple $(c, b) \in B^I$, the variables of V_y get the value b, and those of V_z are assigned with c.

Example 5.12. Consider the query $Q^+(x, y, v) \leftarrow R(u, x, z), S(v, y, z)$ with FDs $\Delta_{Q^+} = \{R\colon u \to x, R\colon u \to z, S\colon y \to v\}$. Using the free-path (x, z, y), the reduction sets $V_x = \{x, u\}$, $V_y = \{y\}$ and $V_z = \{z, u\}$. Given a matrix multiplication instance I with relations A^I and B^I, every tuple $(a, c) \in A^I$ results in a tuple $((a, c), a, c) \in R^{\sigma(I)}$, and every tuple $(c, b) \in B^I$ results in a tuple $(\perp, b, c) \in S^I$. □

We now outline the correctness of this reduction.

Well-defined reduction. For an atom R, either we have $\text{var}(R) \cap V_y = \emptyset$ or $\text{var}(R) \cap V_x = \emptyset$. That is, no atom contains variables from both V_x and V_y. Due to the definition of Q^+, this atom would otherwise also contain both x and y. However, they cannot appear in the same relation according to the definition of a free-path. The reduction is therefore well defined, and it can be constructed in linear time via copy and projection.

Preserving FDs. The construction ensures that if an FD $\gamma \to \alpha$ exists, then γ has all the roles of α. Therefore, either α has no role and corresponds to the constant $\perp$, or every value that appears in α also appears in γ. In any case, all FDs are preserved.

1-1 mapping of answers. If a variable of V_z would have appeared in the head of Q^+, then by the definition of Q^+, some z_i would have been in the head as well. This cannot happen according to the definition of a free-path. Therefore, $V_z \cap \text{free}(Q^+) = \emptyset$, and the head only encodes the x and y values of the matrix multiplication problem, so two different solutions to $\text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$ must differ in either x or y, and correspond to different solutions of $\text{ENUM}_\emptyset\langle\Pi\rangle$. For the other direction, the head necessarily contains the variables x and y. Therefore, two different solutions to $\text{ENUM}_\emptyset\langle\Pi\rangle$ correspond to different solutions of $\text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$.

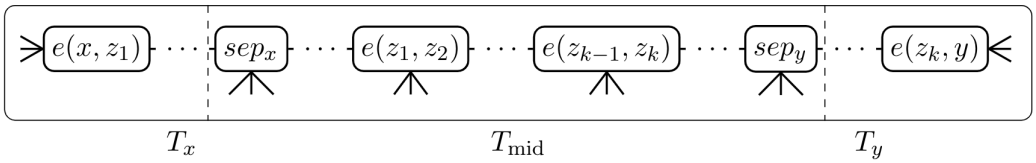


Figure 5.1: The join tree in the proof of Lemma 5.14.

General FDs

Next we show how to lift the idea of this reduction to the case of general FDs. In the case of unary FDs, we ensure that the construction does not violate a given FD $\gamma \to \alpha$, by simply encoding the values of α to γ. In the general case, when allowing more than one variable on the left-hand side of an FD $\gamma_1, \ldots, \gamma_k \to \alpha$, we must be careful when choosing the variables γ_j to which we copy the values of α. Otherwise, as the following example shows, we will not be able to construct the instance in linear time.

Example 5.13. Consider the query $Q(x, y) \leftarrow R_1(x, z, t_1), R_2(z, y, t_1, t_2)$ over a schema with the FD $R_2\colon t_1 t_2 \to y$. Note that $Q = Q^+$ is acyclic but not free-connex, and that (x, z, y) is a free-path in $\mathcal{H}(Q^+)$. To repeat the idea shown in the unary case and ensure that the FDs still hold, the variable on the right-hand side of every FD is encoded to the variables on the left-hand side. If we encode y to t_1, then R_1 would contain the encodings of x, y and z. This means that its size would not be linear in that of the matrix multiplication instance, and we cannot hope for linear time construction. On the other hand, if we choose to encode y only to t_2, the reduction works. □

In the following central lemma, we describe how to carefully pick the variables to which we assign roles in a way that meets the requirements we need for the reduction. We prove requirements 1 and 2 to guarantee a one-to-one mapping between the results of the two problems. Requirement 3 enables linear time construction, while requirement 4 is used to show that all FDs are preserved. The idea is that we consider the join-tree of Q^+ and define a partition of its atoms. We then define V_x and V_y to hold variables that appear only in different parts of the tree, ensuring that no atom contains variables of each. The running intersection property of a join-tree is then used to guarantee that the sets are inclusive enough to correct all FD violations.

Lemma 5.14. *Let Q be a self-join-free CQ over a schema $(\mathcal{R}, \Delta)$ such that Q^+ is acyclic but not free-connex. Further let $(x, z_1, \ldots, z_k, y)$ be a free-path of Q^+. Then, there exist sets of variables V_x, V_y and V_z such that:*

1. *$x \in V_x$, $y \in V_y$, $\{z_1, \ldots z_k\} \subseteq V_z$.*
2. *$V_z \cap \text{free}(Q^+) = \emptyset$.*
3. *For every $R \in \text{atoms}(Q^+)$: $\text{var}(R) \cap V_y = \emptyset$ or $\text{var}(R) \cap V_x = \emptyset$.*
4. *For every $U \to v \in \Delta_{Q^+}$ s.t. $v \in V_\alpha$ with $\alpha \in \{x, y, z\}$: $U \cap V_\alpha \neq \emptyset$.*

Proof. We first define a partition of the atoms of Q into two or three sets: T_x, T_y and possibly T_{mid}. Let T be a join tree of $\mathcal{H}(Q^+)$, and denote the hyperedges on the free-path by $e(x, z_1), \ldots, e(z_k, y)$. Note that, by definition, each hyperedge of the free-path is a vertex of T and an atom of Q^+. By the running intersection property of T and since the path is chordless, we can conclude that there is a simple path P from $e(x, z_1)$ to $e(z_k, y)$ in T, such that $e(z_1, z_2), \ldots, e(z_{k-1}, z_k)$ lie on that path in the order induced by the free-path. Let sep_x be the first atom on the path P that does not contain x. This exists because $e(z_k, y)$ does not contain x, as the free-path is chordless. Similarly, let sep_y be the last atom on P that does not contain y. Let T_x be the set of atoms v such that the unique path from v to $e(x, z_1)$ in T does not go through sep_x. Similarly, let T_y be the set of atoms w such that the unique path from w to $e(z_k, y)$ in T does not go through sep_y. Next set $T_{\text{mid}} = V(T) \setminus (T_x \cup T_y)$. Note that $e(x, z_1) \in T_x$ and $e(z_k, y) \in T_y$, but T_{mid} may be empty (this happens in the case that the free-path is of length two). By definition, the atoms of Q^+ are exactly $T_x \cup T_{\text{mid}} \cup T_y$, and next we show that this union is disjoint. Figure 5.1 depicts the established partition.

We next show that the sets T_x and T_y are disjoint. Assume by contradiction that there is some $v \in T_x \cap T_y$. Let P_x be the unique simple path from v to $e(x, z_1)$, and recall that since $v \in T_x$ it does not go through sep_x. Similarly let P_y be the unique simple path from v to $e(z_k, y)$ that does not go through sep_y.

We first claim that there exists some atoms w that appears in all three paths P, P_x and P_y. Take w to be the first atom on P_x that is also in P and set P_x^w to be the simple path from v to w. Such an atom w exists because the last atom of P_x is $e(x, z_1)$ which is in P. Further set P^w to be the simple path from w to $e(z_k, y)$. Concatenating the paths P_x^w and P^w, we obtain a simple path from v to $e(z_k, y)$. Since the simple paths in a tree are unique, this is exactly P_y, and so w is also in P_y.

Our second claim is that if an atom u is in both P and P_x, then it contains the variable x. Assume by contradiction that such an atom u does not contain x. Then u is an atom on P not containing x, and by definition of sep_x, the simple path from u to $e(x, z_1)$ contains sep_x. As this path is a subpath of P_x, P_x contains sep_x, in contradiction to the fact that $v \in T_x$. Similarly, if an atom is in both P and P_y, then it contains y.

Combining the two claims, we have an atom w containing both x and y, in contradiction to the fact that a free-path is chordless by definition. Therefore we conclude that T_x and T_y are indeed disjoint.

Now we are ready to define the sets of variables V_x, V_y and V_z. We define V_x recursively to contain x and variables that imply those of V_x, but without variables that appear outside of T_x. V_y is defined symmetrically. V_z contains $z_1, \ldots, z_k$ and variables that imply those of V_z but without free variables. Formally,

$$\mathsf{Implies}(V) = \{u \in \text{var}(Q) \mid \exists U \to w \in \Delta_{Q^+} \text{ with } w \in V \text{ and } u \in U\}$$

for $V \subseteq \text{var}(Q)$, and we define via fixpoint iteration the following:

$\boldsymbol{V_x}$: base $V_x := \{x\}$; rule $V_x := (V_x \cup \mathsf{Implies}(V_x)) \setminus \text{var}(T_y \cup T_{\text{mid}})$.

$\boldsymbol{V_y}$: base $V_y := \{y\}$; rule $V_y := (V_y \cup \mathsf{Implies}(V_y)) \setminus \text{var}(T_x \cup T_{\text{mid}})$.

$\boldsymbol{V_z}$: base $V_z := \{z_1, \dots z_k\}$; rule $V_z := (V_z \cup \mathsf{Implies}(V_z)) \setminus \text{free}(Q^+)$.

We now prove that V_x, V_y and V_z meet the requirements of the lemma. Requirements 1 and 2 follow immediately from the definition of the sets. To prove requirement 3, let $R \in \text{atoms}(Q^+)$. If $R \in T_x$, then by definition of V_y we have that $\text{var}(R) \cap V_y = \emptyset$. Otherwise, $R \in T_y \cup T_{\text{mid}}$, and similarly $\text{var}(R) \cap V_x = \emptyset$. It is left to show requirement 4.

Let $\delta = U \to v \in \Delta_{Q^+}$ where $v \in V_\alpha$. We first show the case of $\alpha = z$. If $U \cap V_z = \emptyset$, then $U \subseteq \text{free}(Q^+)$, and by the definition of Q^+, $v \in \text{free}(Q^+)$, which is a contradiction to the definition of V_z. Now we prove the case where $\alpha = x$. The case $\alpha = y$ is symmetric. Denote by $e(U, v)$ an atom containing all variables of δ. As $v \in V_x$, we know that $e(U, v) \notin T_y \cup T_{\text{mid}}$, therefore $e(U, v) \in T_x$. Assume by contradiction that $U \cap V_x = \emptyset$. Let $u \in U$. By definition of V_x, this means that $u \in \text{var}(e_u)$ for some $e_u \in T_y \cup T_{\text{mid}}$. As T_x, T_y and T_{mid} are disjoint, we have that $e_u \notin T_x$, which means that the path between e_u and $e(x, z_1)$ goes through sep_x. This means that the path from e_u to $e(U, v)$ goes through sep_x too, otherwise the concatenation of this path with the path from $e(U, v)$ to $e(x, z_1)$ would result in a path from e_u to $e(x, z_1)$ not going through sep_x. By the running intersection property, $u \in \text{var}(sep_x)$. Since this is true for all $u \in U$, it follows that $v \in \text{var}(sep_x)$ by definition of Q^+, contradicting the fact that $v \in V_x$. □

With the sets V_x, V_y, V_z at hand, we can now perform the reduction for general FDs.

Lemma 5.15. *Let Q be a self-join-free CQ over a schema $(\mathcal{R}, \Delta)$. If Q^+ is acyclic and not free-connex, then* $\text{ENUM}_\emptyset\langle\Pi\rangle \leq_e \text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$.

Proof. Let $I_{A,B} = (A^I, B^I)$ be an instance of $\text{ENUM}_\emptyset\langle\Pi\rangle$ over the domain $\mathcal{D} = \{1, \dots, n\}$. We define an instance $\sigma(I_{A,B})$ of $\text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$ based on the sets V_x, V_y and V_z from Lemma 5.14 and the relations A^I and B^I. Since Q^+ is acyclic but not free-connex, it contains some free-path $(x, z_1, \dots, z_k, y)$.

To define the instance $\sigma(I_{A,B})$, we first fix the functions f_A and f_B:

$$f_A(v,a,c) = \begin{cases} a & : v \in V_x \setminus V_z \\ c & : v \in V_z \setminus V_x \\ (a,c) & : v \in V_x \cap V_z \\ \bot & : \text{otherwise} \end{cases}, \qquad f_B(v,b,c) = \begin{cases} b & : v \in V_y \setminus V_z \\ c & : v \in V_z \setminus V_y \\ (b,c) & : v \in V_y \cap V_z \\ \bot & : \text{otherwise} \end{cases}$$

We partition all relational atoms of Q^+ into two sets: $\mathcal{R}_A^+$ and $\mathcal{R}_B^+$. The set $\mathcal{R}_A^+$ is defined as $\{R^+ \in \text{atoms}(Q^+) \mid \text{var}(R^+) \cap V_y = \emptyset\}$ and $\mathcal{R}_B^+$ is $\text{atoms}(Q^+) \setminus \mathcal{R}_A^+$. To obtain an instance $\sigma(I_{A,B})$ of $\text{ENUM}_{\Delta^+}\langle Q^+\rangle$, we apply f_A to the atoms in $\mathcal{R}_A^+$ using the values of A^I, while the atoms in $\mathcal{R}_B^+$ use f_B and B^I. That is, if $R^+(u_1, \dots, u_m) \in \mathcal{R}_A^+$, then $(R^+)^{\sigma(I_{A,B})}$ is defined to be $\{(f_A(u_1, a, c), \dots, f_A(u_m, a, c)) \mid (a, c) \in A^I\}$. Otherwise, $R^+(u_1, \dots, u_m) \in \mathcal{R}_B^+$, and $(R^+)^{\sigma(I_{A,B})} = \{(f_B(v_1, b, c), \dots, f_B(v_p, b, c)) \mid (c, b) \in B^I\}$. The mapping τ is defined as the projection onto the variables x and y. Note that the instance can be constructed in linear time, and the projection can be computed in constant time.

We now claim that $\sigma(I_{A,B})$ is a database over the schema $(\mathcal{R}^+, \Delta_{Q^+})$, as all the FDs of Δ_{Q^+} are satisfied. Let $\delta = R_j^+ \colon U \to v \in \Delta_{Q^+}$. If $v \notin V_x \cup V_y \cup V_z$, then δ holds as v is assigned the value $\bot$ in every tuple in $(R_j^+)^{\sigma(I_{A,B})}$. Next assume that $v \in V_x \setminus V_z$. By point 4 of Lemma 5.14, there is some $u \in U$ such that $u \in V_x$. Thus in every tuple in $(R_j^+)^{\sigma(I_{A,B})}$, if v is assigned the value a, then u is either assigned the value a or (a, c) for some $c \in \{1, \ldots, n\}$ and in either case δ is satisfied. The proof for the cases where $v \in V_z \setminus (V_x \cup V_y)$ and $v \in V_y \setminus V_z$ is similar. Next assume that $v \in V_x \cap V_z$. By point 4 of Lemma 5.14, there are some $u_1, u_2 \in U$ such that $u_1 \in V_x$ and $u_2 \in V_z$ and for every tuple in $(R_j^+)^{\sigma(I_{A,B})}$, if v is assigned the value (a, c), then u_1 is either assigned the value a or (a, c), u_2 is either assigned the value c or (a, c), and so δ is satisfied. The case $v \in V_y \cap V_z$ is similar. Note that the case where $v \in V_x \cap V_y$ cannot occur due to point 3 of Lemma 5.14.

The structure of the path $(x, z_1 \ldots, z_k, y)$ ensures that answers to $\textsc{Enum}_{\Delta_{Q^+}}\langle Q^+ \rangle$ correspond to those of $\textsc{Enum}_{\emptyset}\langle \Pi \rangle$ and vice versa. Indeed, let $\mu|_{\{x,y\}} \in \Pi(I_{A,B})$. We have that $(\mu(x), \mu(z)) \in A^I$ and $(\mu(z), \mu(y)) \in B^I$. We define $\nu_\mu : \text{var}(Q^+) \to \mathcal{D}$ as follows.

$$\nu_\mu(v) = \begin{cases} \mu(x) & : v \in V_x \setminus V_z, \\ \mu(y) & : v \in V_y \setminus V_z, \\ \mu(z) & : v \in V_z \setminus (V_x \cup V_y), \\ (\mu(x), \mu(z)) & : v \in V_z \cap V_x, \\ (\mu(y), \mu(z)) & : v \in V_z \cap V_y, \\ \bot & : \text{otherwise.} \end{cases}$$

By definition, $f_A(v, \mu(x), \mu(z)) = \nu_\mu(v)$ and $f_B(v, \mu(y), \mu(z)) = \nu_\mu(v)$. Consider any atom $R^+(u_1, \ldots, u_m)$ of Q^+. If $R^+(u_1, \ldots, u_m) \in \mathcal{R}_A^+$, then $\nu_\mu(u_1, \ldots, u_m)$ is equal to $(f_A(u_1, \mu(x), \mu(z)), \ldots, f_A(u_m, \mu(x), \mu(z))) \in (R^+)^{\sigma(I_{A,B})}$. If $R^+(u_1, \ldots, u_m) \in \mathcal{R}_B^+$, we have that $\text{var}(R^+) \cap V_x = \emptyset$ by point 3 of Lemma 5.14. Then, $\nu_\mu(u_1, \ldots, u_m) = (f_B(u_1, \mu(z), \mu(y)), \ldots, f_B(u_m, \mu(z), \mu(y))) \in (R^+)^{\sigma(I_{A,B})}$. Therefore, $\nu_\mu|_{\text{free}(Q^+)}$ is in $Q^+(\sigma(I_{A,B}))$. Since $x \in V_x \setminus V_z$ and $y \in V_y \setminus V_z$, we have that $\nu_\mu(x) = \mu(x)$ and $\nu_\mu(y) = \mu(y)$, and so $\tau(\nu_\mu) = \mu|_{\{x,y\}}$. Moreover, any answer $\mu'|_{\text{free}(Q^+)} \in Q^+(\sigma(I_{A,B}))$ that has $\tau(\mu'|_{\text{free}(Q^+)}) = \mu|_{\{x,y\}}$ assigns $\mu(x)$ to variables in $V_x \setminus V_z$, assigns $\mu(y)$ to variables in $V_y \setminus V_z$ and assigns $\bot$ to variables not in $V_x \cup V_y \cup V_z$. By points 2 and 3 of Lemma 5.14, $V_z \cap \text{free}(Q^+) = \emptyset$ and $V_x \cap V_y = \emptyset$, so these are the only variables in $\text{free}(Q^+)$. Therefore $\mu'|_{\text{free}(Q^+)} = \nu_\mu|_{\text{free}(Q^+)}$.

Next assume that $\mu'|_{\text{free}(Q^+)} \in Q^+(\sigma(I_{A,B}))$. Let R_x^+ be an atom containing x and z_1 (such an atom exists by the definition of the free-path). By point 3 of Lemma 5.14 we have $\text{var}(R^+) \cap V_y = \emptyset$, and $R^+ \in \mathcal{R}_A^+$. By points 1 and 2 of Lemma 5.14 and since x is a free variable, we have $x \in V_x \setminus V_z$ and $z_1 \in V_z$. Thus there exists some $(a, c) \in A^I$ such that $\mu'(x) = f_A(x, a, c) = a$ and $\mu'(z_1) = f_A(z_1, a, c) \in \{a, (a, c)\}$. Similarly, there exists some $(c', b) \in B^I$ such that $\mu'(y) = f_B(y, b, c') = b$ and $\mu'(z_k) = f_B(z_k, b, c') \in \{c', (b, c')\}$. It remains to show that $c = c'$. We show by induction on i that $\mu'(z_i)$ is either c or of the form (t, c) with some value t. We know this fact for $i = 1$ since this is how we define c.

If $k > 1$, then consider an atom $R_i^+(\vec{v}_i)$ containing $\{z_{i-1}, z_i\}$. Then μ' maps $\vec{v}_i$ to some tuple $t \in R_i^+$ that assigns z_{i-1} with a value of the form c or (t, c). Since $z_i \in V_z$, t also assigns z_i with a value of such a form, so $\mu'(z_i)$ is of the form c or (t, c) too. This shows that $c = c'$. Therefore, $\tau(\mu') \in \Pi(I_{A,B})$. Moreover, since τ is simply a projection, $\tau(\mu')$ is uniquely defined. □

We have that $\textsc{Enum}_{\emptyset}\langle\Pi\rangle \leq_e \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle \leq_e \textsc{Enum}_{\Delta}\langle Q\rangle$ by combining Theorem 5.5 and Lemma 5.15. Therefore, having $\textsc{Enum}_{\Delta}\langle Q\rangle \in \mathsf{Enum}\langle\text{lin}, \text{polylog}\rangle$ would mean that $\textsc{Enum}_{\emptyset}\langle\Pi\rangle \in \mathsf{Enum}\langle\text{lin}, \text{polylog}\rangle$, which contradicts the conjecture of the lower bound for matrix multiplication. This concludes the proof of Theorem 5.10. Note that Theorem 5.10 does not contradict the dichotomy of Theorem 2.1: if for a given query Q we have that Q^+ is acyclic but not free-connex, then Q is not free-connex by Proposition 5.4.

As enumeration can be achieved by random-permutation or random-access, Theorem 5.10 also implies the hardness of these two tasks. Together with Corollary 5.9, this proves a dichotomy.

Corollary 5.16. *Let Q be a self-join-free FD-acyclic CQ over the schema $(\mathcal{R}, \Delta)$.*

- *If Q is FD-free-connex, then $\textsc{Enum}_{\Delta}\langle Q\rangle$ is in $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$, $\mathsf{REnum}\langle\text{lin}, \log\rangle$, $\mathsf{RAccess}\langle\text{lin}, \log\rangle$ and $\mathsf{TWAccess}\langle\text{lin}, \log\rangle$.*
- *If Q is not FD-free-connex, then $\textsc{Enum}_{\Delta}\langle Q\rangle$ is not in any of $\mathsf{Enum}\langle\text{lin}, \text{polylog}\rangle$, $\mathsf{REnum}\langle\text{lin}, \text{polylog}\rangle$, $\mathsf{RAccess}\langle\text{lin}, \text{polylog}\rangle$, and $\mathsf{TWAccess}\langle\text{lin}, \text{polylog}\rangle$, assuming* sparseBMM.

5.2.2 FD-Cyclic CQs

In the previous section, we established a classification of FD-acyclic CQs, but we did not consider FD-cyclic queries. Recall that, according to Theorem 2.1 and under certain assumptions, self-join-free cyclic queries are not in $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$. In this section, we therefore explore how FD-extensions can be used to obtain some insight on the implications of this result in the presence of FDs. We show that, under the same assumptions, self-join-free FD-cyclic queries that contain only unary FDs cannot be evaluated in linear time. For schemas containing only unary FDs, this extends the dichotomy from the previous section to all CQs.

Theorem 5.17. *Let Q be a self-join-free CQ over a schema $(\mathcal{R}, \Delta)$, where Δ only contains unary FDs. If Q is FD-cyclic, then $\textsc{Decide}_{\Delta}\langle Q\rangle$ cannot be solved in linear time, assuming* Hyperclique.

Let $\mathcal{H}$ be a hypergraph. We denote by $\textsc{Tetra}(k)$ the decision problem of whether $\mathcal{H}$ contains a k-tetra. Using this notation, Hyperclique is the assumption that $\textsc{Tetra}(k)$ cannot be solved in time linear in the size of the graph for any $k \geq 3$. The original hardness proof for cyclic CQs shows that for every self-join-free cyclic CQ Q there exists

k such that $\textsc{Tetra}(k)$ can be reduced to $\textsc{Decide}_{\emptyset}\langle Q\rangle$. As before, this hardness proof no longer holds in the presence of FDs as the construction is not guaranteed to satisfy the dependencies, and we present a modified reduction that satisfies the FDs. We will show that if Q^+ is cyclic and only unary FDs are present, the problem $\textsc{Tetra}(k)$ for some k can be reduced to $\textsc{Decide}_{\Delta_{Q^+}}\langle Q^+\rangle$. The proof relies on the notion of pseudo-minors.

Definition 5.18 (Pseudo-Minors)**.** A *pseudo-minor* of a hypergraph $\mathcal{H} = (V, E)$ is a hypergraph obtained from $\mathcal{H}$ by a finite series of the following operations:

- *vertex removal*: removing a vertex from V and from all edges in E that contain it;
- *edge removal*: removing an edge e from E if some other $e' \in E$ contains it;
- *edge contraction*: replacing all occurrences of a vertex v (within every edge) with a vertex u if u and v are neighbors.

To perform the said reduction, we will use a tetra pseudo-minor of a hypergraph describing our query.

Definition 5.19. Let $\mathcal{H}$ be a cyclic hypergraph. We denote by $\mathsf{Tet}_{\mathsf{pm}}(\mathcal{H})$ the pairs of pseudo-minors $(\mathcal{H}^a, \mathcal{H}^b)$ of $\mathcal{H}$ such that:

1. $\mathcal{H}^a$ is obtained by a (possibly empty) set of vertex removal and edge removal operations on $\mathcal{H}$.
2. $\mathcal{H}^b$ is obtained by a (possibly empty) set of edge contraction and edge removal operations on $\mathcal{H}^a$.
3. $\mathcal{H}^b$ is a tetra.
4. Either $\mathcal{H}^a = \mathcal{H}^b$ or $\mathcal{H}^a$ is a chordless cycle.

Given a query Q, we denote $\mathsf{Tet}_{\mathsf{pm}}(Q) = \mathsf{Tet}_{\mathsf{pm}}(\mathcal{H}(Q))$.

Brault-Baron [BB13, Theorem 11] showed that a cyclic hypergraph $\mathcal{H}$ admits some k-tetra as a pseudo-minor. We describe the proof here in our terminology.

Lemma 5.20 ([BB13], Theorem 11)**.** *If $\mathcal{H}$ is a cyclic hypergraph, then $\mathsf{Tet}_{\mathsf{pm}}(\mathcal{H}) \neq \emptyset$.*

Proof. If $\mathcal{H}$ contains a chordless cycle C, then removing vertices not in C followed by performing all possible edge removals results in a chordless cycle $\mathcal{H}^a$. Then, $\mathcal{H}^b$ is a 3-tetra obtained by a repeated use of edge-contraction followed by performing all possible edge removals. In this case, $(\mathcal{H}^a, \mathcal{H}^b) \in \mathsf{Tet}_{\mathsf{pm}}(\mathcal{H})$. If $\mathcal{H}$ does not contain a chordless cycle, since it is not acyclic, it is non-conformal. Consider its smallest non-conformal clique. The clique is not contained in any edge (since it is non-conformal), and it is a k-tetra because of its minimality. Therefore, removing all vertices other than the clique, and then performing all possible edge removals, results in a tetra $\mathcal{H}^a = \mathcal{H}^b$. Again, $(\mathcal{H}^a, \mathcal{H}^b) \in \mathsf{Tet}_{\mathsf{pm}}(\mathcal{H})$. □

For the reduction we present next, we first need to show that for an FD-cyclic query Q, no pseudo-minor in $\mathsf{Tet}_{\mathsf{pm}}(Q^+)$ contains all variables of any FD $X \to y$. In the following we assume that Δ only contains non-trivial FDs, meaning $y \notin X$.

Lemma 5.21. *Let Q be a self-join-free FD-cyclic CQ over a schema $(\mathcal{R}, \Delta)$. Let $(\mathcal{H}^a, \mathcal{H}^b) \in \mathsf{Tet}_{\mathsf{pm}}(Q^+)$ and $\mathcal{H}^a = (V, E)$. For every non-trivial $X \to y \in \Delta_{Q^+}$, we have $X \cup \{y\} \not\subseteq V$.*

Proof. We start with an observation regarding the FDs. Note that in $\mathcal{H}(Q^+)$ some edge contains the vertices $X \cup \{y\}$, and by the construction of Q^+, every edge that contains X must also contain y. These properties still hold after any sequence of vertex removals and edge removals as long as none of the vertices $X \cup \{y\}$ are removed. If at least one of $X \cup \{y\}$ is removed, it immediately follows that $X \cup \{y\} \not\subseteq V$ and we are done. Next, we assume by way of contradiction that none of the vertices $X \cup \{y\}$ were removed, and $X \cup \{y\} \subseteq V$. Therefore, there must be an edge in $\mathcal{H}^a$ containing $X \cup \{y\}$, and every edge in $\mathcal{H}^a$ containing X also contains y.

We distinguish two cases. In the first case, $\mathcal{H}^b = \mathcal{H}^a$ is a k-tetra obtained from $\mathcal{H}(Q^+)$ by a sequence of vertex and edge removals. If $X \cup \{y\} \subseteq V$, then by the definition of k-tetra it should contain the edge $V \setminus \{y\}$. This is a contradiction to the fact that every edge in $\mathcal{H}^a$ containing X also contains y. In the second case, $\mathcal{H}^a$ is a cycle, and $\mathcal{H}^b$ is a 3-tetra obtained by performing edge contraction and edge removal operations on it. Since there must be an edge in $\mathcal{H}^a$ containing $X \cup \{y\}$, and all edges are of size 2, it must be that $|X| = 1$. Denote $X = \{x\}$. Since $\mathcal{H}^a$ is a cycle containing both x and y, there should be at least one edge containing x but not containing y. This is again a contradiction to the fact that every edge in $\mathcal{H}^a$ containing X also contains y. □

We are now ready to establish the reduction. Given a k-tetra pseudo-minor of $\mathsf{Tet}_{\mathsf{pm}}(Q^+)$, we can reduce the problem of checking whether a hypergraph contains a k-tetra to finding a Boolean answer to Q^+.

Lemma 5.22. *Let Q be a self-join-free FD-cyclic CQ over a schema $(\mathcal{R}, \Delta)$, where Δ only contains unary FDs. Let $(\mathcal{H}^a, \mathcal{H}^b) \in \mathsf{Tet}_{\mathsf{pm}}(Q^+)$ such that $\mathcal{H}^b$ is a k-tetra for some k. Then, there is a linear time reduction $\textsc{Tetra}(k) \leq_m \textsc{Decide}_{\Delta_{Q^+}}\langle Q^+ \rangle$.*

Proof. Given an input hypergraph $\mathcal{G}$ for the $\textsc{Tetra}(k)$ problem, we define an instance I of $\textsc{Decide}_{\Delta_{Q^+}}\langle Q^+ \rangle$. We consider a sequence of pseudo-minors $\mathcal{H}(Q^+) = \mathcal{H}_1, \mathcal{H}_2, \ldots, \mathcal{H}_m = \mathcal{H}^b$, each pseudo-minor is obtained by performing one operation over the previous one, where $\mathcal{H}_j = \mathcal{H}^a$ for some $1 \leq j \leq m$. We treat hypergraphs as describing CQs. That is, to the hypergraph $\mathcal{H}_i$ we associate a query Q_i such that $\mathcal{H}(Q_i) = \mathcal{H}_i$. Every edge e of $\mathcal{H}_i$ corresponds to an atom in Q_i with a relational symbol R^i_e, and the vertices of e are its variables. We assume that the variables in every atom are sorted by some total order. In the following, we construct instances I_i to these queries. It is possible to define a instance I_m such that deciding $Q_m(I_m)$ solves $\textsc{Tetra}(k)$ [BB13, Lemma 20]. We describe how to inductively build an instance $I_1 = I$ such that deciding $Q^+(I)$ solves the same problem.

Constructing I. We first define I_m. For every edge e of $\mathcal{H}_m$, we define a relation R^m_e that contains all edges of $\mathcal{G}$ that have the same size as e. A tuple of R^m_e consists of the vertices of such an edge sorted by some total order on the vertices of $\mathcal{G}$. We now define I_i given I_{i+1}. We distinguish three cases according to the type of pseudo-minor operation that leads from $\mathcal{H}_i$ to $\mathcal{H}_{i+1}$.

- *edge removal*: For every $e'' \in \mathcal{H}_{i+1}$, set $R^i_{e''} = R^{i+1}_{e''}$. Then, let e be the edge removed, and let e' be an edge containing it. Set R^i_e to be a copy of $R^{i+1}_{e'}$ projected accordingly.
- *edge contraction*: Let v be the vertex replaced by its neighbor u. For any edge $e \in \mathcal{H}_i$ contracting to an edge $e' \in \mathcal{H}_{i+1}$, set R^i_e to be a copy of $R^{i+1}_{e'}$, and assign the attribute v a copy of the value of u in every tuple. Then, if $u \notin e$, project u out of R^i_e. For every other edge $e'' \in \mathcal{H}_i$, set $R^i_{e''} = R^{i+1}_{e''}$.
- *vertex removal*: Let v be the vertex removed from an edge $e \in \mathcal{H}_i$, resulting in an edge $e' \in \mathcal{H}_{i+1}$. Expand $R^{i+1}_{e'}$ to R^i_e by copying $R^{i+1}_{e'}$, and assign v with a constant $\perp$ in every tuple. Next, apply the following FD-correction steps on v:
  1. In every tuple, concatenate to the value of v the values of variables it implies. These variables are defined via fixpoint iteration as $\text{ImpliedBy}(v) = \{v\}$ and $\text{ImpliedBy}(v) = \{w \mid t \to w \in \Delta_{Q^+}, t \in \text{ImpliedBy}(v)\} \cup \text{ImpliedBy}(v)$. For each $w \in \text{ImpliedBy}(v) \setminus \{v\}$, if $R^i(\vec{u})$ is an atom such that $\vec{u}[k] = v$ and $\vec{u}[j] = w$, then in every tuple $t \in R^i$, replace $t[k]$ with $(t[k], t[j])$.
  2. After the value of v is determined, concatenate the value of v to the variables implying it. These are defined via fixpoint iteration as $\text{Implies}(v) = \{v\}$ and $\text{Implies}(v) = \{u \mid u \to t \in \Delta_{Q^+}, t \in \text{Implies}(v)\} \cup \text{Implies}(v)$. For each variable $u \in \text{Implies}(v) \setminus \{v\}$, if $R^i(\vec{u})$ is an atom such that $\vec{u}[k] = v$ and $\vec{u}[j] = u$, then in every tuple $t \in R^i$, replace $t[j]$ with $(t[j], t[k])$.

  For every edge $e'' \in \mathcal{H}_i$ not containing v, set $R^i_{e''} = R^{i+1}_{e''}$.

The overall construction of the instance I can be done in linear time, since there is a constant number of pseudo-minor operations, each requiring a linear number of computational steps.

I satisfies Δ_{Q^+}. We show that I satisfies the FDs in Δ_{Q^+} by induction: we claim that for each $\mathcal{H}_i$ all FDs $x \to y$ such that $x, y \in V(\mathcal{H}_i)$ are satisfied. According to Lemma 5.21, $\mathcal{H}^a$ and therefore all of $\mathcal{H}_j, \ldots, \mathcal{H}_m$ do not contain all variables of any FD. Thus, our claim trivially holds for $\mathcal{H}_j, \ldots, \mathcal{H}_m$. We now prove our claim for $\mathcal{H}_i$ where $i \leq j - 1$. Consider an FD $\delta = x \to y$ such that $x, y \in V(\mathcal{H}_i)$. There are three cases:

- If $x, y \in V(\mathcal{H}_{i+1})$, then by the induction assumption δ is satisfied in $\mathcal{H}_{i+1}$. If $\mathcal{H}_{i+1}$ is obtained by edge removal, then the only new relation in $\mathcal{H}_i$ is a projection of a relation of $\mathcal{H}_{i+1}$, and therefore δ is satisfied in all relations. Otherwise, $\mathcal{H}_{i+1}$ is obtained by vertex removal. If the value of y is the same in R^i_e as in R^{i+1}_e, we are done by the induction assumption. Otherwise, y is changed due to the second FD-correction step, and the vertex removed is some z such that $y \to z$. In this

case, since x transitively implies z, both x and y are concatenated with the same values, and δ is still satisfied.

- If $x \notin V(\mathcal{H}_{i+1})$, then $\mathcal{H}_{i+1}$ is obtained by the removal of the vertex x, and the first FD-correction step ensures that x contains a copy of the values of y in every tuple where they both appear. Therefore δ is satisfied.
- If $y \notin V(\mathcal{H}_{i+1})$, then $\mathcal{H}_{i+1}$ is obtained by the removal of the vertex y. The second FD-correction step ensures that x contains a copy of the values of y, and δ is satisfied.

Correctness. We know [BB13, Lemma 20] that there is a solution to $Q_m(I_m)$ iff there exists a subhypergraph of $\mathcal{G}$ isomorphic to $\mathcal{H}_t$, and in fact every mapping μ that can be used for the evaluation corresponds to such a subhypergraph. We claim that every mapping used for evaluating $Q_{i+1}(I_{i+1})$ corresponds to a mapping that can be used for $Q_i(I_i)$, and vice versa. This was already shown in case $\mathcal{H}_{i+1}$ is obtained by $\mathcal{H}_i$ via edge contraction [BB13, Lemma 15] or edge removal [BB13, Lemma 14], and it was shown for vertex removal [BB13, Lemma 13] when the construction simply assigns the new vertex with a constant and skips the FD-correction steps. Let $\mathcal{H}_{i+1}$ be a pseudo-minor obtained from $\mathcal{H}_i$ via vertex removal, and denote by I_i^0 the instance constructed from I_{i+1} as described but without the FD-correction steps. It is left to show that a mapping μ^0 that satisfies $Q_i(I_i^0)$ corresponds to a mapping μ that satisfies $Q_i(I_i)$, and vice versa. This will conclude that $\mathcal{G}$ has a subhypergraph isomorphic to $\mathcal{H}_t$ (a k-tetra) iff $Q^+(I) \neq \emptyset$.

First we claim that if x implies y and both x and y are present in $\mathcal{H}_i$ (that is, for all $1 \leq j < i$ the operation between $\mathcal{H}_j$ and $\mathcal{H}_{j+1}$ is not the removal of x or y), then y appears in every atom containing x in $\mathcal{H}_i$. This will help us show that we perform the same changes over x in all relations. Since Q^+ is an FD-extension and only unary FDs are present, we are guaranteed that if x implies y, then y is present in every edge of $\mathcal{H}(Q^+) = \mathcal{H}_1$ where x appears. This property is preserved under vertex removal and edge removal operations (as long as x and y are not removed), which are the only operations that can be performed between $\mathcal{H}_1$ and $\mathcal{H}_i$ (this is true since we perform vertex removal on $\mathcal{H}_i$ and, by Definition 5.19, vertex-removal operations only occur between $\mathcal{H}_1$ and $\mathcal{H}^a$, and edge-removal is the only other operation allowed there).

We now show that given μ^0 that satisfies $Q_i(I_i^0)$ there is a mapping μ that satisfies $Q_i(I_i)$, and vice versa. We show this by induction, considering one FD-correction step involving one variable at a time. Consider an FD-correction step of the first type on a vertex v implying w, where both v and w are present in $\mathcal{H}_i$. In any atom $R(\vec{u})$ such that $\vec{u}[k] = v$, we showed that there exists an index j such that $\vec{u}[j] = w$. For every tuple $t_0 \in R^0$, there is a similar tuple $t \in R$ with the only difference being $t[k] = (t_0[k], t_0[j])$. Therefore, by defining $\mu(v) = (\mu_0(v), \mu_0(w))$ and $\mu(u) = \mu_0(u)$ for all other variables $u \neq v$, every tuple that is used in the evaluation of μ^0 in I_i^0 results in a tuple that can be used in the evaluation of μ in I_i. Indeed, μ is a valid evaluation of $Q_i(I_i)$. A

similar argument holds similarly for the opposite direction and for the second type of FD-correction step. For the opposite direction for example, if $\mu(v) = (a_v, a_w)$, we define $\mu^0(v) = a_v$. □

Theorem 5.17 is an immediate consequence of Lemma 5.22.

Proof of Theorem 5.17. Assume for the sake of contradiction that Q is FD-cyclic and $\text{DECIDE}_{\Delta}\langle Q\rangle$ is solvable in linear time. Theorem 5.5 implies a linear time reduction $\text{DECIDE}_{\Delta_{Q^+}}\langle Q^+\rangle \leq_m \text{DECIDE}_{\Delta}\langle Q\rangle$. Therefore, $\text{DECIDE}_{\Delta_{Q^+}}\langle Q^+\rangle$ can be solved in linear time as well. As Q^+ is cyclic, there exists a k-tetra pseudo-minor $\mathcal{H}_{pm} \in \mathsf{Tet_{pm}}(Q^+)$ for some $k \geq 3$. According to Lemma 5.22, $\text{TETRA}(k)$ is also solvable in linear time. □

In terms of enumeration complexity, Theorem 5.17 means that any enumeration algorithm for such a query cannot output a first solution (or decide that there is none) within linear time. As a corollary, we get that for a self-join-free FD-cyclic CQ over a schema containing only contains unary FDs there is no enumeration, random-permutation or random-access algorithm with only linear preprocessing time. Together with Corollary 5.16, this results in a dichotomy.

Corollary 5.23. *Let Q be a self-join-free CQ over a schema $(\mathcal{R}, \Delta)$, where Δ only contains unary FDs.*

- *If Q is FD-free-connex, then* $\text{ENUM}_{\Delta}\langle Q\rangle$ *is in* $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$, $\mathsf{REnum}\langle\text{lin}, \log\rangle$, $\mathsf{RAccess}\langle\text{lin}, \log\rangle$ *and* $\mathsf{TWAccess}\langle\text{lin}, \log\rangle$.
- *If Q is not FD-free-connex, then* $\text{ENUM}\langle Q\rangle$ *is not in any of* $\mathsf{Enum}\langle\text{lin}, \text{polylog}\rangle$, $\mathsf{REnum}\langle\text{lin}, \text{polylog}\rangle$, $\mathsf{RAccess}\langle\text{lin}, \text{polylog}\rangle$ *and* $\mathsf{TWAccess}\langle\text{lin}, \text{polylog}\rangle$ *assuming* sparseBMM.

We conclude this section with a short discussion on its extension to general FDs. The following example shows that our proof for Theorem 5.17 cannot be lifted to general FDs. Exploring this extension is left for future work.

Example 5.24. Consider $Q() \leftarrow R_1(x, y, u), R_2(x, w, z), R_3(y, v, z), R_4(u, v, w)$ over a schema with all possible two-to-one FDs in R_1, R_2 and R_3. That is,

$\Delta = \{xy \to u, yu \to x, ux \to y, zy \to v, yv \to z, vz \to y, xz \to w, zw \to x, wx \to z\}$.

Note that $Q^+ = Q$. The hypergraph $\mathcal{H}(Q^+)$ is cyclic, see Figure 5.2, yet it is unclear whether Q can be solved in linear time, and whether $\text{TETRA}(3)$ can be reduced to answering Q^+. Using Lemma 5.21, $\mathcal{H}(Q^+)$ has triangle pseudo-minors that do not contain all variables of any FD. Consider for example the one obtained by removing all vertices other than x, y, z. A construction similar to that of Lemma 5.22 would assign u with the values of x and y, assign v with the values of y and z, and assign w with the values of x and z. This results in the edge $\{u, v, w\}$ containing all three values of any possible triangle, meaning that this edge cannot be constructed in linear time. Other choices of triangle pseudo-minors lead to similar encoding problems. □

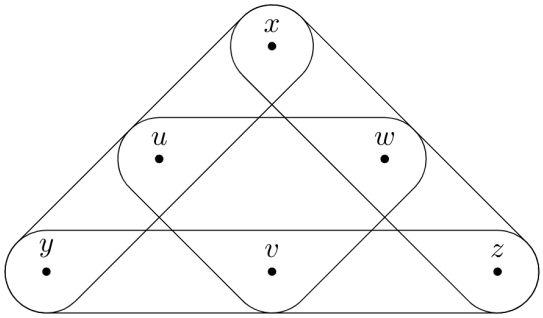


Figure 5.2: The hypergraph $\mathcal{H}(Q) = \mathcal{H}(Q^+)$ for Example 5.24.

5.3 Extended Settings

We devote the final section to extend the results of this chapter to cardinality dependencies, CQs with disequalities and UCQs.

5.3.1 Cardinality Dependencies

In this section, we show that our results also apply to CQs over schemas with cardinality dependencies. *Cardinality Dependencies* (CDs) [AFG16, CFWY14] are a generalization of FDs, where the left-hand side does not uniquely determine the right-hand side, but rather provides a bound on the number of distinct values it can have. Formally, Δ is the set of *CDs* of a schema $\mathcal{S} = (\mathcal{R}, \Delta)$. Every $\delta \in \Delta$ has the form $(R_i \colon A \to B, c)$, where $R_i \colon A \to B$ is an FD and c is a positive integer. A CD δ is *satisfied* by an instance I over $\mathcal{S}$, if every set of tuples $S \subseteq (R_i)^I$ that agrees on the indices of A, but no pair of them agrees on all indices of B, contains at most c tuples. It follows from the definition that δ is an FD if $c = 1$.

Denote by Δ^{FD} the FDs obtained from a set of CDs Δ by setting all c values to one. Given a query Q over $(\mathcal{R}, \Delta)$, we define the *CD-extension* Q^+ of Q to be the same as the FD-extension of Q over $(\mathcal{R}, \Delta^{\text{FD}})$. The schema $\mathcal{S}^+$ is defined with the original c values, and the extended CDs are $\Delta_{Q^+} = \{(R_i^+ \colon A \to b, c) \mid \exists (R_j \colon A \to B, c) \in \Delta, b \in B, A \cup \{b\} \subseteq \text{var}(R_i^+)\}$. Note that FD-extensions are indeed a special case of CD-extensions.

The hardness results extend to CDs because they are not more restrictive than FDs: Since every instance that preserves the FDs Δ^{FD} also preserves the CDs Δ, we conclude that $\textsc{Enum}_{\Delta^{\text{FD}}}\langle Q\rangle \leq_e \textsc{Enum}_{\Delta}\langle Q\rangle$. When only FDs are present, we can apply Theorem 5.5, and get $\textsc{Enum}_{\Delta^{\text{FD}}_{Q^+}}\langle Q^+\rangle \leq_e \textsc{Enum}_{\Delta^{\text{FD}}}\langle Q\rangle$. By combining the two, we get the following lemma.

Lemma 5.25. *Let Q be a CQ over a schema $(\mathcal{R}, \Delta)$, where Δ is a set of CDs, and let Q^+ be its CD-extension. Then* $\textsc{Enum}_{\Delta^{FD}_{Q^+}}\langle Q^+\rangle \leq_e \textsc{Enum}_{\Delta}\langle Q\rangle$.

Defining the classes of *CD-acyclic* and *CD-free-connex* queries similarly to the case with FDs, Lemma 5.25 implies that all lower bounds presented in this chapter hold for

CDs. For example, if Q is self-join-free and CD-acyclic but not CD-free-connex and $\text{ENUM}_{\Delta}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{polylog}\rangle$, then by Lemma 5.25 we have that $\text{ENUM}_{\Delta^{\text{FD}}_{Q^+}}\langle Q^+\rangle$ is in $\mathsf{Enum}\langle \text{lin}, \text{polylog}\rangle$ as well. According to Lemma 5.15 this means that $\text{ENUM}_{\emptyset}\langle \Pi\rangle$ is in $\mathsf{Enum}\langle \text{lin}, \text{polylog}\rangle$, in contradiction to our assumption about the hardness of Boolean matrix multiplication. So $\text{ENUM}_{\Delta}\langle Q\rangle \notin \mathsf{Enum}\langle \text{lin}, \text{polylog}\rangle$ assuming sparseBMM. Similarly, we conclude the hardness of self-join-free CD-cyclic CQs over schemas that contain only unary CDs, of the form $(A \to B, c)$ with $|A| = 1$. By combining Lemma 5.25 with Theorem 5.17, we conclude that it is not possible to find a first answer to such queries in linear time, assuming Hyperclique.

In order to extend the positive results, we need to show that the CD-extension is at least as hard as the original query w.r.t. enumeration. For this, we need a relaxation of exact reductions: The reduction denoted $\text{ENUM}\langle R_1\rangle \leq_{e'} \text{ENUM}\langle R_2\rangle$ requires that one output of $\text{ENUM}\langle R_1\rangle$ corresponds to at most a constant number of outputs of $\text{ENUM}\langle R_2\rangle$ (instead of a bijection between the sets of outputs).

Lemma 5.26. *Let Q be a CQ over a schema $(\mathcal{R}, \Delta)$, where Δ is a set of CDs, and let Q^+ be its CD-extension. Then $\text{ENUM}_{\Delta}\langle Q\rangle \leq_{e'} \text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$.*

Proof. When dealing with FDs, we assume that the right-hand side has only one variable, as we can use such FDs to describe all possible ones. With CDs this no longer holds. Nonetheless, every instance of the schema $\mathcal{S} = (\mathcal{R}, \Delta)$ satisfies $\Delta^1 = \{(R_i\colon A \to b, c) \mid (R_i\colon A \to B, c) \in \Delta, b \in B\}$, so is also an instance of $\mathcal{S}^1 = (\mathcal{R}, \Delta^1)$. Therefore, $\text{ENUM}_{\Delta}\langle Q\rangle \leq_e \text{ENUM}_{\Delta^1}\langle Q\rangle$ using the identity mapping. It is left to show that $\text{ENUM}_{\Delta^1}\langle Q\rangle \leq_{e'} \text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$. The proof idea is the same as in Theorem 5.5, except now, for each tuple extended from R_i^I to $R_i^{I^+}$ we can have at most c new tuples. Since this process is only done a constant number of times, the construction still only requires linear time, and the rest of the proof holds. Note that now one solution of $\text{ENUM}_{\Delta^+}\langle Q^+\rangle$ may correspond to several solutions of $\text{ENUM}_{\Delta^1}\langle Q\rangle$, as some variables were possibly added to the head. However, as the possible values of the added head variables are bounded by CDs, the number of solutions of Q^+ that correspond to one solution of Q is bounded by a constant.

We now formally prove that $\text{ENUM}_{\Delta^1}\langle Q\rangle \leq_{e'} \text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$. Denote Q by $Q(\vec{p}) \leftarrow R_1(\vec{v}_1), \ldots, R_m(\vec{v}_m)$. Given an instance I of $\text{ENUM}_{\Delta}\langle Q\rangle$, we define $\sigma(I)$. We start by removing tuples that interfere with the extended dependencies. For every dependency $\delta = (R_j\colon X \to y, c) \in \Delta^1$ and every atom $R_k(\vec{v}_k)$ that contains $X \cup \{y\}$, we correct R_k according to δ: we only keep tuples of R_k^I that agree with some tuple of R_j^I over the values of $X \cup \{y\}$. Next, we follow the extension of the schema and extend the instance accordingly. This phase results in a sequence of instances $I_0, I_1, \ldots, I_n = \sigma(I)$ that correspond to a sequence of queries $Q = Q_0, Q_1, \ldots, Q_n = Q^+$ such that each query is the result of extending an atom or the head of the previous query according to an FD. If in step i the head was extended, we set $I_{i+1} = I_i$. Now assume some relation R_k is extended according to some CD $(R_j\colon X \to y, c)$. For each tuple $t \in R_k^{I_i}$,

if there is no tuple $s \in R_j^{I_i}$ that agrees with t over the values of X, then we remove t altogether. Otherwise, we consider all values such tuples assign y. Denote those values by $a_1, \ldots, a_m$, and note that due to the CD, $m \leq c$. We copy t to $R_k^{I_{i+1}}$ m times, each time assigning y with a different value of $a_1, \ldots, a_m$. Given an answer $\mu \in Q^+(\sigma(I))$, we define $\tau(\mu)$ to be the projection of μ to $\text{free}(Q)$.

We need to show that $Q(I) = \{\mu|_{\text{free}(Q)} : \mu|_{\text{free}(Q^+)} \in Q^+(\sigma(I))\}$, and that an element of the left-hand side may only appear a constant amount of times on the right-hand side. First, if $\mu|_{\text{free}(Q^+)} \in Q^+(\sigma(I))$, since all tuples of $\sigma(I)$ appear (perhaps projected) in I, then $\mu|_{\text{free}(Q)} \in Q(I)$. It is left to show the opposite direction: if $\mu|_{\text{free}(Q)} \in Q(I)$ then $\mu|_{\text{free}(Q^+)} \in Q^+(I^+)$. We show by induction on $Q = Q_0, Q_1, \ldots, Q_n = Q^+$ that $\mu|_{\text{free}(Q_i)} \in Q_i(I_i)$. The induction base holds since in the cleaning phase we did not remove "useful" tuples. Since $\mu|_{\text{free}(Q)} \in Q(I)$, there exist tuples, one of each relation of the query, that agree on the values of $X \cup \{y\}$ (they all assign them with the values μ assigns them). These tuples were not removed in the cleaning phase, and therefore $\mu|_{\text{free}(Q)} \in Q(I_0)$. Next assume that $\mu|_{\text{free}(Q_i)} \in Q_i(I_i)$, and we want to show that $\mu|_{\text{free}(Q_{i+1})} \in Q_{i+1}(I_{i+1})$. This claim is trivial in case the head was extended. Note that there can be at most $c-1$ different answers $\mu'|_{\text{free}(Q_{i+1})}$ in $Q_{i+1}(I_{i+1})$ such that $\mu|_{\text{free}(Q_{i+1})} \neq \mu'|_{\text{free}(Q_{i+1})}$ but $\mu|_{\text{free}(Q_i)} = \mu'|_{\text{free}(Q_i)}$, as the added variable y is bound by the CD to have at most c possible values. Now consider the case where an atom $R_k(\vec{v}_k)$ was extended according to a CD $(R_j \colon X \to y, c)$. The tuple $\mu(\vec{v}_k) \in R_k^{I_i}$ was extended with the value $\mu(y)$ due to the tuple $\mu(\vec{v}_j) \in R_j^{I_i}$ that agrees with it on the values of X, and so $\mu(\vec{v}_k, y) \in R_k^{I_{i+1}}$. In case of self-joins, other atoms with the relation R_k are extended with a new and distinct variable. Such variables will be mapped to this value $\mu(y)$ as well. Overall, we have that μ (extended by mappings of the fresh variables) is also a homomorphism in $Q_{i+1}(I_{i+1})$. □

We can now extend our positive results to accommodate CDs. Let Q be a CD-free-connex CQ over a schema $(\mathcal{R}, \Delta)$, where Δ contains CDs. According to Lemma 5.26, $\text{ENUM}_{\Delta}\langle Q\rangle \leq_{e'} \text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle \leq_e \text{ENUM}_{\emptyset}\langle Q^+\rangle$, and due to Theorem 2.1, $\text{ENUM}_{\emptyset}\langle Q^+\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$. The class $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$ is closed under this type of reduction since we assume we have access to enough space. To avoid printing duplicates, we need to store previous results in a lookup table, and verify that a generated result is new before printing it. This alone is not enough, as we can have a long sequence of generating known results, and then the delay between generating new results can be larger than constant. For this reason, we delay the results. If every answer to $\text{ENUM}_{\Delta}\langle Q\rangle$ corresponds to at most c answers to $\text{ENUM}_{\emptyset}\langle Q^+\rangle$, we save the newly generated results in a queue, and after generating c results we pop and print a result from the queue. This guarantees that the queue is never empty when accessed, and the results are printed with constant delay. This type of manipulation is formalized in Chapter 3 as part of the Cheater's Lemma (Lemma 3.4). As a result, $\text{ENUM}_{\Delta}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$, and we deduce the following theorem.

Theorem 5.27. *Let Q be a CD-acyclic CQ with no self-joins over the schema $(\mathcal{R}, \Delta)$, where Δ is a set of CDs.*

- *If Q is CD-free-connex, then* $\text{ENUM}_{\Delta}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$.
- *If Q is not CD-free-connex, then* $\text{ENUM}_{\Delta}\langle Q\rangle \notin \mathsf{Enum}\langle \text{lin}, \text{polylog}\rangle$, *assuming* sparseBMM.

Note that, while the hardness results with respect to enumeration imply the hardness of random-permutation and random access, this is not the case for the positive results. In particular, we do not have that the classes $\mathsf{REnum}\langle \text{lin}, \log\rangle$ and $\mathsf{RAccess}\langle \text{lin}, \log\rangle$ are closed with respect to the type of relaxed reduction used in Lemma 5.26. For example, if we have an algorithm that produces outputs with uniformly-random order but every answer appears in an output a different amount of times, the trick of storing results to avoid duplicates does not result in a uniformly-random order.

5.3.2 CQs with Disequalities

As mentioned in Section 3.4.1, in the context of classifying CQs over a general schema with respect to $\mathsf{Enum}\langle \text{lin}, \text{const}\rangle$, disequalities have no effect on the enumeration complexity [BDG07]. In this section, we show that all enumeration results of this chapter apply to CQs with disequalities. To keep the proof ideas clear, we begin by assuming that the schema only contains FDs, and we incorporate CDs at the end of this section. A *CQ with disequalities* over a schema $(\mathcal{R}, \Delta)$ is an expression of the form

$$Q(\vec{p}) \leftarrow R_1(\vec{v}_1), \ldots, R_m(\vec{v}_m), w_1 \neq w_2, \ldots, w_{k-1} \neq w_k$$

where $Q(\vec{p}) \leftarrow R_1(\vec{v}_1), \ldots, R_m(\vec{v}_m)$ is a CQ denoted by Q_{base}, and $w_1, \ldots, w_k$ are variables in $\text{var}(Q)$. We denote $\text{dis}(Q) = \{w_1 \neq w_2, \ldots, w_{k-1} \neq w_k\}$. The evaluation is $Q(I) = \{\mu|_{\vec{p}} \in Q_{\text{base}}(I) | \forall w_i \neq w_j \in \text{dis}(Q) : \mu(w_i) \neq \mu(w_j)\}$.

The structural definitions of CQs with disequalities depend only on their bases. That is, Q is said to be *acyclic* if Q_{base} is acyclic, and we similarly define *cyclic* and *free-connex* CQs with disequalities. The FD-extension of a CQ with disequalities is denoted by Q^+. The base extends as before, and the disequalities remain the same. That is, $(Q^+)_{\text{base}} = Q^+_{\text{base}}$ is the FD-extension of Q_{base}, and $\text{dis}(Q^+) = \text{dis}(Q)$. We say that Q is *FD-acyclic* if Q_{base} is FD-acyclic, and similarly define *FD-free-connex* and *FD-cyclic* CQs with disequalities.

We first show that our positive results apply also to CQs with disequalities. The following lemma is the equivalent of the positive case of Theorem 5.5. In the proof, the reduction from the query to its extension works as before for the base query, and the disequalities are satisfied as they remain the same during the extension.

Lemma 5.28. *Let Q be a CQ with disequalities over a schema $(\mathcal{R}, \Delta)$, and let Q^+ be its FD-extension. Then* $\text{ENUM}_{\Delta}\langle Q\rangle \leq_e \text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$.

Proof. Recall that Q^+_{base} is the FD-extension of Q_{base}. According to the proof of Theorem 5.5, we can show that $\text{ENUM}_{\Delta}\langle Q_{\text{base}}\rangle \leq_e \text{ENUM}_{\Delta_{Q^+}}\langle Q^+_{\text{base}}\rangle$ by using a construction σ

such that $\mu|_{\text{free}(Q)} \in Q_{\text{base}}(I)$ iff $\mu|_{\text{free}(Q^+)} \in Q^+_{\text{base}}(\sigma(I))$. We use this same construction to show that $\text{ENUM}_\Delta\langle Q\rangle \leq_e \text{ENUM}_{\Delta_{Q^+}}\langle Q^+\rangle$. By definition we have that $\mu|_{\text{free}(Q)} \in Q(I)$ iff $\mu|_{\text{free}(Q)} \in Q_{\text{base}}(I)$ and in addition $\forall u \neq w \in \text{dis}(Q) : \mu(u) \neq \mu(w)$. As in the proof of Theorem 5.5, and since $\text{dis}(Q) = \text{dis}(Q^+)$, this is true iff $\mu|_{\text{free}(Q^+)} \in Q^+_{\text{base}}(\sigma(I))$ and $\forall u \neq w \in \text{dis}(Q^+) : \mu(u) \neq \mu(w)$. This is equivalent to $\mu|_{\text{free}(Q^+)} \in Q^+(\sigma(I))$. We conclude that $\mu|_{\text{free}(Q)} \in Q(I)$ iff $\mu|_{\text{free}(Q^+)} \in Q^+(\sigma(I))$. □

Combining Lemma 5.28 with the full version of Theorem 2.1 that holds for CQs with disequalities [BDG07], we have that if Q is an FD-free-connex CQ with disequalities over a schema $(\mathcal{R}, \Delta)$, then $\text{ENUM}_\Delta\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$.

We now discuss the lower bound. In the next lemma, we use the same idea proposed by Bagan et al. [BDG07] to extend their negative results to CQs with disequalities.

Lemma 5.29. *If Q is a CQ with disequalities over a schema $(\mathcal{R}, \Delta)$, then we have that $\text{ENUM}_\Delta\langle Q_{\text{base}}\rangle \leq_e \text{ENUM}_\Delta\langle Q\rangle$.*

Proof. Given an instance I of $\text{ENUM}_\Delta\langle Q_{\text{base}}\rangle$, we construct $\sigma(I)$ by assigning every variable a disjoint domain (this is the same construction used in Chapter 3 as part of Lemma 3.11). Formally, for every atom $R(\vec{v})$ of Q_{base} and every tuple $(c_1, \ldots, c_t) \in R^I$, we have the tuple $((c_1, \vec{v}[1]), \ldots, (c_t, \vec{v}[t]))$ in $R^{\sigma(I)}$. We claim that the answers of Q_{base} over I are exactly those of Q over $\sigma(I)$ if we omit the variable names. That is, we define $\tau : \mathit{dom} \times \text{var}(Q) \to \mathit{dom}$ as $\tau((c, v)) = c$, and show $Q_{\text{base}}(I) = \tau(Q(\sigma(I)))$.

Let $\mu|_{\text{free}(Q)} \in Q_{\text{base}}(I)$. Then, for every atom $R(\vec{v})$ of Q_{base}, there exists a tuple $(\mu(\vec{v}[1]), \ldots, \mu(\vec{v}[t])) \in R^I$. It results in $((\mu(\vec{v}[1]), \vec{v}[1]), \ldots, (\mu(\vec{v}[t]), \vec{v}[t])) \in R^{\sigma(I)}$ by definition of σ. Therefore, as we define the mapping $f_\mu : \text{var}(Q) \to \mathit{dom} \times \text{var}(Q)$ to be $f_\mu(u) = (\mu(u), u)$, we have that $f_\mu|_{\text{free}(Q)} \in Q_{\text{base}}(\sigma(I))$. Note that for all $u, w \in \text{var}(Q)$ with $u \neq w$ we have $f_\mu(u) \neq f_\mu(w)$. Therefore, $f_\mu \in Q(\sigma(I))$ as all disequalities in $\text{dis}(Q)$ are satisfied. Since $\tau \circ f_\mu = \mu$, we have that $\mu|_{\text{free}(Q)} \in \tau(Q(\sigma(I)))$, and this concludes that $Q_{\text{base}}(I) \subseteq \tau(Q(\sigma(I)))$. The opposite direction holds as well. If $\nu|_{\text{free}(Q)} \in Q(\sigma(I))$, then for every atom $R(\vec{v})$ in Q we have that $\nu(\vec{v}) \in R^{\sigma(I)}$. By construction, $\tau(\nu(\vec{v})) \in R^I$, and therefore $\tau \circ \nu|_{\text{free}(Q)} \in Q_{\text{base}}(I)$. □

By combining Theorem 5.5 with Lemma 5.29, we get the following reduction.

Lemma 5.30. *If Q is a CQ with disequalities over a schema $(\mathcal{R}, \Delta)$, then we have that $\text{ENUM}_{\Delta_{Q^+}}\langle Q^+_{\text{base}}\rangle \leq_e \text{ENUM}_\Delta\langle Q\rangle$.*

Lemma 5.30 means that the hardness results of this chapter extend to CQs with disequalities. In particular, if a self-join-free Q is FD-acyclic but not FD-free-connex, then $\text{ENUM}_\emptyset\langle \Pi\rangle \leq_e \text{ENUM}_{\Delta_{Q^+}}\langle Q^+_{\text{base}}\rangle \leq_e \text{ENUM}_\Delta\langle Q\rangle$ by Lemma 5.15 and Lemma 5.30. We get that then $\text{ENUM}_\Delta\langle Q\rangle \notin \mathsf{Enum}\langle \text{lin}, \text{polylog}\rangle$ assuming sparseBMM. Similarly, we can use Lemma 5.22 and Lemma 5.30 to show that if a self-join-free Q is FD-cyclic, and the schema only contains unary FDs, then $\text{DECIDE}_\Delta\langle Q\rangle$ cannot be solved in linear time assuming Hyperclique.

We next consider CQs with disequalities in the presence of CDs. The ideas presented in Section 5.3.1 and Section 5.3.2 can be combined to show the following lemma.

Lemma 5.31. *Let Q be a CQ with disequalities over a schema $(\mathcal{R}, \Delta)$, where Δ is a set of CDs. We have:*

- $\textsc{Enum}_{\Delta}\langle Q\rangle \leq_{e'} \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$
- $\textsc{Enum}_{\Delta^{FD}_{Q^+}}\langle Q^+_{\text{base}}\rangle \leq_e \textsc{Enum}_{\Delta}\langle Q\rangle$

The proof for $\textsc{Enum}_{\Delta}\langle Q\rangle \leq_{e'} \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$ works similarly to that of Lemma 5.28, except it builds upon Lemma 5.26 instead of Theorem 5.5. We use the same reduction as in Lemma 5.26, and the disequalities remain unchanged. By combining Lemma 5.29 with Lemma 5.25, he have that $\textsc{Enum}_{\Delta^{\text{FD}}_{Q^+}}\langle Q^+_{\text{base}}\rangle \leq_e \textsc{Enum}_{\Delta}\langle Q_{\text{base}}\rangle \leq_e \textsc{Enum}_{\Delta}\langle Q\rangle$.

Lemma 5.31 along with Lemma 5.15, Lemma 5.22 and the positive results by Bagan et al. [BDG07, Theorem 13, Theorem 17] prove that all results presented in this chapter apply to CQs with disequalities over schemas with cardinality dependencies. The following theorem summarizes our classification results.

Theorem 5.32. *Let Q be a CQ with disequalities over a schema $(\mathcal{R}, \Delta)$, where Δ is a set of CDs.*

- *If Q is CD-free-connex, then $\textsc{Enum}_{\Delta}\langle Q\rangle \in \mathsf{Enum}\langle \text{lin}, \text{const}\rangle$.*
- *If Q is self-join-free, CD-acyclic but not CD-free-connex, then $\textsc{Enum}_{\Delta}\langle Q\rangle \notin \mathsf{Enum}\langle \text{lin}, \text{polylog}\rangle$, assuming* sparseBMM.
- *If Q is self-join-free, CD-cyclic and Δ contains only unary CDs, then $\textsc{Decide}_{\Delta}\langle Q\rangle$ cannot be solved in linear time, and in particular $\textsc{Enum}_{\Delta}\langle Q\rangle \notin \mathsf{Enum}\langle \text{lin}, \text{polylog}\rangle$, assuming* Hyperclique.

5.3.3 Unions of CQs

Enumeration

The techniques used for the positive results in this chapter and in Chapter 3 are similar: we extend the CQs, and if the extension has a free-connex form, then the original query is tractable. We devote this section to discussing whether there is a need to combine these two forms of extension and how this can be done. First, we discuss the need to combine the two techniques.

Example 5.33. Consider the UCQ $Q = Q_1 \cup Q_2$ with:

$$Q_1(x, w, u) \leftarrow R_1(x, y), R_2(y, z), R_3(z, w), R_4(u)$$
$$Q_2(x, w, u) \leftarrow R_1(x, w), R_2(t, u)$$

over a schema with the FD $R_3 : 2 \to 1$.

The CQ Q_1 in itself is not free-connex, as it contains the free-path (x, y, z, w). Even if we take the FD into account, Q_1 is not tractable. The FD-extension of Q_1 is obtained by adding z to the query head due to the FD $w \to z$.

$$Q_1^+(x, w, u, z) \leftarrow R_1(x, y), R_2(y, z), R_3(z, w), R_4(u).$$

Q_1^+ is not free-connex as it contains the free-path (x, y, z). This means that by using the FD technique alone, Q does not become tractable. If we take the union into account but ignore the FD, Q_2 can provide $\{x, y, z\}$ to Q_1, but adding atoms with any subset of these variables cannot make Q_1 tractable, because it cannot prevent the appearance of a free-path between x and w. It can only shorten it from (x, y, z, w) to (x, z, w). This means that by using the union techniques alone, Q does not become tractable. Nonetheless, if we combine the extensions, we both add z to the head and add an atom with $\{x, y, z\}$, we get the following tractable extension:

$$Q_1^{++}(x, w, u, z) \leftarrow R_1(x, y), R_2(y, z), R_3(z, w), R_4(u), R'(x, y, z).$$

Practically, this means that Q can be solved efficiently by first solving Q_2, then using its results as another atom R', solving Q_1^{++} with the help of R', and translating every answer to Q_1^{++} to an answer to Q_1 by projecting out z. The FD guarantees that the translation phase does not result in duplicates. Therefore, Q is tractable. If we had a cardinality dependency instead of an FD, we would have a constant number of duplicates per answer to Q_1, and the duplicates could be ignored efficiently using the Cheater's Lemma (Lemma 3.4). Thus, with the CD, Q would remain tractable. □

Example 5.33 shows that if we have UCQs over a schema with FDs, we can identify additional tractable cases by combining the two techniques. The two types of extensions can easily be combined in any order. To show this, we just need to modify the definitions of a union extension to account for cardinality dependencies.

Definition 5.34. Let $Q = Q_1 \cup \ldots \cup Q_n$ be a UCQ over a schema with CDs. A *CD-union extension* of Q_1 is obtained by the addition of any number of atoms $R(\vec{v})$ to Q_1 such that $\vec{v}$ is provided by some CD-extension of some $Q_j \in Q$, and R is a fresh relational symbol. By way of recursion, the variables $\vec{v}$ may alternatively be provided by a CD-extension of a union extension of some $Q_j \in Q$.

With this definition, we can show the tractability of UCQs with a free-connex CD-union extension.

Theorem 5.35. *Let Q be a UCQ over a schema with cardinality dependencies. If Q has a free-connex CD-union extension, then* $\textsc{Enum}\langle Q\rangle \in \mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$.

Proof Sketch. Follow the proof of Theorem 3.9. Whenever the proof uses the CDY algorithm for evaluating the individual CQs with linear preprocessing and constant delay, use the adjusted algorithm for CD-extensions (Theorem 5.27) instead. □

Note that in Example 5.33 the order of applying the extension does not matter. That is, first adding R' and then adding z wherever w appears produces the same query as first adding z and then adding R'. This is not always the case. Consider the same UCQ as before but with the FD $R_3 : 1 \to 2$ instead of the reverse FD. When we treat the FD as between variables in Q_1, we get $z \to w$, and we can add w wherever z appears

in Q_1 as an FD-extension. The individual extensions are still not enough: adding w to R_2 results in the free-path (x, y, w), and adding an atom with any subset of $\{x, y, z\}$ does not help as before. The combination of the two extensions results in a free-connex CQ if we first add the provided atom and only then apply the FD. We get:

$$Q_1'(x, w, u) \leftarrow R_1(x, y), R_2(y, z, w), R_3(z, w), R_4(u), R'(x, y, z, w)$$

If we apply the extensions in the opposite order, we do not have w in R', and we have the free-path (x, z, w), so the extension is not tractable.

In conclusion, we saw that the two techniques can and should be combined to find more tractable cases. However, one should be aware that it is not enough to first take the CD-extensions of all CQs in the union and then apply union-extensions. The order in which the different extensions are combined matters.

Random Access and Random Permutation

We showed in Section 5.1 that for any FD-free-connex CQ there is an extended free-connex CQ such that there is a bijection between the two sets of answers. This means that by applying Theorem 4.12 on the extension, and then applying Theorem 5.5 to translate the answers, we can obtain a data structure that allows efficient counting, random-access and inverted-access of FD-free-connex CQs. This data structure can then be used to achieve efficient random-permutation algorithms for UCQs, and we can conclude Theorem 4.26 and Theorem 4.24 for FD-free-connexity as explained in Section 4.3 based on this structure.

Theorem 5.36. *Let Q be a union of FD-free-connex CQs.*

- *There exists a random-permutation algorithm for answering Q that uses linear preprocessing and expected logarithmic delay.*
- *If the intersection of every subset of the CQs is also FD-free-connex, then* $\textsc{Enum}\langle Q\rangle \in \mathsf{REnum}\langle \mathrm{lin}, \log\rangle$.

As mentioned before, in the case of CDs we do not have a bijection between the answers, so we cannot use CDs to extend our results in the same way.

5.4 Note on Space Usage

The algorithms presented in this chapter use linear memory during preprocessing and differ on the amount of memory they use for writing during enumeration. Recall that $\mathsf{CD}\circ\mathsf{Lin}$ denotes the problems that can be solved with the same time bounds as $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$, but with the additional restriction that the available space for writing during the enumeration phase is constant (see Section 3.4.2). All results regarding $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$ presented prior to the section with the extended settings (Section 5.3) also apply for $\mathsf{CD}\circ\mathsf{Lin}$. All lower bounds shown for $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$ imply the same lower bounds for the more restrictive $\mathsf{CD}\circ\mathsf{Lin}$. The positive results rely on Theorem 5.5,

where we show that $\textsc{Enum}_{\Delta}\langle Q\rangle \leq_e \textsc{Enum}_{\Delta_{Q^+}}\langle Q^+\rangle$, and the mapping used between individual results is merely a projection. Hence, we obtain that every FD-free-connex CQ is in $\mathsf{CD}\circ\mathsf{Lin}$. This is also true for CQs with disequalities. The positive results for CDs use the relaxed reduction form in Lemma 5.26. This may cause duplicates, and so we need to store the produced answers to ensure the uniqueness of printed answers and maintain constant delay. This means that our positive results for CDs do not apply also for $\mathsf{CD}\circ\mathsf{Lin}$.

Chapter 6

Enumerating Tree Decompositions

In the previous chapters, we proved the tractability of queries that can take a free-connex form. In this chapter, we inspect what we can do when none of the methods from the previous chapters can be applied. When the queries are neither naturally acyclic nor acyclic via extensions based on dependencies or unions, we can decompose the queries to reach an acyclic structure after a non-linear overhead. As this overhead depends exponentially on the quality of the decomposition, we inspect only the task of decomposition with the query as input. In particular, we no longer use data complexity, since the data is not part of the problem we address. As finding the best decomposition is NP-hard, we devise an anytime algorithm that enumerates decompositions efficiently and allows us to stop when a decomposition found is deemed good enough. It turns out that this problem coincides with the problem of enumerating *minimal triangulations*. The complexity of the latter problem was an open problem that we address in this chapter. Our suggested algorithm runs with the guarantee of incremental polynomial time, and it can incorporate any method for finding a single tree decomposition. It returns the result of this method as a first answer, and then repeatedly applies it (with modified inputs) to try and improve upon the initial result.

This chapter contains joint work with Batya Kenig, Benny Kimelfeld, and Markus Kröll. The findings of this chapter were published in The Symposium on Principles of Database Systems (PODS) 2017 [CKK17], and in an extended version in Discrete Applied Mathematics (DAM) in a special issue for Workshop on Enumeration Problems and Applications (WEPA) 2018 [CKKK20].

Organization. In Section 6.1 we give preliminary definitions and notation, and recall basic results from the literature. The algorithm is established in the next three sections through a series of reductions culminating in the enumeration of maximal independent sets over Succinct Graph Representations (SGRs). The SGR framework is presented in Section 6.2, along with an enumeration algorithm for maximal independent sets. In

Section 6.3 we prove that the graph of minimal separating sets satisfies the tractability requirements needed for the SGR enumeration algorithm, and thereby establish an algorithm for enumerating the minimal triangulations. We show how this algorithm can enumerate the proper tree decompositions in Section 6.4. Then, an experimental study is presented in Section 6.5.

6.1 Preliminaries

In this section we give some basic notation and terminology, and we recall basic theory that we need in this chapter.

6.1.1 Graphs

The graphs in this work are undirected. For a graph g, the set of nodes is denoted by $\mathsf{V}(g)$, and the set of edges (pairs $\{u, v\}$ of distinct nodes) is denoted by $\mathsf{E}(g)$. Let U be a set of nodes of a graph g. We denote by $g_{|U}$ the subgraph of g *induced* by U; that is, $\mathsf{V}(g_{|U}) = U$ and $\mathsf{E}(g_{|U}) = \{\{u, v\} \in \mathsf{E}(g) \mid \{u, v\} \subseteq U\}$. We denote by $g \setminus U$ the graph obtained from g by removing all the nodes in U (along with their incident edges), that is, the graph $g_{|\mathsf{V}(g)\setminus U}$.

Let g be a graph and U a set of nodes of g. We say that U is an *independent set* if it does not contain both endpoints of any edge, and it is a *maximal independent set* if it is an independent set and it is not strictly contained in any other independent set. We denote by $\mathit{MaxInd}(g)$ the set of all the maximal independent sets of g. We say that U is a *clique* (*of* g) if every two nodes of U are connected by an edge, and it is a *maximal clique* (*of* g) if it is a clique that is not strictly contained in any other clique. We denote by $\mathit{MaxClq}(g)$ the set of all the maximal cliques of g. The operation of *saturating* U (*in* g) is that of connecting every non-adjacent pair of nodes in U by a new edge. Hence, if h is obtained from g by saturating U, then U is a clique of h.

6.1.2 Minimal Separators

Let g be a graph, and let S be a subset of $\mathsf{V}(g)$. Let u and v be two nodes of g. We say that S is a (u, v)-*separator* if u and v belong to distinct connected components of $g \setminus S$. We say that S is a *minimal* (u, v)-separator if no strict subset of S is a (u, v)-separator. We say that S is a *minimal separator* if there are two nodes u and v such that S is a minimal (u, v)-separator. We denote by $\mathit{MinSep}(g)$ the set of all the minimal separators of g. We mention that the number of minimal separators (i.e., $|\mathit{MinSep}(g)|$) may be exponential in the number of nodes (i.e., $|\mathsf{V}(g)|$).

Let g be a graph, and let S and T be two minimal separators of g. We say that S *crosses* T, in notation $S \natural_g T$, if there are nodes u and v in T such that S is a (u, v)-separator. If g is clear from the context, we may omit it and write simply $S \natural T$. It is known that $\natural$ is a symmetric relation: if S crosses T then T crosses S [PS97, KKS97].

Hence, if $S \natural T$ then we may also say that S and T are *crossing*. When S and T are non-crossing, we also say that S and T are *parallel*.

6.1.3 Chordality and Triangulation

Let g be a graph. A *chord* of a cycle c of g is an edge $e \in \mathsf{E}(g)$ that connects two nodes that are non-adjacent in c. We say that g is *chordal* if every cycle of length greater than three has a chord. Whether a given graph is chordal can be decided in linear time [TY84]. Dirac [Dir61] has shown a characterization of chordal graphs by means of their minimal separators.

Theorem 6.1 ([Dir61])**.** *A graph g is chordal if and only if every minimal separator of g is a clique.*

Rose [Ros70] has shown that a chordal graph g has fewer minimal separators than nodes (i.e., if g is chordal then $|\mathit{MinSep}(g)| < |\mathsf{V}(g)|$), and Kumar and Madhavan [KM98] have shown that we can find all of these minimal separators in linear time.

Theorem 6.2 ([KM98])**.** *Let g be a chordal graph. The set $\mathit{MinSep}(g)$ can be computed in linear time.*

A *triangulation* of a graph g is a graph h such that $\mathsf{V}(g) = \mathsf{V}(h)$, $\mathsf{E}(g) \subseteq \mathsf{E}(h)$, and h is chordal. The edges in $\mathsf{E}(h) \setminus \mathsf{E}(g)$ are commonly referred to as *fill edges*. A *minimal triangulation* of g is a triangulation h of g with the following property: for every graph h' with $\mathsf{V}(g) = \mathsf{V}(h')$, if $\mathsf{E}(g) \subseteq \mathsf{E}(h') \subsetneq \mathsf{E}(h)$ then h' is non-chordal (or in other words, h' is not a triangulation of g). In particular, if g is already chordal then g is the only minimal triangulation of itself. We denote by $\mathit{MinTri}(g)$ the set of all the minimal triangulations of g.

6.1.4 Tree Decomposition

Let g be a graph. A *tree decomposition* d of g is a pair (t, β), where t is a tree and $\beta : \mathsf{V}(t) \to 2^{\mathsf{V}(g)}$ is a function that maps every node of t into a set of nodes of g, so that all of the following hold: (1) Nodes are covered: for every node $u \in \mathsf{V}(g)$ there is a node $v \in \mathsf{V}(t)$ such that $u \in \beta(v)$; (2) Edges are covered: for every edge $e \in \mathsf{E}(g)$ there is a node $v \in \mathsf{V}(t)$ such that $e \subseteq \beta(v)$; (3) *Junction-tree* (or *running-intersection*) property: for all nodes $u, v, w \in \mathsf{V}(t)$, if v is on the path between u and w, then $\beta(v)$ contains $\beta(u) \cap \beta(w)$. Note that this is equivalent to the running-intersection property in the definition of a join-tree.

Let g be a graph, and let $d = (t, \beta)$ be a tree decomposition of g. For a node v of t, the set $\beta(v)$ is called a *bag* of d. We denote by $\mathrm{bags}(d)$ the set $\{\beta(v) \mid v \in \mathsf{V}(t)\}$, and we denote by $\textsc{saturate}(g, d)$ the graph obtained from g by saturating (i.e., adding an edge between every pair of nodes in) every bag of d. Jordan [Jor02] has shown the following characterization of chordal graphs by means of tree decompositions.

Theorem 6.3 ([Jor02])**.** *A graph g is chordal if and only if it has a tree decomposition d such that every bag of d is a clique of g.*

6.1.5 Enumerating the Minimal Triangulations

A common approach to establish enumeration with polynomial delay is via the technique known as the *branch-and-bound* (or the *flashlight*) method [BEG04]. In this approach, we find a condition ψ over the answers, and then recursively enumerate all of the answers that satisfy ψ and all of the answers that violate ψ (i.e., satisfy $\psi' = \neg\psi$). Hence, in each recursive call, we need to enumerate all the answers that satisfy a conjunction $\psi_1 \wedge \cdots \wedge \psi_m$ of such conditions. For this approach to guarantee polynomial delay, the depth of the recursion should be bounded by a polynomial in the size of the input. Importantly, in each recursive call, we should be able to test whether there is *at least one* answer that satisfies $\psi_1 \wedge \cdots \wedge \psi_m$. Then, in the leaves, we should be able to produce the single answer that satisfies the given constraints.

In this chapter, we devise an algorithm for enumerating the minimal triangulations: given g, enumerate $\mathit{MinTri}(g)$. A branch-and-bound attempt to solve this problem would be, say, to apply the conditions of inclusion and exclusion of fill edges. This approach amounts to testing whether there is a minimal triangulation that contains a given set of edges and excludes another set of edges. Unfortunately, it follows from known hardness results of Golumbic, Kaplan and Shamir [GKS95] that this problem is intractable, as the following proposition shows.

Proposition 6.4. *The following decision problem is NP-complete: given g and two sets I and X of node pairs, is there a minimal triangulation h of g such that $I \subseteq \mathsf{E}(h)$ and $\mathsf{E}(h) \cap X = \emptyset$?*

Proof. Membership in NP is straightforward. To show hardness, we use a reduction from the *chordal sandwich problem*. For a graph property Π, the sandwich problem for Π is that of determining, given graphs g and g'' with $\mathsf{V}(g) = \mathsf{V}(g'')$ and $\mathsf{E}(g) \subseteq \mathsf{E}(g'')$, where there exists a graph g' such that $\mathsf{V}(g') = \mathsf{V}(g)$, $\mathsf{E}(g) \subseteq \mathsf{E}(g') \subseteq \mathsf{E}(g'')$, and g' satisfies Π. This problem is NP-hard for various graph properties Π, including chordality [GKS95]. Now, given g and g'', let X be the set of all the node pairs that are *not* edges of g''. The existence of g' in the chordal sandwich problem is equivalent to the existence of a (minimal) triangulation of g that excludes X. □

Hence, we adopt a different approach to enumerating the minimal triangulations, as we describe in the following section.

6.2 Maximal Independent Sets in Succinct Graphs

The main result of this chapter is an algorithm for enumerating the minimal triangulations of a graph g. As we explain in the next section, this problem amounts to

enumerating the maximal independent sets of a graph h. It is known that the maximal independent sets of a graph can be enumerated with polynomial delay [JPY88]. However, we cannot instantiate h, since the number of nodes of h can be exponential in the size of g. Known algorithms for enumerating maximal independent sets cannot be applied to solve our problem. Nevertheless, h possesses some tractability properties that, in fact, allow us to efficiently enumerate the maximal independent sets of h. In this section, we identify these properties within an abstract framework of *succinct graph representations*, where a graph may be exponentially larger than its representation, and we have access to the nodes and edges through efficient algorithms. Mainly, we devise an algorithm for enumerating the maximal independent sets for such graphs.

6.2.1 Succinct Graph Representations

We begin with the formal definition of a succinct graph representation.

Definition 6.5 (SGR)**.** A *Succinct Graph Representation*, or *SGR* for short, is a triple $(\mathcal{G}, A_{\mathsf{V}}, A_{\mathsf{E}})$ such that:

- $\mathcal{G}$ is a function that maps every input x, called an *instance*, to a graph $\mathcal{G}(x)$;
- A_{V} is an enumeration algorithm that, given an instance x, enumerates the nodes of $\mathcal{G}(x)$;
- A_{E} is a decision algorithm that, given an instance x and nodes v and u of $\mathcal{G}(x)$, determines whether v and u are connected by an edge in $\mathcal{G}(x)$.

An SGR $(\mathcal{G}, A_{\mathsf{V}}, A_{\mathsf{E}})$ is said to be *tractably accessible* if both the following hold.

1. A_{V} enumerates with polynomial delay.
2. A_{E} terminates in polynomial time.

Here, both polynomials are with respect to $|x|$ (the length of x). Observe that in a tractably accessible SGR, the (representation) size of every node v of $\mathcal{G}(x)$ is polynomial in that of x (since writing v is within the polynomial delay).

For efficient enumeration of $\mathit{MaxInd}(\mathcal{G}(x))$, we need additional tractability conditions.

Definition 6.6 (Tractable Expansion)**.** A tractably accessible SGR $(\mathcal{G}, A_{\mathsf{V}}, A_{\mathsf{E}})$ is said to have a *tractable expansion* if both of the following conditions hold.

1. There is a polynomial p such that $|I| \leq p(|x|)$ for all instances x and independent sets I of $\mathcal{G}(x)$.
2. There is a polynomial-time algorithm that, given x and an independent set I of $\mathcal{G}(x)$, either determines that I is maximal or returns a node $v \notin I$ such that $I \cup \{v\}$ is independent.

Following is an example of an SGR that is central to this work.

6.2.2 The Separator Graph as an SGR

The *separator graph* of a graph g is the graph that has the set $\mathit{MinSep}(g)$ of minimal separators as its node set, and an edge between every two minimal separators that are

crossing (i.e., $S, T \in \mathit{MinSep}(g)$ such that $S \natural T$). Throughout this chapter we denote by $\mathsf{MSGraph}$ the SGR $(\mathcal{G}^{\mathsf{ms}}, A_{\mathsf{V}}^{\mathsf{ms}}, A_{\mathsf{E}}^{\mathsf{ms}})$, where:

- $\mathcal{G}^{\mathsf{ms}}$ is a function mapping an input graph g (playing the role of x from Definition 6.5) to its separator graph.
- $A_{\mathsf{V}}^{\mathsf{ms}}$ is an enumeration algorithm that, given a graph g, enumerates its set $\mathit{MinSep}(g)$ of minimal separators. We can use here a variation of the algorithm of Berry et al. [BBC99] that enumerates $\mathit{MinSep}(g)$ with polynomial delay, as we describe later in Section 6.3.2.
- $A_{\mathsf{E}}^{\mathsf{ms}}$ is an algorithm that, given a graph g and two minimal separators S and T, determines whether $S \natural T$ efficiently (e.g., by removing S and testing whether T is split along multiple connected components).

In particular, $\mathsf{MSGraph}$ is a tractably accessible SGR. We will show later (Theorem 6.20) that $\mathsf{MSGraph}$ also has a tractable expansion.

6.2.3 Enumerating Maximal Independent Sets in SGRs

Our main result for this section is the following.

Theorem 6.7. *Let $(\mathcal{G}, A_{\mathsf{V}}, A_{\mathsf{E}})$ be a tractably accessible SGR with a tractable expansion. There is an algorithm that, given an instance x, enumerates the set $\mathit{MaxInd}(\mathcal{G}(x))$ in incremental polynomial time.*

The proof is via the algorithm EnumMIS that is depicted in Algorithm 6.1. This algorithm is an adaptation of the algorithm for computing full disjunctions in databases [CFK+06] that generalizes the problem of enumerating maximal cliques (or maximal independent sets). In turn, that algorithm was based on an improvement of the algorithm of Lawler et al. [LLK80] for generating the maximal independent sets in polynomial total time, and all rely on the general idea of reducing the problem to the *input-restricted problem* that was later introduced by Cohen et al. [CKS08] for enumerating maximal node sets that satisfy a hereditary property.

The underlying idea of that algorithm is to construct a graph over the space of the solutions (maximal independent sets), and traverse the graph in a depth-first-search manner. In the case of maximal independent sets, there is an edge from J to K if K is obtained from J by adding a new node v, removing the neighbours of v, and greedily extending to a maximal independent set.

In this section, we describe the algorithm and prove its correctness and efficiency. In the remainder of this section, we fix a tractably accessible SGR $(\mathcal{G}, A_{\mathsf{V}}, A_{\mathsf{E}})$ with tractable expansion, and an input instance x. Our goal is to enumerate $\mathit{MaxInd}(\mathcal{G}(x))$.

6.2.4 Algorithm Description

As explained earlier, the algorithm extends every maximal independent set J that it generates in the direction of every node v that it generates. By *extending J in the*

Algorithm 6.1 Enumerating maximal independent sets for an SGR

```
1:  procedure EnumMIS((G, A_V, A_E), x)
2:    J :- Extend(x, ∅)
3:    print J
4:    Q :- {J}
5:    P :- ∅
6:    V :- ∅
7:    iterator :- A_V(x)
8:    while Q ≠ ∅ do
9:      J :- Q.pop()
10:     P.push(J)
11:     for all v ∈ V do
12:       J_v :- {v} ∪ {u ∈ J | ¬A_E(x, v, u)}
13:       K :- Extend(x, J_v)
14:       if K ∉ Q ∪ P then
15:         print K
16:         Q :- Q ∪ {K}
17:     while Q = ∅ and iterator.hasNext() do
18:       v :- iterator.next()
19:       V :- V ∪ {v}
20:       for all J' ∈ P do
21:         J'_v :- {v} ∪ {u ∈ J' | ¬A_E(x, v, u)}
22:         K :- Extend(x, J'_v)
23:         if K ∉ Q ∪ P then
24:           print K
25:           Q :- Q ∪ {K}
```

direction of v we mean producing an arbitrary maximal independent set K that contains v and all nodes in J that are non-neighbors of v. As long as there are unprocessed sets, they are extended in the direction of all previously generated nodes. When no unprocessed sets are left, additional nodes are generated, and the previously processed sets are extended in the direction of the new nodes. Put differently, our algorithm adapts the traversal approach by restricting the steps to the solutions that are obtained by extending in the direction of the nodes v that have been produced until that point of time; when a new node v is generated, we revisit the past solutions and take the steps implied by v.

The algorithm maintains two collections, $\mathcal{Q}$ and $\mathcal{P}$, for storing answers (which are maximal independent sets of the graph $\mathcal{G}(x)$). The algorithm inserts answers into $\mathcal{Q}$, and repeatedly *removes* (or *pops*) an answer from $\mathcal{Q}$ and *processes* that answer (while possibly inserting new answers into $\mathcal{Q}$), until $\mathcal{Q}$ is empty. The set $\mathcal{P}$ stores the answers that have already been removed from $\mathcal{Q}$ and processed. Importantly, both collections feature membership testing, element removal and element insertion with a number of comparisons logarithmic in their cardinality (i.e., the number of answers they hold at the time of the operation; this can be achieved for example by using balanced binary

search trees). In addition, the algorithm maintains a collection $\mathcal{V}$ of the nodes of $\mathcal{G}(x)$ generated thus far. The collection $\mathcal{Q}$ is initialized with a single result (which is printed after being generated), which is an arbitrary maximal independent set. This result is obtained through the procedure EXTEND(x, I) that extends a given independent set I into a maximal one. Note that this procedure can be implemented in polynomial time, since $(\mathcal{G}, A_{\mathsf{V}}, A_{\mathsf{E}})$ has a tractable expansion. The sets $\mathcal{P}$ and $\mathcal{V}$ are initialized empty.

The algorithm accesses the nodes of $\mathcal{G}(x)$ through an iterator object that is obtained by executing $A_{\mathsf{V}}(x)$, and features two polynomial-time operations:

- HASNEXT() determines whether there are additional nodes of $\mathcal{G}(x)$ to enumerate.
- NEXT() returns the next node in the iteration.

The algorithm applies the iteration of line 8 until $\mathcal{Q}$ becomes empty, and then terminates. In every iteration, the algorithm pops an element from $\mathcal{Q}$, stores it in $\mathcal{P}$ (lines 9–10), and then processes it. The algorithm iterates through the nodes in $\mathcal{V}$, and for each node v it applies (in lines 12–16) what we call *extension of J in the direction of v*:

1. Generate the set J_v that consists of v and all the nodes in J that are non-neighbors of v, using the algorithm A_{E} for testing adjacency;
2. Extend J_v into an arbitrary maximal independent set K using EXTEND(x, J_v);
3. If K is in neither $\mathcal{Q}$ nor $\mathcal{P}$ (meaning it was not printed before), then print K and add it to $\mathcal{Q}$.

Note that J_v is an independent set, so it is possible to invoke EXTEND(x, J_v) with J_v.

Up to this point, the algorithm is very similar to the algorithm of Cohen et al. [CFK$^+$06] for computing full disjunctions, except that $\mathcal{V}$ does not hold all nodes but only the nodes generated so far. The twist (and the source of extra challenge in proving correctness and efficiency) is in lines 17–25, where we generate additional nodes and compensate for them being missing in the previous iterations. In these lines, the algorithm tests whether it is the case that $\mathcal{Q}$ is empty and the node iterator has additional nodes to process (line 17). While this is the case, the algorithm repeats the following procedure (lines 18–25): generate the next node using the iterator of $A_{\mathsf{V}}(x)$, add it to $\mathcal{V}$, and extend *every* previously processed result (i.e., the results in $\mathcal{P}$) in the direction of the newly generated node v (as previously described).

6.2.5 Correctness and Efficiency

The following lemma states the correctness of the algorithm: the algorithm enumerates *every* element in $\mathit{MaxInd}(\mathcal{G}(x))$, *only* elements in $\mathit{MaxInd}(\mathcal{G}(x))$, and every element is printed exactly once.

Lemma 6.8. ENUMMIS(x) *enumerates* $\mathit{MaxInd}(\mathcal{G}(x))$.

Proof. The algorithm prints only elements that are created by invoking the procedure EXTEND. Therefore, the algorithm prints only elements in $\mathit{MaxInd}(\mathcal{G}(x))$. The tests of lines 14 and 23 ensure that whenever an element is printed, this element has not been

seen before. Hence, no element is printed more than once. It is left to prove that every maximal independent set of $\mathcal{G}(x)$ is printed by the algorithm.

Observe the following. When the algorithm terminates we have $\mathcal{Q} = \emptyset$. Therefore, in the previous iteration the loop of line 17 could only have terminated due to iterator.hasNext() returning false. Therefore, upon termination $\mathcal{V} = \mathsf{V}(\mathcal{G}(g))$.

Suppose, by way of contradiction, that there is some maximal independent set H that is not printed by the algorithm. Let J be a maximal independent set of $\mathcal{G}(x)$, among all the printed ones, that contains a maximal number of elements from H. The set J must exist, since the algorithm prints at least one maximal independent set. Let H_m be the intersection $H \cap J$. Since $H \neq H_m$ (or else H is not maximal), there is at least one node in $H \setminus J$; let v be such a node.

At this point we established that before the algorithm terminates, *(a)* the node v was generated, *and (b)* J was printed. We now branch into two cases, as follows.

1. The set J was inserted into $\mathcal{P}$ before the node v was generated. Immediately after v is generated (in line 18), the set $J_v = \{v\} \cup \{u \in J \mid \neg A_{\mathsf{E}}(x, v, u)\}$ will be constructed (in line 21) and expanded to a maximal independent set K that contains J_v.
2. The node v was generated before J was inserted into $\mathcal{P}$. At the iteration when J is inserted into $\mathcal{P}$, we have $v \in \mathcal{V}$, and so the set $J_v = \{v\} \cup \{u \in J \mid \neg A_{\mathsf{E}}(x, v, u)\}$ will be constructed (in line 12) and expanded to a maximal independent set K that contains J_v.

So, we have established that before the algorithm terminates, the set J_v is generated and expanded to a maximal independent set K that contains J_v. Furthermore, $H_m \cup \{v\} \subseteq J_v$ (since $H_m \subseteq J$, and does not contain any neighbor of v), and therefore $H_m \cup \{v\} \subseteq K$. According to the algorithm, one of the following options must hold: (1) K is inserted into $\mathcal{Q}$, (2) K is already in $\mathcal{Q}$ (3) K was in $\mathcal{Q}$ in the past and is now in $\mathcal{P}$. Since the algorithm prints every maximal independent set that is inserted into $\mathcal{Q}$, we get a contradiction to the maximality of H_m. □

We now prove that the algorithm EnumMIS enumerates with incremental polynomial time. We do so in two steps. We first define an algorithm that is similar to EnumMIS, but with a small twist that makes it easier to prove incremental polynomial time. Then, we prove a general result that will imply that, if the new algorithm enumerates in incremental polynomial time then so does EnumMIS.

The new algorithm is similar to EnumMIS, except that each of the print commands (lines 3, 15 and 24) is replaced with an operation that takes the time of the printing, but is actually void (e.g., printing to `/dev/null` in Unix). Instead, each maximal independent set is printed immediately after being removed from $\mathcal{Q}$ (line 9). Hence, answers are *held* until removed from $\mathcal{Q}$. We refer to the resulting algorithm as EnumMISHold. For theory purposes, it would have been enough to discuss only EnumMISHold, which is easier to analyze. However, since delaying the results is not required to obtain the

theoretical guarantees, we also discuss EnumMIS where we print the results as soon as we have them. Next, we prove that EnumMISHold enumerates in incremental polynomial time. Observe that to bound the delay EnumMISHold, we only need to bound the time between two executions of line 9 of EnumMIS.

Lemma 6.9. EnumMISHold(x) *enumerates with incremental polynomial time.*

Proof. We begin by showing that the size of the node set $\mathcal{V}$ is polynomial in the size of the printed result set $\mathcal{P}$. Whenever a new node v is inserted into $\mathcal{V}$ (line 19), the set $\mathcal{Q}$ is empty. The following calls to Extend (line 22) will generate maximal independent sets containing v. Each of these maximal independent sets is either already in $\mathcal{P}$, or it is inserted into $\mathcal{Q}$ (line 25). Therefore, at the end of the iteration of the main loop in which v was inserted into $\mathcal{V}$, all maximal independent sets in $\mathcal{Q}$ contain v. In the next iteration of the main loop, if such an iteration exists, one of these newly generated independent sets will be printed and inserted into $\mathcal{P}$ (line 10). That is, at the beginning of every iteration of the algorithm (specifically, line 11), every node $v \in \mathcal{V}$ belongs to some maximal independent set that has already been printed (and thus part of $\mathcal{P}$). Since we assume tractable expansion, each independent set in $\mathcal{P}$ contains at most $p(|x|)$ nodes, and we can conclude that $|\mathcal{V}| \leq p(|x|) \cdot |\mathcal{P}|$.

We now bound the time between two executions of line 9 of EnumMIS. Line 10 takes polynomial time in $|x|$ (since there are at most exponentially many independent sets, $(\mathcal{G}, A_{\mathsf{V}}, A_{\mathsf{E}})$ has a tractable expansion, and operations on $\mathcal{P}$ require a logarithmic number of comparisons in the cardinality). The number of iterations of line 11 is at most the size of $\mathcal{V}$, which is polynomial in the number of answers printed so far (due to the above observation). Each operation in that iteration takes time polynomial in $|x|$.

The loop of line 17 repeats (at most) until a node that belongs to none of the printed answers is generated. Hence, the observation that this number is polynomial in the size of the output, along with the tractable expansion, again implies that the number of times we iterate is polynomial in the number of answers printed so far. The loop of line 20 repeats at most as many times as the number of answers in $\mathcal{P}$, and these were printed before. Besides the loops, each of lines 18–25 takes polynomial time in $|x|$. □

Lemma 6.9 shows that EnumMISHold enumerates with incremental polynomial time. Next, we show the same for EnumMIS. The key point is that every answer is printed in EnumMIS *no later* than it is printed in EnumMISHold. Note that this holds even though the two algorithms do not necessarily enumerate in the same order (as we make no assumptions about the order of removal in $\mathcal{Q}$), since we assume that EnumMISHold spends on void the printing time of EnumMIS. We will prove that this suffices to conclude that if EnumMISHold enumerates in incremental polynomial time, then so does EnumMIS. We prove here a general result. Let P be an enumeration problem, and let A be a solver for P. For input x and answer $y \in P(x)$, we denote by $\text{time}_{A,x}(y)$ the time in which y is printed. We prove the following theorem.

Theorem 6.10. *Let P be an enumeration problem, and let A and B be two solvers for P. Suppose that for all instances x and for all answers $y \in P(x)$ we have* $\text{time}_{A,x}(y) \leq \text{time}_{B,x}(y)$*. If B enumerates in incremental polynomial time, then so does A.*

Theorem 6.10 is not a vacuous statement, since the order of results may differ between A and B. Furthermore, the corollary no longer holds when substituting "incremental polynomial time" with "polynomial delay." For example, imagine two algorithms that print all subsets of an input set. The first prints a new answer after every two time ticks, while the second prints them after every single time tick, except for the last answer which is printed at the same time in both algorithms. The first algorithm meets the guarantee of polynomial delay, and even though the second algorithm prints every answer no later than the first, the second algorithm does not enumerate in polynomial delay as its delay before the last answer is exponential.

Let P be an enumeration problem, let A be a solver for P, and let x be input for A. If τ is a time tick during the execution of $A(x)$, then we denote by $\text{out}_{A,x}(\tau)$ the answers $y \in P(x)$ that have been printed before time τ is reached. We have the following lemma.

Lemma 6.11. *Let P be an enumeration problem, and A a solver for P. The following are equivalent.*

1. *A enumerates in incremental polynomial time.*
2. *There is a polynomial p such that for all input x and time tick τ it holds that*
$$p(|x| + |\text{out}_{A,x}(\tau)|) > \tau\,.$$

Proof. Denote the time of the Nth result by t_N.

$1 \Rightarrow 2$ If A enumerates in incremental polynomial time, there exists a polynomial p_1 such that $t_{N+1} - t_N \leq p_1(|x| + N)$. Without loss of generality, we assume that p_1 is monotone (as every polynomial is upper bounded by some monotone polynomial, and we can replace p_1 with such polynomial). We get the following on the printing time of the Nth result.
$$t_N = \sum_{i=1}^{N} t_i - t_{i-1} \leq \sum_{i=1}^{N} p_1(|x| + i - 1) \leq N \cdot p_1(|x| + N - 1)$$
In this case we get that for any time τ there exists a polynomial p_2 such that the following holds.
$$\tau < t_{|\text{out}_{A,x}(\tau)|+1} \leq (|\text{out}_{A,x}(\tau)| + 1) \cdot p_1(|x| + |\text{out}_{A,x}(\tau)|) \leq p_2(|x| + |\text{out}_{A,x}(\tau)|)$$

$2 \Rightarrow 1$ Assume now that $p_3(|x| + |\text{out}_{A,x}(\tau)|) > \tau$ for any time τ. Consider the delay after the Nth answer.
$$t_{N+1} - t_N \leq t_{N+1} < p_3(|x| + N + 1)$$
This shows that there exists a polynomial p_4 such that $t_{N+1} - t_N < p_4(|x| + N)$, meaning that A enumerates in incremental polynomial time. □

We can now prove Theorem 6.10.

Proof. Using the characterization of Lemma 6.11, let p be a polynomial such that for all x and τ we have $p(|x| + |\text{out}_{B,x}(\tau)|) > \tau$. The condition of the theorem implies that at every time tick τ, the set of answers printed by B is a subset of the set of answers printed by A, and therefore, $|\text{out}_{A,x}(\tau)| \geq |\text{out}_{B,x}(\tau)|$. Again since we can assume monotonicity, we conclude that $p(|x| + |\text{out}_{A,x}(\tau)|) > \tau$ as well. We use Lemma 6.11 to conclude that A enumerates in incremental polynomial time. □

Using the algorithms EnumMIS and EnumMISHold as A and B in Theorem 6.10, respectively, the combination with Lemma 6.9 implies that EnumMIS enumerates in incremental polynomial time, as claimed.

6.2.6 Tightness of the Algorithm

In what follows, we show that the time bounds that EnumMIS achieves are tight since it is not possible to solve the same problem with polynomial delay under the SETH hypothesis (which we describe next).

We recall that k-SAT is the satisfiability problem over n variables, where every clause contains at most k literals. The SETH hypothesis states that there is no algorithm for solving k-SAT in $O(2^{(1-\varepsilon)n})$ time for a fixed ε and all k.

Definition 6.12 (The Strong Exponential Time Hypothesis)**.** For every $\varepsilon > 0$ there exists a k such that k-SAT requires time larger than $2^{(1-\varepsilon)n}$ where n is the number of variables.

Let $(\mathcal{G}, A_V, A_E)$ be the tractable expansion of a tractably accessible SGR. We denote by $\mathsf{SMIS}(\mathcal{G}, \mathsf{A_V}, \mathsf{A_E})$ the following enumeration problem: Given an instance x, enumerate all $\mathit{MaxInd}(\mathcal{G}(x))$.

Proposition 6.13. *There exists some tractably accessible SGR with a tractable expansion $(\mathcal{G}, A_V, A_E)$, such that $\mathsf{SMIS}(\mathcal{G}, \mathsf{A_V}, \mathsf{A_E})$ cannot be enumerated with a polynomial delay, assuming the* SETH*.*

Proof. Let $k \geq 3$, and let ϕ be an instance of k-SAT with $\text{var}(\phi) = \{x_1, \ldots, x_n\}$ (for readability, we assume that $n \geq 2$ is even). We will show that a polynomial delay algorithm for enumerating $\mathit{MaxInd}(\mathcal{G}(x))$ will decide the satisfiability of ϕ within time $2^{\frac{n}{2}} \cdot \mathsf{poly}(|\phi|)$. This is true for any choice of k, which is not possible assuming the SETH.

We first describe the SGR $(\mathcal{G}, A_V, A_E)$. For any string x that is not a k-SAT formula, $\mathcal{G}(x) = \emptyset$. Otherwise, given a k-SAT instance ϕ, we define $\mathcal{G}(x)$ as follows: The set of vertices represents all possible truth assignments on $\frac{n}{2}$ variables twice, with two additional nodes $\bot_A$ and $\bot_B$. Intuitively, V_A corresponds to all possible truth assignments on the variables $x_1, \ldots, x_{\frac{n}{2}}$, and V_B corresponds to all possible truth assignments on the

remaining variables $x_{\frac{n}{2}+1}, \ldots, x_n$. That is,

$$V_A = \{A\} \times \{0,1\}^{\frac{n}{2}}$$
$$V_B = \{B\} \times \{0,1\}^{\frac{n}{2}}$$
$$V(\mathcal{G}(\phi)) = V_A \cup V_B \cup \{\bot_A\} \cup \{\bot_B\}.$$

To define the set of edges, we first start with edges between the set V_A and V_B. There is an edge (u, v) for $u \in V_A$, $v \in V_B$ if and only if u and v together encode a truth assignment that does not satisfy ϕ. Moreover, we also add all edges between nodes in V_A, between nodes in V_B and certain connections to the nodes $\bot_A$ and $\bot_B$ as follows:

$$\begin{aligned} E_{unsat} =& \{\{u,v\} | \exists a_1, \ldots, a_n \in \{0,1\} \text{ s.t. } u = (A, a_1, \ldots, a_{\frac{n}{2}}) \in V_A, \\ & v = (B, a_{\frac{n}{2}+1}, \ldots, a_n) \in V_B \text{ and } \phi(a_1, \ldots, a_n) = \mathsf{false}\}. \\ E(\mathcal{G}(\phi)) =& E_{unsat} \cup \{\bot_A, \bot_B\} \cup \{\{u,v\} \mid u,v \in V_A\} \cup \{\{u,v\} \mid u,v \in V_B\} \\ & \cup \{\{u, \bot_A\} \mid u \in V_A\} \cup \{\{u, \bot_B\} \mid u \in V_B\} \end{aligned}$$

We first note that this SGR is tractably accessible. Indeed, the set of nodes can be enumerated with a polynomial (even constant) delay, and for any $u, v \in \mathcal{G}(\phi)$, we can check whether $\{u, v\} \in E(\mathcal{G}(\phi))$ in polynomial time, since evaluation of any k-SAT formula can be done within polynomial (or even linear) time. To show that this SGR also has a tractable expansion, we note that the set of maximal independent sets of $\mathcal{G}(\phi)$ is given as the union of the sets I_A, I_B and I_{sat} with

$$I_A = \{\{u, \bot_B\} \mid u \in V_A\}, \quad I_B = \{\{u, \bot_A\} \mid u \in V_B\} \text{ and}$$
$$\begin{aligned} I_{sat} = \{\{u,v\} | \exists a_1, \ldots, a_n \in \{0,1\} \text{ s.t. } u = (A, a_1, \ldots, a_{\frac{n}{2}}) \in V_A, \\ v = (B, a_{\frac{n}{2}+1}, \ldots, a_n) \in V_B \text{ and } \phi(a_1, \ldots, a_n) = \mathsf{true}\}. \end{aligned}$$

Every maximal independent set of $\mathcal{G}(\phi)$ is of size 2, satisfying the first condition of a tractable expansion. For the second condition, we note that every subset I of $V(\mathcal{G})$ of size one can be extended trivially to a maximal independent set (by adding either $\bot_A$, $\bot_B$, or in case that $I \subset \{\bot_A, \bot_B\}$ some arbitrary element from V_A or V_B), and for any subset of size two, we can check whether I is (maximally) independent within polynomial time.

Note that ϕ is satisfiable if and only if $\mathit{MaxInd}(\mathcal{G}(x))$ contains more than the sets I_A and I_B. Assume that we can enumerate $\mathit{MaxInd}(\mathcal{G}(x))$ with a polynomial delay. We can output $2 \cdot 2^{\frac{n}{2}}$ many solutions within time $2^{\frac{n}{2}} \cdot \mathsf{poly}(|\phi|)$, meaning that we can decide whether there are more than $2 \cdot 2^{\frac{n}{2}}$ many maximal independent sets of $\mathcal{G}(\phi)$ within in the same time bound. Since ϕ is satisfiable iff $\mathcal{G}(\phi)$ has at least $2 \cdot 2^{\frac{n}{2}} + 1$ maximal independent sets, we are done. □

6.2.7 Note on Space Usage

We conclude Section 6.2 with a discussion on the space usage. Note that our algorithm may reach an exponential space as it relies on remembering all past answers to avoid the

production of duplicates. This cost is already incurred in the enumerators of maximal independent sets that form the basis of our algorithm [CFK+06, CKS08, LLK80]. However, several algorithms for enumerating maximal independent sets (and more generally maximal sets w.r.t. different properties) guarantee both polynomial delay and polynomial space, including the *reverse search* [AF96], the algorithm of Conte et al. [CGM+17], and the *proximity search* [CU19]. However, it is not clear to us how these algorithms can be adapted to enumerating the maximal independent sets of an SGR in a manner that limits the space, given that the set of nodes is not known upfront (and in light of Proposition 6.13). Moreover, note that the exponential space of our algorithm is also required for storing the (possibly exponential number of) past generated nodes of the SGR.

A natural question then remains open: can Theorem 6.7 be improved to require only polynomial space (at least when ignoring the space used by invoking the SGR functions)? Particularly, we leave open the question of whether and how the aforementioned polynomial-space algorithms can be adapted to enumerating the maximal independent sets of an SGR, and whether we can avoid storing all produced nodes. It appears that further assumptions on the SGR are required to this aim, and establishing these assumptions is left as a future direction.

6.3 Minimal Triangulations

In Section 6.2.2 we introduced MSGraph and claimed that it is an SGR. In this section, we use known results to reduce the problem of enumerating the minimal triangulations of a graph to the problem of enumerating the maximal independent sets for MSGraph. We will describe how to enumerate the nodes of MSGraph with polynomial delay, concluding that it is in fact an SGR. We will further show that MSGraph has a tractable expansion (Definition 6.6), and therefore Theorem 6.7 can be applied to conclude that the minimal triangulations can be enumerated in incremental polynomial time.

6.3.1 Reduction

We use the following notation. Let g be a graph. We denote by $\mathit{ClqMinSep}(g)$ the set of minimal separators S of g, such that S is a clique of g. Let φ be a subset of $\mathit{MinSep}(g)$. We denote by $g_{[\varphi]}$ the graph that results from saturating the minimal separators in φ.

Parra and Scheffler [PS97] have shown the following connection between minimal triangulations and maximal sets of *pairwise-parallel minimal separators* (that is, every two minimal separators in the set are non-crossing).

Theorem 6.14 (Parra and Scheffler [PS97])**.** *Let g be a graph.*

1. *If φ is a maximal set of pairwise-parallel minimal separators of g, then $g_{[\varphi]}$ is a minimal triangulation of g, and $\mathit{MinSep}(g_{[\varphi]}) = \varphi$.*

2. *If h is a minimal triangulation of g, then the set $\varphi = MinSep(h)$ is a maximal set of pairwise-parallel minimal separators in g, and $h = g_{[\varphi]}$.*

Theorem 6.14, combined with Theorem 6.2, gives the desired reduction in the following corollary. Recall that the graph $\mathcal{G}^{\mathsf{ms}}(g)$ is defined in Section 6.2.2, as part of the SGR $\mathsf{MSGraph} = (\mathcal{G}^{\mathsf{ms}}, A_{\mathsf{V}}^{\mathsf{ms}}, A_{\mathsf{E}}^{\mathsf{ms}})$.

Corollary 6.15. *For a graph g, there is a polynomial-time-computable bijection between the following two sets:*

- *$MaxInd(\mathcal{G}^{\mathsf{ms}}(g))$, that is, the set of all maximal sets of pairwise-parallel minimal separators of g.*
- *$MinTri(g)$, that is, the set of all minimal triangulations of g.*

Hence, it suffices to prove that $\mathsf{MSGraph}$ has a tractable expansion, as we do next.

6.3.2 Enumerating Minimal Separators

We now describe a variation of the algorithm of Berry et al. [BBC99] that, given a graph g, enumerates its set $MinSep(g)$ of minimal separators. Their algorithm enumerates with polynomial total time, and with a simple change (that we explain next) can enumerate with polynomial delay. Our variation is depicted in Algorithm 6.2. There, for $v \in \mathsf{V}(g)$ we denote by $N(v)$ the set of neighbors of v. For $U \subseteq \mathsf{V}(g)$ we denote by $N(U)$ the set of neighbors of nodes in U, excluding the nodes of U themselves; that is,

$$N(U) \stackrel{\text{def}}{=\!=} \Big(\bigcup_{v \in U} N(v) \Big) \setminus U \,.$$

We also denote by $\mathscr{C}(U)$ the set of connected components of the graph $g \setminus U$ (the graph obtained from g by removing all the nodes of U).

Algorithm 6.2 Enumerating minimal separators with polynomial delay

```
1:  procedure PDelayAllMinSep(g)
2:      Q :- ∅
3:      P :- ∅
4:      for all v ∈ V(g) do
5:          for all C ∈ C({v} ∪ N(v)) do
6:              Q :- Q ∪ {N(C)}
7:      while Q ≠ ∅ do
8:          S :- Q.pop()
9:          for all x ∈ S do
10:             S' :- {N(C) | C ∈ C(S ∪ N(x))}
11:             if S' ∉ P then
12:                 Q :- Q ∪ {S'}
13:         P :- P ∪ {S}
14:         print S
```

The algorithm remains intrinsically the same as that of Berry et al. [BBC99]. Minimal separators are considered as neighborhoods of connected components. The algorithm

finds minimal separators contained in a set $U \subseteq \mathsf{V}(g)$ by taking the neighborhoods of the connected components of $g \setminus U$, that is, $N(C)$ for all $C \in \mathscr{C}(U)$. Initially, the minimal separators that are contained in the neighborhoods of single nodes are generated (lines 3–5). Then, every previously generated minimal separator S is processed to produce more minimal separators that are *close* to S (lines 7–12). For every node v in the minimal separator S, it produces minimal separators that are contained in $S \cup N(v)$.

Our modification is in the data structures and the time of printing answers. In Algorithm 6.2, $\mathcal{Q}$ and $\mathcal{P}$ play the role of $\mathscr{S} \setminus \mathscr{T}$ and $\mathscr{T}$ of the original algorithm [BBC99], respectively. There, $\mathscr{S}$ holds all minimal separators generated, and $\mathscr{T}$ is a subset that holds the separators that were processed. The easy access to the separators yet to be processed (i.e. $\mathscr{S} \setminus \mathscr{T}$), along with printing answers when processed (in line 13, rather then when revealed in line 11), provides the polynomial delay. Correctness is derived directly by the correctness of the original algorithm, and the polynomial delay can be easily verified. In particular, the time between two consecutive results is $O(|\mathsf{V}(g)|^3)$.

6.3.3 Tractable Expansion

Recall that Rose [Ros70] proved that a chordal graph has fewer minimal separators than nodes. Combined with this result, Theorem 6.14 gives the first of the two conditions of Definition 6.6.

Corollary 6.16. *Let g be a graph. If I is a (maximal) independent set of $\mathcal{G}^{\mathsf{ms}}(g)$, then $|I| < \mathsf{V}(g)$.*

Proof. Suppose that I is a maximal set of pairwise-parallel minimal separators of g. Then by Theorem 6.14, $h = g_{[I]}$ is a minimal triangulation of g, and $\mathit{MinSep}(h) = I$. The graph h is chordal, hence from Rose [Ros70] we get that $|\mathit{MinSep}(h)| < |V(h)| = |V(g)|$. □

We now turn to proving the second condition of Definition 6.6. We do so by describing a general procedure for extending a set of pairwise-parallel minimal separators of a graph g to a maximal such set. Algorithm Extend of Algorithm 6.3 can apply any known polynomial time triangulation heuristic, referred to as Triangulate, as a black box. It uses the following procedures as subroutines.

- Saturate(g, S) receives a graph g and a set $S \subseteq V(g)$ of vertices, and saturates S (i.e., modifies g such that S becomes a clique).
- Triangulate(g) receives a graph g and returns a (not necessarily minimal) triangulation g' of g. We assume that this procedure runs in polynomial time. (For example, a naive implementation would be to add every possible edge; later we discuss smarter alternatives.)
- MinTriSandwich(g, g') receives a graph g and a triangulation g' of g, and returns a *minimal* triangulation of g. Using one of the known algorithms [Dah97, Pey01, BHT01], this procedure runs in time that is polynomial in the size of the graph.

Note that this is different than the task described in Proposition 6.4 since here g' is necessarily chordal.

- ExtractMinSeps(h) receives a chordal graph h and returns its set of minimal separators. Using the algorithm of Kumar [KM98], the execution time of this procedure is linear in h.

Extend takes as input a graph g and a set φ of pairwise-parallel minimal separators. It then proceeds by saturating the separators in φ, resulting in $g_{[\varphi]}$. At this stage it passes $g_{[\varphi]}$ to the triangulation heuristic Triangulate. We note that Triangulate does not have to produce a minimal triangulation. This is important since it allows us to incorporate *any* method for triangulation or tree decomposition. (We discuss in detail the translation between triangulations and tree decompositions in Section 6.4.)

The problem of transforming a non-minimal triangulation into a minimal one is called the *minimal triangulation sandwich problem* [Heg06]. Various polynomial-time algorithms for this problem exist [Dah97, Pey01], and these were reported to perform well in practice [BHT01].

At this stage we have a minimal triangulation h of $g_{[\varphi]}$. Theorem 6.17 shows that h is also a minimal triangulation of g. Lemma 6.18 shows that the set of minimal separators of h contains φ, which is essential as we need to *extend* φ. Finally, we can apply the algorithm of Kumar [KM98] to extract the minimal separators of the (chordal) graph h in linear time.

Algorithm 6.3 Extending a set of pairwise-parallel minimal separators

```
procedure Extend(g,φ)
  g_t :- Triangulate(g_[φ])
  h :- MinTriSandwich(g_[φ], g_t)
  return ExtractMinSeps(h)
```

To prove correctness of the algorithm Extend of Algorithm 6.3, we need the following result by Heggernes [Heg06].

Theorem 6.17 (Heggernes [Heg06])**.** *Given a graph g, let φ be an arbitrary set of pairwise-parallel minimal separators of g. Obtain the graph $g_{[\varphi]}$ by saturating each separator in φ. The following hold:*

1. *$\varphi \subseteq ClqMinSep(g_{[\varphi]})$, that is, φ consists of clique minimal separators of $g_{[\varphi]}$.*
2. *$ClqMinSep(g) \subseteq MinSep(g_{[\varphi]})$; that is, every clique minimal separator of g is a (clique) minimal separator of $g_{[\varphi]}$.*
3. *Every minimal triangulation of $g_{[\varphi]}$ is a minimal triangulation of g.*

The next lemma builds on Theorems 6.14 and 6.17.

Lemma 6.18. *Let g be a graph, and φ a set of pairwise-parallel minimal separators of g. Let h be a minimal triangulation of $g_{[\varphi]}$. Then $\varphi \subseteq MinSep(h)$.*

Proof. By Part 1 of Theorem 6.17 we have that $\varphi \subseteq \mathit{ClqMinSep}(g_{[\varphi]})$. Since h is a minimal triangulation of $g_{[\varphi]}$ then by Part 2 of Theorem 6.14, h is the result of saturating a maximal set, say φ', of pairwise-parallel minimal separators of $g_{[\varphi]}$. Therefore, by Part 2 of Theorem 6.17 we have $\mathit{ClqMinSep}(g_{[\varphi]}) \subseteq \mathit{MinSep}(h)$. This implies that $\varphi \subseteq \mathit{MinSep}(h)$, as claimed. □

We then conclude the correctness of the algorithm.

Lemma 6.19. *Let φ be a set of pairwise-parallel minimal separators of a graph g.* Extend(g, φ) *returns a maximal set I of pairwise-parallel minimal separators of g such that $\varphi \subseteq I$. Furthermore, the algorithm terminates in polynomial time.*

Proof. Assuming correctness of procedures Triangulate, and MinTriSandwich, the graph h is a minimal triangulation of $g_{[\varphi]}$. By Part 3 of Theorem 6.17, we have that h is a minimal triangulation of g. Consequently, from Part 2 of Theorem 6.14 we get that $\mathit{MinSep}(h) = I$ is a maximal set of pairwise-parallel minimal separators of g. By Lemma 6.18 it holds that $\varphi \subseteq \mathit{MinSep}(h)$, making I an extension of φ. All of the procedures in Algorithm 6.3 run in time that is polynomial in the size of the graph making it polynomial as well. □

From Corollary 6.16 and Lemma 6.19 we get the main result of this part.

Theorem 6.20. *The SGR* MSGraph *has a tractable expansion of independent sets.*

This theorem allows us to establish the main result of this section.

6.3.4 Main Result

From Theorems 6.20 and 6.7 we conclude that it is possible to enumerate the maximal independent sets of MSGraph in incremental polynomial time. Applying the bijection of Corollary 6.15, we get the main result of this section.

Corollary 6.21. *Given a graph, the minimal triangulations can be enumerated in incremental polynomial time.*

In the next section, we will use this result for enumerating tree decompositions.

6.4 Proper Tree Decompositions

In this section, we define the notion of a *proper* tree decomposition, which is essentially a tree decomposition that is, intuitively, not deemed redundant due to another tree decomposition. Our ultimate goal is to enumerate *only* the proper tree decompositions, and we will show that this translates to enumerating the minimal triangulations.

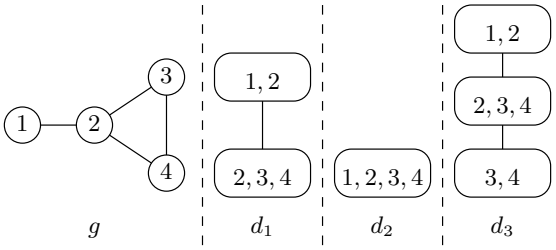


Figure 6.1: Examples of proper (d_1) and improper ($d2, d3$) decompositions of a graph g.

6.4.1 Proper Tree Decompositions

Let d_1 and d_2 be two tree decompositions of a graph g. We say that d_1 and d_2 are *bag equivalent*, denoted $d_1 \equiv_{\mathsf{b}} d_2$, if $\text{bags}(d_1) = \text{bags}(d_2)$. We denote by $d_1 \sqsubseteq d_2$ the fact that for every bag $b_1 \in \text{bags}(d_1)$ there exists a bag $b_2 \in \text{bags}(d_2)$ such that $b_1 \subseteq b_2$.

Let g be a graph, and let d and d' be tree decompositions of g. We say that d' *strictly subsumes* d if $d' \sqsubseteq d$ and $\text{bags}(d) \not\subseteq \text{bags}(d')$ in multiset notation (i.e., some bag appears in d more times than it appears in d'). A tree decomposition is *proper* if it is not strictly subsumed by any tree decomposition, and it is *improper* otherwise.

Figure 6.1 shows examples of proper and improper tree decompositions. It can be shown that d_1 is proper (e.g., since every clique of g is contained in some bag of d, as we prove in Proposition 6.24). But d_2 is not proper, since it is subsumed by d_1; that is, every bag of d_1 is contained in some bag of d_2, but the bag $\{1, 2, 3, 4\}$ is not a bag of d_1. For the same reason, d_2 is subsumed also by d_3. Finally, d_3 is subsumed by d_1 since every bag of d_1 is a bag of d_3, but the bag $\{3, 4\}$ is not a bag of d_1.

6.4.2 Enumeration

The main result of this section is the following, showing that enumerating the proper tree decompositions reduces to enumerating the minimal triangulations.

Theorem 6.22. *Let g be a graph. There is a bijection M between $\mathit{MinTri}(g)$ and the equivalence classes of $\equiv_{\mathsf{b}}$ over the proper tree decompositions of g. Moreover, given a minimal triangulation h of g, the proper tree decompositions in the class $M(h)$ can be enumerated with polynomial delay.*

Combined with Corollary 6.21, we get the following.

Corollary 6.23. *The set of proper tree decompositions of a given graph can be enumerated in incremental polynomial time.*

Next, we discuss the proof of Theorem 6.22, and in particular show how M is defined. We first need some propositions. The following proposition is a folklore, and it is using the fact that every collection of subtrees of a tree satisfies the *Helly property* [Gol80].

Proposition 6.24. *If d is a tree decomposition of a graph g, then every clique of g is contained in some bag of d.*

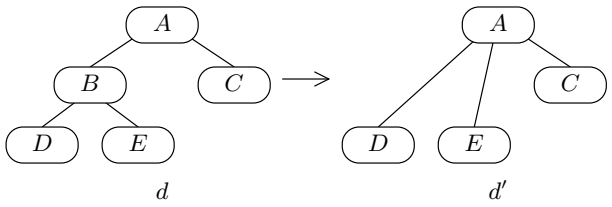


Figure 6.2: A strictly subsuming tree decomposition in the proof of Proposition 6.25.

Proof. We use the fact that the junction-tree property of a tree decomposition is equivalent to the property that for every node v of the graph, the bags of the tree decomposition that contain v form a (connected) subtree. Denote $d = (t, \beta)$ and let C be a clique of g. Every node v in C defines a subtree of t that is induced by the bags that contain v. Since d covers the edges of g, every two nodes in C must share some bag in d, and hence, their subtrees must share a vertex. It is known that every collection of subtrees of a tree satisfies the *Helly property* [Gol80]: if every two subtrees share a vertex, then there exists a vertex that is shared by all the subtrees. In particular, there exists a vertex in d common to all of these subtrees; this shared node corresponds to a bag that contains C. □

The following proposition states that in a proper tree decomposition, there is no containment among bags.

Proposition 6.25. *If d is a proper tree decomposition of a graph g, then* $\text{bags}(d)$ *is an antichain w.r.t. set inclusion (that is, no bag contains another).*

Proof. We need to show that a proper tree decomposition cannot have two bags with one contained in the other. Assume, by way of contradiction, that d is a proper tree decomposition of g with two bags $B, C \in \text{bags}(d)$ where $B \subseteq C$. Let A be the second bag in the path from B to C. Since d is a tree decomposition and A is on the path from B to C, we get that $B = B \cap C \subseteq A$.

Define d' to be the graph obtained from d by removing B and connecting A to all other neighbors of B, as illustrated in Figure 6.2. We will show that d' is a tree decomposition for g. The first two properties of the tree decomposition still hold because A contains B. Consider the path between two bags α and β of d'. If the path between them is the same as in d, the third property still holds. If it changed, then the path used to go through B, and the only new bag that may appear in this path is A. In this case, $\alpha \cap \beta \subseteq B \subseteq A$, and the third property holds as well. We have found a tree decomposition d' for g that strictly subsumes d, hence d is improper, and this is a contradiction. □

From Theorem 6.3, the following easily follows.

Proposition 6.26. *If d is a tree decomposition of a graph g, then* $\textsc{saturate}(g, d)$ *is a triangulation of g.*

Proof. According to the definitions, d is a tree decomposition of $\textsc{saturate}(g, d)$. Hence, since every bag of d is a clique of $\textsc{saturate}(g, d)$, it follows from Theorem 6.3 that $\textsc{saturate}(g, d)$ is chordal. □

The definition of M is based on Lemma 6.27, stating that a chordal graph g has a single proper tree decomposition, up to the equivalence $\equiv_{\mathsf{b}}$, with the set of bags being precisely the set of maximal cliques.

Lemma 6.27. *If g is a chordal graph and d is a proper tree decomposition of g, then* $\text{bags}(d) = \mathit{MaxClq}(g)$.

Proof. According to Proposition 6.24, every clique of g is contained in some bag of d, and according to Theorem 6.3, g has some tree decomposition, say d', where all the bags are cliques of g. So we have that $d' \sqsubseteq d$. If $\text{bags}(d) \not\subseteq \text{bags}(d')$, then d' strictly subsumes d, in contradiction to the fact that d is proper. Hence $\text{bags}(d) \subseteq \text{bags}(d')$, meaning that the bags of d are cliques of g. It thus follows that every maximal clique is a bag of d. That is, $\mathit{MaxClq}(g) \subseteq \text{bags}(d)$. Finally, Proposition 6.25 states that the bags of d are an antichain w.r.t. set inclusion, and hence, $\text{bags}(d) \subseteq \mathit{MaxClq}(g)$. We conclude that $\text{bags}(d) = \mathit{MaxClq}(g)$, as claimed. □

Based on Lemma 6.27, we define M to be the function that maps every $h \in \mathit{MinTri}(g)$ to the equivalence class of the proper tree decomposition of h. Lemma 6.28 states that M has the required properties.

Lemma 6.28. *Let g be a graph. The mapping M is a bijection between $\mathit{MinTri}(g)$ and the equivalence classes of $\equiv_{\mathsf{b}}$ over the proper tree decompositions of g.*

Proof. We show that M has the correct range and that it is bijective.

M has a proper range Let h be a minimal triangulation of g, and let d be a proper tree decomposition of h in $M(h)$. Then d is also a tree decomposition of g, as the three properties of a tree decomposition still hold. We need to show that d is a *proper* tree decomposition of g. According to Lemma 6.27, we have $\text{bags}(d) = \mathit{MaxClq}(h)$, and therefore, $\textsc{saturate}(g, d) = h$. Assume, by way of contradiction, that d is improper. Then d is strictly subsumed by some tree decomposition d' of g, meaning that $d' \sqsubseteq d$. Let h' be the graph $\textsc{saturate}(g, d')$. From Proposition 6.26 it follows that h' is a triangulation of g. From $d' \sqsubseteq d$ and the fact that every bag of d is a clique of h, we conclude that $\mathsf{E}(h') \subseteq \mathsf{E}(h)$. And since h is a minimal triangulation, we get that $h = h'$. We can now conclude that d' is also a tree decomposition of h: the junction-tree property holds and the nodes are covered since it is a tree decomposition of g, and the edges are covered since those are the edges of h' that are covered by its definition. We get that both d and d' are tree decompositions of h, and d is strictly subsumed by d', which contradicts the fact that d is a proper tree decomposition of h.

M **is injective** Let h_1 and h_2 be two minimal triangulations such that $h_1 \neq h_2$. Without loss of generality, assume that the edge $\{u, v\}$ is in h_1 but not in h_2. The nodes u and v are part of some maximal clique of h_1, so they share a bag in $M(h_1)$. But they are not part of any clique of h_2, so they do not share any bag in $M(h_2)$. Therefore, $M(h_1) \neq M(h_2)$.

M **is surjective** Given a proper tree decomposition d of g, we need to show that there exists a minimal triangulation h of g such that $d \in M(h)$. Consider the graph $h = \text{SATURATE}(g, d)$. We show that h is a minimal triangulation and that $d \in M(h)$.

We first show that h is a minimal triangulation of g. According to Proposition 6.26, h is a triangulation of g. Assume, by way of contradiction, that h is not minimal. Then there exists a minimal triangulation h' of g obtained from h by removing some edges; denote one of these edges by e. Consider a tree decomposition $d' \in M(h')$. The clique containing e in h is not a clique in h', and so $\text{bags}(d) \not\subseteq \text{bags}(d')$. Also note that since $h' \subseteq h$, every maximal clique of h' is contained in some maximal clique of h, and therefore $d' \sqsubseteq d$. Then d' strictly subsumes d, in contradiction to d being proper.

Finally, we need to show that d is a proper tree decomposition of h. The nodes of h are covered in d, and the junction-tree property holds, since d is a tree decomposition of g. The new edges of h are covered in d since they are all a result of saturation of the bags of d. So d is a tree decomposition of h, and we claim that it is proper. Assume, by way of contradiction, that d is not a proper tree decomposition of h. Then, the tree decomposition d' that strictly subsumes it is also a tree decomposition for g, contradicting the fact that d is a proper tree decomposition of g. □

To complete the proof of Theorem 6.22, we explain how the proper tree decompositions in the class $M(h)$ can be enumerated with polynomial delay for $h \in \mathit{MinTri}(g)$. Jordan [Jor02] shows that, given a chordal graph h, a tree over the bags that represent the maximal cliques of h is a tree decomposition if and only if it is a maximal spanning tree, where the weight of an edge between two bags is the size of their intersection. Hence, this enumeration problem is reduced to enumerating all maximal spanning trees, which can be solved in polynomial delay [YKW10]. Since the number of maximal cliques is at most the number of nodes in chordal graphs [Gav74], we have a polynomial delay algorithm for enumerating the tree decompositions. This concludes the proof.

According to Corollary 6.23, we can enumerate all proper tree decompositions with incremental polynomial time. Note that this section also implies another alternative: we can enumerate only one representative of every equivalence class with the same complexity guarantees. That is, we can enumerate one proper tree-decomposition of each possible bag configuration with incremental polynomial time. The choice of which variation to use depends on the application at hand. For some applications, different tree-decompositions with the same bags may be of different quality, while for others only the bags matter.

6.5 Experimental Evaluation

We now describe an experimental study over an implementation of our enumeration algorithm for minimal triangulations, namely the algorithm EnumMIS of Algorithm 6.1 for the SGR $(\mathcal{G}^{\mathsf{ms}}, A_{\mathsf{V}}^{\mathsf{ms}}, A_{\mathsf{E}}^{\mathsf{ms}})$, calling the procedure Extend of Algorithm 6.3. The goal of the experimental study is twofold. First, we wish to understand how practical the execution cost of the algorithm is for enumerating many minimal triangulations (and tree decompositions). Second, we wish to study the ability of the algorithm to produce many *high-quality* triangulations, given an underlying triangulation algorithm (for Extend), and even to improve upon standard quality measures of the underlying algorithm itself. In Section 6.5.1 we describe the experimental setup, in Section 6.5.2 we report on the efficiency of the algorithm in terms of its delay, and in Sections 6.5.3 and 6.5.4 we study the quality of the results.

6.5.1 Experimental Setup

We first describe the general setup for our study.

Implementation and Hardware. We implemented all algorithms in C++, with STL data structures.[1] All experiments were carried out on a 2.6GHz dual-core laptop with 8GB of RAM running Windows 10 professional.

Triangulation Algorithms. We implemented two well known triangulation algorithms as the procedure Triangulate in line 2 of the procedure Extend (Algorithm 6.3). Both algorithms apply the general technique of *node-elimination ordering* [OCF76], where nodes are eliminated from the graph in turn, by adding a subset of fill edges between the eliminated node and its neighbors in the (leftover) graph. Both algorithms guarantee a minimal triangulation (hence there was no need to call MinTriSandwich$(g_{[\varphi]}, g_t)$ in line 3 of Extend).

- MCS_M [BBH02]. This is an extension of the *Maximum Cardinality Search* (MCS) algorithm for recognizing chordal graphs [TY84], which finds a minimal elimination ordering along with its corresponding minimal triangulation.
- LB_TRIANG [BBH+06]. This algorithms guarantees minimality of the triangulation by adding only a subset of the fill edges at each of the elimination steps, and allows for complete flexibility in determining the elimination order. We applied the *min fill* heuristic that selects, at each iteration, the node whose elimination results in the smallest number of edges to add.

Datasets. We used three types of datasets: probabilistic graphical models, database queries, and random (synthetic) graphs. For the first type, we used the following

[1] The code is available online: `https://github.com/NofarCarmeli/MinTriangulationsEnumeration`

benchmark networks from the UAI probabilistic inference challenge.[2] The datasets Alchemy and DBN from the challenge are not described here as each of their graphs had only one or two minimal triangulations, and the enumeration ended instantaneously.

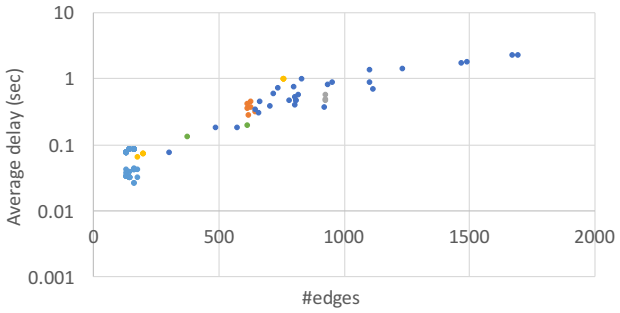
- **Promedas**: The Promedas (PRObabilistic MEdical Diagnostic Advisory System) Markov networks represent medical diagnosis cases, and consist of binary variables that were converted from layered noisy-or Bayesian networks. The dataset includes 33 graphs with 26-1039 nodes and 36-1696 edges, and many of them are considered too difficult for exact inference.[3]
- **Object detection**: Markov Random Fields for object-detection tasks in computer vision. It includes 79 instances of connected networks, each containing 60 nodes and between 135 to 180 edges.
- **Image segmentation**: Bayesian networks generated from image-segmentation tasks. It includes 6 graphs with 226-235 nodes and 617-647 edges.
- **Grids**: An $N \times N$ grid network. Such networks that are common in image processing [BKR11], and networks that model problems such as medical diagnosis and object detection. This dataset includes 8 grids with $N = 10$ and $N = 20$, resulting in graphs with 100 or 400 nodes, and 180-760 edges.
- **Pedigree**: Bayesian networks used to model genetic information [FG02]. The data set includes 3 graphs, each has 385 nodes and 930 edges.
- **CSP**: Constraint-satisfaction problems. There are 3 instances in the dataset, with 67-100 nodes and 226-619 edges.

The datasets of second and third types are as follows.

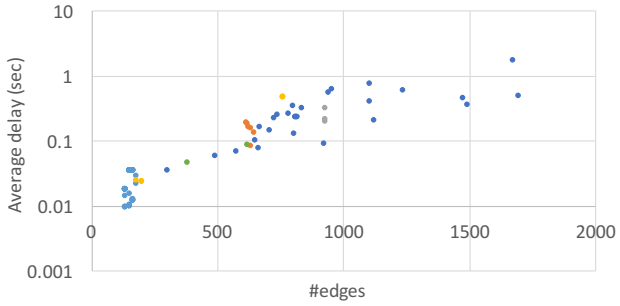
- **TPC-H**: Graphs induced from TPC-H. These are the Gaifman (primal) graphs of joins for implementing the TPC-H benchmark queries in LogiQL, the Datalog variant of LogicBlox [AtCG$^+$15].[4] The queries include up to 22 nodes, and up to 46 edges, and their treewidth is up to 7.
- **Random**: Random $G(n, p)$ graphs in the Erdős-Rényi model. The number of nodes is n and every pair of nodes is connected by an edge with probability p (independently). We generated 54 random graphs for varying n between 30 and 200, and three values of p: 0.3 (sparsest), 0.5 and 0.7 (densest).

As a baseline approach, we implemented the algorithm of DunceCap [TR15] for generating all of the generalized hypertree decompositions (each involving an underlying tree decomposition). However, this algorithm is designed to handle small join queries and to span a much greater space of objects (namely, the generalized hypertree decompositions). In particular, on the TPC-H dataset we observed that on the smaller queries our algorithm is faster by 3 to 4 orders of magnitude, and on some of the larger queries (Q7 and Q9) we could not get their algorithm to terminate in less than two hours (while our algorithms terminated in a few seconds, as we later discuss). Therefore,

[2] `http://www.cs.huji.ac.il/project/PASCAL/showNet.php`
[3] `http://graphmod.ics.uci.edu/uai08/Evaluation/Report/Benchmarks`
[4] The queries, provided to us by LogicBlox, are used for benchmarking the engine.



(a) LB_TRIANG



(b) MCS_M

Figure 6.3: Average delay (in seconds) for the two triangulation algorithms over the probabilistic-graphical-model benchmarks: Object Detection (●), Segmentation (●), Pedigree (●), Grids (●), Promedas (●), CSP (●) .

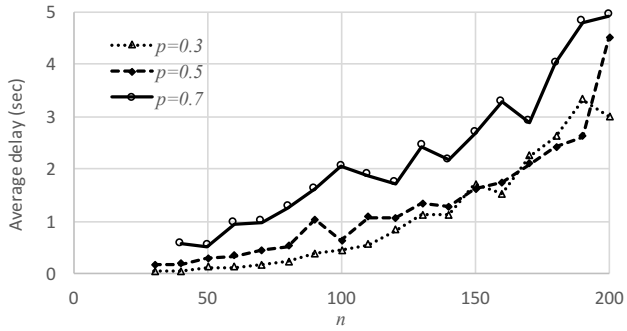
we decided to exclude comparisons to this implementation. As of today, we are not aware of any other published algorithms for enumerating (minimal) triangulations or tree decompositions with guarantees of correctness (completeness).

6.5.2 Execution Cost

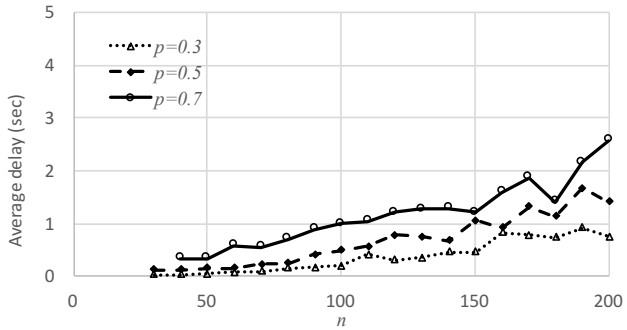
In what follows we report on the delay of the two variants of the implementation, corresponding to the two triangulation algorithms LB_TRIANG and MCS_M.

Probabilistic graphical models

We measured the average delay between minimal triangulation printouts for the network datasets from the UAI challenge. The measurements were conducted during 30 minutes executions. 5 of the graphs in Promedas, and one graph of CSP completed the enumeration within this time. We plotted the delay of the other graphs against the number of their edges. The plots, corresponding to the two minimal triangulation algorithms LB_TRIANG and MCS_M, are presented in Figures 6.3a and 6.3b, respectively, using log-scale. Overall, we see that the delay increases with the size of the graph. However, this trend varies between the different benchmarks. While this dependency is apparent for the Promedas data set, the average delay for object detection has little correlation with the number of edges in the graph.



(a) LB_TRIANG



(b) MCS_M

Figure 6.4: Average delay over 54 graphs randomly generated from the Erdős-Rényi $G(n, p)$ for varying n and p.

Random graphs

We measured the average delay (in seconds) between the printout of consecutive minimal triangulations during a 30 minute execution. The plots in Figures 6.4a and 6.4b show the average delay vs. the size of the graph for the two variants. We can see that the delay increases with the size of the graph, and that the general trend is that the delay is larger for denser graphs. We also see that for LB_TRIANG the delay is generally longer than for MCS_M.

Database queries

We evaluated our enumeration algorithm over a set of 22 queries from the TPC-H dataset. The graphs of these queries are quite small when compared to the UAI datasets (< 23 nodes). Moreover, half of these graphs are chordal to begin with (i.e., have only one minimal triangulation—the graph itself), and hence, irrelevant for us. Except for two queries, all of the rest had at most 5 minimal triangulations. The remaining two queries are Q7 (Volume shipping Query) and Q9 (Product Type Profit Measure Query), and they have a considerable number of minimal triangulations: 700 and 588, respectively. When considering the minimum-width tree decomposition for each of the queries, the largest bag was of size 8; this is due to a relation of arity 8 in the query. In fact the largest bag in each of the queries had at most two variables more than the size

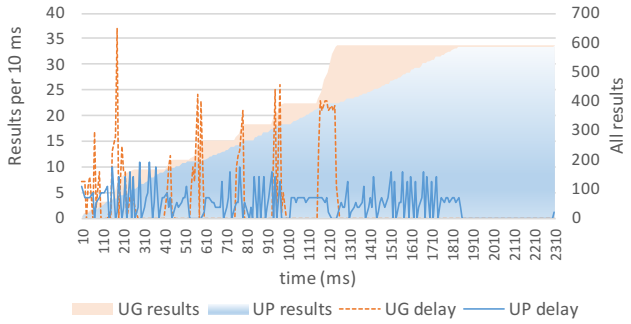


Figure 6.5: Delay behavior in two printing modes: UG (Upon Generation, as in EnumMIS), and UP (Upon Pop, as in EnumMISHold).

Dataset	**#trng**	**min-w**	**#≤$\mathbf{w}_1$ (%)**	**%w↓ (max)**
MCS_M				
Promedas (28)	11064.5	25.8	3713.6 (33.6%)	2.2 (15.2)
Grids (8)	40319.8	18.4	93.6 (0.2%)	0.0 (0.0)
Obj. Detection (79)	100349.9	6.1	42743.9 (42.6%)	0.4 (12.5)
Segmentation (5)	12836.5	23.0	20.5 (0.2%)	0.0 (0.0)
Pedigree (3)	7789.0	31.7	3087.3 (39.6%)	0.0 (0.0)
CSP (2)	29450.5	16.5	26741.5 (90.8%)	13.2 (26.3)
LB_TRIANG				
Promedas (28)	4220.7	18.6	2352.0 (55.7%)	1.9 (16.7)
Grids (8)	13881.3	24.5	1273.0 (9.2%)	3.0 (8.7)
Obj. Detection (79)	33295.4	5.8	15709.3 (47.2%)	0.0 (0.0)
Segmentation (5)	5174.2	21.8	2141.8 (41.4%)	10.3 (20.7)
Pedigree (3)	3646.0	23.7	3227.7 (88.5%)	5.3 (14.8)
CSP (2)	11772.0	16.5	3760.5 (31.9%)	0.0 (0.0)

Table 6.1: Width statistics on generated triangulations following 30 minutes execution.

of the largest relation. The execution for all 22 queries completed within 5 seconds.

In one of the queries we compared the delays for two modes of printing: the one of EnumMIS and the one of EnumMISHold that prints upon extraction from the queue, as described in Section 6.2.5. We refer to the former as UG (Upon Generation) and to the latter as UP (Upon Pop). Recall that both modes guarantee incremental polynomial time (Theorem 6.10). This gives us a sense of the practical impact of printing the solutions as soon as possible compared to holding the solutions to attain an easy-to-prove incremental polynomial time algorithm. The results are in Figure 6.5. While the dotted line (of UG) has bursts of high-frequency prints followed by periods where no new triangulation is created, the solid line (UP) has a more steady pace as can be seen by the constant slope in Figure 6.5. As expected, despite the fact that the last result of UG is printed earlier that of UP, termination is at the same time in both modes, as the algorithm needs to check that there are no additional minimal triangulations.

Dataset	**#trng**	**min-f**	#≤$\mathbf{f}_1$ (%)	%**f**↓ (max)
MCS_M				
Promedas (28)	11064.5	3353.4	8136.0 (73.5%)	18.1 (49.9)
Grids (8)	40319.8	2752.6	15771.4 (39.1%)	4.2 (28.1)
Obj. Detection (79)	100349.9	30.0	27614.1 (27.5%)	19.9 (47.1)
Segmentation (5)	12836.5	2555.2	5269.7 (41.1%)	5.9 (12.5)
Pedigree (3)	7789.0	3525.7	743.0 (9.5%)	2.8 (3.5)
CSP (2)	29450.5	46.0	18815.5 (63.9%)	35.2 (55.8)
LB_TRIANG				
Promedas (28)	4220.7	1239.4	175.0 (4.1%)	0.2 (11.1)
Grids (8)	13881.3	1600.3	1.0 (0.0%)	0.0 (0.0)
Obj. Detection (79)	33295.4	27.6	5110.7 (15.3%)	10.4 (27.6)
Segmentation (5)	5174.2	1402.0	130.2 (2.5%)	1.2 (4.2)
Pedigree (3)	3646.0	1491.0	1.0 (0.0%)	0.0 (0.0)
CSP (2)	11772.0	34.5	664.0 (5.6%)	1.4 (3.0)

Table 6.2: Fill statistics on generated triangulations following 30 minutes execution

6.5.3 Quality

In what follows we report on the quality of the generated minimal triangulations in terms of two standard measures of quality for triangulations and tree decompositions: *fill* and *width*. *Fill* refers to the total number of edges added in order to make the resulting graph chordal, while *width* refers to the size of the largest clique in the generated triangulation (minus one).[5] The natural benchmark for the quality of the triangulations is the first result our enumeration returns, as it is the result we would get by running the minimal triangulation algorithm we used, on the original input graph.

For each graph of the probabilistic inference dataset, we executed the enumeration algorithm for 30 minutes. The results in Table 6.1 include only the experiments where the enumeration did not complete. For each graph we measured the following: the number of generated triangulations (**#trng**), the minimum observed width over all printed triangulations (**min-w**), the number of printed triangulations of width at most that of the first (#≤$\mathbf{w}_1$), the average reduction in width (over the dataset) and the maximum improvement in parentheses (**w**↓ (%)). In Table 6.2 we show the same results for fill instead of width (**min-f**, #≤$\mathbf{f}_1$ and **f**↓ (%)).

We can see that the algorithm, in both variants, is able to generate a significant number of triangulations of high quality, in terms of both width and fill. Moreover, it amplifies the quality of the underlying triangulation, by means of width, and much more by means of fill. According to the number of triangulations printed, MCS_M enables generating twice as many triangulations as LB_TRIANG. However, with the exception of only a handful of the graphs tested, the triangulations generated by LB_TRIANG are

[5]Recall that is NP-hard to find a triangulation that minimizes the fill [Yan81a] or the width [ACP87].

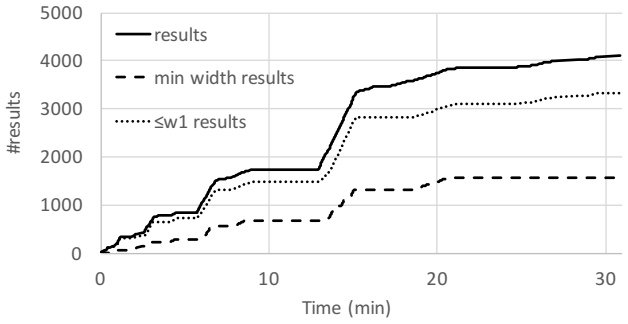


Figure 6.6: Cumulative number of triangulations.

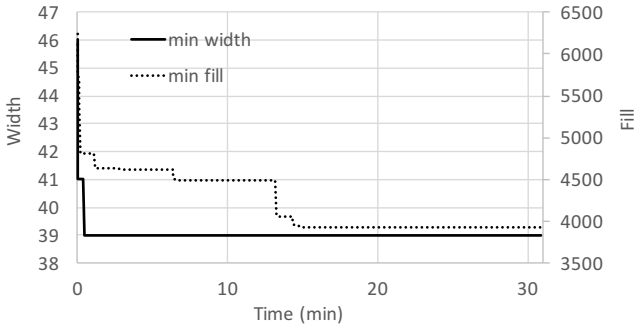


Figure 6.7: Minimum width and fill over time.

superior in both the width and fill metrics (this is especially apparent for the Promedas and Pedigree datasets). Furthermore, this set of superior triangulations accounts for a larger portion of the total number of triangulations that generated.

6.5.4 Case Study

In this section we take a closer look at the behaviour of the enumeration during a single execution. We use a graph from the Promedas dataset. In Figure 6.6 we show the cumulative number of results generated over time. We consider three types of results: *(a)* all minimal triangulations, *(b)* minimal triangulations of the minimum width (where the minimum is taken over the printed triangulations), *and (c)* those with a width at most that of the first triangulation ($\leq$w1), which is the one that we would obtain by using only the triangulation algorithm at hand. The reduction in the number of new triangulations over time is consistent with the increase in the delay entailed in the guarantee of incremental polynomial time, rather than polynomial delay.

Figure 6.7 presents the reduction in the minimum width and minimum fill obtained during the execution of the algorithm. Each time slice records the minimum width (solid curve) and minimum fill (dotted curve) observed up to that time. We can see that both the width and the fill reduce over time, but the minimum observed width is reached very quickly, while attaining the minimum observed fill takes longer.

Chapter 7

Conclusions

We conclude the thesis with a summary of the main results and point at some open problems following our work.

We studied when CQs and UCQs can be answered efficiently: with linear preproceesing time and constant or logarithmic time per answer. To evaluate the queries, we demand either enumeration (with arbitrary order), random permutation or random access. Our queries are either over general schemas or in the presence of functional dependencies. We showed how in several settings, even though the queries do not have a naturally acyclic tractable structure, we can identify implicit acyclicity and utilize it to answer additional queries efficiently. We also established lower bounds for other queries.

Regarding CQs without self-joins, we saw that the free-connex queries are exactly the tractable ones, and that $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle = \mathsf{Enum}\langle \mathrm{lin}, \log\rangle = \mathsf{REnum}\langle \mathrm{lin}, \log\rangle = \mathsf{RAccess}\langle \mathrm{lin}, \log\rangle$, assuming conventional hypotheses. The same holds when the schema contains unary FDs, but then the tractable CQs are the ones with a free-connex FD-extension. That is, additional queries may become tractable in the presence of FDs. This is also true when the FDs are replaced with CDs and when the CQs contain disequalities. In the presence of general (non-unary) FDs or CDs, we showed similar results if the extension is acyclic.

The picture is more involved in the case of UCQs, where $\mathsf{RAccess}\langle \mathrm{lin}, \log\rangle$ is a proper subset of $\mathsf{Enum}\langle \mathrm{lin}, \log\rangle$. A union of intractable CQs may be tractable with respect to $\mathsf{Enum}\langle \mathrm{lin}, \mathrm{const}\rangle$. In contrast, a UCQ may be intractable with respect to random access even if every CQ it contains is tractable. To obtain random permutation, we suggested two alternatives for a union of free-connex CQs: an algorithm with logarithmic delay that applies for UCQs with a free-connex intersection, and an algorithm with logarithmic delay in expectation that applies to all unions of free-connex CQs. Regarding enumeration, we formalized how CQs within a union can make each other easier by providing variables, introduced union extensions, and showed that UCQs with a free-connex union extension are tractable. Our techniques involving dependencies also apply to UCQs, and they can make otherwise intractable UCQs tractable with respect to either of our three tasks.

We proved conditional lower bounds showing that our techniques for the enumeration of UCQs can be applied to all tractable queries in some cases. In case of a union of two intractable CQs or two acyclic body-isomorphic CQs, free-connex union extensions fully capture the tractable cases, under conventional complexity assumptions. Then, we defined the unbalanced triangle detection hypothesis, showed that it exactly captures the hardness of some UCQs, and that if we assume this hypothesis, we get a dichotomy for unions of two binary UCQs: these are tractable if and only if they have a free-connex union extension.

Lastly, we tackled the task of enumerating proper tree-decompositions, which can be used as query plans that extract acyclicity from cyclic queries. We proved that enumerating the tree decompositions can be done via the enumeration of minimal triangulations, and showed how to solve the latter problem with incremental polynomial time. We did so by introducing the concept of a succinct graph representation (SGR) and reducing the problem of enumerating the minimal triangulations to the enumeration of the maximal independent sets of an SGR. Our experimental study shows that the algorithm is effective on graphs of various domains: it is able to enhance off-the-shelf algorithms for triangulation by generating many high-quality different triangulations.

Future Research

We describe some directions for future work next.

Completing characterizations. In Sections 3.2 and 3.3, we saw conditional lower bounds for the enumeration of UCQ answers. Can we complete a dichotomy and show these lower bounds for all UCQs that do not admit a free-connex union extension? In particular, we defined UTD and showed that if these extensions capture all tractable cases, then UTD necessarily holds. Can we complete our lower bounds based on UTD and show the hardness of all UCQs with no tractable extension based on this hypothesis? Our characterization regarding FD-cyclic CQs is not complete either. Our proof for the hardness of FD-cyclic CQs assumes that all FDs are unary. It remains open to answer whether this result fold for general FDs and to classify Example 5.24.

Tools for enumeration lower bounds. The conditional lower bounds we build upon all rely either on the decision problem (e.g., Hyperclique) or the total time to find all answers (e.g., BMM). We are currently lacking tools for showing lower bounds for enumeration based on the delay between answers. In particular, can we find techniques for showing lower bounds for random permutation in cases where we can solve enumeration efficiently?

Reasoning about self-joins. Another gap following our techniques for lowers bounds is that they all assume no self-joins. This is a restriction that applies also in the CQ

results that we build upon [BDG07, BB13], and it is in place to ensure that we can freely encode different atoms with different relations. Note that our upper bounds allow for self-joins, so do not suffer from the same restriction. Can a query with self-joins be tractable while a query with the same structure and distinct relations is intractable? If not, can we extend our lower bounds to apply also for self-joins?

Finding the tractable extensions. We showed that any UCQ with a free-connex union extension is tractable with respect to enumeration, but how do we get such an extension? For the fragments of UCQs where we proved lower bounds, we saw that for every tractable UCQ, we can get a free-connex extension by covering one difficult structure at a time (Definition 3.46). Is this true in general for all UCQs? Also, when the UCQs are over a schema with dependencies, we showed that we can benefit from taking an FD-extension of a union extension (Section 5.3.3). Can we get all tractable unions with respect to enumeration by first applying a union extension and then applying the FD-extension or do we sometimes need to apply them in the opposite order?

Efficient random permutation for more queries. We showed that UCQs with a free-connex union extension are tractable with respect to enumeration (Section 3.1.4), and we showed efficient algorithms for the random permutation of unions of free-connex CQs (Section 4.3). Can we achieve efficient random permutation for UCQs that contain intractable CQs but have a free-connex union extension? Also, we showed that we can achieve efficient random access and random permutation for FD-free-connex CQs. Can we do the same for CD-free-connex CQs? (see Section 5.3.1.)

Random access and counting. We defined a random access algorithm to return an out-of-bound message when necessary. Then, efficient random access implies the ability to count the answers (see the proof of Theorem 4.9). What happens if we define random access such that it gives no guarantee on the answer when it receives an index too large? Can we find an alternative hardness proof that does not rely on the out-of-bound behavior? If not, can we find an efficient random access algorithm even in cases where it takes too long to count the number of answers?

Free-connex tree decompositions. Tree decompositions can be used to transform cyclic queries into acyclic ones. However, we know that in the presence of projections, acyclicity is not the best structure we can hope for: acyclicity only allows for linear-delay enumeration [BDG07], while we would like free-connexity that can be used to achieve constant delay. Can we find an efficient way to take the projections into account when generating decompositions and create a free-connex decomposition in which the free variables form a subtree?

Improved tree decompositions enumeration. It is left open whether the enumeration of the minimal triangulations can be carried out with polynomial delay. As discussed in in Section 6.3 and Section 6.2.6, it is not clear how to do this with known techniques, and the abstraction used here cannot achieve this time bound. Polynomial delay is possible in the case that the number of minimal separators of the input graph is polynomial in the size of the input graph. In a follow up work to the original publication of the work in Chapter 6, Ravid et al. [RMK19] showed how to perform in such cases ranked enumeration under a wide class of cost functions that generalizes width and fill-in. If the number of minimal separations is not bounded, the question of incorporating some order remains open.

Reducing space requirements. Our research focuses on time complexity, and several of our solutions rely on storing all produced answers. An interesting question is whether we can avoid this large use of memory while achieving the same time bounds. This question remains open for: UCQs with a tractable union extension (see Section 3.4.2), CD-free-connex CQs (see Section 5.4) and tree decompositions (see Section 6.2.7).

Exploring additional settings. In Section 3.4.1, we saw that the complexity of answering UCQs changes with the addition of disequalities; we also saw some cases of tractable unions containing an intractable CQ. Can we find a characterization for all such UCQs with disequalities? In addition, there is a dichotomy for enumerating CQs with negation [BB13]. Can we extend it to accommodate FDs? What happens to the enumeration complexity when we combine unions with negation? Also, we showed that FDs and CDs have a positive impact on the complexity of answering queries. Does the same hold for other types of dependencies (e.g., inclusion dependencies)?

Performance in practice. Even though this work is mainly theoretical, some of this research is accompanied by experiments. We presented an experimental study for the enumeration of tree-decompositions (Section 6.5). In fact, the algorithm presented here holds many opportunities for optimization over real-life graphs, and an optimized version of the code is available online.[1] The results regarding random permutation were also implemented, and an experimental study shows that our algorithms outperform the sampling-with-rejection alternatives [CZB+20]. Yet, there is still much room to perform extensive practical research and compare the acyclicity-based approach we take here to the approach common in current industrial database management systems.

[1] `https://github.com/TechnionTDK/efficient-td-enum`

Bibliography

[ACP87] Stefan Arnborg, Derek G Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in ak-tree. *SIAM Journal on Algebraic Discrete Methods*, 8(2):277–284, 1987.

[AF96] David Avis and Komei Fukuda. Reverse search for enumeration. *Discret. Appl. Math.*, 65(1-3):21–46, 1996.

[AFG16] Myrto Arapinis, Diego Figueira, and Marco Gaboardi. Sensitivity of counting queries. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, Rome, Italy*, pages 120:1–120:13, 2016.

[AGPR99] Swarup Acharya, Phillip B. Gibbons, Viswanath Poosala, and Sridhar Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, pages 275–286. ACM Press, 1999.

[AHV95] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, Boston, MA, USA, 1995.

[AP09] Rasmus Resen Amossen and Rasmus Pagh. Faster join-projects and sparse matrix multiplications. In *ICDT*, pages 121–126, 2009.

[AtCG+15] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the logicblox system. In *SIGMOD*, pages 1371–1382. ACM, 2015.

[AW14] Amir Abboud and Virginia Vassilevska Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *Foundations of Computer Science (FOCS), 2014 IEEE 55th Annual Symposium on*, pages 434–443. IEEE, 2014.

[AYZ97] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

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

[BBC99] Anne Berry, Jean Paul Bordat, and Olivier Cogis. Generating all the minimal separators of a graph. In Peter Widmayer, Gabriele Neyer, and Stephan Eidenbenz, editors, *WG*, volume 1665 of *Lecture Notes in Computer Science*, pages 167–172. Springer, 1999.

[BBH02] Anne Berry, Jean R. S. Blair, and Pinar Heggernes. Maximum cardinality search for computing minimal triangulations. In *WG*, WG '02, pages 1–12, London, UK, UK, 2002. Springer-Verlag.

[BBH+06] Anne Berry, Jean-Paul Bordat, Pinar Heggernes, Geneviève Simonet, and Yngve Villanger. A wide-range algorithm for minimal triangulation from an arbitrary ordering. *Journal of Algorithms*, 58(1):33 – 66, 2006.

[BDG07] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

[BEG04] Endre Boros, Khaled M. Elbassioni, and Vladimir Gurvich. Algorithms for generating minimal blockers of perfect matchings in bipartite graphs and related problems. In *ESA*, volume 3221 of *Lecture Notes in Computer Science*, pages 122–133. Springer, 2004.

[BFMY83] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *J. ACM*, 30(3):479–513, 1983.

[BGS20] Christoph Berkholz, Fabian Gerhardt, and Nicole Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020.

[BHT01] Jean R.S. Blair, Pinar Heggernes, and Jan Arne Telle. A practical algorithm for making filled graphs minimal. *Theoretical Computer Science*, 250(1–2):125 – 141, 2001.

[BKR11] Andrew Blake, Pushmeet Kohli, and Carsten Rother. *Markov Random Fields for Vision and Image Processing*. The MIT Press, 2011.

[BKS17] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering conjunctive queries under updates. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of*

Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017, pages 303–318, 2017.

[BKS18] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. In *21st International Conference on Database Theory, ICDT 2018, March 26-29, 2018, Vienna, Austria*, pages 8:1–8:19, 2018.

[CFK+06] Sara Cohen, Itzhak Fadida, Yaron Kanza, Benny Kimelfeld, and Yehoshua Sagiv. Full disjunctions: Polynomial-delay iterators in action. In *VLDB*, pages 739–750. ACM, 2006.

[CFWY14] Yang Cao, Wenfei Fan, Tianyu Wo, and Wenyuan Yu. Bounded conjunctive queries. *PVLDB*, 7(12):1231–1242, 2014.

[CGM+17] Alessio Conte, Roberto Grossi, Andrea Marino, Takeaki Uno, and Luca Versari. Listing maximal independent sets with minimal space and bounded delay. In *International Symposium on String Processing and Information Retrieval*, pages 144–160. Springer, 2017.

[CK18] Nofar Carmeli and Markus Kröll. Enumeration complexity of conjunctive queries with functional dependencies. In *21st International Conference on Database Theory (ICDT 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[CK19a] Nofar Carmeli and Markus Kröll. Enumeration complexity of conjunctive queries with functional dependencies. *Theory of Computing Systems*, pages 1–33, 2019.

[CK19b] Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 134–148, 2019.

[CKK17] Nofar Carmeli, Batya Kenig, and Benny Kimelfeld. Efficiently enumerating minimal triangulations. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 273–287. ACM, 2017.

[CKKK20] Nofar Carmeli, Batya Kenig, Benny Kimelfeld, and Markus Kröll. Efficiently enumerating minimal triangulations. *Discrete Applied Mathematics*, 2020.

[CKS08] Sara Cohen, Benny Kimelfeld, and Yehoshua Sagiv. Generating all maximal induced subgraphs for hereditary and connected-hereditary graph properties. *J. Comput. Syst. Sci.*, 74(7):1147–1159, 2008.

[CM77] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, page 77–90, New York, NY, USA, 1977. Association for Computing Machinery.

[CMN99] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. On random sampling over joins. In *SIGMOD*, pages 263–274. ACM Press, 1999.

[CS18] Florent Capelli and Yann Strozecki. Incremental delay enumeration: Space and time. *Discrete Applied Mathematics*, 2018.

[CU19] Alessio Conte and Takeaki Uno. New polynomial delay bounds for maximal subgraph enumeration by proximity search. In *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing*, pages 1179–1190, 2019.

[CZB+20] Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 393–409, 2020.

[Dah97] Dias Dahlhaus. *WG*, chapter Minimal elimination ordering inside a given chordal graph, pages 132–143. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[Dir61] Gabriel A. Dirac. On rigid circuit graphs. Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg 25, Universität Hamburg, 1961.

[DK19] Shaleen Deep and Paraschos Koutris. Ranked enumeration of conjunctive query results. *CoRR*, abs/1902.02698, 2019.

[Dur64] Richard Durstenfeld. Algorithm 235: Random permutation. *C. ACM*, 7(7):420, 1964.

[Dur20] Arnaud Durand. Fine-grained complexity analysis of queries: From decision to counting and enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 331–346, 2020.

[Fag83] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, July 1983.

[FFG02] Jörg Flum, Markus Frick, and Martin Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.

[FG02] Maáyan Fishelson and Dan Geiger. Exact genetic linkage computations for general pedigrees. In *ISMB/ECCB*, pages 189–198, 2002.

[Gav74] F. Gavril. The intersection graphs of subtrees in trees are exactly the chordal graphs. *J. Combinatorial Theory*, 16:47–56, 1974.

[GGM+05] Georg Gottlob, Martin Grohe, Nysret Musliu, Marko Samer, and Francesco Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *WG*, volume 3787 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2005.

[GGS05] Georg Gottlob, Gianluigi Greco, and Francesco Scarcello. Pure nash equilibria: Hard and easy games. *J. Artif. Intell. Res. (JAIR)*, 24:357–406, 2005.

[GKS95] M.C. Golumbic, H. Kaplan, and R. Shamir. Graph sandwich problems. *Journal of Algorithms*, 19(3):449 – 473, 1995.

[GKS08] Konstantin Golenberg, Benny Kimelfeld, and Yehoshua Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, pages 927–940. ACM, 2008.

[GLS02] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579 – 627, 2002.

[GMS09] Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM*, 56(6), 2009.

[GO95] Anka Gajentaan and Mark H Overmars. On a class of o (n2) problems in computational geometry. *Computational geometry*, 5(3):165–185, 1995.

[Gol80] Martin Charles Golumbic. *CHAPTER 4 - Triangulated Graphs*, pages 81 – 104. Academic Press, 1980.

[Gra96] Etienne Grandjean. Sorting, linear time and the satisfiability problem. *Ann. Math. Artif. Intell.*, 16:183–236, 1996.

[Heg06] Pinar Heggernes. Minimal triangulations of graphs: A survey. *Discrete Mathematics*, 306(3):297 – 317, 2006. Minimal Separation and Minimal Triangulation.

[HH99] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298. ACM Press, 1999.

[HP02] Vagelis Hristidis and Yannis Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, pages 670–681. Morgan Kaufmann, 2002.

[IUV17] Muhammad Idris, Martin Ugarte, and Stijn Vansummeren. The dynamic yannakakis algorithm: Compact and efficient query processing under updates. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 1259–1274, New York, NY, USA, 2017. ACM.

[Jor02] M. Jordan. *An Introduction to Probabilistic Graphical Models*, chapter 17. University of California, Berkeley, 2002.

[JPY88] David S. Johnson, Christos H. Papadimitriou, and Mihalis Yannakakis. On generating all maximal independent sets. *Inf. Process. Lett.*, 27(3):119–123, 1988.

[Kaz13] Wojciech Kazana. *Query evaluation with constant delay.* Theses, École normale supérieure de Cachan - ENS Cachan, September 2013.

[KEK16] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. Flexible caching in trie joins. *CoRR*, abs/1602.08721, 2016.

[KG15] Batya Kenig and Avigdor Gal. On the impact of junction-tree topology on weighted model counting. In *SUM*, volume 9310 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2015.

[Kim12] Benny Kimelfeld. A dichotomy in the complexity of deletion propagation with functional dependencies. In *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '12, pages 191–202, New York, NY, USA, 2012. ACM.

[KKS97] Ton Kloks, Dieter Kratsch, and Jeremy P. Spinrad. On treewidth and minimum fill-in of asteroidal triple-free graphs. *Theor. Comput. Sci.*, 175(2):309–335, 1997.

[KLM89] Richard M Karp, Michael Luby, and Neal Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429 – 448, 1989.

[KM98] P.Sreenivasa Kumar and C.E.Veni Madhavan. Minimal vertex separators of chordal graphs. *Discrete Applied Mathematics*, 89(1–3):155 – 168, 1998.

[KPP16] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In Robert Krauthgamer, editor, *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2016, Arlington, VA, USA, January 10-12, 2016*, pages 1272–1287. SIAM, 2016.

[KV00] Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. *J. Comput. Syst. Sci.*, 61(2):302–332, 2000.

[LG14] François Le Gall. Powers of tensors and fast matrix multiplication. In *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, ISSAC '14, pages 296–303, New York, NY, USA, 2014. ACM.

[LLK80] Eugene L. Lawler, Jan Karel Lenstra, and A. H. G. Rinnooy Kan. Generating all maximal independent sets: Np-hardness and polynomial-time algorithms. *SIAM J. Comput.*, 9(3):558–565, 1980.

[LS88] S. Lauritzen and D. J. Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society*, B, 50(2):157–224, 1988.

[LWW18] Andrea Lincoln, Virginia Vassilevska Williams, and R. Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1236–1252, 2018.

[LWYZ19] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. Wander join and XDB: online aggregation via random walks. *ACM Trans. Database Syst.*, 44(1):2:1–2:41, 2019.

[Mar10] Dániel Marx. Approximating fractional hypertree width. *ACM Trans. Algorithms*, 6(2), 2010.

[MS91] Bernard M. E. Moret and Henry D. Shapiro. *Algorithms from P to NP: Volume 1: Design & Efficiency.* Benjamin-Cummings, 1991.

[OCF76] Tatsuo Ohtsuki, Lap Kit Cheung, and Toshio Fujisawa. Minimal triangulation of a graph and optimal pivoting order in a sparse matrix.

Journal of Mathematical Analysis and Applications, 54(3):622 – 633, 1976.

[OZ15] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015.

[Pey01] Barry W. Peyton. Minimal orderings revisited. *SIAM J. Matrix Analysis Applications*, 23(1):271–294, 2001.

[PS97] Andreas Parra and Petra Scheffler. Characterizations and algorithmic applications of chordal graph embeddings. *Discrete Applied Mathematics*, 79(1-3):171–188, 1997.

[PY99] Christos H. Papadimitriou and Mihalis Yannakakis. On the complexity of database queries. *J. Comput. Syst. Sci.*, 58(3):407–427, 1999.

[RMK19] Noam Ravid, Dori Medini, and Benny Kimelfeld. Ranked enumeration of minimal triangulations. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, PODS '19, pages 74–88, New York, NY, USA, 2019. ACM.

[Ros70] Donald J Rose. Triangulated graphs and the elimination process. *Journal of Mathematical Analysis and Applications*, 32(3):597 – 609, 1970.

[RS84] Neil Robertson and P.D Seymour. Graph minors. iii. planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49 – 64, 1984.

[Seg15] Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Rec.*, 44(1):10–17, May 2015.

[SK01] W. Nick Street and YongSeog Kim. A streaming ensemble algorithm (SEA) for large-scale classification. In *KDD*, pages 377–382. ACM, 2001.

[Str10] Y. Strozecki. *Enumeration complexity and matroid decomposition.* PhD thesis, Université Paris Diderot - Paris 7, 2010.

[SV17] Luc Segoufin and Alexandre Vigny. Constant delay enumeration for FO queries over databases with local bounded expansion. In *20th International Conference on Database Theory, ICDT 2017, Venice, Italy*, pages 20:1–20:16, 2017.

[TAG+19] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *CoRR*, abs/1911.05582, 2019.

[TR15] Susan Tu and Christopher Ré. DunceCap: Query plans using generalized hypertree decompositions. In *SIGMOD*, pages 2077–2078. ACM, 2015.

[TY84] Robert E. Tarjan and Mihalis Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput.*, 13(3):566–579, July 1984.

[WW10] Virginia Vassilevska Williams and Ryan Williams. Subcubic equivalences between path, matrix and triangle problems. In *51th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2010, Las Vegas, Nevada, USA*, pages 645–654, 2010.

[Yan81a] M. Yannakakis. Computing the minimum fill-in is np-complete. *SIAM Journal on Algebraic Discrete Methods*, 2(1):77–79, 1981.

[Yan81b] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 82–94. VLDB Endowment, 1981.

[YKW10] Takeo Yamada, Seiji Kataoka, and Kohtaro Watanabe. Listing all the minimum spanning trees in an undirected graph. *Int. J. Comput. Math.*, 87(14):3175–3185, 2010.

[ZCL+18] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. Random sampling over joins revisited. In *SIGMOD*, pages 1525–1539. ACM, 2018.

[ZMC06] Jizhen Zhao, Russell L. Malmberg, and Liming Cai. Rapid *ab initio* RNA folding including pseudoknots via graph tree decomposition. In *WABI*, pages 262–273, 2006.

כל תוצאות האפיון שהזכרנו עד כה תקפות למסדי נתונים כלליים, כשלא ניתן להניח דבר על הקלט. בפועל, לעיתים קרובות יש תלויות בין תכונות שונות השמורות במסד הנתונים. במקרים כאלה, תוצאות הקושי שהזכרנו לא תקפות, ולמעשה אנחנו מראים ששאילתות נוספות הן קלות כשיש תלויות במסד הנתונים. אנחנו בוחנים מקרים בהם יש תלויות פונקציונליות, מגדירים הרחבה מבוססת תלויות, ומראים ששאילתות עם הרחבה מבוססת תלויות מחוברת־חופשיים הן קלות. בנוסף, אנחנו מראים הרחבה לדיכוטומיות שתיארנו עד כה במקרה של תלויות בהן משתנה אחד גורר אחר. במקרה כזה, השאילתה היא קלה אם ורק אם ההרחבה מבוססת התלויות שלה היא בעלת מבנה קל לפי הדיכוטומיה הידועה עבור מסדי נתונים כלליים.

עבור שאילתות המאופיינות כקשות, לא ניתן להשתמש בהרחבות מבוססות איחוד או בהרחבות מבוססות תלויות כדי לשכתב את השאילתה לצורה חסרת מעגלים שקולה. כחלופה, אנחנו יכולים לפרק את השאילתה. בעזרת פירוק כזה, ניתן לשכתב את השאילתה לצורה חסרת מעגלים שקולה לאחר זמן עיבוד מקדים גדול מלינארי [GGS05]. למרות שהפתרון הזה לא נותן את הזמן האופטימלי שהצלחנו להשיג במקרה של שאילתות קלות, הוא יכול להוריד את זמן החישוב משמעותית בהשוואה לחישוב ישיר של השאילתה המעגלית הנתונה. לכן, אנחנו בוחנים את המשימה של מציאת פירוק עצי איכותי לגרף המייצג את השאילתה.

מציאת פירוק עצי אופטימלי היא בעיה שהוכחה כקשה [ACP87]. לכן, נפוץ השימוש בהיוריסטיקות שמוצאות עצים טובים. עם זאת, זמן החישוב רגיש מאוד לאיכות הפירוק, וייתכן הבדל משמעותי בזמן הריצה בין הפירוק האופטימלי לפירוק שנמצא בגישה היוריסטית. אנחנו מציעים לנקוט בגישה של מניית כל עצי הפירוק שאינם יתירים. מתברר שהמשימה הזו שקולה למציאת כל הטריאנגולציות המינימליות של גרף. סיבוכיות מציאת הטריאנגולציות היתה בעיה פתוחה, ואנחנו פותרים אותה בתזה הזאת. אנחנו מציעים אלגוריתם שיכול להתבסס על כל פתרון קיים למציאת פירוק בודד על מנת ליצור היצע רחב של פירוקים איכותיים. אז, המשתמש יכול להחליט מתי פירוק שהוחזר הוא טוב מספיק, ולעצור את ריצת האלגוריתם.

שלנו מאלגוריתם יעיל ומספקת לו פחות זמן כולל למציאת כל התשובות. דיכוטומיה ידועה עבור שאילתות צירוף עם הטלה ללא צירופים עצמיים מראה: שאילתה כזו היא קלה אם ורק אם היא חסרת מעגלים "ומחוברת־חופשיים" [BDG07]. שאילתה נקראת מחוברת־חופשיים אם היא נשארת חסרת מעגלים גם לאחר הוספת אטום חדש המכיל את המשתנים החופשיים.

הצעד הטבעי הבא הוא לבחון שאילתות המורכבות מאיחוד של צירוף עם הטלה. זו מחלקה חשובה של שאילתות, כי ניתן לבטא בעזרתה כל שאילתת אלגברה רלציונית חיובית. כשמאפשרים איחוד, התמונה נהיית יותר מורכבת, ומציאת הסיבוכיות הופכת למשימה יותר מאתגרת. איחוד של שאילתות קלות הוא תמיד קל. אנחנו בוחנים גם את המקרה שהאיחוד מכיל שאילתה קשה, ומראים שאפילו אם האיחוד הוא רק של שאילתות קשות, השאילתה כולה יכולה להיות קלה. לצורך זה, אנחנו מזהים כיצד חלקים שונים באיחוד יכולים לפשט זה את זה בעזרת אספקת משתנים. אנחנו מגדירים הרחבת שאילתות מבוססת איחוד, ומראים שכל שאילתה עם הרחבה מחוברת־חופשיים היא קלה.

אנחנו גם מוכיחים שעבור מחלקות מסוימות של שאילתות, הרחבה מחוברת־חופשיים מאפיינת בדיוק את כל השאילתות הקלות. זה תקף במקרה שהאיחוד מכיל שתי שאילתות צירוף עם הטלה קשות, או במקרה ששתי השאילתות באיחוד זהות עד כדי הטלה. בנוסף, אנחנו מגדירים השערה על מציאת משולשים בגרפים לא מאוזנים, ומראים קשור הדוק בין הקושי של מציאת המשולשים לקושי של חישוב שאילתות איחוד שאין להן הרחבה מחוברת־חופשיים. בפרט, אנחנו מראים שאם ההשערה נכונה – ההרחבה מאפיינת את כל השאילתות הקלות במקרה של איחוד של שתי שאילתות בינאריות חסרות צירופים עצמיים, ואם הרחבה כזו מאפיינת את כל השאילתות הקלות – ההשערה בהכרח נכונה.

לאחר מכן, אנחנו בוחנים כיצד ניתן להפיק תועלת נוספת משאילתות קלות מעבר לדרישות המנייה שדנו בהן עד כה. המטרה שלנו בחלק הזה היא לענות על שאילתות בסדר אקראי. הבטחה כזו על הסדר נדרשת אם השימוש בתוצאות הביניים של השאילתה מניח שיש להן משמעות סטטיסטית והן מייצגות את כל התוצאות. אנחנו מראים שניתן להשיג מנייה כזאת אם יש ברשותנו אלגוריתם יעיל לגישה אקראית לתוצאות השאילתה: מציאת תשובה בהינתן מיקומה ברשימת התשובות. אנחנו חוקרים את הסיבוכיות של שתי המשימות האלה בהשוואה לסיבוכיות של מניית התשובות ללא דרישות על הסדר.

אנחנו מראים שבמקרה של שאילתות צירוף עם הטלה, השאילתות מחוברות־החופשיים הן בדיוק השאילתות הקלות ביחס לכל שלושת המשימות. במקרה של מנייה בסדר אקראי ושל גישה אקראית, אנחנו מאפשרים עיכוב לוגריתמי בין תשובות עוקבות. במקרה של שאילתות איחוד, התמונה יותר מורכבת: איחוד של שאילתות קלות הוא תמיד קל ביחס למנייה, אבל הוא יכול להיות קשה ביחס לגישה אקראית. אנחנו מציעים שני פתרונות למנייה בסדר אקראי: אלגוריתם עם תוחלת עיכוב לוגריתמית, ואלגוריתם נוסף עם חסם לוגריתמי על העיכוב שניתן להשתמש בו רק בחלק מהמקרים.

תקציר

חישוב שאילתות מעל מסדי נתונים הוא בעיה בסיסית בתחום ניהול המידע. עם עליית כמות המידע הזמין בעולם, גדלה חשיבותו של ניתוח המידע ביעילות, ומתחזק הצורך באפיון השאילתות שניתן לחשב ביעילות. לכן, בשנים האחרונות חלה התקדמות משמעותית בהבנת הסיבוכיות המדויקת של מניית התשובות של שאילתות.

כמות התשובות של שאילתה יכולה להיות גדולה בסדרי גודל ממסד הנתונים עליו היא מתבססת. במקרים כאלה, לא ניתן לקוות להחזיר את כל התוצאות בזמן לינארי בגודל מסד הנתונים, כי אלגוריתם לחישוב שאילתה כזו נדרש לקרוא את הקלט ואז להדפיס את כל התוצאות. לכן, אנחנו משתמשים במדדי סיבוכיות מתחום סיבוכיות המנייה, ואנחנו בוחנים את מחלקת הבעיות שניתן לפתור בזמן עיבוד מקדים לינארי ואז בעיכוב קבוע בין כל שתי תשובות עוקבות. במובן מסוים, אלה הן בעיות המנייה הקלות ביותר שאינן טריויאליות. בתקציר הזה נקרא לבעיות במחלקה הזו "קלות", ולבעיות האחרות נקרא "קשות".

אנחנו מתייחסים לשאילתה כנתונה מראש, והבעיה שאנחנו מעוניינים לפתור היא: בהינתן מסד נתונים כקלט, יש להחזיר את כל התשובות של השאילתה על מסד הנתונים כפלט. בפרט, לצורך ניתוח הסיבוכיות אנחנו משתמשים במדד של סיבוכיות מידע: גודל השאילתה (שהוא בדרך כלל קטן משמעותית מגודל מסד הנתונים) נחשב קבוע.

הפעולה הבסיסית ביותר שעומדת בלב שאילתות נפוצות היא צירוף טבלאות. זו גם בדרך כלל הפעולה הכי יקרה חישובית. נדון תחילה בשאילתות המורכבות מצירוף בלבד. עבור שאילתות צירוף המכילות מעגלים, אם השאילתה לא מכילה צירופים עצמיים (אין טבלה שמופיעה פעמיים בשאילתה), לא ניתן לקבוע האם יש לה תשובות בזמן לינארי [BB13]. לעומת זאת, קיים אלגוריתם ידוע שמחשב שאילתות צירוף חסרות מעגלים ביעילות [Yan81b]. כלומר, יש דיכוטומיה עבור שאילתות צירוף ללא צירופים עצמיים: שאילתות כאלה הן קלות אם ורק אם הן חסרות מעגלים. תוצאות הקושי כאן, כמו גם כל תוצאות הקושי שאנחנו מראים בתזה, מותנות בהנחות קושי מסוימות הנהוגות בתחום הסיבוכיות המדויקת.

כשמוסיפים לשאילתות צירוף גם הטלה, הדיכוטומיה שהזכרנו כבר לא מתקיימת, ורמת הקושי של חישוב השאילתה יכולה לעלות. הסיבה היא שייתכן ששתי תשובות שונות לצירוף יהפכו לזהות לאחר ההטלה. מכיוון שאנחנו לא מאפשרים להדפיס כפילויות ואנחנו מספקים לאלגוריתם רק זמן קבוע לתשובה, ההטלה מקשיחה את הדרישות

המחקר בוצע בהנחייתו של פרופסור בני קימלפלד, בפקולטה למדעי המחשב.

חלק מן התוצאות בחיבור זה פורסמו כמאמרים מאת המחברת ושותפיה למחקר בכנסים ובכתבי־עת במהלך תקופת מחקר הדוקטורט של המחברת, אשר גרסאותיהם העדכניות ביותר הינן:

Nofar Carmeli and Markus Kröll. Enumeration complexity of conjunctive queries with functional dependencies. In *21st International Conference on Database Theory (ICDT 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

Nofar Carmeli and Markus Kröll. Enumeration complexity of conjunctive queries with functional dependencies. *Theory of Computing Systems*, pages 1–33, 2019.

Nofar Carmeli and Markus Kröll. On the enumeration complexity of unions of conjunctive queries. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 134–148, 2019.

Nofar Carmeli, Batya Kenig, and Benny Kimelfeld. Efficiently enumerating minimal triangulations. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 273–287. ACM, 2017.

Nofar Carmeli, Batya Kenig, Benny Kimelfeld, and Markus Kröll. Efficiently enumerating minimal triangulations. *Discrete Applied Mathematics*, 2020.

Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 393–409, 2020.

אני מודה לטכניון על התמיכה הכספית הנדיבה בהשתלמותי.

חשיבות חוסר המעגליות בסיבוכיות בעיות מנייה במסדי נתונים

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

נופר כרמלי

הוגש לסנט הטכניון --- מכון טכנולוגי לישראל
אלול התש"פ חיפה ספטמבר 2020

חשיבות חוסר המעגליות בסיבוכיות בעיות מנייה במסדי נתונים

נופר כרמלי