Answering (Unions of) Conjunctive Queries using Random Access and Random-Order Enumeration

NOFAR CARMELI, Technion, Israel and DI ENS, ENS, Université PSL, CNRS, Inria, France
SHAI ZEEVI, Technion, Israel
CHRISTOPH BERKHOLZ, Humboldt-Universität zu Berlin, Germany
ALESSIO CONTE, University of Pisa, Italy
BENNY KIMELFELD, Technion, Israel
NICOLE SCHWEIKARDT, Humboldt-Universität zu Berlin, Germany

As data analytics becomes more crucial to digital systems, so grows the importance of characterizing the database queries that admit a more efficient evaluation. We consider the tractability yardstick of answer enumeration with a polylogarithmic delay after a linear-time preprocessing phase. Such an evaluation is obtained by constructing, in the preprocessing phase, a data structure that supports polylogarithmic-delay enumeration. In this article, we seek a structure that supports the more demanding task of a "random permutation": polylogarithmic-delay 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 meaningful manner. An even more demanding task is that of "random access": polylogarithmic-time retrieval of an answer whose position is given.

We establish that the free-connex acyclic CQs are tractable in all three senses: enumeration, random-order enumeration, and random access; and in the absence of self-joins, it follows from past results that every other CQ is intractable by each of the three (under some fine-grained complexity assumptions). However, the three yardsticks are separated in the case of a union of CQs (UCQ): while a union of free-connex acyclic CQs has a tractable enumeration, it may (provably) admit no random access. We identify a fragment of such UCQs where we can guarantee random access with polylogarithmic access time (and linear-time preprocessing) and a more general fragment where we can guarantee tractable random permutation. For general unions of free-connex acyclic CQs, we devise two algorithms with relaxed guarantees: one has logarithmic delay in expectation, and the other provides a permutation that is almost uniformly distributed. Finally, we present an implementation and an empirical study that show a considerable practical superiority of our random-order enumeration approach over state-of-the-art alternatives.

CCS Concepts: • **Theory of computation → Database theory**; *Complexity classes*; *Database query languages (principles)*; *Database query processing and optimization (theory).*

Additional Key Words and Phrases: unions of conjunctive queries, enumeration, complexity

ACM Reference Format:
Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Alessio Conte, Benny Kimelfeld, and Nicole Schweikardt. 2023. Answering (Unions of) Conjunctive Queries using Random Access and Random-Order Enumeration. 1, 1 (January 2023), 49 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Nofar Carmeli, Technion, Haifa, Israel, 32000 , DI ENS, ENS, Université PSL, CNRS, Inria, Paris, France, snofca@cs.technion.ac.il; Shai Zeevi, Technion, Haifa, Israel, 32000, shai.zeevi@cs.technion.ac.il; Christoph Berkholz, Humboldt-Universität zu Berlin, Berlin, Germany, 12489, berkholz@informatik.hu-berlin.de; Alessio Conte, University of Pisa, Pisa, Toscana, Italy, conte@di.unipi.it; Benny Kimelfeld, Technion, Haifa, Israel, 32000, bennyk@cs.technion.ac.il; Nicole Schweikardt, Humboldt-Universität zu Berlin, Berlin, Germany, 12489, schweikn@informatik.hu-berlin.de.

2023. Manuscript submitted to ACM

1 INTRODUCTION

In the effort of reducing the computational cost of database queries to the very least possible, recent years have seen substantial progress in understanding the fine-grained data complexity of enumerating the query answers. The seminal work of Bagan, Durand, and Grandjean [6] has established that the *free-connex acyclic* conjunctive queries (or just *free-connex CQs* for short) can be evaluated using an enumeration algorithm with a constant delay between consecutive answers, after a linear-time preprocessing phase. Moreover, their work, combined with that of Brault-Baron [9], established that, in the absence of self-joins (i.e., when every relation occurs at most once), the free-connex CQs are *precisely* the CQs that have such an evaluation. The lower-bound part of this dichotomy requires some lower-bound assumptions in fine-grained complexity (namely, that neither *sparse Boolean matrix-multiplication*, nor *triangle detection*, nor *hyperclique detection* can be done in linear time). Later generalizations consider unions of CQs (UCQs) [8, 13] and the presence of constraints [8, 12].

As a query-evaluation paradigm, the enumeration approach has the important guarantee that the number of intermediate 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. Some examples are the following.

- In *machine-learning* pipelines, we may use query answers as the features of a model [29, 30] for the task of learning a model. Within such pipelines, the intermediate results can be used to save time by invoking the next-step processing without waiting for the full result to be materialized (e.g., as in streaming learning algorithms [36]), and possibly allow the user to stop the execution when the learning process stabilises.
- Database clients often allow the user to execute queries over a user-interface application (e.g., a Web browser) and present a long list of answers with *paging* to allow the user to get a sense of the space of answers (or search for a specific answer) before the entire result is computed.
- In data-exploration frameworks, we might wish to visualize approximate summaries of query answers early on, before completing full query evaluation, and let these summaries improve over time (e.g., as in online aggregation [26, 31]).

Yet, these 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 merely an artifact of the tree selected to utilize free-connexity, and hence, intermediate answers may feature an extreme bias. Importantly, there has been a considerable recent progress in understanding the ability to enumerate the answers not just efficiently, but also with a guarantee on the sorting order [20, 37]: which order (e.g., lexicographic order) allows for efficient enumeration and which order is intractable?

To be a statistically meaningful representation of the space of answers, the enumeration order needs to be provably random. In this article, 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 by a randomized algorithm, so that each output comes from a uniform distribution over the remaining answers. In particular, the enumeration 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 [20, 37], our focus here is on achieving a *logarithmic* or *polylogarithmic 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 (poly)logarithmic-time per sample. Note that sampling *with* replacement has been studied in the past [2, 16] and recently gained a renewed attention [42].

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 were 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 polylogarithmic time. We show that, having this structure at hand, we can use the Fisher-Yates shuffle [22] 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 polylogarithmic 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 algorithm for free-connex CQs has been established by Brault-Baron [9]. 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*, namely the operation of finding the index of a given answer, 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., [42]) 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[1] 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 [8, 13]. Interestingly, this is no longer the case for random access! The reason is as follows. An efficient-random access algorithm 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. At this point, we are investigating three questions:

(1) Can we identify a nontrivial class of UCQs with an efficient random access?
(2) Can we identify a nontrivial class of UCQs with an efficient random permutation, without requiring random access?
(3) Can we achieve efficient random permutation for a substantially larger class of UCQs at the cost of (slightly) relaxing the required guarantees?

For the first question, we identify the class of *mutually-compatible UCQs* (mc-UCQs) and show that every such UCQ has an efficient random-access algorithm. For the second question, we identify a more general class of UCQs that allows for random permutation, by building on the following insight: Knowing how many answers are in the intersection of two free-connex acyclic CQs suffices to guarantee an efficient random permutation. As for the third question, we show that the answer is positive if we allow one of two relaxations. First, there is a random permutation where *each delay* is a

[1]The *coupon collector problem* asks how many samples with replacement are needed in expectation before we see every sample at least once. The name refers to the situation where one purchases boxes with random coupons inside (and no double discount allowed) until every coupon is collected.

geometric random variable with a logarithmic mean; in particular, each delay is logarithmic *in expectation*. Second, there is a permutation in worst-case polylogarithmic delay (and logarithmic delay in expectation) where the order is *almost random*; in particular, the probability of the next answer can be arbitrarily close to its probability in a uniform distribution, and the probability of a whole permutation is the one of the uniform distribution up to a factor of two.

Finally, we present an implementation of our random-access and random-permutation algorithms, and present an empirical evaluation. Over the TPC-H benchmark, we compare our random permutation to the approach of using a state-of-the-art random sampler [42], which is designed to produce a uniform sample with replacement, and then remove duplicates as they are detected. The experiments show that our algorithms feature not only complexity and statistical guarantees, but also a significant practical improvement. Moreover, the theoretical improvement of Fisher-Yates over our random access for mc-UCQs, compared to our generic algorithm for UCQ random permutation, is not consistently evident in the experiments; there is work to be done on how to leverage the advantages of this class in practice.

A short version of this article appeared in a conference proceedings [15]. Compared to that version, the major additions in this manuscript are as follows. First, we significantly clarified the randomness requirement in our definitions (Section 3). Second, we added proofs of correctness for the CQ algorithm (Section 4.2). Third, we added two algorithms for UCQs, one that assumes that we can count the answers in the intersection of the CQs (Section 5.4) and one with guaranteed polylogarithmic delay (Section 5.3). We also added the two new algorithms to the experimental evaluation (Section 7). Fourth, we added a new section that analyses the space usage (Section 6). Lastly, throughout the paper we added variants of our algorithms that use van Emde Boas trees and, thereby, guarantee a reduced delay at the cost of a slightly increased preprocessing time.

The remainder of the manuscript is structured as follows. The basic notation is fixed in Section 2. In Section 3 we introduce three classes of enumeration problems and discuss the relationship between them. Sections 4 and 5 are devoted to our results concerning CQs and UCQs, respectively. Section 6 comments on the space consumption of our algorithms. Section 7 presents our experimental study. Additional information concerning the experimental study is provided in the Appendix.

2 PRELIMINARIES

In this section, we provide basic definitions and notation that we will use throughout this article. For integers ℓ, m we write $[\ell, m]$ for the set of all integers i with $\ell \leq i \leq m$.

Databases and Queries. A (relational) *schema* $\mathbf{S}$ is a collection of *relation symbols* R, each with an associated arity $\text{ar}(R)$. A *relation* r is a set of tuples of *constants*, where each tuple has the same arity (length). A *database* D (over the schema $\mathbf{S}$) associates with each relation symbol R a finite relation r, which we denote by R^D, such that $\text{ar}(R) = \text{ar}(R^D)$. Notationally, we identify a database D with its finite set of *facts* $R(c_1, \dots, c_k)$, stating that the relation R^D over the k-ary relation symbol R contains the tuple $(c_1, \dots, c_k)$.

A *conjunctive query* (CQ) over the schema $\mathbf{S}$ is a relational query Q defined by a first-order formula of the form $\exists \vec{y}\, \varphi(\vec{x}, \vec{y})$, where φ is a conjunction of atomic formulas of the form $R(\vec{t})$ with variables among those in $\vec{x}$ and $\vec{y}$. We write a CQ Q shortly as a logic rule, that is, an expression of the form $Q(\vec{x})$:- $R_1(\vec{t}_1), \dots, R_n(\vec{t}_n)$ where each R_i is a relation symbol of $\mathbf{S}$, each $\vec{t}_i$ is a tuple of variables and constants with the same arity as R_i, and $\vec{x}$ is a tuple of k variables from $\vec{t}_1, \dots, \vec{t}_n$. We call $Q(\vec{x})$ the *head* of Q, and $R_1(\vec{t}_1), \dots, R_n(\vec{t}_n)$ the *body* of Q. Each $R_i(\vec{t}_i)$ is an *atom* of Q. We use $Vars(Q)$ and $Vars(\alpha)$ to denote the sets of variables that occur in the CQ Q and the atom α, respectively. The variables occurring in the head are called the *head variables*, and we make the standard safety assumption that every head variable occurs at least once in the body. The variables occurring in the body but not in the head are existentially quantified, and are called the *existential variables*. A CQ with no existential variables is called a *full join query*.

We usually omit the explicit specification of the schema $\mathbf{S}$, and simply assume that it is the one that consists of the relation symbols that occur in the query at hand.

A *homomorphism* from a CQ Q to a database D is a mapping of the variables in Q to the constants of D, such that every atom of Q is mapped to a fact of D. Each such homomorphism h yields an *answer* to Q, which is obtained from $\vec{x}$ by replacing every variable in $\vec{x}$ with the constant it is mapped to by h. That is, an answer can be seen as an assignment to $\vec{x}$ that can be extended into a homomorphism. In particular, two homomorphisms may differ only on the $\vec{y}$ part, hence producing the same answer. We denote by $Q(D)$ the set of all answers to Q on D. We say that a database D is *globally consistent* with respect to Q if each fact in D *agrees* with some answer in $Q(D)$; that is, there exists a homomorphism from Q to D and an atom of Q such that the homomorphism maps the atom to the fact.

A *self-join* in a CQ Q is a pair of distinct atoms over the same relation symbol. We say that Q is *self-join-free* if it has no self-joins, that is, every relation symbol occurs at most once in the body.

To each CQ $Q(\vec{x})$:- $\alpha_1(\vec{x},\vec{y}),\dots,\alpha_k(\vec{x},\vec{y})$ we associate a hypergraph $\mathcal{H}_Q$ where the nodes are the variables in $\mathit{Vars}(Q)$, and the edges are $E = \{e_1, ..., e_k\}$ such that $e_i = \mathit{Vars}(\alpha_i)$. Hence, the nodes of $\mathcal{H}_Q$ are $\vec{x} \cup \vec{y}$, and the hyperedge e_i includes all the variables that appear in α_i. A CQ Q is *acyclic* if its hypergraph is *α-acyclic*. That is, there exists a tree T (called a *join-tree* of Q) such that $\mathit{nodes}(T) = \mathit{edges}(\mathcal{H}_Q)$, and for every $v \in \mathit{nodes}(\mathcal{H}_Q)$, the nodes of T that contain v form a (connected) subtree of T. The CQ Q is *free-connex* if Q is acyclic and $\mathcal{H}_Q$ remains acyclic when adding a hyperedge that consists of the free variables of Q.

A *union of CQs* (UCQ) is a query of the form $Q_1(\vec{x}) \cup \dots \cup Q_m(\vec{x})$, where every Q_i is a CQ with the sequence $\vec{x}$ of head variables. The set of answers to $Q_1(\vec{x}) \cup \dots \cup Q_m(\vec{x})$ over a database D is, naturally, the union $Q_1(D) \cup \dots \cup Q_m(D)$.

Computation Model and Data Structures. 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* $\mathcal{A}$ for an enumeration problem P is an algorithm that consists of two phases: *preprocessing* and *enumeration*. During preprocessing, $\mathcal{A}$ is given an input I, and it builds certain 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 denote the running time of the preprocessing phase by t_p. The *delay*, denoted by t_d, is an upper bound on the time between printing any two answers during the enumeration phase, the time from the beginning of the enumeration phase and until printing the first answer, and the time after printing the last answer and until the algorithm terminates.

In this article, an enumeration problem will refer to a query (namely, a CQ or a UCQ) Q, the input I is a database D, and the answer set Y is $Q(D)$. Hence, we adopt *data complexity*, where the query is treated as fixed. We use a variant of the *Random Access Machine* (RAM) model with uniform cost measure that has been adopted as a standard computation model in a substantial part of the database theory literature, cf. e.g. [5, 24, 27]. This model enables the construction of lookup tables of polynomial size that can be queried in constant time. In particular, it is possible to compute the semi-join of two relations in linear time. For the randomized version of this model, it is reasonable to assume that it takes constant time to draw a random number of size polynomial in the input size.

In the RAM model, we are able to implement a *van Emde Boas tree* (vEB-tree) [38], which is a data structure that represents a set $S \subseteq [N]$, for any fixed number $N = n^{O(1)}$ that is polynomial in the input size n. A vEB-tree supports the following operations in $O(\log\log n)$ time for a given $i \in [N]$: test whether $i \in S$, insert i into S, delete i from S, find the largest $j < i$ such that $j \in S$. Using lazy initialization, the data structure can be built in time $O(|S| \cdot \log\log n)$.

For simplicity, we will describe our algorithms using standard $O(\log n)$ binary search to maintain a dynamic set and perform predecessor search. Whenever possible, we then discuss the improvements from $O(\log n)$ to $O(\log \log n)$ gained by using vEB-trees instead. Note, however, that in our setting this improvement comes at the cost of increasing the preprocessing time from "linear" to $O(|D| \cdot \log \log |D|)$.

Complexity Hypotheses. Our conditional optimality results rely on the following hypotheses on the hardness of algorithmic problems.

The hypothesis sparse-BMM states that there is no algorithm that multiplies two Boolean matrices A and B (represented as lists of their non-zero entries) over the Boolean semiring in time $m^{1+o(1)}$, where m is the number of non-zero entries in A, B, and AB. The best known algorithms for this problem require at least $\Omega(m^{4/3})$ [4, 19, 41].[2] As it is not clear how further improvement of (dense) matrix multiplication could be used to improve this bound below $m^{4/3}$, sparse-BMM is a valid conjecture even if the matrix multiplication exponent[3] ω is equal to 2.

By Triangle we denote the hypothesis that there is no $O(m)$-time algorithm that detects whether a graph with m edges contains a triangle. The best known algorithm for this problem runs in time $m^{2\omega/(\omega+1)+o(1)}$ [3], which is $\Omega(m^{4/3})$ even if $\omega = 2$. The Triangle hypothesis is also implied by a slightly stronger conjecture in [1].

A $(k{+}1, k)$-hyperclique is a set of $k{+}1$ vertices in a hypergraph such that every k-element subset is a hyperedge. By Hyperclique we denote the hypothesis that for every $k \geq 3$ there is no time $O(m)$ algorithm for deciding the existence of a $(k{+}1, k)$-hyperclique in a k-uniform hypergraph with m hyperedges. This hypothesis is implied by the (l, k)-Hyperclique conjecture proposed in [32].

While the three hypotheses are not as established as classical complexity assumptions (like P $\neq$ NP), their refutation would lead to unexpected breakthroughs in algorithms, which would be achieved when improving the relevant methods in our article.

3 ENUMERATION CLASSES

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

3.1 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.

Definition 3.1. *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 $\mathcal{A}$ where every emission is done uniformly at random. Yet, to make this requirement formal, we need to be precise about the source of randomness and probability space underlying this requirement. We require the preprocessing phase of $\mathcal{A}$ to be deterministic, and each emission execution can be probabilistic; in other words, $\mathcal{A}$ is allowed to use random bits in the enumeration phase but not in the preprocessing phase. Moreover, we require that each of the remaining answers has the same probability

[2]Amossen and Pagh [4] proved an upper bound of $\tilde{O}(m_{\text{in}}^{2/3} m_{\text{out}}^{2/3} + m_{\text{in}}^{0.862} m_{\text{in}}^{0.408})$ where m_{in} and m_{out} are the number of ones in the input and output, respectively (hence implying $m^{4/3+o(1)}$). However, it has been pointed out by Deep et al. [19] that this bound only holds if $m_{\text{in}} < m_{\text{out}}$.
[3]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. The currently best bound on ω is $\omega < 2.373$ and it is conjectured that $\omega = 2$ [25, 39].

over the random bits of the corresponding emission execution of $\mathcal{A}$. That is, for every input I, if $|P(I)| = n$ and $1 \leq j \leq n$, then the jth emission $\mathcal{A}$ can toss a set B_j of random bits and, consequently, it emits a random answer X_j. We require that

$$\Pr_{B_j}(X_j = a) \;=\; \frac{1}{n-j+1}$$

for each answer a among the remaining $n-j+1$ answers that have not yet been emitted. In particular, every ordering of the answer set $P(I)$ has probability $\frac{1}{n!}$ of representing the order in which $\mathcal{A}$ prints the answers.

Definition 3.2. *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 3.3. *By definition,* $\mathsf{REnum}\langle \text{lin}, d\rangle \subseteq \mathsf{Enum}\langle \text{lin}, d\rangle$ *for all d.*

Remark 3.4. *The choice of the randomness model is worth a discussion. The requirement from the random-permutation algorithm is strong in the sense that it guarantees more than just a good behavior of the distribution over the permutations. For example, according to our definition, an emission execution cannot determine (randomly) the* next two answers *since the latter answer will not be uniformly distributed over the random bits of the next execution (but rather be decided deterministically). However, such an algorithm could, potentially, still guarantee a uniform distribution over the permutations. According to our definition, even different executions of the emission from* the same state *of the machine (e.g., due to parallelization or restoration of a previous version) are guaranteed to behave uniformly, that is, cast probabilistically independent (uniformly distributed) answers.*

One could consider alternative, more permissive definitions of a random permutation. For example, we could require the uniform distribution over the remaining answers to be over the entire collection of random bits *in all of the execution of $\mathcal{A}$ up to the jth emission, with or without randomness in the preprocessing phase. We have chosen the strictest definition to ensure that our algorithms, designed to meet the strict requirement, apply in weaker variants as well. Note, however, that our lower bounds on the complexity of random permutation are applicable regardless of any distribution requirements (hence, apply also to weaker alternatives).* □

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 $t_1, ..., t_n$ and called *the enumeration order of* $\mathcal{A}$ such that, when the access routine is called with parameter i, it returns t_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* t_i; the time it takes to access a tuple is called *access time* and denoted t_a.

Definition 3.5. *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 3.6. *By definition,* $\mathsf{RAccess}\langle \text{lin}, d\rangle \subseteq \mathsf{Enum}\langle \text{lin}, d\rangle$ *for all d.*

Algorithm 1 Random Permutation

```
1: procedure SHUFFLE($n$)
2:     assume $a[0], ..., a[n-1]$ are uninitialized
3:     for $i$ in $0, \dots, n-1$ do
4:         choose $j$ uniformly from $i, \dots, n-1$
5:         if $a[i]$ is uninitialized then
6:             $a[i] = i$
7:         if $a[j]$ is uninitialized then
8:             $a[j] = j$
9:         swap $a[i]$ and $a[j]$ ; output $a[i]$
```

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 3.7. *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 3.8. *By definition,* $\mathsf{TWAccess}\langle \text{lin}, d\rangle \subseteq \mathsf{RAccess}\langle \text{lin}, d\rangle$ *for all d.*

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

3.2 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 much larger than the size of the input; however, we want 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* [22], so that it works with constant delay after constant preprocessing time. The original version of the Fisher-Yates Shuffle (also known as *Knuth Shuffle*) [22] 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 containing the numbers $0, \dots, n-1$. 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 3.9 describes an adaptation of this procedure that runs with constant delay and constant preprocessing time in the RAM model.

Proposition 3.9. *A random permutation of $0, \dots, n-1$ can be generated with constant delay and constant preprocessing time.*

Proof. Algorithm 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.[4] 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 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 1 runs with constant delay, 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 $\{0, \ldots, n-1\}$, we can now argue that 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 by 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 3.10. *If $P \in \mathsf{RAccess}\langle \text{lin}, \log^c \rangle$ and P has polynomially many answers, then $P \in \mathsf{REnum}\langle \text{lin}, \log^c \rangle$, for all $c \geq 1$.*

Proof. Let P be an enumeration problem in $\mathsf{RAccess}\langle \text{lin}, \log^c \rangle$, and let $\mathcal{A}$ be the associated random-access algorithm for P. When given an input I, our random-permutation algorithm proceeds as follows. It 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-bounds 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 3.9 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 CONJUNCTIVE QUERIES

In this section, we discuss random access for CQs. For enumeration, the characterization of CQs with respect to $\mathsf{Enum}\langle \text{lin}, \log \rangle$ follows from known results of Bagan, Durand, Grandjean, and Brault-Baron.

[4]To keep track of the array positions that are still uninitialized, one can use standard methods for *lazy array initialization* [33].

Theorem 4.1 ([6, 9]). *Let Q be a CQ. If Q is free-connex, then it is in* $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$. *Otherwise, if it is also self-join-free, then it is not in* $\mathsf{Enum}\langle\text{lin}, \text{polylog}\rangle$ *assuming* sparse-BMM, Triangle, *and* Hyperclique.

Indeed, if the query Q is self-join-free and non-free-connex, there are two cases. If Q is cyclic, then it is not possible to determine whether there exists a first answer to Q in linear time assuming Triangle and Hyperclique [9]. Therefore, Q is not in $\mathsf{Enum}\langle\text{lin}, \text{lin}\rangle$. Otherwise, if Q is acyclic, the proof follows along the same lines as the one presented by Bagan et al. [6]. Using the same reduction as defined there, if any acyclic non-free-connex CQ is in $\mathsf{Enum}\langle\text{lin}, \log^c\rangle$, then any two Boolean matrices A and B 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. This contradicts sparse-BMM. These lower bound techniques apply only when the query has no self-joins; this restriction is required in order to encode the input to the problem we assume to be hard as a database instance, where every atom refers to a different relation. The characterization of non-free-connex CQs with self-joins with respect to enumeration complexity is an open problem, but we do know that some such queries are in $\mathsf{Enum}\langle\text{lin}, \text{const}\rangle$ [7].

Remark 4.2. *Theorem 4.1 applies for general databases. In the common case where the database exhibits dependencies between attributes, there are in fact more tractable cases, and they can be solved via algorithms for free-connex CQs (even when the CQ is not free-connex). To handle the case where the dependencies can be modeled as* Functional Dependencies (FDs)*, Carmeli and Kröll [12] defined the "FD-extension" of a CQ—a variation of the CQ that accounts for the schema's FDs. They showed an* exact reduction *from any CQ to its FD-extension. This means that if a CQ has a free-connex FD-extension, then it is possible to construct an equivalent pair of free-connex CQ and database instance during a linear preprocessing. Then, algorithms that answer this equivalent free-connex CQ also answer the original CQ. As a consequence, positive results for free-connex CQs, including all positive results presented in this article, also apply to CQs that are not free-connex but rather have a free-connex FD-extension. As an example, consider the following CQ:*

$$Q(\textit{employee}, \textit{manager}) \text{ :- } \mathrm{Team}(\textit{employee}, \textit{team}), \mathrm{Manages}(\textit{manager}, \textit{team}).$$

This CQ finds the list of employees and their managers. If we know that every team has one manager, the extended CQ adds the manager *variable wherever the team variable appears, and the* Team *relation has an increased arity in the extended database. The FD-extension is as follows:*

$$Q^{+}(\textit{employee}, \textit{manager}) \text{ :- } \mathrm{Team}(\textit{employee}, \textit{team}, \textit{manager}), \mathrm{Manages}(\textit{manager}, \textit{team}).$$

Note that while Q is not free-connex, its FD-extension Q^{+} is free-connex. Any algorithm for free-connex CQs can be used to solve Q following the linear-time translation of the query and database. □

An implication of Theorem 4.1 is that the answers to free-connex CQs can be enumerated with logarithmic delay. Since Brault-Baron [9] 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 $\mathsf{RAccess}\langle\text{lin}, \log\rangle$. According to Theorem 3.10, this also shows the tractability of a random-order enumeration, that is, membership in $\mathsf{REnum}\langle\text{lin}, \log\rangle$.

4.1 Two-Way-Access Algorithm

We next present a two-way-access algorithm for free-connex CQs. Compared to Brault-Baron [9], 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 5. To proceed, we use the following folklore result.

Algorithm 2 Preprocessing: initialize bucket$[\cdot,\cdot]$, startIndex$(\cdot)$, and w$(\cdot)$

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

Proposition 4.3. *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., [27, 34]). 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 [40] 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 in a way that leaves us with an equivalent 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. A recent tutorial by Berkholz et al. [7, Section 4.1] explains this procedure in more details along with its correctness.

So, it is left to design a random-access algorithm for full acyclic CQs. We do so in the remainder of this section. The idea is to construct a data structure that represents the set of answers in a specific order that we define as follows. Let $R_1, \dots, R_k$ be an ordering of the relations obtained by a DFS traversal on the join tree. We identify the answer t with a sequence $(u_1, \dots, u_k)$ from $R_1 \times \cdots \times R_k$. Then the order that the structure represents is a lexicographic order over the $(u_1, \dots, u_k)$ where each position i is ordered as the original input relation R_i. Then we establish random access from this structure as it supports efficient retrieval of the tuple at a given position in this order. Moreover, we establish efficient inverted access as the structure can be used for efficiently computing the number of answers preceding a given answer. Algorithm 2 describes the preprocessing phase that builds the data structure and computes the count (i.e., the number $|Q(D)|$ of answers). Then, Algorithm 3 provides random-access to the answers, and Algorithm 4 provides inverted-access.

Given a relation R, denote by $\mathrm{pAtts_R}$ the attributes that appear both in R and in its parent. If R is the root, then $\mathrm{pAtts_R} = \emptyset$. Given a relation S and an assignment a of a database constant to each attribute of $\mathrm{pAtts_S}$, we denote by bucket$[S, a]$ all tuples in S that agree with a over the attributes that S and a have in common. (Intuitively, bucket$[S, a]$ is $S \ltimes \{a\}$ when $\{a\}$ is viewed as a singleton relation.) We use this notation also when a is a tuple, by treating the tuple as an assignment from the attributes of its relation to the values it holds.

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 only

Algorithm 3 Random-Access

```
1:  procedure ACCESS($j$)
2:      if $j \geq \mathrm{w}(\mathrm{bucket}[root, \emptyset])$ then
3:          return out-of-bound
4:      else
5:          $answer = \emptyset$
6:          SUBTREEACCESS($\mathrm{bucket}[root, \emptyset]$, $j$)
7:          return $answer$

8:  procedure SUBTREEACCESS($B \subseteq R$, $j$)
9:      find $t \in B$ s.t. $\mathrm{startIndex}(t) \leq j < \mathrm{startIndex}(t{+}1)$    ▷ Implemented by either binary search or via vEB-trees.
10:     $answer = answer \cup \{\mathrm{Atts}_{\mathrm{R}} \longrightarrow \mathrm{Atts}_{\mathrm{R}}(t)\}$
11:     let $R_1, \ldots, R_m$ be the children of $R$
12:     $j_1, \ldots, j_m$ = SPLITINDEX($j - \mathrm{startIndex}(t)$,
13:                     $\mathrm{w}(\mathrm{bucket}[R_1, t]), \ldots, \mathrm{w}(\mathrm{bucket}[R_m, t])$)
14:     for $i$ in $1, \ldots, m$ do
15:         SUBTREEACCESS($\mathrm{bucket}[R_i, t]$, $j_i$)
```

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 $\mathrm{startIndex}(t)$ 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. Intuitively, every tuple t gives rise to $\mathrm{w}(t)$ subanswers, and we are interested in knowing which tuple we will get at a specific index if we duplicate every tuple t $\mathrm{w}(t)$ times; startIndex helps us in this vein by telling us where the sequence of these subanswers begins. At the end of preprocessing, the root relation has one bucket (since $\mathrm{pAtts}_{root} = \emptyset$), and the weight of this bucket represents the number of answers to the query. We remark that this dynamic programming solution that computes this weight function was previously used for counting all query answers [35] and for sampling [42].

The random-access is done recursively in a root-to-leaf order: we start from the 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 attributes of the current relation R with the constants that appear in this tuple t; this is denoted $\mathrm{Atts}_{\mathrm{R}} \longrightarrow \mathrm{Atts}_{\mathrm{R}}(t)$ in the pseudo-code. Next, we assign the rest of the search to the children of the current relation, restricted to the bucket that corresponds to t. Finding t in line 9 can be done in logarithmic time using binary search. Alternatively, we could store the tuples t indexed by their startIndex number in a van Emde Boas tree over $[N]$, where $N = |D|^{|Vars(Q)|}$ is an upper bound on the number of output tuples. As a consequence, we can find t using an $O(\log\log(|D|))$ predecessor search at the cost of increasing the preprocessing time to $O(|D| \cdot \log\log(|D|))$ for building the data structure. The remaining index $j' = j - \mathrm{startIndex}(t)$ 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

Algorithm 4 Inverted-Access

```
1:  procedure InvertedAccess($answer$)
2:      return InvertedSubtreeAccess($root$, $answer$)

3:  procedure InvertedSubtreeAccess($R$, $answer$)
4:      find $t \in R$ s.t. $\mathrm{Atts_R}(t) = \mathrm{Atts_R}(answer)$
5:      if $t$ was not found then
6:          return not-an-answer
7:      let $R_1, \ldots, R_m$ be the children of $R$
8:      for $i$ in $1, \ldots, m$ 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[$R_1$, $answer$]), $j_1, \ldots,$
13:                            w(bucket[$R_m$, $answer$]), $j_m$)
14:     return startIndex($t$) + $offset$
```

arrays: if the last bucket is of weight m, its index is $j' \bmod m$, and the other buckets recursively split between them the index $\lfloor \frac{j'}{m} \rfloor$.

Algorithm 4 works similarly to Algorithm 3, but while the search down the tree in Algorithm 3 is guided by the index and the answer is the assignment, in Algorithm 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 3. Recursively, $\textsc{CombineIndex}(w_1, j_1, \ldots, w_m, j_m) = j_m + w_m \cdot \textsc{CombineIndex}(w_1, j_1, \ldots, w_{m-1}, j_{m-1})$ with $\textsc{CombineIndex}() = 0$.

Line 4 can be supported in constant time after an appropriate indexing of the buckets at preprocessing (in our RAM model). Since Algorithm 4 has a constant number of operations (in data complexity), inverted-access can be done in constant time (after the linear preprocessing provided by Algorithm 2).

The next theorem, parts of which are already given in [9], summarizes the algorithms presented so far.

Theorem 4.4. *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. A variant of the data structure using vEB-trees can be built in time $O(|D| \cdot \log\log(|D|))$ and additionally supports random-access in $O(\log\log(|D|))$.*

Example 4.5. *Consider the CQ*

$$Q(v, w, x, y, z) \text{ :- } R_1(x, v, w), R_2(v, y), R_3(w, z)$$

with the join-tree with R_1 as root, and R_2 and R_3 are its children. Figure 1 depicts an example of the computed information available at the end of preprocessing over an input database. There, the startIndex *value is denoted s.*

Let us now demonstrate how this structure is used during access. Calling Access(13) *finds $(a_2, b_2, c_1) \in R_1$. Then, the remaining $13 - 8 = 5$ is split to $5 \bmod 3 = 2$ 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 the tuples $(b_2, d_3) \in R_2$ and $(c_1, e_3) \in R_3$. Overall, the obtained answer 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 the index* startIndex$(b_2, d_3) = 1$ *from a bucket of weight 2, and calling* InvertedSubtreeAccess *on*

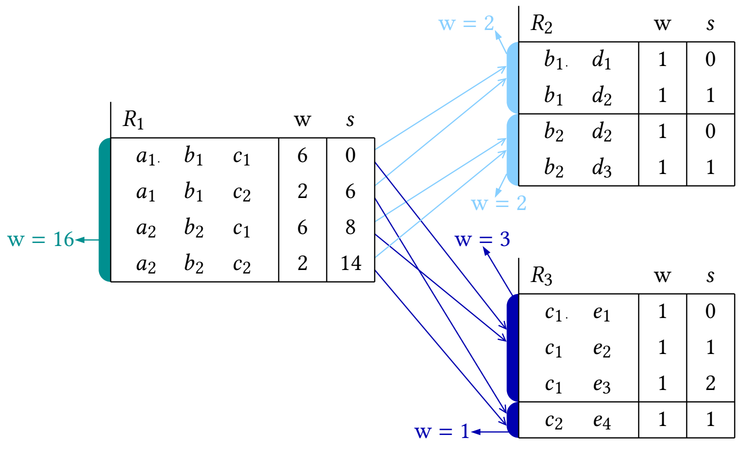


Fig. 1. A data structure constructed during preprocessing for Example 4.5.

R_3 returns $\text{startIndex}(c_1, e_3) = 2$ *from a bucket of weight* 3. *The call for* $\textsc{CombineIndex}(2, 1, 3, 2)$ *returns* $2 + 3 \cdot 1 = 5$, *and the result is* $8 + 5 = 13$. □

4.2 Correctness

We prove the correctness of Algorithms 2, 3 and 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 $\text{join}(T_R)$ the join of the relations in T_R.

Claim 4.6. *Let $t \in R$. The number of answers in* $\text{join}(T_R)$ *that agree with t is* $\text{w}(t)$.

Proof. We prove that $\text{w}(t) = |\{a \in \text{join}(T_R) \mid a \text{ agrees with } t\}|$ by induction on the join tree. If R is a leaf, then $\text{join}(T_R) = R$, and every tuple agrees only with itself. Therefore, $|\{a \in \text{join}(T_R) \mid a \text{ agrees with } t\}| = 1$. By definition of w on leaves, $\text{w}(t) = 1$. 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 impossible that there exists a variable in both R_i and R_j that does not appear in R. Therefore, for every pair of answers $a_i \in \text{join}(T_{R_i})$ and $a_j \in \text{join}(T_{R_j})$, if both of them agree with t, then they also agree with each other. This means that the answers in $\text{join}(T_R)$ 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.

$$|\{a \in \text{join}(T_R) \mid a \text{ agrees with } t\}| \overset{(1)}{=} \prod_{i=1}^{m} |\{a \in \text{join}(T_{R_i}) \mid a \text{ agrees with } t\}|$$

$$\overset{(2)}{=} \prod_{i=1}^{m} \sum_{\{s \in R_i \mid s \text{ agrees with } t\}} |\{a \in \text{join}(T_{R_i}) \mid a \text{ agrees with } s\}| \overset{(3)}{=} \prod_{i=1}^{m} \sum_{\{s \in R_i \mid s \text{ agrees with } t\}} \text{w}(s)$$

$$\overset{(4)}{=} \prod_{i=1}^{m} \sum_{s \in \text{bucket}[R_i, t]} \text{w}(s) \overset{(5)}{=} \prod_{i=1}^{m} \text{w}(\text{bucket}[R_i, t]) \overset{(6)}{=} \text{w}(t)$$

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 4.6, $\mathrm{w}(\mathrm{bucket}[\mathit{root}, \emptyset]) = |Q(D)|$.

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 3 and Algorithm 4. We denote by $\mathrm{parent}(i)$ an index j such that R_j is the parent of R_i in T. The following algorithm 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 5 go over the tuples in each bucket in the same order that was used to set startIndex during preprocessing.

Algorithm 5 Enumeration

```
1: procedure Enumerate($D$, $T$)
2:     for $t_1$ in bucket[$R_1$, ()] do
3:         for $t_2$ in bucket[$R_2$, $t_{\mathrm{parent}(2)}$] do
4:             ...
5:             for $t_n$ in bucket[$R_n$, $t_{\mathrm{parent}(n)}$] do
6:                 output $\bigcup_{i=1}^{n} \mathrm{Atts}_{\mathrm{R_i}} \rightarrow \mathrm{Atts}_{\mathrm{R_i}}(t_i)$
```

Claim 4.7. *Algorithm 5 outputs $Q(D)$.*

Proof. The correctness follows from the correctness of the classic Yannakakis algorithm for acyclic join queries [40]. The semi-join reductions were performed at preprocessing (before applying Algorithm 2), and when Algorithm 5 is performed, we are left with a full acyclic join over a globally consistent database instance. Therefore, Algorithm 5 only needs to join the relations along the join tree. □

We consider the ordering of $Q(D)$ produced by Algorithm 5, and prove that Algorithm 3 is correct with respect to that order.

Claim 4.8. *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 ($\mathrm{Atts}_{\mathrm{R}} \rightarrow \mathrm{Atts}_{\mathrm{R}}(t)$)
```

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 $R_1, \ldots, R_m$. 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 5 are ordered by a DFS, it behaves as follows:

```
procedure Enumerate($D$, $T_B$)
    for $t$ in $B$ do
for $\{a_1 \in \text{join}(T_{R_1}) \mid a_1 \text{ agrees with } t\}$ do
    ...
    for $\{a_m \in \text{join}(T_{R_m}) \mid a_m \text{ agrees with } t\}$ do
        output $(\text{Atts}_R \to \text{Atts}_R(t)) \cup \bigcup_{i=1}^{m} a_i$
```

According to Claim 4.6, 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 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 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 $\{a_i \in \text{join}(T_{R_i}) \mid a_i \text{ agrees with } t\}$. 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 4.6, $\text{w}(\text{bucket}[\mathit{root}, \emptyset]) = |Q(D)|$. Therefore, Access$(j)$ recognizes out-of-bound correctly. Due to Claim 4.8, 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 4.7), this proves that Algorithm 3 is a random-access algorithm for $Q(D)$. It is left to prove the correctness of the inverted-access.

Claim 4.9. *Algorithm 4 is an inverted-access algorithm with respect to Algorithm 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 $\text{join}(T_R)$. 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 $\text{startIndex}(t + 1) = \text{startIndex}(t) + \text{w}(t + 1) = \text{startIndex}(t) + 1$ 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, $\text{startIndex}(t) \leq j < \text{startIndex}(t{+}1) = \text{startIndex}(t) + 1$, and so $\text{startIndex}(t)$ is exactly j. When InvertedSubtreeAccess(R, a) is called, $\mathit{offset} = 0$ as R is a leaf, and so $\text{startIndex}(t) = j$ 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 finds the tuple t used in Algorithm 3. Let $j_1, \ldots, j_m$ be the indices obtained in line 13 of Algorithm 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 sets *offset* to be $j - \text{startIndex}(t)$. InvertedSubtreeAccess(R, a) then returns $j - \text{startIndex}(t) + \text{startIndex}(t) = j$. □

4.3 A Dichotomy for CQs

Theorem 4.4 along with Theorem 3.10 implies that the dichotomy of Theorem 4.1 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 4.10. *For every CQ Q, the following holds: If Q is free-connex, then Q is in each of* $\mathsf{TWAccess}\langle\mathrm{lin},\log\rangle$, $\mathsf{RAccess}\langle\mathrm{lin},\log\rangle$, $\mathsf{REnum}\langle\mathrm{lin},\log\rangle$ *and* $\mathsf{Enum}\langle\mathrm{lin},\log\rangle$.[5] *If Q is self-join-free and not free-connex, then it is not in any of* $\mathsf{TWAccess}\langle\mathrm{lin},\mathrm{polylog}\rangle$, $\mathsf{RAccess}\langle\mathrm{lin},\mathrm{polylog}\rangle$, $\mathsf{REnum}\langle\mathrm{lin},\mathrm{polylog}\rangle$, *and* $\mathsf{Enum}\langle\mathrm{lin},\mathrm{polylog}\rangle$ *assuming* SPARSE-BMM, TRIANGLE, *and* HYPERCLIQUE.

PROOF. According to Theorem 4.4, every free-connex CQ is in the class $\mathsf{TWAccess}\langle\mathrm{lin},\log\rangle$, and also the number of answers to the query can be computed during the linear-time preprocessing phase. By definition, this also means $Q \in \mathsf{RAccess}\langle\mathrm{lin},\log\rangle$. From Theorem 3.10 we obtain that the CQ also is in $\mathsf{REnum}\langle\mathrm{lin},\log\rangle$; and since we have $\mathsf{REnum}\langle\mathrm{lin},\log\rangle \subseteq \mathsf{Enum}\langle\mathrm{lin},\log\rangle$, it is also in the class $\mathsf{Enum}\langle\mathrm{lin},\log\rangle$.

According to Theorem 4.1, self-join-free non-free-connex CQs are not in $\mathsf{Enum}\langle\mathrm{lin},\log\rangle$ (assuming SPARSE-BMM, TRIANGLE, and HYPERCLIQUE); since $\mathsf{REnum}\langle\mathrm{lin},\log\rangle \subseteq \mathsf{Enum}\langle\mathrm{lin},\log\rangle$, they are also not in $\mathsf{REnum}\langle\mathrm{lin},\log\rangle$. According to Theorem 3.10, these CQs are also not in $\mathsf{RAccess}\langle\mathrm{lin},\log\rangle$, and therefore they are not in $\mathsf{TWAccess}\langle\mathrm{lin},\log\rangle$. □

REMARK 4.11. *An alternative way of achieving a random permutation with good expected delay in theory is using counting, sampling (with replacement) and enumeration. First, compute the number of answers. Then, repeatedly sample, while rejecting answers that were already seen, until producing half of the answers. In parallel, enumerate and randomly permute all answers, and delete the answers seen during sampling. As a final step, go over the produced permutation and print all new answers. Since we only produce half of the answers during the sampling phase, the probability of rejecting a sample is at most half. Hence, we are expected to find a new answer after 2 samples or less, and by using an $O(1)$ in expectation sampling algorithm (e.g., using the alias method [17, 42]), this phase has $O(1)$ expected delay. The number of operations required for the production and permutation of all answers is known in advance and linear in the number of answers. Thus, this computation can be interleaved with the first half of the sampling phase (i.e., when producing a quarter of the answers by sampling), while only requiring an additional constant number of operations in every delay step (c.f. the* Cheater's Lemma *[13]). The second half of the sampling phase can be used to erase two already seen answers in every delay step. We just need to make sure to store the permutation in a way that allows finding and erasing an answer in constant time. Then, the final phase only needs to read from memory in $O(1)$ delay. Hence, this approach can compute a random permutation for free-connex CQs with linear preprocessing and constant expected delay. Note that, unlike the algorithm we devised in this section, the delay guarantees for this solution are in expectation, and in particular, there is no worst-case guarantee on the total time. We briefly discuss the practicality of this alternative solution in Section 7.*

In the rest of this article, we call the random-permutation algorithm for CQs established in this section CQRA-PERMUTE.

5 UNIONS OF CQS

In this section, we discuss the availability of random-permutation and random-access in UCQs. We first show that not all UCQs that have efficient enumeration also have efficient random-access algorithms. Then, we propose several

[5]Note that the stated upper bounds are not optimal, e. g., $\mathsf{Enum}\langle\mathrm{lin},\mathrm{const}\rangle$ [6] or $\mathsf{RAccess}\langle\mathrm{lin}\cdot\log\log,\log\log\rangle$ (by Theorem 4.4) are possible as well. We stated it this way to provide a unique conditional dichotomy for the cases where (poly)logarithmic access time/delay is possible.

solutions for random-permutation for UCQs, either by relaxing the requirements on the solution or by restricting the class of queries supported by the algorithm.

If several CQs are in $\mathsf{Enum}\langle\mathrm{lin}, d\rangle$, for some d, then their union can also be enumerated within the same time bounds [13, 21]. Since our goal is to answer queries in random order, a natural question arises: does the same apply to queries in $\mathsf{RAccess}\langle\mathrm{lin}, d\rangle$ and $\mathsf{REnum}\langle\mathrm{lin}, d\rangle$? We show that it does not apply to CQs in $\mathsf{RAccess}\langle\mathrm{lin}, d\rangle$. 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 $\mathsf{RAccess}\langle\mathrm{lin}, \mathrm{lin}\rangle$ under Triangle.

Example 5.1. *Consider the CQs $Q_1(x, y, z) \text{ :- } R(x, y), S(y, z)$ and $Q_2(x, y, z) \text{ :- } S(y, z), T(x, z)$. Let $Q_\cup = Q_1 \cup Q_2$. 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 4.4. 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 $Q_\cup \in \mathsf{RAccess}\langle\mathrm{lin}, \mathrm{lin}\rangle$. 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" $Q_\cap(x, y, z) \text{ :- } R(x, y), S(y, z), T(x, z)$ 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 Triangle. *Thus, under* Triangle, *the UCQ $Q_\cup$ does not belong to* $\mathsf{RAccess}\langle\mathrm{lin}, \mathrm{lin}\rangle$.

Example 5.1 shows that (assuming Triangle) $\mathsf{RAccess}\langle\mathrm{lin}, \log\rangle$ is not closed under union. It also shows that, when considering UCQs, we have that $\mathsf{Enum}\langle\mathrm{lin}, \mathrm{const}\rangle \not\subseteq \mathsf{RAccess}\langle\mathrm{lin}, \mathrm{lin}\rangle$. In particular, this means that $\mathsf{Enum}\langle\mathrm{lin}, \log\rangle \neq \mathsf{RAccess}\langle\mathrm{lin}, \log\rangle$, which is not the case when only considering CQs. In Section 5.5, we devise a sufficient condition for UCQs to have a $\mathsf{RAccess}\langle\mathrm{lin}, \mathrm{polylog}\rangle$ computation, and conclude that in this case we also have efficient random-permutation. In the rest of this section we inspect different solutions for random-permutation that do not go through random-access for the union but only through random-access to the individual CQs. These solutions are phrased as random-permutation algorithms for unions of sets, where the sets are assumed to admit efficient counting, uniform sampling, membership testing, and deletion. In Section 5.1, we show that answers to CQs support such operations. We propose three solutions that rely on these operations: a solution that relaxes the delay requirements to logarithmic time *in expectation* in Section 5.2, a solution that relaxes the order requirements to uniformly-random *with high probability* in Section 5.3, and a solution with strict guarantees that can be applied *only* when we can compute during preprocessing the number of answers in the intersection of the different CQs. The following table summarizes the algorithms we propose for the random permutation of answers to unions of free-connex CQs.

Section	Algorithm	Delay Guarantee	Remark
5.2	CQRA-repeat	–	$O(\log(\lvert I\rvert))$ delay in expectation
5.3	CQRA-default	$O(\log^2(\lvert I\rvert))$	$O(\log(\lvert I\rvert))$ delay in expectation, almost random permutation
5.4	CQRA-double	$O(\log(\lvert I\rvert))$	applicable when knowing the intersection sizes
5.5	MCRA-permute	$O(\log^2(\lvert I\rvert))$	applicable for mc-UCQs

Table 1. Random permutation algorithms for unions of free-connex CQs.

5.1 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 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 5.2. *It is possible to support counting, testing, sampling and deletion of a set initialized as* $\{0, \ldots, n-1\}$ *with constant time per operation.*

Algorithm 6 Counting, testing, sampling and deletion for $0, \ldots, n-1$

```
1:  procedure Initialize($n$)
2:      assume $a[0], \ldots, a[n-1]$ are uninitialized
3:      assume $b[0], \ldots, b[n-1]$ are uninitialized
4:      $i = 0$
5:  procedure Count
6:      output $n - i$
7:  procedure Sample
8:      choose $j$ uniformly from $i, \ldots, n-1$
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 $b[k] \geq i$
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:     $i = i + 1$
```

Proof. Algorithm 6 describes a data structure supporting the required operations. As in Algorithm 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, we do not wish to initialize such arrays as n can be very big, and we want the initialization to be done in constant time. So, the arrays can be implemented as lookup tables as in Algorithm 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 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 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 $k \neq$ not-an-answer $\wedge$ D.TEST($k$)
10: procedure DELETE($k$)
11:     $k$ =P.INVERTEDACCESS($a$)
12:     if $k \neq$ not-an-answer then
13:         D.DELETE($k$)
```

If we have counting, random-access and inverted-access procedures for some enumeration problem, we can use Algorithm 6 on the indices of the answers in order to support counting, testing, sampling and deletion of the answers. This is described in Algorithm 7. During initialization, we count the number of answers to our problem P, and initialize Algorithm 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 5.3. *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 4.4), we can apply Lemma 5.3 to free-connex CQs and conclude that they support sampling, testing, deletion and counting in logarithmic time (after linear preprocessing). We conclude the following lemma, which will we use in the next three sections.

LEMMA 5.4. *Given a free-connex CQ Q and a database D, it is possible to build in $O(|D|)$ time a data structure that is initialized to hold $Q(D)$ and supports uniform sampling, membership testing, deletion and counting in $O(\log(|D|))$ time. A variant of the data structure using vEB-trees can be built in time $O(|D| \cdot \log\log(|D|))$ and supports uniform sampling, membership testing, deletion and counting in $O(\log\log(|D|))$ time.*

Algorithm 8 Random-Order Enumeration of $S_1 \cup \cdots \cup S_k$ (CQRA-REPEAT)

```
1: while $\sum_{j=1}^{k} S_j.\textsc{Count}() > 0$ do
2:     $chosen$ = choose $i$ with probability $\frac{S_i.\textsc{Count}()}{\sum_{j=1}^{k} S_j.\textsc{Count}()}$
3:     $element = S_{chosen}.\textsc{Sample}()$
4:     $providers = \{S_j \mid S_j.\textsc{Test}(element) = True\}$
5:     $owner = \min\{j \mid S_j \in providers\}$
6:     for $S_j \in providers \setminus \{S_{owner}\}$ do
7:         $S_j.\textsc{delete}(element)$
8:     if $owner = chosen$ then
9:         $S_{chosen}.\textsc{delete}(element)$ ; output $element$
```

5.2 Random-Permutation with Expected Logarithmic Delay

In order to provide a random-permutation algorithm for UCQs, we devise an algorithm for the union of sets, which are assumed to each support efficient counting, uniform sampling, membership testing, and deletion. If the number of sets in the union is constant, the algorithm also 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 [28] extended by tuple deletions that allow for sampling without replacement. We prove the following lemma.

Lemma 5.5. *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.*

Algorithm 8 enumerates the union of several sets in uniformly random order. Every iteration starts by choosing a random set and a random element it contains. The choice of set is weighted by the number of elements it contains. If the algorithm would have always printed the element at that stage (after line 3), then an element that appears in two sets would have had twice the probability of being chosen compared to an element that appears in only one set. The following lines correct this bias. 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 8 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 $1 - \frac{|\bigcup_{j=1}^{k} S_j|}{\sum_{j=1}^{k} |S_j|}$ 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 8 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 5.5.

Combining Lemma 5.4 with Lemma 5.5, we have an algorithm for answering UCQs with random order. In the rest of this article, we call this solution CQRA-REPEAT.

THEOREM 5.6. *Let Q be a union of free-connex CQs. CQRA-REPEAT is a random-permutation algorithm for answering Q that uses linear preprocessing and expected logarithmic delay. In addition, this algorithm has the guarantee of amortized logarithmic delay.*

REMARK 5.7. *Combining the vEB-tree variant from Lemma 5.4 with Lemma 5.5, one obtains a random-permutation algorithm for answering Q with expected $O(\log\log(|D|))$ delay after $O(|D| \cdot \log\log(|D|))$ preprocessing.*

REMARK 5.8. *It is worth remarking an alternative way to achieve random permutation for a union of k CQs, using the $O(1)$-expected-delay sampling method mentioned in Remark 4.11. The three ingredients we need to use this approach are: counting, sampling (with replacement), and enumeration. As for counting, we cannot always exactly count the number of answers to the UCQ. However, we can count the number of answers to each individual CQ in the union and use half of the size of the largest one. This will give us the two guarantees we need: the number of answers is at least twice this number (which means the number of unseen answer is at least half); and this number is at most a constant fraction of the number of answers (which means we can split the enumeration of all answers to a constant number of operations in each delay step). For sampling, given an algorithm for sampling a single CQ, we can use Algorithm 8 without the deletion only until producing one answer. If we use an $O(1)$ in expectation CQ sampling algorithm in line 3 (e.g., Remark 4.11), we will get an $O(1)$ in expectation UCQ sampling algorithm, since the chance of line 9 being successful is always at least $\frac{1}{2k} = \frac{1}{O(1)}$ (due to the fact that the unseen answers are at least half). Finally, it is known that a union of free-connex CQs can be enumerated efficiently [13, 21]. This approach therefore computes a random permutation for a union of free-connex CQs with linear preprocessing and constant expected delay. Note that, unlike the algorithm we devised in this section, this solution has no worst-case guarantee on the total time. We discuss the practical merit of this alternative in Section 7.*

5.3 Random-Permutation with Guaranteed Logarithmic Delay

In this section, we devise a variation of Algorithm 8 such that the delay is guaranteed in exchange for the chance of a non-uniform output. Assume, as in the previous section, that we are given k sets, each supporting sampling, testing, deletion and counting in time at most t. We wish to generate a random permutation of the n elements in the union of these sets. Algorithm 9 behaves similarly to Algorithm 8, with the addition of limited tolerance to rejections: if the condition in line 9 is not satisfied for γ consecutive times, then the algorithm performs an "out of character" choice and outputs the sampled element anyway on line 12. However, we will prove that a suitable choice of γ can make the probability of line 12 ever being executed essentially negligible, while preserving poly-logarithmic guaranteed delay.

Algorithm 9 Random-Order Enumeration of $S_1 \cup \dots \cup S_k$ with $O(\gamma \log n)$ delay (CQRA-DEFAULT)

```
1: $gap = 0$
2: while $\sum_{j=1}^{k} S_j.\textsc{Count}() > 0$ do
3:     $chosen$ = choose $i$ with probability $\frac{S_i.\textsc{Count}()}{\sum_{j=1}^{k} S_j.\textsc{Count}()}$
4:     $element = S_{chosen}.\textsc{Sample}()$
5:     $providers = \{S_j \mid S_j.\textsc{Test}(element) = True\}$
6:     $owner = \min\{j \mid S_j \in providers\}$
7:     for $S_j \in providers \setminus \{S_{owner}\}$ do
8:         $S_j.\textsc{delete}(element)$
9:     if $owner = chosen$ then
10:        $S_{chosen}.\textsc{delete}(element)$ ; output $element$ ; $gap = 0$    ▷ element chosen with uniform probability
11:    else if $gap = \gamma$ then
12:        $S_{owner}.\textsc{delete}(element)$ ; output $element$ ; $gap = 0$    ▷ element chosen with non-uniform probability
13:    else $gap{+}{+}$
```

Choice of γ. Fix a constant $c \geq 2$. Assuming that $k \geq 2$, let us set $\gamma = \lceil (c+1) \log_{\frac{k}{k-1}}(n') \rceil$, where $n' = \sum_{j=1}^{k} S_j.\textsc{Count}()$. Note that, since $n \leq n' \leq kn$, we have that $\gamma = O(c \log n)$. As a result, the worst-case delay of Algorithm 9 is $O(ktc \log(n))$. Note also that $\gamma \geq (c+1) \log_{\frac{k}{k-1}}(n)$, as this will be useful in the analysis that follows.

In the following we study how likely Algorithm 9 is to enumerate the solutions in a uniformly random order, and when it does not, how close its result is to a random permutation.

Likelihood of non-uniformity. Next, we call the event of an execution of line 12 a *bad element*, while an execution of line 10 is called a *good element*. A bad element occurs after γ consecutive times where an element is sampled not from its owner in line 4. As each element has exactly one owner and at most k providers, the probability of extracting an element from its owner is at least $\frac{1}{k}$. The chance of γ consecutive non-owner extractions is thus:

$$P(\text{bad element}) \leq \left(1 - \frac{1}{k}\right)^{\gamma} = \left(\frac{k}{k-1}\right)^{-\gamma} \leq \left(\frac{k}{k-1}\right)^{-(c+1)\log_{\left(\frac{k}{k-1}\right)}(n)} = n^{-(c+1)}$$

We call the event of an entire run of Algorithm 9 containing no bad element a *good permutation*, while a run that contains a bad element is called a *bad permutation*. Algorithm 9 prints elements exactly n times, so the chance of a bad element occurring throughout *the whole execution* is:

$$P(\text{bad permutation}) \leq P(\text{bad element}) \cdot n \leq n^{-c}$$

We can thus say that *with high probability* line 12 is never reached during an execution of Algorithm 9 and the algorithm outputs a uniformly random permutation.

Bounding the probabilities of obtained permutations. While the algorithm yields a uniformly random permutation with high probability, there is still a small chance of a bad permutation. In the following, we assess how skewed Algorithm 9 is compared to the uniform distribution. In particular, we examine how likely different permutations are, compared to the $\frac{1}{n!}$ probability we would have if the algorithm behaved completely uniformly. First, note that if there is at most one element in the union, any enumeration algorithm trivially returns a uniformly random permutation. In the following, we assume that both $n > 1$ and $k \leq n$ hold.[6]

We start by showing a lower bound on the probability of any permutation. As we established before, Algorithm 9 has probability at least $1 - n^{-c}$ of a good permutation. If this happens, the output is uniformly random, and every permutation has probability $\frac{1}{n!}$. Thus, for every permutation π of the elements in the union,

$$P(\pi) \geq P(\pi \text{ and good permutation}) = P(\text{good permutation})P(\pi|\text{good permutation}) \geq (1 - n^{-c})\frac{1}{n!}$$

We next show an upper bound on the probability of any permutation. Let us consider the probability of Algorithm 9 extracting a specific element e as the next element, in the case where i elements are left. This probability $P(e)$ is given by $P(\text{bad element})P(e|\text{ bad element}) + P(\text{good element})P(e|\text{ good element})$. As we noted above, the chance of Algorithm 9 extracting a *bad element* is some value $\phi \leq n^{-(c+1)}$. In case of a bad element, Algorithm 9 extracts a sample uniformly at random from the multi-union of the k sets. There are at most k copies of b out of the possible copies of elements left to sample, which are of course at least i (and up to ki), so the probability of extracting e from the multi-union is *at most* $\frac{k}{i}$. It follows that:

$$\begin{aligned} P(e) &= P(\text{bad element})P(e|\text{ bad element}) + P(\text{good element})P(e|\text{ good element}) \\ &\leq (\phi)\,\frac{k}{i} + (1-\phi)\,\frac{1}{i} = \frac{1}{i}(1 + \phi(k-1)) \leq \frac{1}{i}\left(1 + \frac{k-1}{n^{c+1}}\right) \end{aligned}$$

Thus, we can upper bound the likelihood of any permutation π of the elements in the union as follows:

$$P(\pi) \leq \prod_{i \in \{n,\dots,1\}} \left(\frac{1}{i} \cdot \left(1 + \frac{k-1}{n^{c+1}}\right)\right) = \frac{1}{n!}\left(1 + \frac{k-1}{n^{c+1}}\right)^n$$

We proceed to analyze the right-most element. As by assumption $k \leq n$, we get: $1 + \frac{k-1}{n^{c+1}} \leq 1 + n^{-c} = e^{\ln(1+n^{-c})}$, where $\ln(\cdot)$ is the natural logarithm. As $\ln(1 + n^{-c}) \leq n^{-c}$ for all $n > 1$, we further have that $(e^{\ln(1+n^{-c})})^n \leq (e^{n^{-c}})^n = e^{n^{-c+1}} = e^{\left(\frac{1}{n^{c-1}}\right)}$. Thus, we can conclude that $\left(1 + \frac{k-1}{n^{c+1}}\right)^n \leq e^{\frac{1}{n^{c-1}}} \leq 2$, meaning:

$$P(\pi) \leq \frac{1}{n!}\left(1 + \frac{k-1}{n^{c+1}}\right)^n \leq e^{\frac{1}{n^{c-1}}} \cdot \frac{1}{n!} \leq 2 \cdot \frac{1}{n!}$$

In other words, the probability of any permutation being output by Algorithm 9 is at most twice that of the uniform distribution, i.e., $\frac{1}{n!}$, and quickly approaches $\frac{1}{n!}$ as c increases, or even as n increases as long as $c \geq 2$ (e.g., if $c = 2$ and $n \geq 100$ we obtain $P(\pi) < \frac{1.01}{n!}$).

It could be said that Algorithm 9 behaves "approximately" like a random-permutation algorithm, since some permutations are slightly more or slightly less likely to be obtained, but these probabilities get arbitrarily close to the uniform as we increase c.[7] More formally, we say that a randomized enumeration algorithm E *behaves uniformly* with probability $1 - \epsilon$ if E can be defined as a random permutation with rejection (i.e., it can decide not to return any answer), except that it behaves differently in the case of rejection (e.g., it returns an arbitrary element), so that the result of the algorithm is always a valid permutation, and the probability that it reaches this alternative of rejection is at most ϵ.

The following lemma summarizes our analysis of the guarantees we obtain by using Algorithm 9.

[6]We make this assumption for the sake of the analysis that follows, but our algorithm can be adapted straightforwardly the handle the case where it is violated. For example, we can introduce a preprocessing phase that produces up to k answers and, if it finds all answers, simply prints them after permuting them uniformly at random; otherwise, $n > k \geq 1$ and the algorithm described here will only be called.

[7]By setting c high enough that $\gamma > n$ one could observe that all permutations become equally likely, at the cost of a higher worst-case delay, as the algorithm becomes equivalent to Algorithm 8.

Lemma 5.9. *Let $S_1, \ldots, S_k$ be sets, with $k \geq 2$, each supporting sampling, testing, deletion and counting in time t. Let $c \geq 2$ be some real number. Then, there is an algorithm for enumerating $\bigcup_{j=1}^{k} S_j$ such that (in the following, $n = |\bigcup_{j=1}^{k} S_j|$):*

- *the delay is $O(kt)$ in expectation and $O(ktc\log(n))$ in the worst case.*
- *assuming $n \geq k$, each permutation π of the elements is obtained with probability:*

$$\frac{1-n^{-c}}{n!} \leq P(\pi) \leq e^{\frac{1}{n^{c-1}}} \cdot \frac{1}{n!} \leq \frac{2}{n!}$$

- *on top of the previous guarantee, with probability at least $1 - n^{-c}$ the algorithm behaves uniformly (i.e., picking each permutation π with probability exactly $\frac{1}{n!}$).*

As before, since we know free-connex CQs support the required operations in logarithmic time, we can combine Lemma 5.9 with Lemma 5.4 to obtain an algorithm for UCQs with these guarantees. We call the obtained algorithm CQRA-default, and we conclude the following theorem.

Theorem 5.10. *Let Q be a union of free-connex CQs. CQRA-default is an algorithm for answering Q that uses linear preprocessing time, having worst-case polylogarithmic and expected logarithmic delay, that behaves uniformly with high probability. Furthermore, each possible permutation of the answers to Q has a guaranteed probability of being returned that deviates from the uniform one by a factor not larger than 2.*

Remark 5.11 (Comparison to Algorithm 8). *The difference between Algorithm 9 and Algorithm 8 is only in the behaviour when many consecutive iterations reject. In that case, Algorithm 9 has spent all of the time it is willing to spend on that next output, and it prints the output candidate it has, while compromising the uniformity of the output order. At the same situation, Algorithm 8 stubbornly continues to try and find an output from the required distribution, while compromising the delay. The analysis given in this section shows that it is not likely that we will reach the problematic case where we have to decide between these two options. Thus, it also shows that the probability of Algorithm 8 having a long delay is very small. In fact, Section 7 shows that in practice this limit is not reached, and so both algorithms behave the same. The trade-off presented here is therefore mainly for the purpose of theoretical worst-case guarantees.*

Remark 5.12. *Here again, we can obtain an alternative approach using the $O(1)$-expected-delay sampling in the approach of Remarks 4.11 and 5.8. We can use an algorithm similar to that of Remark 4.11, with the difference that we build a pool of solutions with a constant-delay enumeration algorithm running in the background, from which we draw an unseen answer every time we reach $O(\log(n))$ rejections due to seen answers. Once a constant fraction of the answers have been produced this way, the background algorithm terminates and we move on to enumerating the still unseen answers that it produced (and already permuted in parallel). Hence, we need approximate counting (as in Remark 5.8), constant-time enumeration, constant-time deletion, and constant-time sampling with replacement . With that, we have the following guarantees: First, as in the proof of Lemma 5.9, the background enumeration is invoked with a probability as small as we wish (where here we can simply use the union bound over all the iterations), so we are close to uniform. Second, we get a better guarantee on the expected delay, which becomes* constant *rather than $O(\log(n))$. Whether we have a guarantee on the probability of each individual permutation, similarly to Lemma 5.9, is left open, and the practical merit of using these alternative techniques is discussed in Section 7.*

5.4 UCQs that Allow for Random-Permutation

Algorithm 8 has the disadvantage that the delay is not bounded by a logarithm (rather it is logarithmic in expectation), while Algorithm 9 does not have this problem, but in exchange it has a small probability of producing a non-uniform output. In this section, we describe an algorithm that does not suffer from any of these issues, but it can only be applied in cases where the size of the intersection is known. We first show this algorithm for a union of two sets, and prove the following lemma.

LEMMA 5.13. *Let S_1 and S_2 be sets, each supporting 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 sample from $S_1 \cup S_2$ in time $O(t)$ and enumerate $S_1 \cup S_2$ in uniformly random order with $O(t)$ delay.*

Lines 2 and 3 of Algorithm 8 sample an element, but the intersection elements are twice as likely as the others to be sampled since they may be sampled from either S_1 or S_2. Algorithm 8 and Algorithm 9 correct this bias by rejecting with a certain probability. We next propose Algorithm 10, which corrects the same bias by sampling twice and then randomly choosing between the two sampled elements. If one of the sampled elements appears in the intersection 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 $|S_1 \cup S_2| = |S_1| + |S_2| - |S_1 \cap S_2|$.

Algorithm 10 Random-Order Enumeration of $S_1 \cup S_2$ (CQRA-DOUBLE)

```
1:  Compute $size_\cap = |S_1 \cap S_2|$
2:  while $|S_1| + |S_2| > 0$ do
3:      $element$ = SAMPLE-UNION($S_1, S_2, size_\cap$)
4:      if $element \in S_1 \cap S_2$ then $size_\cap = size_\cap - 1$
5:      $S_1$.DELETE($element$); $S_2$.DELETE($element$)
6:      output $element$
7:
8:  procedure SAMPLE-UNION($S_1, S_2, |S_1 \cap S_2|$)        ▷ Sets $S_1, S_2$ with $|S_1 \cap S_2| = size_\cap$.
9:      for $k \in \{1, 2\}$ do
10:         $chosen_k$ = choose $i \in \{1, 2\}$ with probability $\frac{|S_i|}{|S_1|+|S_2|}$
11:         $element_k = S_{chosen_k}$.SAMPLE()
12:     if $element_1 \in S_1 \cap S_2$ and $element_2 \notin S_1 \cap S_2$ then
13:         $p_1 = \frac{|S_1 \cup S_2| - |S_1 \cap S_2|}{4|S_1 \cup S_2|}$; $p_2 = 1 - p_1$        ▷ $|S_1 \cup S_2| = |S_1| + |S_2| - size_\cap$.
14:     else if $element_1 \notin S_1 \cap S_2$ and $element_2 \in S_1 \cap S_2$ then
15:         $p_2 = \frac{|S_1 \cup S_2| - |S_1 \cap S_2|}{4|S_1 \cup S_2|}$; $p_1 = 1 - p_2$
16:     else
17:         $p_1 = 1$; $p_2 = 0$
18:     $element$ = choose $element_i$ with probability $p_i$
19:     return $element$
```

We now show that Algorithm 10 prints the results in a uniformly random order. For this we need to argue that SAMPLE-UNION($S_1, S_2, size_\cap$) samples from $S_1 \cup S_2$ uniformly at random. As we show in the proof of Algorithm 8, for every $i \in \{1, 2\}$ and $e \in S_i$, we have that $(chosen, element) = (e, i)$ with probability $\frac{1}{|S_1|+|S_2|}$. Thus, for every $e \notin S_1 \cap S_2$, the probability of $element = e$ 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 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

$e = \mathit{element}_1$ again, but $\mathit{element}_2 \in S_1 \cap S_2$. 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 third case is that $e = \mathit{element}_2$, and $\mathit{element}_1 \in S_1 \cap S_2$. This case has the same probability as the second case. Overall, the probability of choosing e is:

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

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 equal in Algorithm 10, 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 10 generates the elements in uniformly random order. For the time complexity, since every iteration prints an element after a constant number of operations, each taking $O(t)$ time, the delay is bounded by $O(t)$. This proves Lemma 5.13.

Next, we generalize Lemma 5.13 to a union of an arbitrary number of sets $S_1, \dots, S_k$.

Lemma 5.14. *Let $S_1, \dots, S_k$ be sets, each supporting sampling, testing, deletion and counting in time t. If for every $I \subseteq [1, k]$ we can also count $\bigcap_{i \in I} S_i$ in time $O(t)$, then it is possible to sample from $\bigcup_{i \in [k]} S_i$ in time $O(2^k t)$ and enumerate $\bigcup_{i \in [k]} S_i$ in uniformly random order with $O(2^k t)$ delay.*

Proof. Let us write $f(k)$ to denote the time needed to uniformly sample a union of k of these sets. We will show that $f(k) = O(2^k t)$. Then, as in Algorithm 10, we can repeatedly sample and delete the sampled element (while keeping track of the sizes of all intersections) to obtain a random permutation.

When $k = 1$, we can sample S_1 in time $O(t)$, hence $f(1) = O(t)$. When $k > 1$, we call Sample-union(T_1, T_2, $\mathit{size}_\cap$) where $T_1 = \bigcup_{i=1}^{\lfloor \frac{k}{2} \rfloor} S_i$ is the union of the first $\lfloor \frac{k}{2} \rfloor$ sets and $T_2 = \bigcup_{i=\lfloor \frac{k}{2} \rfloor+1}^{k} S_i$ is the union of the other sets. We have shown that this procedure samples an element from the union $T_1 \cup T_2$ uniformly at random, provided that the conditions of Lemma 5.13 are met. To verify this, note that for each $j \in \{1, 2\}$ we can sample from T_j in time $f(\frac{k}{2})$. Furthermore, since we can test and delete for all S_i in time t we can also test and delete for T_1 and T_2 in time $O(kt)$. Counting for T_1, T_2, and $T_1 \cup T_2$ is possible using inclusion-exclusion in time $O(2^k t)$ with the help of the counting procedures for the sets $\bigcap_{i \in I} S_i$. Then, we can count $T_1 \cap T_2$ as $|T_1 \cap T_2| = |T_1| + |T_2| - |T_1 \cup T_2|$. This leads to the recursion $f(k) = t \cdot 2^k + 2f(\frac{k}{2})$ with $f(1) = t$. Unwinding the recursion shows that $f(k) = t\left(\sum_{j=0}^{\log k} 2^{j+\frac{k}{2^j}}\right) + k \cdot f(1)$. The proof is concluded by noting that $\sum_{j=0}^{\log k} 2^{j+\frac{k}{2^j}} \leq \sum_{\ell=0}^{k} 2^\ell \leq 2^{k+1}$: For simplicity, assume that $k = 2^n$ for some $n \geq 4$. Then, $g(j) := j + \frac{k}{2^j} = j + 2^{n-j}$. We have $g(0) = 2^n$ and $g(1) = 2^{n-1}$+1. For every j with $2 \leq j \leq n$ we have $g(j) \leq 2^{n-1}$ (because $g(j) \leq n + 2^{n-2}$ and $n \leq 2^{n-2}$). In particular, this guarantees that $g(j) \leq 2^{n-1}$+j for all $1 \leq j \leq n$. Thus, $\sum_{j=0}^{n} 2^{g(j)} \leq \sum_{\ell=2^{n-1}+1}^{2^n} 2^\ell \leq \sum_{\ell=0}^{k} 2^\ell$. This proves Lemma 5.14. □

As free-connex CQs support the required operations (see Lemma 5.4), Lemma 5.14 yields the following theorem.

Theorem 5.15. *If Q is a union of free-connex CQs where the conjunction of every subset of the set of CQs comprising Q is also free-connex, then $Q \in \mathsf{REnum}\langle \text{lin}, \log\rangle$. A witnessing random-permutation algorithm, called CQRA-double, is obtained by combining Algorithm 10 with Lemma 5.14.*

Note that, since free-connexity can be tested by an algorithm, so can the condition of Theorem 5.15.

Remark 5.16. *The statements of Lemma 5.14 and Theorem 5.15 require the tractability of the conjunction of every subset of the set of CQs in the union. Note that this requirement is needed in order to count the union of all sets using inclusion-exclusion when applying the induction step in Lemma 5.14. Note also that it is not enough to require that the intersection of all CQs is free connex in order to deduce that the conjunction of every subset of the CQs is free-connex. As an example, consider the union of*

$$\begin{aligned} Q_1(x,y,z) &\text{ :- } E_1(x,y), C_1(z), & Q_2(x,y,z) &\text{ :- } E_2(y,z), C_2(x), \\ Q_3(x,y,z) &\text{ :- } E_3(z,x), C_3(y), & Q_4(x,y,z) &\text{ :- } R(x,y,z). \end{aligned}$$

Even though each CQ is free-connex and the intersection of all of them is free-connex, the intersection $Q_1 \cap Q_2 \cap Q_3$ is not acyclic.

5.5 UCQs that Allow for Random-Access

We now identify a class of UCQs that allow for random-access with polylogarithmic access time and linear preprocessing (and hence, via Theorem 3.10 also allow for random-order enumeration with linear preprocessing and polylogarithmic delay).

Assume two sets A and A' such that $A' \subseteq A$. An order over A' is *compatible* with an order over A if the former is a subsequence of the latter, that is, the precedence relationship of the elements of A' is the same in both orders.

Definition 5.17. *A* mutually compatible *UCQ, or* mc-UCQ *for short, is a UCQ $Q = Q_1 \cup \dots \cup Q_m$ such that for all $\emptyset \neq I \subseteq [1,m]$, the CQ $Q_I := \bigcap_{i \in I} Q_i$ is free-connex and, moreover, there are $\mathsf{RAccess}\langle \text{lin}, \log\rangle$-algorithms $\mathcal{A}_I$ for Q_I that:*

(a) provide inverted access in logarithmic time, and
(b) are compatible in the sense that on every database D and $\emptyset \neq I \subseteq [1,m]$, the enumeration order of $\mathcal{A}_I$ on input D is compatible with the enumeration order of $\mathcal{A}_{\{\min(I)\}}$ on input D.

This subsection's main result proves the following.

Theorem 5.18. *Every mc-UCQ Q belongs to $\mathsf{RAccess}\langle \text{lin}, \log^2\rangle$. By combining this with Theorem 3.10 we obtain a random-permutation algorithm called MCRA-permute which witnesses that every mc-UCQ Q belongs to $\mathsf{REnum}\langle \text{lin}, \log^2\rangle$.*

An example of an mc-UCQ is $Q_7^S \cup Q_7^C$ used in the experiments in Section 7. This UCQ is comprised of two acyclic CQs with the same structure, except they use different relations (formed by different selections applied on the same initial relations). These CQs have the following structure for $i \in \{S, C\}$: $Q_7^i(o,c,a,b,p,s,l,m,n)$:- $R(s,a), L(o,p,s,l), O(o,c), B(c,b), N^i(a,m), M^i(b,n)$. Applying Theorem 4.4 on Q_7^S, Q_7^C and $Q_7^S \cap Q_7^C$, we can construct algorithms for random-access and inverted-access in a compatible order.

Remark 5.19. *The definition of an mc-UCQ is, on the face of it and conceivably, more demanding than that of Theorem 5.15 where we require only that every intersection is free-connex. Interestingly, it turns out to be a nontrivial challenge to find a UCQ that is* not *an mc-UCQ but still satisfies the condition of Theorem 5.15. In particular, we leave for future investigation the question of whether being an mc-UCQ is indeed a strictly stronger requirement.*

Let us present a sufficient condition which guarantees a UCQ $Q = Q_1 \cup \cdots \cup Q_m$ to be an mc-UCQ: Let $\vec{x}$ be the free variables of Q, let $F(\vec{x})$ be a new atom, where F is a fresh relation symbol, and let Q' be the CQ obtained as the intersection of all the CQs $Q_1, \ldots, Q_m$ and the new atom $F(\vec{x})$. We claim that if Q' is β-acyclic, *then Q is an mc-UCQ. Recall that a CQ is β-acyclic if, and only if, every subhypergraph of the CQ's hypergraph is α-acyclic [23]. Applying this to the particular CQ Q', we obtain that the CQ Q_I is free-connex for every $\emptyset \neq I \subseteq [1, m]$. To see that also the compatibility condition of Definition 5.17 is satisfied, we can argue as follows. It is known that β-acyclic hypergraphs stay β-acyclic after removing any set of nodes (i.e., variables of the CQ) and, furthermore, there exists an ordering of the CQ's variables which is an elimination order of all queries obtained by taking a subset of the CQ's atoms [10]. This yields an ordering of the variables of Q' that is an elimination order for* all *the queries Q_I (for every $\emptyset \neq I \subseteq [1, m]$). In turn, this implies that there is an ordering $<$ of the UCQ's free variables $\vec{x}$ that does not have a disruptive trio (cf. [14]) with respect to any CQ Q_I. Using results of (Section 3 of) [14], we then obtain* RAccess⟨lin, log⟩*-algorithms $\mathcal{A}_I$ for Q_I for all $\emptyset \neq I \subseteq [1, m]$ that also provide inverted access in logarithmic time and that, moreover, are compatible in the sense that they enumerate the query's result tuples in the same lexicographic order, namely the lexicographic order induced by the ordering $<$ of the UCQ's free variables $\vec{x}$. This proves that the UCQ Q is indeed an mc-UCQ, provided that the CQ Q' is β-acyclic.*

A concrete example is the UCQ $Q = Q_1 \cup Q_2$ with

$$\begin{aligned} Q_1(x, y, z, u) &\quad :\!- \quad R_1(x, y), R_2(y, z), R_3(x, u, v) \\ Q_2(x, y, z, u) &\quad :\!- \quad R_1(x, y), R_4(x, y, z), R_5(x, u) \end{aligned}$$

It can easily be verified that the associated CQ

$$Q'(x, y, z, u) \quad :\!- \quad R_1(x, y), R_2(y, z), R_3(x, u, v), R_4(x, y, z), R_5(x, u), F(x, y, z, u)$$

is β-acyclic. Consequently, Q is an mc-UCQ.

The remainder of this section describes the algorithm for proving Theorem 5.18. By Theorem 3.10 we can focus on RAccess⟨lin, $\log^2$⟩.

Random-access for unions of sets. We start with the abstract setting of providing random-access for a union of sets (of arbitrary elements) and then turn to the specific setting where these sets are the results of the CQs that a given UCQ consists of.

We build upon Durand and Strozecki's *union trick* [21], which can be described as follows. Assume that A and B are two (not necessarily disjoint) subsets of a certain universe U, and for each of these sets, we have available an algorithm that enumerates the elements of the set. Furthermore, assume that for the set B we also have available an algorithm for testing membership in B. The goal is to enumerate $A \cup B$ (and, as usual, all enumerations are without repetitions). The pseudocode for the union trick is provided in Algorithm 11. Here, "a = A.First()" means that the enumeration algorithm for A is started and a shall be the first output element. Similarly, "a = A.Next()" means that the next output element of the enumeration algorithm for A is produced and that a shall be that element. In case that all elements of A have already been enumerated, A.Next() will return the end-of-enumeration message EOE; and in case that A is the empty set, A.First() will return EOE.

Algorithm 11 starts by enumerating all elements of A in the same order as the enumeration algorithm for A, but every time it encounters $a \in A \cap B$, it ignores this element and instead outputs the next available element produced by the enumeration algorithm for B. Once the enumeration of A has terminated, the algorithm proceeds by producing the remaining elements of B. Clearly, Algorithm 11 enumerates all elements in $A \cup B$; and the algorithm's delay is

Algorithm 11 Durand-Strozecki's Union Trick for $A \cup B$

```
1: $a = A.\textsc{First}()$ ; $b = B.\textsc{First}()$
2: while $a \neq \texttt{EOE}$ do
3:     if $a \notin B$ then
4:         output $a$ ; $a = A.\textsc{Next}()$
5:     else
6:         output $b$                                ▷ Note that $b \neq \texttt{EOE}$ at this point.
7:         $b = B.\textsc{Next}()$ ; $a = A.\textsc{Next}()$
8: while $b \neq \texttt{EOE}$ do { output $b$ ; $b = B.\textsc{Next}()$ }
```

$O(d_A + d_B + t_B)$ where d_A and d_B are the delay of the enumeration algorithms for A and B, respectively, and t_B is the time needed for testing membership in B.

The idea is to provide random-access to the jth output element produced by Algorithm 11. Let us write $a_1, a_2, \ldots, a_n$ and $b_1, b_2, \ldots, b_{n'}$ for the elements of A and B, repectively, as they are produced by the given enumeration algorithms for A and for B. Let us first consider the case where $j \leq |A|$. Clearly, the jth output element of Algorithm 11 will be a_j if $a_j \notin B$; and in case that $a_j \in B$, the jth output element of Algorithm 11 will be b_k for the particular number $k = |\{a_1, \ldots, a_j\} \cap B|$. In case that $j > |A|$, the jth output element of Algorithm 11 will be b_ℓ for $\ell = j - |A| + |A \cap B|$.

But how can we compute $k = |\{a_1, \ldots, a_j\} \cap B|$ efficiently upon input of j? Following is a sufficient condition. Assume we have available an algorithm that enumerates $A \cap B$, and its enumeration order is *compatible* with that of the enumeration algorithm for A in the sense defined above. Furthermore, assume that we have available a routine "$(A \cap B).\textsc{InvAcc}(c)$" that, upon input of an arbitrary $c \in A \cap B$ returns the particular number i such that c is the ith element produced by the enumeration algorithm for $A \cap B$. (We say that i is the *rank* of c in $A \cap B$.) Then we can compute $k = |\{a_1, \ldots, a_j\} \cap B|$ by using that $|\{a_1, \ldots, a_j\} \cap B| = (A \cap B).\textsc{InvAcc}(a_j)$. This immediately leads to the random-access algorithm for $A \cup B$ whose pseudocode is given in Algorithm 12.

Algorithm 12 Random-access for $A \cup B$

```
1: function $(A \cup B).\textsc{Access}(j)$
2:     $a = A.\textsc{Access}(j)$
3:     if $a \neq \texttt{Error}$ then
4:         if $a \notin B$ then
5:             output $a$
6:         else
7:             $k = (A \cap B).\textsc{InvAcc}(a)$ ;  $b = B.\textsc{Access}(k)$ ; output $b$
8:     else  $\ell = j - |A| + |A \cap B|$ ; $b = B.\textsc{Access}(\ell)$ ; output $b$
```

The following lemma summarizes the prerequisites and specifies the running time of this algorithm.

Lemma 5.20. *Let A and B be sets. Assume we have available enumeration algorithms for A, for B, and for $A \cap B$ such that*

(1) the enumeration order for $A \cap B$ is compatible with that for A,

(2) after having carried out the preprocessing phase for B,

- *upon input of a number j the routine $B.\textsc{Access}(j)$ returns, within time t_B, the j-th output element of the enumeration algorithm for B,*
- *upon input of an arbitrary u it takes time t_T to test if $u \in B$,*

(3) after having carried out the preprocessing phase for A we know its cardinality $|A|$, and upon input of a number j, the routine $A.\textsc{Access}(j)$ returns, within time t_A, the j-th output element of the enumeration algorithm for A,

(4) after having carried out the preprocessing phase for $A \cap B$, we know its cardinality $|A \cap B|$, and upon input of an arbitrary $c \in A \cap B$, its rank $(A \cap B).\textsc{InvAcc}(c)$ according to the enumeration order for $A \cap B$ can be computed within time t_I.

Then, after having carried out the preprocessing phases for A, for B, and for $A \cap B$, Algorithm 12 provides random-access to $A \cup B$ in such a way that upon input of an arbitrary number j it takes time $O(t_A + t_B + t_T + t_I)$ to output the j-th element enumerated by Algorithm 11.

Our next goal is to generalize this to the union of m sets $S_1, \ldots, S_m$ for an arbitrary $m \geq 2$. We proceed by induction on m and have already established the basis for $m = 2$. Let us now consider the induction step from $m-1$ to m. We let $A = S_1$ and $B = S_2 \cup \cdots \cup S_m$ and use Algorithm 11 to enumerate $A \cup B = S_1 \cup \cdots \cup S_m$, where the routines $B.\textsc{First}()$ and $B.\textsc{Next}()$ are provided by the induction hypothesis. We would like to use Algorithm 12 to provide random-access to the j-th element that will be enumerated from $A \cup B$. By assumption, we know how to compute $|A|$ and $a = A.\textsc{Access}(j)$; and by the induction hypothesis, we already know how to compute $b = B.\textsc{Access}(j)$. What we still need in order to execute Algorithm 12 is a way to compute $|A \cap B|$ and a workaround with which we can replace the command $k = (A \cap B).\textsc{InvAcc}(a)$; recall that this command was introduced to compute the number $k = |\{a_1, \ldots, a_j\} \cap B|$ for $a = a_j$. Applying the two modifications, which we describe in the next paragraph, leads to our recursive random access procedure for $\bigcup_{i=1}^m S_i$ as shown in Algorithm 13.

Algorithm 13 Recursive random-access algorithm for $\bigcup_{i=1}^m S_i$

```
 1: function $(\bigcup_{i=1}^m S_i).\textsc{Access}(j)$
 2:     $a = S_1.\textsc{Access}(j)$
 3:     if $a \neq$ Error then
 4:         if $a \notin \bigcup_{i=2}^m S_i$ then
 5:             output $a$
 6:         else
 7:             $k = \textsc{Compute-}k(a)$ ; $b = (\bigcup_{i=2}^m S_i).\textsc{Access}(k)$ ; output $b$
 8:     else
 9:         $\ell = j - |S_1| + \sum_{\emptyset \neq I \subseteq [2,m]} (-1)^{|I|+1} |T_{1,I}|$;      ▷ Requires counting for $T_{1,I} := S_1 \cap \bigcap_{i \in I} S_i$
10:         $b = (\bigcup_{i=2}^m S_i).\textsc{Access}(\ell)$ ; output $b$
11:
12: function $\textsc{Compute-}k\ (a)$
13:     for each $I$ with $\emptyset \neq I \subseteq [2,m]$ do
14:         $b = T_{1,I}.\textsc{Largest}(a)$ ; $n_{1,I} = T_{1,I}.\textsc{InvAcc}(b)$      ▷ Requires Largest and InvAcc for $T_{1,I} := S_1 \cap \bigcap_{i \in I} S_i$
15:     $k = \sum_{\emptyset \neq I \subseteq [2,m]} (-1)^{|I|+1} n_{1,I}$ ;  output $k$
```

Computing $|A \cap B|$ for $A = S_1$ and $B = S_2 \cup \cdots \cup S_m$ is easy: we can use the inclusion-exclusion principle and obtain

$$|A \cap B| = \left| \bigcup_{i=2}^m (S_1 \cap S_i) \right| = \sum_{\emptyset \neq I \subseteq [2,m]} (-1)^{|I|+1} \left| \bigcap_{i \in I} (S_1 \cap S_i) \right|.$$

Thus, we can compute $|A \cap B|$ provided that for each $I \subseteq [2,m]$, we can compute the cardinality $|T_{1,I}|$ of the set $T_{1,I} := S_1 \cap \bigcap_{i \in I} S_i$. Under the assumption that we can compute the numbers $|T_{1,I}|$, we can replace the computation of ℓ in line 8 of Algorithm 12 by the expression in line 9 of Algorithm 13. Let us now discuss how to compute $k = |\{a_1, \ldots, a_j\} \cap B|$. Again using the inclusion-exclusion principle, we obtain that

$$|\{a_1, \ldots, a_j\} \cap B| = \sum_{\emptyset \neq I \subseteq [2,m]} (-1)^{|I|+1} \left| \bigcap_{i \in I} (\{a_1, \ldots, a_j\} \cap S_i) \right| .$$

We can compute this number if for each $\emptyset \neq I \subseteq [2, m]$ we can compute $n_{1,I} := \left| \{a_1, \ldots, a_j\} \cap \bigcap_{i \in I} S_i \right|$. To compute $n_{1,I}$, assume we have available an algorithm that enumerates $T_{1,I}$, and its enumeration order is compatible with that of the algorithm for $A = S_1$. Furthermore, assume we have available a routine $T_{1,I}.\textsc{InvAcc}(c)$ that, given $c \in T_{1,I}$, returns the particular i such that c is the ith element produced by the enumeration algorithm for $T_{1,I}$. In addition, assume that we have available a routine $T_{1,I}.\textsc{Largest}(a)$ that, given $a \in S_1$, returns the particular $c \in T_{1,I}$ such that c is the largest element of $T_{1,I}$ that is less than or equal to a in the enumeration order of S_1. Then, we can compute $n_{1,I}$ by using that $n_{1,I} = T_{1,I}.\textsc{InvAcc}(b)$ for $b := T_{1,I}.\textsc{Largest}(a_j)$. In summary, we replace the computation of k in line 7 of Algorithm 12 by the procedure $\textsc{Compute-}k$ provided in Algorithm 13. The next lemma states the main properties of our random access algorithm for unions of m sets.

Lemma 5.21. *Let $m \geq 2$ and let $S_1, \ldots, S_m$ be sets. For each $\ell \in [1, m]$ and each I with $\emptyset \neq I \subseteq [\ell{+}1, m]$ let $T_{\ell,I} := S_\ell \cap \bigcap_{i \in I} S_i$. Assume that for each $\ell \in [1, m]$ we have available an enumeration algorithm for S_ℓ, and for each $\emptyset \neq I \subseteq [\ell{+}1, m]$ we have available an enumeration algorithm for $T_{\ell,I}$ such that*

(1) the enumeration order for $T_{\ell,I}$ is compatible with that for S_ℓ,

(2) after having carried out the preprocessing phase for S_ℓ we know its cardinality $|S_\ell|$, and
- *upon input of a number j the routine $S_\ell.\textsc{Access}(j)$ returns, within time t_{acc} the j-th output element of the enumeration algorithm for S_ℓ, and*
- *upon input of an arbitrary u it takes time t_{test} to test if $u \in S_\ell$,*

(3) after having carried out the preprocessing phase for $T_{\ell,I}$ we know its cardinality $|T_{\ell,I}|$ and
- *upon input of an arbitrary $c \in T_{\ell,I}$ its rank $T_{\ell,I}.\textsc{InvAcc}(c)$ can be computed within time $t_{inv\text{-}acc}$, and*
- *upon input of an arbitrary $a \in S_\ell$ it takes time t_{lar} to return the particular $c \in T_{\ell,I}$ such that c is the largest element of $T_{\ell,I}$ that is less than or equal to a according to the enumeration order of S_ℓ.*

Then, after having carried out the preprocessing phases for S_ℓ and $T_{\ell,I}$ for all $\ell \in [1, m]$ and all $\emptyset \neq I \subseteq [\ell{+}1, m]$, Algorithm 13 provides random-access to $S_1 \cup \cdots \cup S_m$ in such a way that upon input of an arbitrary number j it takes time

$$O(\, m{\cdot}t_{acc} + m^2{\cdot}t_{test} + 2^m{\cdot}t_{inv\text{-}acc} + 2^m{\cdot}t_{lar}\,)$$

to output the j-th element that is returned by the enumeration algorithm for $S_1 \cup \cdots \cup S_m$ obtained by an iterated application of Algorithm 11 (starting with $A = S_1$ and $B = S_2 \cup \cdots \cup S_m$).

Proof. The proof follows the sketch described above. By applying the time bound obtained from Lemma 5.20 we obtain the following recursion for describing the time $f(m)$ used for providing access to the j-th element of the union of m sets:

$$f(m) \;=\; t_{acc} + (m{-}1){\cdot}t_{test} + (2^{m-1}{-}1){\cdot}(t_{inv\text{-}acc} + t_{lar}) + f(m{-}1)$$

Solving this recursion provides the claimed time bound. □

Finally, to prove Theorem 5.18, we show the algorithms for the different components. The quadratic-logarithmic part is due to the t_{lar} component, and we show that it suffices for $\textsc{Largest}(a)$.

Proof of Theorem 5.18. Let $Q = Q_1 \cup \cdots \cup Q_m$ be the given mc-UCQ, and let $\mathcal{A}_I$, for all $\emptyset \neq I \subseteq [1,m]$, be RAccess⟨lin, log⟩-algorithms which witness that Q is an mc-UCQ.

Upon input of a database D, we perform the linear-time preprocessing of all the algorithms $\mathcal{A}_I$ on input D. Now consider an arbitrary $\ell \in [1,m]$ and an $I \subseteq [\ell{+}1,m]$. For the sets $S_\ell := Q_{\{\ell\}}(D)$ and $T_{\ell,I} := Q_{\{\ell\}\cup I}(D)$, we then immediately know that all the assumptions of Lemma 5.21 are satisfied, except for the last one (i.e., the last bullet point in item (3) of Lemma 5.21). Furthermore, we know that each of the time bounds t_{test}, t_{acc}, and $t_{inv\text{-}acc}$ are at most logarithmic in the size $|D|$ of D. To finish the proof, it therefore suffices to show the following for each $\ell \in [1,m]$ and each $\emptyset \neq I \subseteq [\ell{+}1,m]$:

($*$) On input of an arbitrary $a \in S_\ell$, within time $O(\log^2 |D|)$ we can output the particular $c \in T_{\ell,I}$ such that c is the largest element of $T_{\ell,I}$ that is less than or equal to a according to the enumeration order of S_ℓ.

We can achieve this by doing a binary search on indexes w.r.t. the enumeration orders on S_ℓ and $T_{\ell,I}$ by using the routines $T_{\ell,I}$.Access and S_ℓ.InvAcc. More precisely, we start by letting $j = S_\ell$.InvAcc(a), $c = T_{\ell,I}$.Access(1), and $j_c = S_\ell$.InvAcc(c). If $j_c = j$ we can safely return c. If $j_c > j$, we return an error message indicating that $T_{\ell,I}$ does not contain any element less than or equal to a. If $j_c < j$, we let $k_c = 1$, $k_d = |T_{\ell,I}|$, $d = T_{\ell,I}$.Access(k_d), and $j_d = S_\ell$.InvAcc(d). If $j_d \leq j$ we can safely return d. Otherwise, we do a binary search based on the invariant that c, d are elements of $T_{\ell,I}$ with $c < a < d$ (where $<$ refers to the enumeration order of S_ℓ), j_c, j_d are their indexes in S_ℓ, and k_c, k_d are their indexes in $T_{\ell,I}$: we let $k' = \lfloor (k_c + k_d)/2 \rfloor$, and in case that $k' = k$ we can safely terminate with output c. Otherwise, we let $c' = T_{\ell,I}$.Access(k') and $j' = S_\ell$.InvAcc(c'). If $j' = j$ we can safely terminate and return c'. If $j' < j$ we proceed letting $(c, d, j_c, j_d, k_c, k_d) = (c', d, j', j_d, k', k_d)$. If $j' > j$ we proceed letting $(c, d, j_c, j_d, k_c, k_d) = (c, c', j_c, j', k_c, k')$. The number of iterations is logarithmic in $|T_{\ell,I}|$, and each iteration invokes a constant number of calls to $T_{\ell,I}$.Access and S_ℓ.InvAcc. Since each such call is answered in time $O(\log |D|)$, we have achieved ($*$). The random access part of Theorem 5.18 now follows from Lemma 5.21 and the random enumeration part then follows by Theorem 3.10. □

Remark 5.22. *The definition of mc-UCQs requires the existence of compatible random access algorithms with linear preprocessing and logarithmic access time. Now suppose we change the definition and require compatible random access algorithms with $O(n \cdot \log\log n)$ preprocessing and $O(\log\log n)$ access time, as provided by the vEB-tree variant in Theorem 4.4. By following the proof of Theorem 5.18 while using the alternative random access procedures as a black box we obtain a random access algorithm with $O(n \cdot \log\log n)$ preprocessing and $O(\log n \cdot \log\log n)$ access time (at the cost of larger space consumption). It remains open whether the additional* $\log n$ *factor introduced by the binary search performed in the proof of Theorem 5.18 can be improved as well.*

6 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 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 6.1. *Any random-permutation algorithm for a problem with n answers must use at least n bits.*

Proof. Since a correct random-permutation algorithm must be able to produce every possible permutation of the answers with a positive probability, for any subset of the answers, such an algorithm may reach a state where exactly these answers were not printed yet. The algorithm must distinguish the 2^n possible states corresponding to the unprinted answers in order to guarantee printing every answer exactly once. (If the algorithm reaches the same state after producing two different subsets of the answers, where some answer a appears in one subset but not the other, then regardless of whether the algorithm chooses to print a following this state, it will be wrong in one of the cases by either printing a twice or not printing a at all.) 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 6.1 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 1 that requires no more than the optimal amount of space.

We propose using a bitset: $\frac{n}{\log n}$ registers, where the ith register contains bits representing the answers numbered $i \log n, \dots, i \log n + \log n - 1$, and each bit is on iff the answer of this index was not printed yet. In order to search these registers efficiently, we store them as leaves in a complete binary search tree where each inner node stores the number of on bits in the leaves of its subtree. Given such a tree, the operations over the registers, required by our algorithms, can be done as follows:

- Test k: return the value of bit number $k - \lfloor \frac{k}{\log n} \rfloor$ in register number $\lfloor \frac{k}{\log n} \rfloor$.
- Delete k: set bit number $k - \lfloor \frac{k}{\log n} \rfloor$ in register number $\lfloor \frac{k}{\log n} \rfloor$ to zero, and reduce 1 from all counts on the path from this register to the root.
- Count: return the value stored at the root.
- Sample: first, draw a random number t between 0 and the count. Then, follow a path from the root to a leaf as follows: if t is larger than the value stored in the left child, go left; otherwise, subtract this value from t and go right. When reaching a leaf, return the index represented by the tth on bit in this register (if this is register a and bit b, return $a \log n + b$).

The number of nodes in such a tree is $O(\frac{n}{\log n})$, and its depth is $O(\log n)$. 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 6.2. *The set $\{0, \dots, n-1\}$ 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 1 with the solution we propose here, we obtain a random-permutation algorithm for free-connex CQs and mc-UCQs with the same time guarantees as before and optimal space consumption. By applying this same solution in Lemma 5.3, we also obtain our results for UCQs with optimal space consumption.

Theorem 6.3. *Algorithms CQRA-permute, CQRA-repeat, CQRA-default, CQRA-double, and MCRA-permute can be realized with the optimal space consumption of $O(n)$ bits, where n is the number of answers.*

Remark 6.4. *Note that the discussion of this section has been relevant to the main algorithms we presented in the manuscript, and not to the vEB-tree versions that we mentioned in the remarks; those would require analyzing the space requirements of the vEB-trees themselves and would result in higher space consumption.*

7 IMPLEMENTATION AND EXPERIMENTS

In this section, we present an implementation and an experimental evaluation of the random-order enumeration algorithms presented in this article. [8] Our algorithm for random-order CQ enumeration proposed in Section 4 is denoted as CQRA-PERMUTE, and the algorithms for UCQs are denoted as in Table 1. The goal of our experiments is twofold. First, we examine the practical execution cost of CQRA-PERMUTE compared to the alternative of repeatedly applying a state-of-the-art sampling algorithm (without replacement) [42] and removing duplicates. Second, we compare the performance of the suggested algorithms for UCQs to one another and examine their empirical overhead compared to the cumulative cost of running CQRA-PERMUTE for each CQ separately. We describe the implementations in Section 7.1, the experimental setup in Section 7.2, and the results in Section 7.3.

7.1 Implementation

We implemented the algorithms suggested in this article in c++14 using the standard library (STL), and mainly the unordered STL containers. For instance, we use an unordered map to partition a table into the buckets of Algorithms 2 and 3. Other than STL, the implementation uses Boost to hash complex types such as vectors.

The CQRA-PERMUTE implementation uses a query compiler that generates c++ code for the specific CQ and database schema. Specifically, the code is generated via templates, which are files of c++ code with *placeholders*. These placeholders stand for query-specific parameters such as the relation names, the attributes and their types, the tree structure of the query, and its head variables. Once these placeholders are filled in and function calls are ordered according to the tree structure, the result is valid c++ code.

CQRA-REPEAT, CQRA-DEFAULT and CQRA-DOUBLE use CQ enumerators as black boxes with an interface of four methods: *count*, *sample*, *test*, and *delete*. Therefore, in addition to the counting and sampling provided by CQRA-PERMUTE, we implemented deletion and testing as explained in Section 5.1. The latter two methods require inverted-access, which we described in Algorithm 4. The inverted-access is compiled only when needed as part of a UCQ enumeration, as it requires non-negligible preprocessing (to support line 4 of Algorithm 4). Hence, CQRA-PERMUTE meets the four requirements when inverted-access is activated and the shuffler capable of deletion is used.

MCRA-PERMUTE uses the underlying index CQRA-PERMUTE for random-access, testing, and inverted-access of all CQs, as well as all intersection CQs. We created MCRA-PERMUTE by using the shuffler described in Algorithm 1 on the random-access for mcUCQs described in Section 5.5. Doing so requires knowing the number of answers after linear time preprocessing. The cardinality of a mcUCQ $Q_1(I) \cup \ldots \cup Q_m(I)$ is simple to compute recursively via the formula $|Q_1(I)| + |Q_2(I) \cup \cdots \cup Q_m(I)| - |Q_1(I) \cap (Q_2(I) \cup \cdots \cup Q_m(I))|$, for which we have all elements after linear time preprocessing. A minor difference between the implementation and the definition in Section 5.5 is that instead of computing the largest answer less than or equal to our current answer and then applying inverted-access on it, we compute that index directly (using the same binary-search concept as in the proof of Theorem 5.18).

7.2 Experimental Setup

We now describe the setup of our experimental study.

Algorithms. To the best of our knowledge, this article is the first to suggest a provably uniform random-order algorithm for CQ enumeration. Therefore, we compare our CQRA-PERMUTE to a sampling solution by Zhao et al. [42] via an implementation from their public repository, also using c++14. Their algorithm generates a uniform sample, and we

[8] The source code can be accessed at: https://github.com/TechnionTDK/cq-random-enum

naively transform it into a sampling-without-replacement algorithm by duplicate elimination (i.e., rejecting previously encountered answers).[9] Zhao et al. [42] suggest four different ways to initialize their algorithm, denoted *RS*, *EO*, *OE*, and *EW*. We compare our algorithm to EW as it consistently outperformed all other methods in our experiments (see Section A.2 in the Appendix for a comparison of the four methods). We denote this variant by Sample(EW). Sample(EW) samples by iterating over the join tree from root to leaves and maintaining an assignment of the variables. At each relation, the algorithm chooses a tuple that agrees with the assignment such that the probability of each tuple being chosen is proportional to its weight. The weights, just like the weights in our algorithm, represent the number of answers the tuple produces when only examining the sub-query rooted in its relation. Hence, the approach taken by Sample(EW) is very similar to our approach when restricted to the first sample. One difference between the solutions is that we start preprocessing with the reduction described in Proposition 4.3 (even if the query is full), meaning we invest time at the beginning of preprocessing to remove useless database facts in order to avoid handling them in later phases (which include the rest of preprocessing). Another difference between the approaches lies in the way of handling projections. The algorithms suggested in this article adhere to set semantics with projection included. In Sample(EW), projection is applied on top of sampling, and the uniform distribution applies to the results prior to projection. As projection may result in duplicates, the sampling can be interpreted as over bag semantics (i.e., if an answer can be explained in two different ways, it is twice as likely to be sampled). This does not pose a problem in our comparisons since we only use full queries in our experiments, where the two interpretations meet. Hence, we consider two algorithms for CQs: CQRA-permute and Sample(EW), and four algorithms for UCQs: CQRA-repeat, CQRA-default, CQRA-double, and MCRA-permute.

In Remarks 4.11, 5.8 and 5.7 we also discuss an alternative algorithm that repeatedly invokes a sampler with replacement and handles repetitions in different ways. Our empirical study shows that this approach is inferior practically, and so, we do not include this algorithm in our list of studied solutions. We refer to this alternative again later in Remark 7.1.

Dataset. We used the TPC-H benchmark as the database for the experiments. We generated a database using the TPC-H *dbgen* tool with a scale factor of $sf = 5$. The database has been instantiated once in memory, and all experiments use the exact same database.

Queries. We compare our CQRA-permute to Sample(EW) using the six free-connex CQs on which Sample(EW) is implemented in the online repository. These are CQs over the TPC-H schema. As we mentioned when discussing the algorithms, Sample(EW) samples under bag semantics while CQRA-permute samples under set semantics. In order to mitigate this difference and allow for a fair comparison, we added attributes from the *LineItem* relation to Q_3, Q_7, Q_9, and Q_{10}. This addition eliminates the projections, which implies that all queries we consider are full, and we obtain an equivalence between set semantics and bag semantics. We note that, in order to support free-connex CQs with projection, the only additional operations that CQRA-permute performs on top of those for full CQs are projecting away some attributes at preprocessing, so this is not expected to have a significant implication on performance. For lack of benchmarks, we phrased UCQs that we believed would form a natural extension to the TPC-H queries. The full description of our queries can be found in the Appendix (Section A.1).

[9]The application of this approach as an enumeration algorithm has also been discussed by Capelli and Strozecki [11].

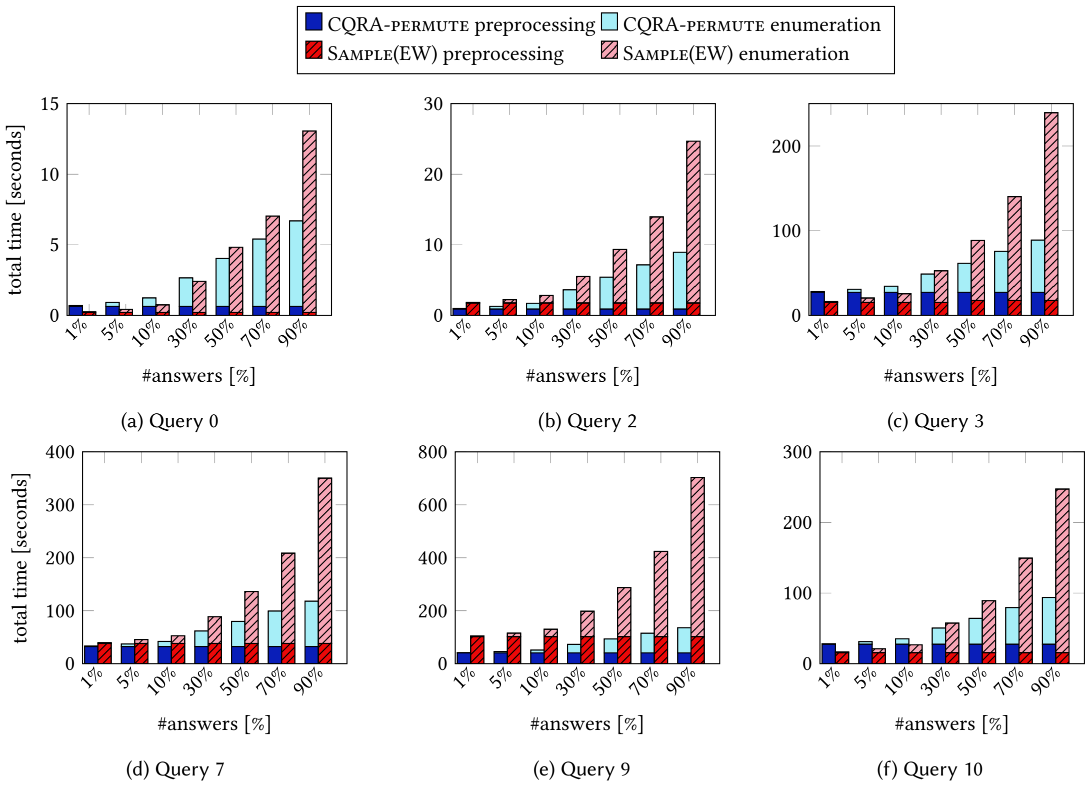


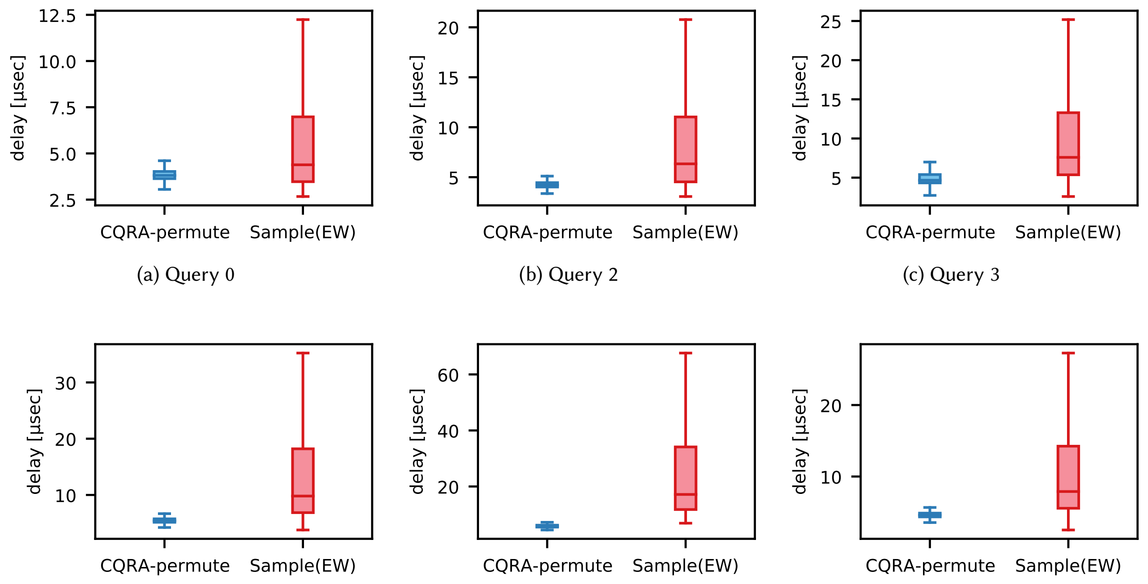
Fig. 2. Total enumeration time of CQs when requesting different percentages of answers. In each bar, the bottom (darker) part refers to the preprocessing phase and the top (lighter) part to the enumeration phase.

Hardware and system. The experiments were executed on an Intel(R) Xeon(R) CPU 2.50GHz machine with 768KB L1 cache, 3MB L2 cache, 30MiB L3 cache, and 496 GB of RAM, running Ubuntu 16.04.01 LTS. Code compilations used the O3 flag and no other optimization flag.

7.3 Experimental Results

We now describe the results of our experimentation. The CQ experiments analyze CQRA-permute in terms of the total enumeration time (Section 7.3.1) and delay (Section 7.3.2), while the UCQ experiments analyze the suggested algorithms in terms of the total enumeration time, as well as the rejection rate of the relevant algorithms (Section 7.3.3). Each result is the average over three runs, except for Figures 3 and 4 that show a single run. We omit from all preprocessing times the portion devoted to reading the relations.

7.3.1 CQ running time. To characterize the total enumeration time of CQRA-permute, we compare it to that of Sample(EW) for the TPC-H CQs. In the experiment, we task each algorithm with enumerating k distinct answers for increasing values of k. The different values of k were chosen as a percentage of the query results (1%, 5%, 10%, 30%, 50%, 70%, 90%). For each task, we measure the total enumeration time, that is, the time elapsed from the beginning of the preprocessing phase to when k distinct answers were supplied. The results of this experiment are presented



(a) Query 0 (b) Query 2 (c) Query 3

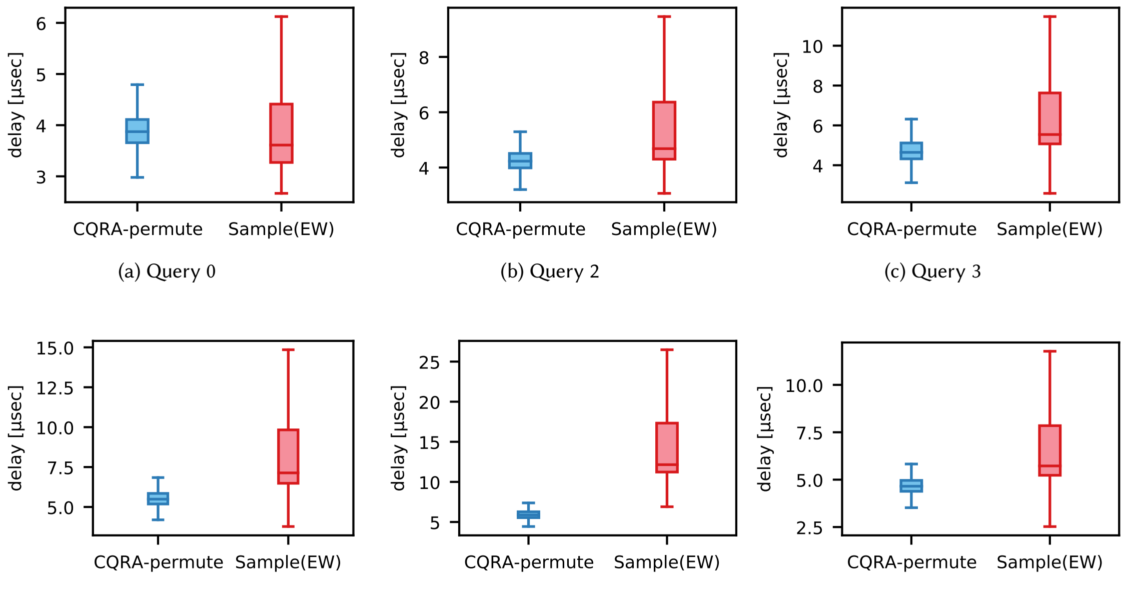
(d) Query 7 (e) Query 9 (f) Query 10

Fig. 3. The delay in a full enumeration.

in Figure 2 with a chart per query. The results indicate that, as k grows, the total enumeration time of Sample(EW) grows more rapidly in comparison to CQRA-permute. This can be attributed to the duplicate elimination, which was implemented in a straight-forward manner, as we described. The total time of CQRA-permute increases slower as it does not reject answers. Hence, Sample(EW) seems comparable for smaller k values, but is consistently outperformed by CQRA-permute for larger values of k. This is especially true when preprocessing time becomes negligible in comparison to the time it takes to enumerate k distinct answers. CQRA-permute performs better, relative to Sample(EW), on queries with more relations (Q_2 ,Q_7, Q_9) than ones with fewer relations.

Remark 7.1. *In Remarks 4.11, 5.8 and 5.7, we discussed an alternative way of transforming sampling with replacements to sampling without replacements that works similarly to what we do in our experiments with Sample(EW) for the first half of the answers, and then uses a different approach to handle the last half of the answers (were rejections are more likely). In our experiment, as evident by Figure 2, CQRA-permute consistently outperforms Sample(EW) even when producing only half of the answers. We conclude that this alternative approach will not bring practical gain, at least with the implementation of Sample(EW) that we use. Importantly, this implementation takes $O(\log n)$ time per sample, yet Chen and Yi [17] explain how it can be changed to obtain the guarantee of $O(1)$ time per sample (with replacements) using the* alias *method. Nevertheless, we are not aware of any published implementation that guarantees the $O(1)$ time bound. We leave it for future research to reimplement the sampling-with-replacement algorithm using the alias method and study its performance empirically.*

7.3.2 CQ delay. To examine the delay of CQRA-permute and Sample(EW), we record the delay of each answer and depict it in box-and-whisker plots. Each query has two box plots: enumeration of all answers (Figure 3) and enumeration of 50% of the answers (Figure 4). Outliers that fell outside the whiskers are not shown, since some are several orders



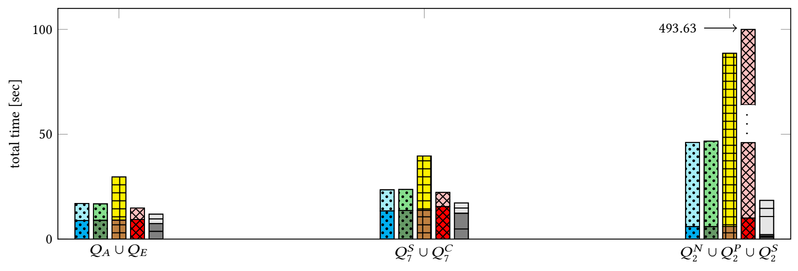
(a) Query 0 (b) Query 2 (c) Query 3

(d) Query 7 (e) Query 9 (f) Query 10

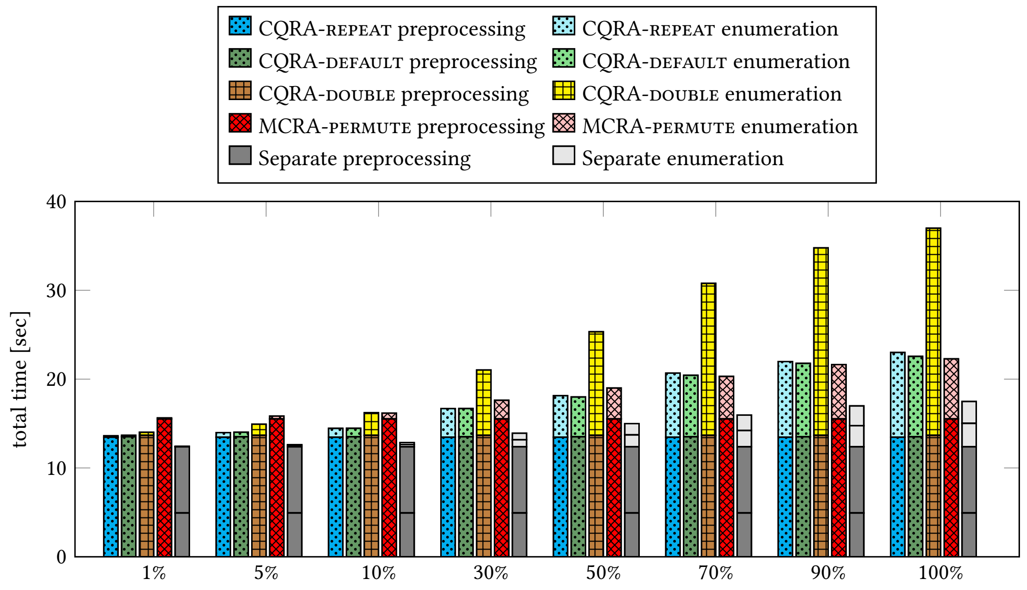
Fig. 4. The delay when enumerating 50% of the answers.

of magnitude larger than the median. However, we note that CQRA-PERMUTE consistently had less outliers than SAMPLE(EW), and these outliers had a lower mean and standard deviation. Further information regarding the results of this experiment, as well as the outliers dropped, is in the Appendix (Section A.3). We can see that in a full enumeration, CQRA-PERMUTE always shows a lower median value, smaller variation, and a smaller interquartile range (the difference between the value of the third quartile and that of the first quartile, which we denote by IQR). Smaller IQR and whiskers show that half of the delay samples fall within a smaller range, meaning that the delay is more stable and predictable. When enumerating 50% of the answers, the variation and IQR remain smaller across all queries. However, in Q_0 we see that SAMPLE(EW) actually exhibits a smaller median. In general, we noticed that CQRA-PERMUTE usually shows better results on larger queries in comparison to SAMPLE(EW). For instance, SAMPLE(EW) has a better median in Q_0 than in its larger counterpart Q_2. We conjecture this is because on smaller queries the impact of our algorithm's bookkeeping is more significant.

7.3.3 UCQ running time. In this section, we investigate empirically the total enumeration time of our suggested UCQ algorithms. We measured the running time of the different UCQ algorithms on three UCQs, and the results are displayed in Figures 5a and 5b. Figure 5a shows the total running time of a full enumeration in the three UCQs, while Figure 5b focuses on one UCQ, namely $Q_7^S \cup Q_7^C$, and measures the time of the different UCQ algorithms as it varies when producing a different portion of the answers (as was done in Section 7.3.1 for CQs). In both cases, we compare the different UCQ algorithms to each other and to the cumulative running time of CQRA-PERMUTE on the CQs comprising the union (the "Separate" alternative). We stress that running a random-permutation algorithm separately on the individual CQs is *not* an alternative to a random-permutation algorithm for UCQs—it produces duplicates and does not



(a) Total time of a full enumeration for different UCQs.



(b) Time of producing varying percentage of the answers for one UCQ.

Fig. 5. The total time of UCQ algorithms. To show the overhead of handling the union, we include in gray the separate executions of CQRA-PERMUTE for the two individual CQs (not an alternative to UCQ enumeration).

provide a uniformly random order. We perform this comparison to get a sense of the overhead of the UCQ algorithms over the CQ algorithm.

First, let us examine the preprocessing time as shown in 5a. The algorithms CQRA-PERMUTE, CQRA-DEFAULT and CQRA-DOUBLE rely on the same preprocessing, and so they require the same preprocessing time. The difference in preprocessing time between these algorithms and the CQ algorithm (Separate) is that in these three algorithms we also build an index that supports testing whether a given answer to one CQ is also an answer to another CQ. Meanwhile, the preprocessing of MCRA-PERMUTE adds to that of the other UCQ algorithms the need to preprocess CQs defined by the intersection of the CQs comprising the union. Hence, MCRA-PERMUTE always requires the largest preprocessing time. That said, in case we are interested in producing all answers, we should focus more on the enumeration phase, as the difference in the running time there is more significant.

The overhead of CQRA-repeat compared to its CQ counterpart (CQRA-permute) is mostly attributed to the mechanism that handles answers in the intersection of multiple CQs. In particular, a union of m CQs calls the inverted-access method $m-1$ times per answer. Figure 5a demonstrates that the slowdown between CQRA-permute and CQRA-repeat depends largely on the intersection size, as $Q_2^N \cup Q_2^P \cup Q_2^S$ has a large intersection and $Q_A \cup Q_E$ has no intersection at all. In general, two disjoint queries will be much faster than two identical queries because for the latter the algorithm will reject half of the answers on average.

In our experiments, the "timeout" for CQRA-default (Line 12 of Algorithm 9) was never reached, and so CQRA-repeat and CQRA-default behave identically. This is not a coincidence, since we chose the threshold γ specifically to ensure that it will be reached only with very low probability (as explained in Section 5.3). The choice between these two algorithms is therefore relevant for the theoretical guarantees rather than the expected practical behaviour. These two algorithms are faster in our experiments than CQRA-double, despite the latter offering better theoretical guarantees. More specifically, CQRA-double has roughly doubled the enumeration cost, which can be explained by the double sampling incurred in every iteration.

The difference in running time between MCRA-permute and CQRA-repeat depends largely on the number of CQs in the union. MCRA-permute seems comparable or even preferable to CQRA-repeat on queries that consist of a union of two CQs, but it does not scale well to additional CQs to the union. MCRA-permute performs poorly on $Q_2^N \cup Q_2^P \cup Q_2^S$, as its delay depends exponentially on the number of CQs in the union. When considering a union of two CQs, both algorithms benefit from a small number of answers in the intersection, but MCRA-permute maintains its lead. This can be seen in Figure 5a as $Q_A \cup Q_E$ is a disjoint union, unlike $Q_7^S \cup Q_7^C$. When the union is disjoint, MCRA-permute never calls the inverted access of the intersection (line 7 of Algorithm 12 is never reached). In CQRA-repeat, a disjoint union causes no rejections (Line 9 of Algorithm 8 is reached in every iteration). So, in both algorithms, a disjoint union causes a guaranteed logarithmic delay (not polylogarithmic and not only in expectation). However, CQRA-repeat still pays for testing membership in the other CQs as it does not know in advance that the union is disjoint.

Figure 5b shows the middle column of Figure 5a as it changes during the course of enumeration. It shows that the increase in total delay is rather steady in all UCQ algorithms, and that MCRA-permute starts being preferable over CQRA-repeat and CQRA-default when producing about 60% of the answers or more.

7.3.4 UCQ rejection rate. We now focus on the algorithms that use rejection, namely CQRA-repeat and CQRA-default, and inspect how much of their time they spend in iterations that lead to rejections compared to the amount of time that leads to accepted answers. Figure 6 shows how this time changes along the course of the full enumeration for a single UCQ. We can see that the time spent on rejections decays over the course of enumeration. The reason is that, as we explain next, the number of answers that belong to multiple CQs (shared answers) drops faster than that of non-shared answers, and so rejections become less likely. Let us imagine two sets, the shared answers and the non-shared answers, and inspect how they evolve over time. If an answer is reached through its owner, it is deleted everywhere, and so it will be removed from its corresponding set (this happens regardless of whether this answer is shared); however, if the answer is reached through a non-owner provider, it will become non-shared, so the number of shared answers will decrease while the number of non-shared answers will increase.

7.3.5 Conclusions. Our experimental study indicates that the merits of CQRA-permute are not only in its complexity and statistical guarantees—a fairly direct implementation features a significant improvement in practical performance compared to the state-of-the-art approach. The UCQ alternatives we inspected differ theoretically by expected vs. worst-case guarantee or a logarithmic factor. These differences are so minor in practice that they do not correspond to the

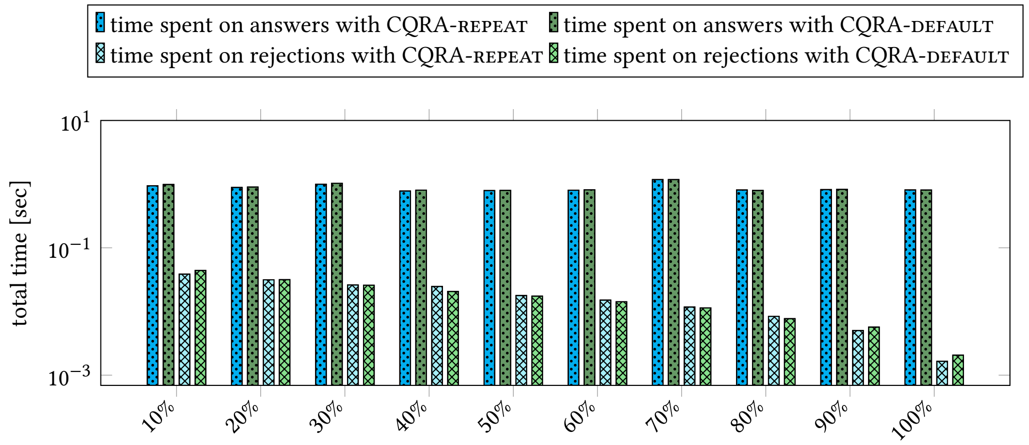


Fig. 6. Time spent on producing answers vs. time spent on rejections across a full enumeration of $Q_7^S \cup Q_7^C$ (log scale).

differences in performance observed in our experiments. Our experimental study shows that CQRA-REPEAT is usually the best choice, while CQRA-DEFAULT performs similarly, CQRA-DOUBLE performs worse, and MCRA-PERMUTE is comparable or slightly preferable for unions of two CQs but significantly worse for larger unions. The overhead of the UCQ algorithms we inspected, compared to CQs, is non-negligible. While this overhead is reasonable for the case of binary union, it is an important future challenge to reduce this overhead for larger unions.

8 CONCLUSIONS

We studied the problems of answering queries in a random permutation and via a random-access. We established that for CQs without self-joins it holds that $\mathsf{Enum}\langle\text{lin}, \log\rangle = \mathsf{RAccess}\langle\text{lin}, \log\rangle = \mathsf{REnum}\langle\text{lin}, \log\rangle$. We also studied the generalization to unions of free-connex CQs where, in contrast, we have $\mathsf{Enum}\langle\text{lin}, \log\rangle \neq \mathsf{RAccess}\langle\text{lin}, \log\rangle$ and random-access may be intractable even if tractable for each CQ in the union. (Here, tractability is with respect to the yardstick of a polylogarithmic time per answer after a linear-time preprocessing phase.) We then studied four alternatives for UCQs: (1) MCRA-PERMUTE uses the random-access approach for the restricted class of mc-UCQs and achieves guaranteed $\log^2$ delay; (2) CQRA-DOUBLE provides random permutation with log delay by relying on the assumption that the intersection of every subset of the CQs is free-connex; (3) CQRA-REPEAT finds a random permutation directly for any UCQ comprising of free-connex CQs and achieves log delay in expectation; and (4) CQRA-DEFAULT also provides random permutation for any UCQ comprising of free-connex CQs and achieves log delay in expectation, but now guarantees a worst-case of polylogarithmic delay in the cost of relaxing the uniform distribution to an *almost* uniform distribution. We described an implementation of our algorithms, and presented an experimental study that compares between the alternatives and shows that our algorithms outperform the sampling-with-rejection alternatives.

Concerning the restriction of being an mc-UCQ, we showed that it is guaranteed by a sufficient condition that involves the concept of β-acyclicity. A different yet highly restrictive sufficient condition is being a union comprising of the same CQ with different selection conditions (as in our experiments). In future work we intend to seek additional syntactic properties that ensure being an mc-UCQ.

For CQs, we achieved random permutation through random access, but we had no requirement regarding the specific answer order entailed by our random-access algorithm. Further investigation of the specific orders that can be efficiently achieved goes beyond the scope of this work, and indeed it was carried out following the publication of the conference version of this article. Carmeli et al. [14] characterized the orders that can be achieved with logarithmic access time after linear preprocessing. They explained that the order we achieve here is lexicographical, and built upon our random-access algorithm for CQs to design an improved algorithm that supports any tractable lexicographic order specified by the user.

We know that (under complexity assumptions) non-free-connex CQs do not have a random permutation algorithm with linear preprocessing and polylogarithmic delay. However, it would be interesting to understand more precisely the times required for guaranteeing enumeration in a random permutation, for example, by building on top of recent work on sampling cyclic CQs [18]. Regarding UCQs, it is an open problem to find which UCQs admit efficient fine-grained enumeration, even without order guarantees [13]. However, we do know that UCQs comprising of free-connex CQs admit efficient enumeration. This work opens the question of finding an exact characterisation for when such a UCQ admits random-access or random-permutation in polylogarithmic delay.

ACKNOWLEDGMENTS

Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project numbers 412400621; 414325841. The work of Nofar Carmeli was funded by the French government under the management of Agence Nationale de la Recherche as part of the “Investissements d’avenir” program, reference ANR-19-P3IA-0001 (PRAIRIE 3IA Institute).

REFERENCES

[1] A. Abboud and V. V. Williams. Popular conjectures imply strong lower bounds for dynamic problems. In *FOCS*, pages 434–443, 2014.
[2] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, pages 275–286. ACM Press, 1999.
[3] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.
[4] R. R. Amossen and R. Pagh. Faster join-projects and sparse matrix multiplications. In *ICDT*, pages 121–126, 2009.
[5] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.
[6] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *CSL*, pages 208–222. Springer, 2007.
[7] C. Berkholz, F. Gerhardt, and N. Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020.
[8] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering UCQs under updates and in the presence of integrity constraints. In *ICDT*, pages 1–19, 2018.
[9] J. Brault-Baron. *De la pertinence de l’énumération: complexité en logiques propositionnelle et du premier ordre.* PhD thesis, Université de Caen, 2013.
[10] J. Brault-Baron. Hypergraph acyclicity revisited. *ACM Comput. Surv.*, 49(3), dec 2016.
[11] F. Capelli and Y. Strozecki. Incremental delay enumeration: Space and time. *Discrete Applied Mathematics*, 268:179–190, 2019.
[12] N. Carmeli and M. Kröll. Enumeration complexity of conjunctive queries with functional dependencies. *Theory of Computing Systems*, pages 1–33, 2019.
[13] N. Carmeli and M. Kröll. On the enumeration complexity of unions of conjunctive queries. In *PODS*, PODS ’19, pages 134–148, New York, NY, USA, 2019. ACM.
[14] N. Carmeli, N. Tziavelis, W. Gatterbauer, B. Kimelfeld, and M. Riedewald. Tractable orders for direct access to ranked answers of conjunctive queries. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS 2021)*, pages 325–341, 2021.
[15] N. Carmeli, S. Zeevi, C. Berkholz, B. Kimelfeld, and N. Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration. In *PODS*, pages 393–409. ACM, 2020.
[16] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *SIGMOD*, pages 263–274. ACM Press, 1999.
[17] Y. Chen and K. Yi. Random sampling and size estimation over cyclic joins. In *23rd International Conference on Database Theory (ICDT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.
[18] Y. Chen and K. Yi. Random sampling and size estimation over cyclic joins. In C. Lutz and J. C. Jung, editors, *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*, volume 155 of *LIPIcs*, pages 7:1–7:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.
[19] S. Deep, X. Hu, and P. Koutris. Fast join project query evaluation using matrix multiplication. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland,*

OR, USA], June 14-19, 2020, pages 1213–1223. ACM, 2020.
[20] S. Deep and P. Koutris. Ranked enumeration of conjunctive query results. *CoRR*, abs/1902.02698, 2019.
[21] A. Durand and Y. Strozecki. Enumeration complexity of logical query problems with second-order variables. In *CSL*, volume 12 of *LIPIcs*, pages 189–202, 2011.
[22] R. Durstenfeld. Algorithm 235: Random permutation. *C. ACM*, 7(7):420, 1964.
[23] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.
[24] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.
[25] F. L. Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303, 2014.
[26] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298. ACM Press, 1999.
[27] M. Idris, M. Ugarte, and S. Vansummeren. The Dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *SIGMOD*, pages 1259–1274. ACM, 2017.
[28] R. M. Karp, M. Luby, and N. Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429 – 448, 1989.
[29] M. A. Khamis, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. Learning models over relational data using sparse tensors and functional dependencies. *ACM Trans. Database Syst.*, 45(2):7:1–7:66, 2020.
[30] B. Kimelfeld and C. Ré. A relational framework for classifier engineering. *SIGMOD Rec.*, 47(1):6–13, 2018.
[31] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join and XDB: online aggregation via random walks. *ACM Trans. Database Syst.*, 44(1):2:1–2:41, 2019.
[32] A. Lincoln, V. V. Williams, and R. Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *Proc. SODA*, pages 1236–1252. SIAM, 2018.
[33] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP: Volume 1: Design & Efficiency.* Benjamin-Cummings, 1991.
[34] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015.
[35] R. Pichler and S. Skritek. Tractable counting of the answers to conjunctive queries. *J. Comput. Syst. Sci.*, 79(6):984–1001, sep 2013.
[36] W. N. Street and Y. Kim. A streaming ensemble algorithm (SEA) for large-scale classification. In *KDD*, pages 377–382. ACM, 2001.
[37] N. Tziavelis, D. Ajwani, W. Gatterbauer, M. Riedewald, and X. Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *CoRR*, abs/1911.05582, 2019.
[38] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977.
[39] V. V. Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898, 2012.
[40] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, VLDB '81, pages 82–94. VLDB Endowment, 1981.
[41] R. Yuster and U. Zwick. Fast sparse matrix multiplication. *ACM Trans. Algorithms*, 1(1):2–13, 2005.
[42] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In *SIGMOD*, pages 1525–1539. ACM, 2018.

APPENDIX

A ADDITIONS TO SECTION 7

A.1 Queries

This section describes the CQs and UCQs used in Section 7. The keyword "DISTINCT" was added to all SQL code to emphasize the fact that we discuss set semantics evaluation. The keyword does not affect the actual result since the queries have no projections.

The following six queries are those used to compare CQRA-permute to Sample(EW).

Query Q_0: a chain join between the relations PartSupp, Supplier, Nation, and Region. It returns the suppliers that sell products (parts) along with their nation and region.

```
SELECT DISTINCT r_regionkey, n_nationkey,
                s_suppkey, ps_partkey
  FROM region, nation, supplier, partsupp
 WHERE r_regionkey = n_regionkey AND
       n_nationkey = s_nationkey AND
       s_suppkey = ps_suppkey
```

Query Q_2: similar to Q_0, except for the addition of the *PART* table with `ps_partkey = p_partkey`.

```
SELECT DISTINCT r_regionkey, n_nationkey,
                s_suppkey, ps_partkey
  FROM region, nation, supplier, partsupp, part
 WHERE r_regionkey = n_regionkey AND
       n_nationkey = s_nationkey AND
       s_suppkey = ps_suppkey AND
       ps_partkey = p_partkey
```

Query Q_3: the join of three relations: Customer, LineItems and Orders. We added the three attributes `l_partkey`, `l_suppkey`, and `l_linenumber` to the output to ensure equivalence between set semantics and bag semantics (see Section 7.2).

```
SELECT DISTINCT o_orderkey, c_custkey, l_partkey,
                l_suppkey, l_linenumber
  FROM customer, orders, lineitems
 WHERE c_custkey = o_custkey AND
       o_orderkey = l_orderkey;
```

Query Q_7: similar to Q_3, except it also joins Supplier and Nation for the customer and the supplier.

```
SELECT DISTINCT o_orderkey, c_custkey,
                n1.n_nationkey, s_suppkey,
                l_partkey, l_linenumber,
                n2.n_nationkey
  FROM supplier, lineitem, orders, customer,
       nation n1, nation n2
 WHERE s_suppkey = l_suppkey AND
       o_orderkey = l_orderkey AND
       c_custkey = o_custkey AND
       s_nationkey = n1.n_nationkey AND
       c_nationkey = n2.n_nationkey;
```

Query Q_9: the join of the relations Nation, Supplier, LineItem, PartSupp, Orders, and Part. As in Q_3, we added the attributes `l_partkey`, `l_suppkey`, and `l_linenumber` to the output to ensure an equivalence between bag and set semantics (see Section 7.2).

```
SELECT DISTINCT n_nationkey, s_suppkey,
                o_orderkey, l_linenumber, p_partkey
  FROM nation, supplier, lineitem,
       partsupp, orders, part
 WHERE n_nationkey = s_nationkey AND
       s_suppkey = l_suppkey AND
       s_suppkey = ps_suppkey AND
       o_orderkey = l_orderkey AND
       l_partkey = p_partkey AND
       p_partkey = ps_partkey;
```

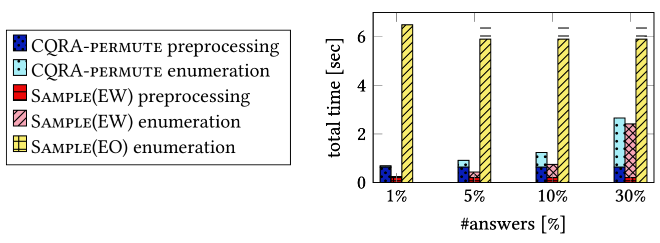
Query Q_{10}: similar to Q_3, except it also joins Nation.

```
SELECT DISTINCT o_orderkey, c_custkey, l_partkey,
                l_suppkey, l_linenumber, n_nationkey
  FROM lineitem, orders, customer, nation
 WHERE o_orderkey = l_orderkey AND
       c_custkey = o_custkey AND
       c_nationkey = n_nationkey;
```

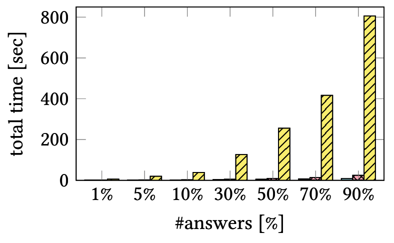
The UCQ experiments use $Q_7^S \cup Q_7^C$, $Q_2^N \cup Q_2^P \cup Q_2^S$, and $Q_A \cup Q_E$, with the following CQs:

Query Q_A: the query deals with orders whose suppliers are from the United States of America. That is done by applying a condition to a full chain join of the tables Order, LineItem, Supplier, Nation, and Region.

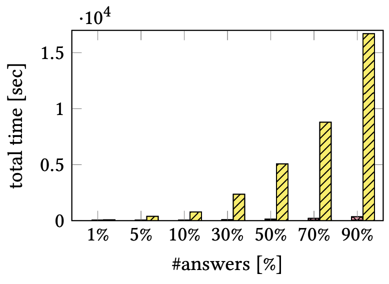
```
SELECT DISTINCT o_orderkey, s_suppkey,
                n_nationkey, r_regionkey, r_name
  FROM orders, lineitem, supplier, nation, region
```



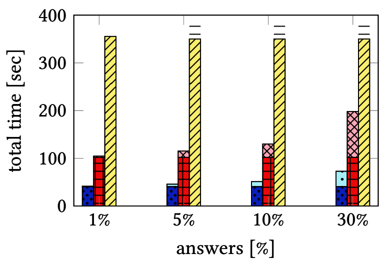
(a) Query Q_0



(b) Query Q_2



(c) Query Q_7



(d) Query Q_9

Fig. 7. Total enumeration time of CQs when requesting different percentages of answers. In each bar, the bottom (darker) part refers to the preprocessing phase and the top (lighter) part to the enumeration phase.

```
WHERE o_orderkey = l_orderkey AND
      l_suppkey = s_suppkey AND
      s_nationkey = n_nationkey AND
      n_regionkey = r_regionkey AND
      n_nationkey = 24
```

Query Q_E: similar to Q_A, except for the demand that the supplier be from the United Kingdom. Meaning, it has the same SQL expression as Q_A, but the constant 24 (United States) is replaced by 23 (United Kingdom).

Query Q_7^S: similar to Q_7, except for the addition of the constraint `n1.n_name = "UNITED STATES"`. Meaning, the output should only include orders where the supplier is American.

Query Q_7^C: similar to Q_7, except we replace `n1.n_name = "UNITED STATES"` with `n2.n_name = "UNITED STATES"`. Meaning, demanding the customer is American (instead of the supplier being American).

Query Q_2^N: similar to Q_2, except for the addition of the constraint `n_nationkey = 0`. Meaning, the supplier must be from the first country in the database.

Query Q_2^P: similar to Q_2, except for the addition of the constraint `n_partkey mod 2 = 0`. Meaning, the part identifier must be even.

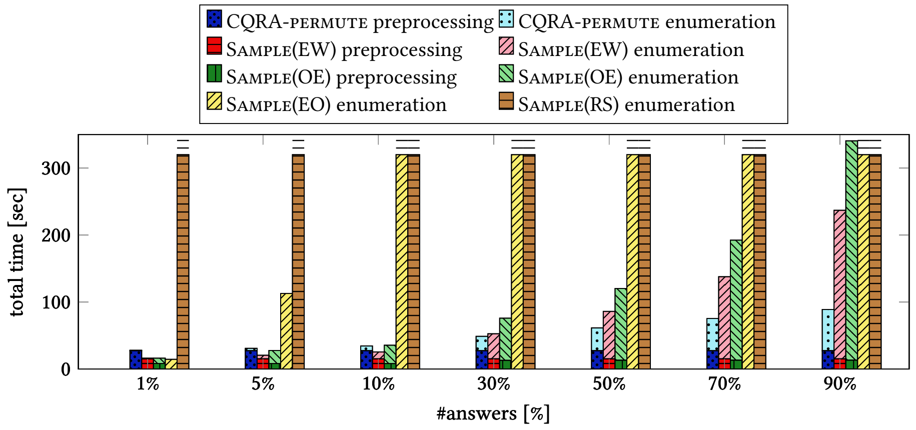


Fig. 8. Time spent on producing varying percentage of the answers to Q_3. In the bars that contain two colors, the bottom (darker) part refers to the preprocessing phase and the top (lighter) part to the enumeration phase.

Query Q_2^S: similar to Q_2, except for the addition of the constraint `n_suppkey mod 2 = 0`. Meaning, the supplier identifier must be even.

A.2 Additional methods by Zhao et al.

As mentioned in Section 7.2, Zhao et al. [42] discuss 4 different ways of initializing their sampling algorithm, denoted as *RS*, *EO*, *OE*, and *EW*. In the implementation of Zhao et al. , *EW* and *EO* were implemented for every query. In addition, there is an implementation of RS and OE for Q_3. Here we review *EO*, *OE*, and *RS* (in Sections A.2.1, A.2.2, and A.2.3 respectively) in order to explain our comparison to SAMPLE(EW) alone.

A.2.1 EO. As SAMPLE(EO) may reject, it possesses a much longer sampling time (as evident by our experiments). Figure 7 repeats the experiment made in Section 7.3.1 (depicted in Figure 2) with the addition of SAMPLE(EO). We omit the SAMPLE(EO) preprocessing, as Zhao et al. [42] did in their paper, since this only consists of computing basic statistical information. Note also that SAMPLE(EO) underperforms compared to SAMPLE(EW) regardless. When running SAMPLE(EO), we used a timeout and halted when it took longer than 100 times the sampling time of its *EW* counterpart. We omit Q_{10}, as SAMPLE(EO) did not produce 1% of the answers within the time limit. Figure 7 shows that with the exception of Q_3 at 1%, *EO* is significantly slower than both CQRA-PERMUTE and SAMPLE(EW).

A.2.2 OE. Out of our six queries, SAMPLE(OE) was implemented on Q_3 alone. Figure 8 shows the results of Section 7.3.1 with SAMPLE(OE) added. In our experiments with Q_3, SAMPLE(EW) has always out-preformed SAMPLE(OE).

A.2.3 RS. SAMPLE(RS) was also implemented only on Q_3. SAMPLE(RS) was unable to produce a sample of 1% of the answers to Q_3 in less than an hour. It took SAMPLE(RS) about 6.8 seconds to gather a sample of 100000 distinct answers, which is roughly 0.33% of all answers. Therefore, SAMPLE(RS) would be slower than SAMPLE(EW) even if it were to proceed and sample 1% with no deterioration due to repeating samples. In Figure 8, we omit the SAMPLE(RS) preprocessing, as Zhao et al. [42] did in their work, and as it underperforms compared to SAMPLE(EW) regardless.

algorithm	query	mean (μ)	SD (σ)	outliers [%]
CQRA-permute	Q_0	3.964625	26.77761	2.3527
Sample(EW)	Q_0	3.965105	155.8571	6.6181
CQRA-permute	Q_2	4.35985	29.14075	3.38665
Sample(EW)	Q_2	5.455966	136.3711	6.1348
CQRA-permute	Q_3	4.443927	198.3520	3.07209
Sample(EW)	Q_3	6.599028	519.8907	6.17242
CQRA-permute	Q_7	5.342141	191.1972	2.91181
Sample(EW)	Q_7	8.471535	534.6212	7.12741
CQRA-permute	Q_9	5.664519	200.8911	2.8047
Sample(EW)	Q_9	14.90882	537.2356	8.26721
CQRA-permute	Q_{10}	4.652109	195.7905	2.89414
Sample(EW)	Q_{10}	6.866843	519.3847	6.330277

algorithm	query	mean (μ)	SD (σ)	outliers [%]
CQRA-permute	Q_0	3.891264	18.9752	2.685475
Sample(EW)	Q_0	20.28385	2848.952	12.0188
CQRA-permute	Q_2	4.319361	20.74831	3.662525
Sample(EW)	Q_2	38.02335	4815.667	12.412425
CQRA-permute	Q_3	4.347431	140.2731	3.65655
Sample(EW)	Q_3	49.84528	20863.52	12.3539
CQRA-permute	Q_7	5.254392	135.2184	3.47616
Sample(EW)	Q_7	72.04367	30804.44	12.50615
CQRA-permute	Q_9	5.57028	142.0814	3.2938
Sample(EW)	Q_9	141.239102	56781.80	12.75
CQRA-permute	Q_{10}	4.564015	138.4678	3.43488
Sample(EW)	Q_{10}	49.30842	11648.17	12.4268

Fig. 9. The mean and standard deviation of the delay during *(a)* an enumeration of 50% of answers using each algorithm (left), *and (b)* a full enumeration using each algorithm (right).

A.3 More on the CQ delay experiment

This section gives further information regarding the experiment described in Section 7.3.2. Figure 9 shows the mean, standard deviation (SD) and number of delay samples that counted as outliers during the enumeration of 50% of the answers (on the top of Figure 9) and a full enumeration (on the bottom of Figure 9). We see that CQRA-permute always possesses a smaller mean than Sample(EW), sometimes by an order of magnitude. We also see that CQRA-permute always has considerably lower standard deviation than that of Sample(EW). That holds even when the two are close in median as is the case with Q_0. Finally, the number of outliers in CQRA-permute boxplots is also consistently smaller. The smaller number of outliers and lower standard deviation indicates the predictability of the delay, as it does not grow rapidly.