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

Nofar Carmeli
Technion
snofca@cs.technion.ac.il

Shai Zeevi
Technion
shai.zeevi@cs.technion.ac.il

Christoph Berkholz
Humboldt-Universität zu Berlin
berkholz@informatik.hu-berlin.de

Benny Kimelfeld
Technion
bennyk@cs.technion.ac.il

Nicole Schweikardt
Humboldt-Universität zu Berlin
schweikn@informatik.hu-berlin.de

ABSTRACT

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 paper, 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 valuable manner. An even more demanding task is that of a "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. For such UCQs we devise a random-order enumeration whose delay is logarithmic in expectation. We also identify a subclass of UCQs for which we can provide random access with polylogarithmic access time. 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).*

KEYWORDS

unions of conjunctive queries, enumeration, complexity

ACM Reference Format:
Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. 2020. Answering (Unions of) Conjunctive Queries using Random Access and Random-Order Enumeration. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3375395.3387662

1 INTRODUCTION

In the effort of reducing the computational cost of database queries to the very least possible, recent years have seen a 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 [8], 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) [7, 11] and the presence of constraints [7, 10].

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 such as machine learning, summarisation, and search. The intermediate results can be used to save time by invoking the next-step processing (e.g., as in streaming learning algorithms [28]), computing approximate summaries that improve in time (e.g., as in online aggregation [20, 24]), and presenting the first pages of search results (e.g., as in keyword search over structured data [19, 21]). Yet, at least the latter two applications make the implicit assumption that the collection of intermediate results is a representative of the entire space of answers. In contrast, the aforementioned constant-delay algorithms enumerate in an order that is a merely an artifact of the tree selected to utilize free-connexity, and hence, intermediate answers may feature

PODS'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-7108-7/20/06…$15.00
https://doi.org/10.1145/3375395.3387662

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 order [14, 29].

Yet, to be a statistically meaningful representation of the space of answers, the enumeration order needs to be provably random. In this paper, we investigate the task of enumerating answers in a uniformly random order. To be more precise, the goal is to enumerate the answers without repetitions, and the output induces a uniform distribution over the space of permutations over the answer set. We refer to this task as *random permutation*. Similarly to the recent work on ranked enumeration [14, 29], 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, 12] and recently gained a renewed attention [33].

One way of achieving a random permutation is via *random access*—a structure that is tied to some enumeration order and, given a position i, returns the ith answer in the order. Random access, in general, can be seen as an efficient way of accessing the query answers as if they are already computed and stored in an array. One could imagine additional uses for an efficient random-access algorithm. For example, a server implementing a random-access algorithm can provide answers to concurrent users in a stateless manner: the users ask for a range of indices, and the server does not need to keep track of the answers already sent to each user. To satisfy our target of an efficient permutation, we need a random-access structure that can be constructed in linear time (preprocessing) and supports answer retrieval (given i) in polylogarithmic time. We show that, having this structure at hand, we can use the Fisher-Yates shuffle [16] 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 for free-connex CQs has been established by Brault-Baron [8]. Here, we devise our own random-access algorithm for free-connex CQs that is simpler and better lends itself to a practical implementation. Moreover, we design our algorithm in such a way that it is accompanied by an *inverted access* that is needed for our later results on UCQs.

Note that an alternative approach to our algorithm for random permutation would be to repeatedly sample tuples uniformly with replacement (using known techniques, e.g., [33]) and reject tuples that have already been produced. In expectation, this alternative would have the same *total* time as our algorithm, namely $O(M \log M)$ where M is the number of answers, due to the *coupon collector* argument and the fact that the delay of our algorithm is $O(\log M)$. However, this alternative approach would *not* have the strict (and deterministic) guarantee that we provide on the delay, and would not even be counted as an enumeration algorithm with a sublinear delay.

The tractability of enumeration generalizes from free-connex CQs to *unions* of free-connex CQs [7, 11]. Interestingly, this is no longer the case for random access! The reason is as follows. An efficient random access 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 two questions:

(1) Can we identify a nontrivial class of UCQs with an efficient random access?
(2) Can we get an efficient random permutation for unions of free-connex CQs, without requiring a random access?

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. As for the second question, we show that the answer is positive under the following weakening of the delay guarantee: there is a random permutation where *each delay* is a geometric random variable with a logarithmic mean. In particular, each delay is logarithmic in expectation.

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 [33], which is designed to produce a uniform sample with replacement, and then remove duplicates as they are detected. The experiments show that our algorithms are not only featuring 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.

The paper 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 presents our experimental study. Due to space restrictions, some details had to be deferred to an appendix.

2 PRELIMINARIES

In this section, we provide basic definitions and notation that we will use throughout this paper. 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* **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 **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 **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 $Q(\vec{x})$:- $R_1(\vec{t}_1), \ldots, R_n(\vec{t}_n)$ where each R_i is a relation symbol of 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 $\vec{t}_1, \ldots, \vec{t}_n$. We call $Q(\vec{x})$ the *head* of Q, and $R_1(\vec{t}_1), \ldots, R_n(\vec{t}_n)$ the *body* of Q. Each $R_i(\vec{t}_i)$ is an *atom* of Q. We use *Vars*(Q) and *Vars*(α) 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 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. 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 $Q(\vec{x})$:- $\alpha_1(\vec{x}, \vec{y}), \ldots, \alpha_k(\vec{x}, \bar{y})$ we associate a hypergraph $\mathcal{H}_Q$ where the nodes are the variables in *Vars*(Q), and the edges are $E = \{e_1, ..., e_k\}$ such that $e_i = Vars(\alpha_i)$. 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 acyclic. That is, there exists a tree T (called a *join-tree* of Q) such that $nodes(T) = edges(\mathcal{H}_Q)$, and for every $v \in nodes(\mathcal{H}_Q)$, 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 $Q_1(\vec{x}) \cup \cdots \cup Q_m(\vec{x})$, where every Q_i is a CQ with the sequence $\vec{x}$ of head variables. The set of answers to $Q_1(\vec{x}) \cup \cdots \cup Q_m(\vec{x})$ over a database D is, naturally, the union $Q_1(D) \cup \cdots \cup Q_m(D)$.

Computation Model. 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 time between printing any two answers during the enumeration phase is called *delay*, and is denoted by t_d.

In this paper, 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, 17, 22]. 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.

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 (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 running time for this problem is $O(m^{4/3})$ [4], which remains true even if the matrix multiplication exponent[1] 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 an k-uniform hypergraph with m hyperedges. This hypothesis is implied by the (l, k)-HYPERCLIQUE conjecture proposed in [25].

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

3 ENUMERATION CLASSES

In this section, we define three 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 where every emission is done uniformly at random. That is, at every emission, every until then not yet emitted tuple has equal probability of being emitted. As a result,

[1] 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 $\omega = 2$ [18, 31].

if $|P(I)| = n$, every ordering of the answers $P(I)$ 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.*

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

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

Fact 3.5. *By definition,* $\mathsf{RAccess}\langle \text{lin}, d\rangle \subseteq \mathsf{Enum}\langle \text{lin}, d\rangle$ *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* [16], so that it works with constant delay after constant preprocessing time. The original version of the Fisher-Yates Shuffle (also known as *Knuth Shuffle*) [16] 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 $0, \ldots, n-1$. Then, at each step i, it chooses a random index, j, greater than or equal to i and swaps the chosen cell with the ith cell. At the end of this procedure, the array contains a random permutation. Proposition 3.6 describes an adaptation of this procedure that runs with constant delay and constant preprocessing time in the RAM model.

Algorithm 1 Random Permutation

```
procedure SHUFFLE(n)
    assume a[0], ..., a[n−1] are uninitialized
    for i in 0, . . . , n−1 do
        choose j uniformly from i, . . . , n−1
        if a[i] is uninitialized then
            a[i] = i
        if a[j] is uninitialized then
            a[j] = j
        swap a[i] and a[j] ; output a[i]
```

Proposition 3.6. *A random permutation of $0, \ldots, n-1$ 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.[2] 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 3.7. *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 $c \geq 1$.*

[2] To keep track of the array positions that are still uninitialized, one can use standard methods for *lazy array initialization* [26].

Proof. Let P be an enumeration problem in 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 bound error. We can use this to do a binary search for the number of answers using $O(\log(|P(I)|))$ calls to $\mathcal{A}$'s access procedure. Since $|P(I)|$ is polynomial in the size of the input, $\log(|P(I)|) = O(\log(|I|))$. Each access costs time $O(\log^c(|I|))$. In total, the number $|P(I)|$ is thus computed in time $O(\log^{c+1}(|I|))$, which still is in $O(|I|)$.

During the enumeration phase, we use Proposition 3.6 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 RANDOM-ACCESS FOR CQS

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

Theorem 4.1 ([6, 8]). *Let Q be a CQ. If Q is free-connex, then it is in* Enum$\langle\text{lin}, \text{const}\rangle$*. Otherwise, if it is also self-join-free, then it is not in* 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 [8]. Therefore, Q is not in 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 Enum$\langle\text{lin}, \log^c\rangle$, then any two Boolean matrices of size $n \times n$ can be multiplied in $O(m_1 + m_2 + m_3 \cdot \log^c(n))$ time, where m_1, m_2, and m_3 are the number of non-zero entries in A, B, and AB, respectively. This contradicts sparse-BMM.

An implication of Theorem 4.1 is that free-connex CQs can be answered with logarithmic delay. Since Brault-Baron [8] 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 RAccess$\langle\text{lin}, \log\rangle$. According to Theorem 3.7, this also shows the tractability of a random-order enumeration, that is, membership in REnum$\langle\text{lin}, \log\rangle$.

In this section, we present a random-access algorithm for free-connex CQs that, compared to Brault-Baron [8], is simpler and better lends itself to a practical implementation. In addition, we devise the algorithm in such a way that it is accompanied by an *inverted-access* that is needed for our results on UCQs in Section 5. An inverted-access I_A is an enhancement of a random-access algorithm A with the inverse operation: given an element e, the inverted-access returns $I_A[e] = j$ such that $A[j] = e$. That is, the jth answer in the random-access is e. If e is not an answer, then the inverted-access indicates so by returning "not-an-answer."

To proceed, we use the following folklore result.

Algorithm 2 Preprocessing

procedure Preprocessing(D, Q)
  for R in leaf-to-root order **do**
    Partition R to buckets according to pAtts_R
    for bucket B in R **do**
      for tuple t in B **do**
        if R is a leaf **then**
          $\text{w}(t) = 1$
        else
          let C be the children of R
          $\text{w}(t) = \prod_{S \in C} \text{w}(\text{bucket}[S, t])$
        let P be the tuples preceding t in B
        $\text{startIndex}(t) = \sum_{s \in P} \text{w}(s)$
      $\text{w}(B) = \sum_{t \in B} \text{w}(t)$

Proposition 4.2. *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., [22, 27]). 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 [32] for acyclic join queries. Then, we utilize the fact that Q is *free-connex*, which enables us to drop some atoms and attributes that correspond to quantified variables and be left with an acyclic CQ that contains exactly the free-variables. This leaves us with a full acyclic join that has the same answers as the original free-connex CQ.

So, it is left to design a random-access algorithm for full acyclic CQs. We do so in the remainder of this section. Algorithm 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 pAtts_R the attributes that appear both in R and in its parent. If R is the root, then $\text{pAtts}_R = \emptyset$. Given a relation R and an assignment a, we denote by $\text{bucket}[S, a]$ all tuples in S that agree with a over the attributes that S and a have in common. 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 (intuitively this is $S \ltimes a$).

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 $\text{w}(t)$ for each tuple t. This weight represents the number of different answers this tuple agrees with when only joining the relations of the subtree rooted in the current relation. The weight is computed in a leaf-to-root order, where tuples of a leaf relation have weight 1. The weight of a tuple t in a non-leaf relation R is determined by the product of the weights of the corresponding tuples in the children's relations. These corresponding tuples are the ones that agree with t on the attributes that R shares with its child. The weight of each bucket is the sum of the weights of the tuples it contains. In addition, we

Algorithm 3 Random-Access

```
1:  procedure Access(j)
2:      if j ≥ w(bucket[root, ∅]) then
3:          return out-of-bound
4:      else
5:          answer = ∅
6:          SubtreeAccess(bucket[root, ∅], j)
7:          return answer

8:  procedure SubtreeAccess(B ⊆ R, j)
9:      find t ∈ B s.t. startIndex(t) ≤ j < startIndex(t+1)
10:     answer = answer ∪ {Atts_R → Atts_R(t)}
11:     let R_1, ..., R_m be the children of R
12:     j_1, ..., j_m = SplitIndex(j − startIndex(t),
13:                     w(bucket[R_1, t]), ..., w(bucket[R_m, t]))
14:     for i in 1, ..., m do
15:         SubtreeAccess(bucket[R_i, t], j_i)
```

Algorithm 4 Inverted-Access

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

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

assign each tuple t with an index range that starts with startIndex(t) and ends with the startIndex of the following tuple in the bucket (or the total weight of the bucket if this is the last tuple). This represents a partition of the indices from 0 to the bucket weight, such that the length of the range of each tuple is equal to its weight. At the end of preprocessing, the root relation has one bucket (since $\text{pAtts}_{root} = \emptyset$), and the weight of this bucket represents the number of answers to the query.

The random-access is done recursively in a root-to-leaf order: we start from the single bucket at the root. At each step we find the tuple t in the current relation that holds the required index in its range (we denote by t+1 the tuple that follows t in the bucket). Then, we assign the rest of the search to the children of the current relation, restricted to the bucket that corresponds to t. Finding t can be done in logarithmic time using binary search [3]. The remaining index $j' = j -$ startIndex(t) is split into search tasks for the children using the method SplitIndex. The split can be seen as representing j' in a mixed radix numeral system where the units are the bucket weights. In other words, it is done in the same way as an index is split in standard multidimensional arrays: if the last bucket is of weight m, its index is j' mod m, and the other buckets recursively split between them the index $\lfloor \frac{j'}{m} \rfloor$.

Algorithm 4 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, CombineIndex$(w_1, j_1, \dots, w_m, j_m)$ is given by $j_m + w_m \cdot$ CombineIndex$(w_1, j_1, \dots, w_{m-1}, j_{m-1})$ with CombineIndex$()=0$.

Line 4 can be supported in constant time after an appropriate indexing of the buckets at preprocessing (in our RAM model). Since Algorithm 4 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).

[3] In fact, it seems conceivable that this search can be improved to $O(\log\log(|I|))$ using a van Emde Boas tree.

The next theorem, parts of which are already given in [8], summarizes the algorithms presented so far.

Theorem 4.3. *Given a free-connex CQ Q and a database D, it is possible to build in linear time a data structure that allows to output the count $|Q(D)|$ in constant time and provides random-access in logarithmic time, and inverted-access in constant time.*

Example 4.4. *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. The following is an example of an input database for such a query and the computed information available at the end of preprocessing. Here, the startIndex *value is denoted s.*

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

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

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

Calling Access(13) *finds $(a_2, b_2, c_1) \in R_1$. Then, the remaining $13 - 8 = 5$ is split to 5 mod 3 = 2 in the top bucket of R_3 and $\lfloor \frac{5}{3} \rfloor = 1$ in the bottom bucket of R_2. These in turn find 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 R_3 returns* startIndex$(c_1, e_3) = 2$ *from a bucket of weight* 3. *The call for* CombineIndex(2, 1, 3, 2) *returns $2 + 3 \cdot 1 = 5$, and the result is $8 + 5 = 13$.* □

Theorem 4.3 along with Theorem 3.7 implies that the dichotomy of Theorem 4.1 extends to the problems REnum⟨lin, log⟩ and RAccess⟨lin, log⟩. This also means that for self-join-free CQs, the classes of efficient random-access, random-permutation and enumeration collapse. This is summarized by the next corollary.

Corollary 4.5. *For every CQ Q, the following holds: If Q is free-connex, then Q is in each of* RAccess⟨lin, log⟩, REnum⟨lin, log⟩

and Enum⟨lin, log⟩*. If Q is self-join-free and not free-connex, then it is not in any of* RAccess⟨lin, polylog⟩*,* REnum⟨lin, polylog⟩*, and* Enum⟨lin, polylog⟩ *assuming* SPARSE-BMM*,* TRIANGLE*, and* HYPERCLIQUE*.*

5 UNIONS OF CQS

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

If several CQs are in Enum⟨lin, d⟩, for some d, then their union can also be enumerated within the same time bounds [11, 15]. Since our goal is to answer queries in random order, a natural question arises: does the same apply to queries in RAccess⟨lin, d⟩ and REnum⟨lin, d⟩? We show that it does not apply to CQs in RAccess⟨lin, d⟩. This means that for UCQs we cannot 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 RAccess⟨lin, lin⟩ under TRIANGLE.

EXAMPLE 5.1. *Consider the CQs $Q_1(x, y, z)$:- $R(x, y), S(y, z)$ and $Q_2(x, y, z)$:- $S(y, z), T(x, z)$. 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 4.3. 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 $Q_\cup \in$ RAccess⟨lin, lin⟩. 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" $Q_\cap(x, y, z)$:- $R(x, y), S(y, z), T(x, z)$ 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* RAccess⟨lin, lin⟩*.*

Example 5.1 shows that (assuming TRIANGLE) RAccess⟨lin, log⟩ is not closed under union. It also shows that, when considering UCQs, we have that Enum⟨lin, const⟩ $\not\subseteq$ RAccess⟨lin, lin⟩. In particular, this means that Enum⟨lin, log⟩ $\neq$ RAccess⟨lin, log⟩, which is not the case when only considering CQs. In Section 5.2, we devise a sufficient condition for UCQs to have a RAccess⟨lin, polylog⟩ computation. In Section 5.1, we show that if we relax the bound to logarithmic time *in expectation*, we can enumerate in a random-order any union comprised of free-connex CQs.

5.1 Random-Permutation with Expected Logarithmic Delay

In order to provide a random-permutation algorithm for UCQs, we start by devising a general algorithm for the union of sets, and then show how it can be applied for UCQs. The sets are assumed to have 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 [23] extended by tuple deletions that allow for sampling without repetitions. We prove the following lemma.

Algorithm 5 Random-Order Enumeration of $S_1 \cup \cdots \cup S_k$

```
1: while Σ_{j=1}^{k} S_j.COUNT() > 0 do
2:     chosen = choose i with probability S_i.COUNT() / Σ_{j=1}^{k} S_j.COUNT()
3:     element = S_chosen.SAMPLE()
4:     providers = {S_j | S_j.TEST(element) = True}
5:     owner = min{j | S_j ∈ providers}
6:     for S_j ∈ providers \ {S_owner} do
7:         S_j.DELETE(element)
8:     if S_owner = S_chosen then
9:         S_chosen.DELETE(element) ; output element
```

LEMMA 5.2. *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 5 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 5 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. 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 5 performs is bound 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 5.2.

In order to use Algorithm 5, the sets in question need to support counting, sampling, testing and deletion. We next show how to support these operations using the shuffle mechanism provided in Algorithm 1, assuming that the sets in question support efficient counting, random-access and inverted-access. Then, we will be able use this algorithm to answer UCQs.

We describe the construction of the data structure. First, we count the number of answers n. 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 $a[0], \dots, a[i-1]$ represent the indices of the deleted elements, while $a[i], \dots, a[n-1]$ hold the indices that remain in the set. We also use a reverse index b: whenever we set $a[i] = j$, we also set $b[j] = i$. Conceptually, a and b start initialized with $a[j] = b[j] = j$ and $i = 0$. Practically, the arrays can be implemented as lookup tables as in Algorithm 1. When *sampling*, we generate a uniformly random number $k \in \{i, \dots, n-1\}$. We then return element number $a[k]$ using the random-access routine. When *testing* membership, we call the inverted-access routine and return "True" iff we obtain a valid index. When *deleting*, we use the inverted-access routine to find the index m of the item to be deleted. We then find an index k such that $a[k] = m$, swap $a[k]$ with $a[i]$, and increase i by one. In order to efficiently find k, we use the reverse index b. When *counting*, we return $n-i$. The correctness of these procedures follows along the same lines of that of Algorithm 1. This proves the following lemma.

Lemma 5.3. *If an enumeration problem supports 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, we can apply this result to UCQs. Combining Theorem 4.3 with Lemma 5.2 and Lemma 5.3, we have an algorithm for answering UCQs with random order.

Theorem 5.4. *Let Q be a union of free-connex CQs. There exists a random-permutation algorithm for answering Q that uses linear preprocessing and expected logarithmic delay.*

5.2 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 3.7 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. A *mutually compatible* UCQ, or *mc-UCQ* for short, is a UCQ $Q = Q_1 \cup \cdots \cup Q_m$ 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 RAccess⟨lin, log⟩-algorithms $\mathcal{A}_I$ for Q_I that: *(a)* provide inverted access in logarithmic time; *(b)* are *compatible* in the sense that on every database D and $\emptyset \neq I \subseteq [1, m]$ we have that $\mathcal{A}_I$ is compatible with $\mathcal{A}_{\{\min(I)\}}$. We can prove the following.

Algorithm 6 Durand-Strozecki's Union Trick for $A \cup B$

```
a = A.First() ; b = B.First()
while a ≠ EOE do
    if a ∉ B then
        output a ; a = A.Next()
    else
        output b ; b = B.Next() ; a = A.Next()
while b ≠ EOE do { output b ; b = B.Next() }
```

Theorem 5.5. *Every mc-UCQ Q belongs to* RAccess⟨lin, $\log^2$⟩ *and to* REnum⟨lin, $\log^2$⟩.

An example of an mc-UCQ is $Q_7^S \cup Q_7^C$ used in the experiments in Section 6. 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 4.3 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.

The remainder of this section describes the algorithm for proving Theorem 5.5. By Theorem 3.7 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* [15], 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 6. 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.

This algorithm 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 6 enumerates all elements in $A \cup B$; and the algorithm's delay is $O(d_A + d_B + t_B)$ 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.

Algorithm 7 Random-access for $A \cup B$

```
1: function (A ∪ B).Access(j)
2:     a = A.Access(j)
3:     if a ≠ Error then
4:         if a ∉ B then
5:             output a
6:         else
7:             k = (A ∩ B).InvAcc(a) ;  b = B.Access(k) ; output b
8:     else  ℓ = j − |A| + |A ∩ B| ; b = B.Access(ℓ) ; output b
```

The idea is to provide random-access to the jth output element produced by Algorithm 6. 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 6 will be a_j if $a_j \notin B$; and in case that $a_j \in B$, the jth output element of Algorithm 6 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 6 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 "$(A \cap B)$.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 $|\{a_1, \ldots, a_j\} \cap B| = (A \cap B)$.InvAcc$(a_j)$. This immediately leads to the random-access algorithm for $A \cup B$ whose pseudocode is given in Algorithm 7.

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 6 to enumerate $A \cup B = S_1 \cup \cdots \cup S_m$, where the routines B.First() and B.Next() are provided by the induction hypothesis. We would like to use Algorithm 7 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 $a = A$.Access(j); and by the induction hypothesis, we already know how to compute $b = B$.Access(j). What we still need in order to execute Algorithm 7 is a way to compute $|A \cap B|$ and a workaround with which we can replace the command $k = (A \cap B)$.InvAcc(a); recall that this command was introduced to compute the number $k = |\{a_1, \ldots, a_j\} \cap B|$.

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 $|A \cap B| =$

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

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 $T_{1,I} := S_1 \cap \bigcap_{i \in I} S_i$.

Algorithm 8 Workaround for line 7 of Algorithm 7 for $S_1 \cup \cdots \cup S_m$. We assume that $a \in S_1$.

```
procedure Compute-k (a)
    for each I with ∅ ≠ I ⊆ [2, m] do
        b = T_{1,I}.Largest(a) ; n_{1,I} = T_{1,I}.InvAcc(b)
    k = ∑_{∅≠I⊆[2,m]} (−1)^{|I|+1} n_{1,I}  ;  output k
```

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 $T_{1,I}$.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 $T_{1,I}$.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 $n_{1,I} = T_{1,I}$.InvAcc(b) for $b := T_{1,I}$.Largest(a_j). In summary, we can replace the first command in line 7 of Algorithm 7 by Algorithm 8.

To recap, we obtain the following for $S_1 \cup \cdots \cup S_m$. For $\ell \in [1, m]$ 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 every $\ell \in [1, m]$ we have available an enumeration algorithm for S_ℓ, and for every $\emptyset \neq I \subseteq [\ell{+}1, m]$ we have available an enumeration algorithm for $T_{\ell,I}$, so that all of the following hold.

(1) The enumeration for $T_{\ell,I}$ is compatible with that for S_ℓ.
(2) After having carried out the preprocessing phase for S_ℓ: *(a)* we know its cardinality $|S_\ell|$, *(b)* given j, the routine S_ℓ.Access(j) returns in time t_{acc} the jth output element of the enumeration algorithm for S_ℓ, *and (c)* given u, it takes time t_{test} to test whether $u \in S_\ell$.
(3) After having carried out the preprocessing phase for $T_{\ell,I}$: *(a)* we know its cardinality $|T_{\ell,I}|$, *(b)* given $c \in T_{\ell,I}$, the rank $T_{\ell,I}$.InvAcc(c) can be computed in time $t_{inv\text{-}acc}$, *and (c)* given $a \in S_\ell$, it takes time t_{lar} to return largest element of $T_{\ell,I}$ that does not succeed a in 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 $\emptyset \neq I \subseteq [\ell{+}1, m]$, we can provide 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 6 (starting with $A = S_1$ and $B = S_2 \cup \cdots \cup S_m$).

Finally, to prove Theorem 5.5, 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 Largest(a).

6 IMPLEMENTATION AND EXPERIMENTS

In this section, we present an implementation and an experimental evaluation of the random-order enumeration algorithms presented in this paper. [4] Our algorithm for random-order CQ enumeration proposed in Section 4 is denoted as REnum(CQ), the algorithm for UCQs from Section 5.1 is denoted REnum(UCQ), and the algorithm for mc-UCQs from section 5.2 is denoted REnum(mcUCQ). The goal of our experiments is twofold. First, we examine the practical execution cost of REnum(CQ) compared to the alternative of repeatedly applying a state-of-the-art sampling algorithm (without replacement) [33] and removing duplicates. Second, we examine the empirical overhead of REnum(UCQ) and REnum(mcUCQ) compared to the cumulative cost of running each REnum(CQ) for each CQ separately. We describe our implementation of REnum(CQ), REnum(UCQ), and REnum(mcUCQ) in Section 6.1, the experimental setup in Section 6.2, and the results in Section 6.3.

6.1 Implementation

REnum(CQ), REnum(UCQ), and REnum(mcUCQ) are implemented 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 REnum(CQ) 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.

As described in Algorithm 5, REnum(UCQ) uses 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 REnum(CQ), we implemented deletion and testing as explained in Section 5.1. The latter two require an 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, REnum(CQ) meets the four requirements when inverted-access is activated and the shuffler capable of deletion is used. Other than that, our implementation of Algorithm 5 is fairly straightforward.

REnum(mcUCQ) uses the underlying index REnum(CQ) for random-access, testing, and inverted-access of all CQs, as well as all intersection CQs. We created REnum(mcUCQ) by using the shuffler described in Algorithm 1 on the random-access for mcUCQs described in Section 5.2. 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 5.2 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 5.5).

6.2 Experimental Setup

We now describe the setup of our experimental study.

Algorithms. To the best of our knowledge, this paper is the first to suggest a provably uniform random-order algorithm for CQ enumeration. Therefore, we compare our REnum(CQ) to a sampling algorithm by Zhao et al. [33] via an implementation from their public repository. Their algorithm generates a uniform sample, and we naively transform it into a sampling-without-replacement algorithm by duplicate elimination (i.e., rejecting previously encountered answers).[5] Zhao et al. [33] 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 B.2 in the Appendix). 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 signify the number of answers the tuple produces when only examining the sub-query rooted in its relation. This sampling algorithm is also implemented in c++14. Hence, we consider four algorithms: REnum(CQ), Sample(EW), REnum(UCQ), and REnum(mcUCQ).

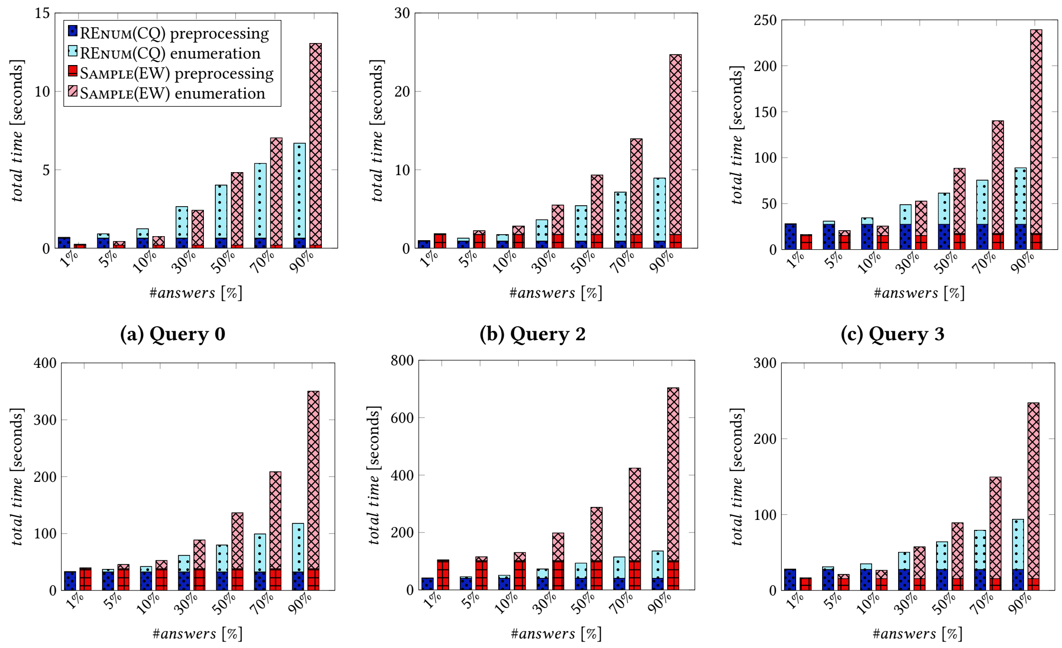
Dataset. We used the TPC-H benchmark as the database for the experiments. We generated a database using the TPC-H *dbgen* tool with a scale factor of $sf = 5$. The database has been instantiated once in memory, and all experiments use the exact same database.

Queries. We compare our REnum(CQ) to Sample(EW) using the six free-connex CQs on which Sample(EW) is implemented in the online repository. These are full-join (projection-free) CQs over the TPC-H schema. For lack of benchmarks, we phrased UCQs that we believed would form a natural extension to the TPC-H queries. The algorithms suggested in this paper 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. In order to mitigate this difference, we add attributes from the *LineItem* relation to Q_3, Q_7, Q_9, and Q_{10}. This addition achieves an equivalence between set semantics and bag semantics. The full description of our queries can be found in the Appendix (Section B.1). Each result is the average over three runs, except for Figures 2 and 3 that show a single run.

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.

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

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



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

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

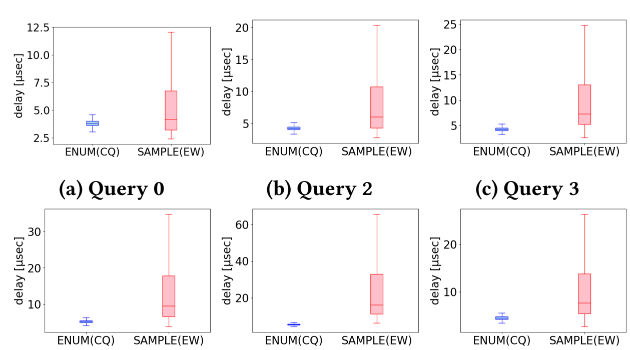
Figure 1: Total enumeration time of CQs when requesting different percentages of answers. In each bar, the bottom (darker) part refers to the preprocessing phase and the top (lighter) part to the enumeration phase.

6.3 Experimental Results

We now describe the results of our experimentation with CQs and UCQs. The CQ experiments analyze REnum(CQ) in terms of the total enumeration time (Section 6.3.1) and delay (Section 6.3.2), while the UCQ experiments analyze REnum(UCQ) and REnum(mcUCQ) in terms of the total enumeration time, as well as the rejection rate of REnum(UCQ) (Section 6.3.3). We omit from all preprocessing times the portion devoted to reading the relations.

6.3.1 CQ running time. To characterize the total enumeration time of REnum(CQ), 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 in Figure 1 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 REnum(CQ). Generally, the total time of REnum(CQ) increases slower as it does not reject answers. Hence, Sample(EW) seems better or comparable for smaller k values, but is consistently outperformed by REnum(CQ) 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. REnum(CQ) performs better, relative to Sample(EW), on queries with more relations (Q_2 ,Q_7, Q_9) than ones with fewer relations.

6.3.2 CQ delay analysis. To examine the delay of REnum(CQ) 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 2) and enumeration of 50% of the answers (Figure 3). Outliers that fell outside the whiskers are not shown, since some are several orders of magnitude larger than the median. Further information regarding the results of this experiment, as well as the outliers dropped, is in the Appendix (Section B.3). We can see that in a full enumeration, REnum(CQ) always shows a lower median value, smaller variation, and a smaller interquartile range (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 addition, REnum(CQ) usually shows better results on larger



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

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

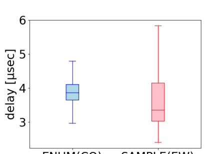
Figure 2: The delay in a full enumeration.

queries, in comparison to SAMPLE(EW). For instance, SAMPLE(EW) has a better median in Q_0 than in its larger counterpart Q_2.

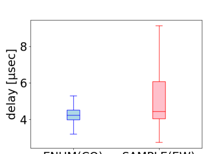
6.3.3 UCQ enumeration. This section analyzes the total enumeration time of RENUM(UCQ) and RENUM(MCUCQ), as well as the time spent on rejection of RENUM(UCQ) in three experiments, shown in Figures 4a, 4b, and 5, respectively. The 1st experiment measures the length of a full enumeration (with RENUM(UCQ) or RENUM(MCUCQ)) in three UCQs, while the second focuses on one UCQ and measures the total time of both UCQ algorithms as it varies when producing a different portion of the answers (as in Section 6.3.1). In both experiments, we compare RENUM(UCQ) and RENUM(MCUCQ) to the cumulative running time of RENUM(CQ) on the CQs comprising the union. We stress that running RENUM(CQ) on the independent CQs is *not* an alternative to an actual UCQ enumeration—it produces duplicates and does not provide a uniform random order. We perform this comparison to measure the overhead of the UCQ algorithms over RENUM(CQ). The 3rd experiment examines the time that RENUM(UCQ) spends on producing rejected answers during a single run. It shows how this time changes along the course of a full enumeration.

The difference in preprocessing time between RENUM(CQ) and RENUM(UCQ) is that for the latter, we need to build an index that supports Line 4 in Algorithm 5. Figure 4a shows that this difference can be quite small, as seen in $Q_7^S \cup Q_7^C$ and $Q_A \cup Q_E$. Meanwhile, the preprocessing of RENUM(MCUCQ) adds to that of RENUM(UCQ) the need to preprocess CQs defined by intersection of CQs from the union. Hence, RENUM(MCUCQ) always has the largest preprocessing time. Nevertheless, we see that the difference in the enumeration phase is more significant for both algorithms.

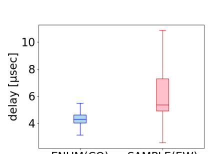
The slowdown of RENUM(UCQ) compared to RENUM(CQ) is mostly attributed to the effort to avoid duplicates by multiple CQs. A union of m CQs calls the inverted-access $m-1$ times per answer. Also, the enumeration phase is slowed down by the deletion mechanism and rejections. Figure 4a also shows that the slowdown between RENUM(CQ) and RENUM(UCQ) depends largely on the intersection size. $Q_2^N \cup Q_2^P \cup Q_2^S$ 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 two identical queries the algorithm will reject half of the answers on average.



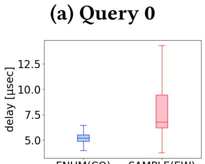
(a) Query 0



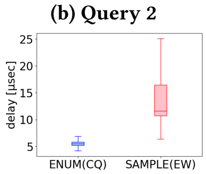
(b) Query 2



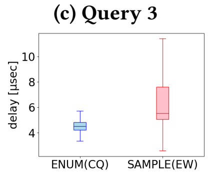
(c) Query 3



(d) Query 7

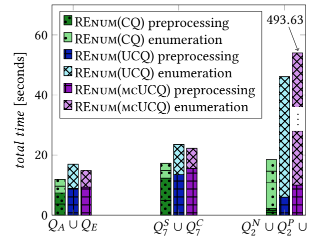


(e) Query 9

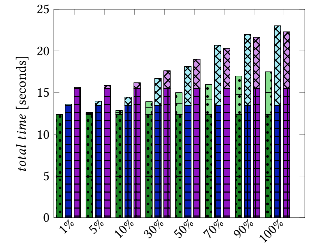


(f) Query 10

Figure 3: The delay when enumerating 50% of the answers.



(a) Total time.



(b) Varying percentage.

Figure 4: The total time of UCQs with RENUM(UCQ) vs. the total time of CQs comprising the union with RENUM(CQ).

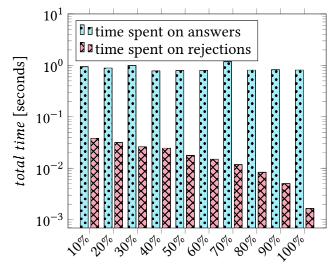


Figure 5: time spent on producing answers vs. time spent on rejections across a full enumeration of $Q_7^S \cup Q_7^C$ (log scale).

Figure 4a demonstrates that the difference in running time between RENUM(MCUCQ) and RENUM(UCQ) depends on the number of CQs in the union. For two CQs, RENUM(MCUCQ) outperforms RENUM(UCQ). $Q_A \cup Q_E$ is a disjoint union, while $Q_7^S \cup Q_7^C$ is not. Both algorithms benefit from a disjoint union, but RENUM(MCUCQ) maintains its lead. If the union is disjoint, line 7 of Algorithm 7 will never be called, so the running time of the inverted-access is saved. In RENUM(UCQ), a disjoint union causes no rejections, so the delay is also guaranteed log-time (not only in expectation). However, RENUM(UCQ) still tests membership in the other queries (as we do not know in advance that the union is disjoint), so it is more costly. RENUM(MCUCQ) on $Q_2^N \cup Q_2^P \cup Q_2^S$ suffers from a larger number of CQs in the union. As the delay depends exponentially on the number of CQs in the union, this has a dramatic effect.

Figure 4b shows the middle column of Figure 4a as it changes during the course of enumeration. It shows that the increase in total delay is rather steady in both UCQ algorithms, and that RENUM(MCUCQ) starts being preferable over RENUM(UCQ) when producing about 60% of the answers or more.

Finally, Figure 5 shows that the time RENUM(UCQ) spends on producing rejected answers decays over time. The reason for this decay is that the number of answers that belong to multiple CQs (shared answers) drops faster than that of non-shared answers, for two reasons. First, shared answers have a higher probability of being selected. Second, when a non-shared answer is selected, it is deleted everywhere, while a shared answer may become non-shared.

6.3.4 Conclusions. Our experimental study indicates that the merits of RENUM(CQ) are not only in its complexity and statistical guarantees—a fairly simple implementation of it features a significant improvement in practical performance compared to the state-of-the-art approach. Moreover, the overhead of REnum(UCQ) 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. Finally, although REnum(mcUCQ) has the advantage of guaranteed delay (unlike that of REnum(UCQ) which is expected), our empirical evaluation shows that REnum(UCQ) is usually comparable to REnum(mcUCQ) or more efficient.

7 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. We then studied two alternatives: (1) REnum(mcUCQ) uses the random-access approach for the restricted class of mc-UCQs and achieves guaranteed $\log^2$ delay; (2) REnum(UCQ) finds a random permutation directly for any UCQ comprising of free-connex CQs and achieves log delay in expectation. Our experimental study shows that the two solutions are comparable on a union of two CQs, but REnum(UCQ) preforms better on larger unions. We described an implementation of our algorithms, and presented an experimental study showing that our algorithms outperform the sampling-with-rejection alternatives.

It seems conceivable that by using van Emde Boas trees [30], the delay can be improved from log to log log; we plan to provide details in this paper's journal version. Concerning the restricted class of mc-UCQs, an obvious future task is to find syntactic properties which ensure that a given UCQ is an mc-UCQ. A simple, but very restrictive sufficient condition is a union comprising of the same CQ with different selection conditions (like we have done in our experiments).

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 [13]. Regarding UCQs, it is an open problem to find which UCQs admit efficient fine-grained enumeration, even without order guarantees [11]. 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 also supported by the Irwin and Joan Jacobs Fellowship and by the Google PhD Fellowship.

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, J. Keppeler, and N. Schweikardt. Answering UCQs under updates and in the presence of integrity constraints. In *ICDT*, pages 1–19, 2018.
[8] J. Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre.* PhD thesis, Université de Caen, 2013.
[9] F. Capelli and Y. Strozecki. Incremental delay enumeration: Space and time. *Discrete Applied Mathematics*, 268:179–190, 2019.
[10] N. Carmeli and M. Kröll. Enumeration complexity of conjunctive queries with functional dependencies. In *ICDT*, pages 11:1–11:17, 2018.
[11] 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.
[12] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *SIGMOD*, pages 263–274. ACM Press, 1999.
[13] 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.
[14] S. Deep and P. Koutris. Ranked enumeration of conjunctive query results. *CoRR*, abs/1902.02698, 2019.
[15] 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.
[16] R. Durstenfeld. Algorithm 235: Random permutation. *C. ACM*, 7(7):420, 1964.
[17] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.
[18] F. L. Gall. Powers of tensors and fast matrix multiplication. In *ISSAC*, pages 296–303, 2014.
[19] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD*, pages 927–940. ACM, 2008.
[20] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298. ACM Press, 1999.
[21] V. Hristidis and Y. Papakonstantinou. DISCOVER: keyword search in relational databases. In *VLDB*, pages 670–681. Morgan Kaufmann, 2002.
[22] 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.
[23] R. M. Karp, M. Luby, and N. Madras. Monte-carlo approximation algorithms for enumeration problems. *Journal of Algorithms*, 10(3):429 – 448, 1989.
[24] 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.
[25] 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.
[26] B. M. E. Moret and H. D. Shapiro. *Algorithms from P to NP: Volume 1: Design & Efficiency.* Benjamin-Cummings, 1991.
[27] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015.
[28] W. N. Street and Y. Kim. A streaming ensemble algorithm (SEA) for large-scale classification. In *KDD*, pages 377–382. ACM, 2001.
[29] 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.
[30] P. van Emde Boas. Preserving order in a forest in less than logarithmic time. In *16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975*, pages 75–84, 1975.
[31] V. V. Williams. Multiplying matrices faster than coppersmith-winograd. In *STOC*, pages 887–898, 2012.
[32] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, VLDB '81, pages 82–94. VLDB Endowment, 1981.
[33] 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 PROOFS FOR SECTION 5

The description of the algorithm for the random access for $A \cup B$ gives the following lemma.

Lemma A.1. *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.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.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 $(A \cap B)$.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 7 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 6.

The generalization to a union of an arbitrary number of sets is formzlied by the following lemma.

Lemma A.2. *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_ℓ.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 $T_{\ell,I}$.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 $\emptyset \neq I \subseteq [\ell{+}1, m]$, we can provide 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 6 (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 A.1 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. □

Theorem 5.5. *Every mc-UCQ Q belongs to* RAccess⟨lin, $\log^2$⟩ *and to* REnum⟨lin, $\log^2$⟩.

Proof. 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 $\mathcal{A}_I$ 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 A.2 are satisfied, except for the last one (i.e., the last bullet point in item (3) of Lemma A.2). 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 (∗). Theorem 5.5 now follows from Lemma A.2. □

B ADDITIONS TO SECTION 6

B.1 Queries

This section describes the CQs and UCQs used in Section 6. 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 REnum(CQ) to Sample(EW).

Query Q_0: a chain join between the tables *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 tables: *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 6.2).

```
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 tables *NATION*, *SUPPLIER*, *LINEITEMS*, *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 6.2).

```
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*.

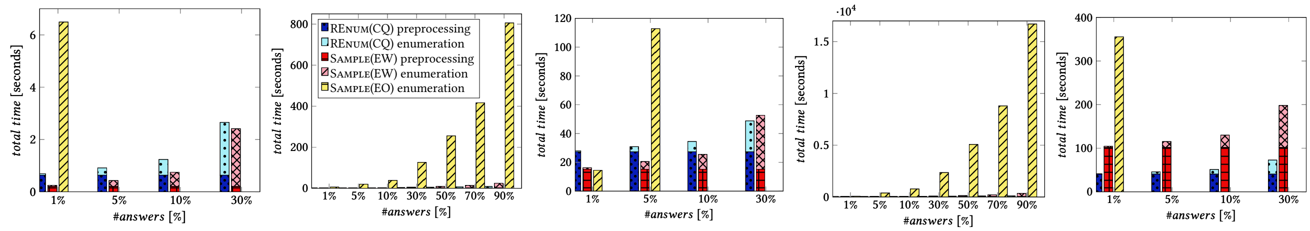
```
SELECT DISTINCT o_orderkey, s_suppkey,
                n_nationkey, r_regionkey, r_name
  FROM orders, lineitem, supplier, nation, region
 WHERE o_orderkey = l_orderkey AND
       l_suppkey = s_suppkey AND
       s_nationkey = n_nationkey AND
       n_regionkey = r_regionkey AND
       n_nationkey = 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.



(a) Query 0 (b) Query 2 (c) Query 3 (d) Query 7 (e) Query 9

Figure 6: 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.

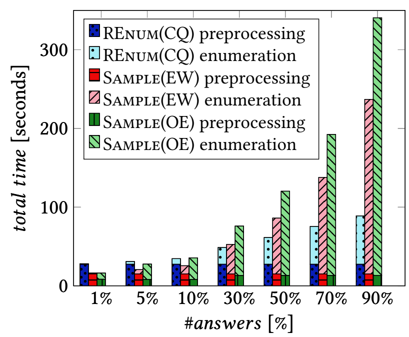


Figure 7: Total enumeration time of Q_3 when requesting different percentages of answers. In each bar, the bottom (darker) part refers to the preprocessing phase and the top (lighter) part to the enumeration phase.

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.

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.

B.2 Additional methods by Zhao et al.

As mentioned in Section 6.2, Zhao et al. [33] 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 B.2.1, B.2.2, and B.2.3 respectively) in order to explain our comparison to Sample(EW) alone.

B.2.1 EO. As Sample(EO) may reject, it possesses a much longer sampling time (as evident by our experiments). Figure 6 repeats the experiment made in section 6.3.1 (depicted in Figure 1) with the addition of Sample(EO). We omit the Sample(EO) preprocessing, as Zhao et al. [33] did in their work, and as it 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. When Sample(EO) timed-out, the corresponding bar in Figure 6 is omitted. In addition, we omit Q_10, as Sample(EO) did not produce 1% of the answers within the time limit. Figure 6 shows that with the exception of Q_3 at 1%, *EO* is significantly slower than both REnum(CQ) and Sample(EW).

B.2.2 OE. Out of our six queries, Sample(OE) was implemented on Q_3 alone. Figure 7 shows the results of section 6.3.1 with Sample(OE) added. In our experiments with Q_3, Sample(EW) has always outpreformed Sample(OE).

B.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.

B.3 More on the CQ delay experiment

This section describes further information that was left out of section 6.3.2 to save space. The following tables show the mean, standard deviation (SD) and number of delay samples that counted as outliers during the enumeration of 50% of the answers (on the left of Figure 8) and a full enumeration (on the right of Figure 8). We see that REnum(CQ) always possesses a smaller mean than Sample(EW), sometimes by an order of magnitude. We also see that REnum(CQ) 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 REnum(CQ) 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.

algorithm	query	mean (μ)	SD (σ)	outliers [%]
REnum(CQ)	Q_0	3.964625	26.77761	2.3527
Sample(EW)	Q_0	3.965105	155.8571	6.6181
REnum(CQ)	Q_2	4.35985	29.14075	3.38665
Sample(EW)	Q_2	5.455966	136.3711	6.1348
REnum(CQ)	Q_3	4.443927	198.3520	3.07209
Sample(EW)	Q_3	6.599028	519.8907	6.17242
REnum(CQ)	Q_7	5.342141	191.1972	2.91181
Sample(EW)	Q_7	8.471535	534.6212	7.12741
REnum(CQ)	Q_9	5.664519	200.8911	2.8047
Sample(EW)	Q_9	14.90882	537.2356	8.26721
REnum(CQ)	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 [%]
REnum(CQ)	Q_0	3.891264	18.9752	2.685475
Sample(EW)	Q_0	20.28385	2848.952	12.0188
REnum(CQ)	Q_2	4.319361	20.74831	3.662525
Sample(EW)	Q_2	38.02335	4815.667	12.412425
REnum(CQ)	Q_3	4.347431	140.2731	3.65655
Sample(EW)	Q_3	49.84528	20863.52	12.3539
REnum(CQ)	Q_7	5.254392	135.2184	3.47616
Sample(EW)	Q_7	72.04367	30804.44	12.50615
REnum(CQ)	Q_9	5.57028	142.0814	3.2938
Sample(EW)	Q_9	141.239102	56781.80	12.75
REnum(CQ)	Q_{10}	4.564015	138.4678	3.43488
Sample(EW)	Q_{10}	49.30842	11648.17	12.4268

Figure 8: 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).