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

Shai Zeevi
Answering (Unions of) Join Queries using Random Access and Random-Order Enumeration

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Shai Zeevi

Submitted to the Senate of the Technion — Israel Institute of Technology
Nisan 5780 Haifa April 2020
This research was carried out under the supervision of Associate Prof. Benny Kimelfeld, in the Faculty of Computer Science.

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

Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration, 2020. To appear in PODS.

Acknowledgements

I would like to thank my advisor, Prof. Benny Kimelfeld, for his guidance and support throughout this work. I would also like to thank Nofar Carmeli for all her help, as well as Nicole Schweikardt and Christoph Berkholz for their assistance. In addition, I would like to thank all the friends I made during this experience and all my colleagues at the Technion Data and Knowledge laboratory. Finally, I would like to thank my family for their continual love and support, without which this achievement would not have been possible.

The generous financial help of the Technion is gratefully acknowledged.
Contents

List of Figures

Abstract

1 Introduction

2 Preliminaries

3 Enumeration Classes
   3.1 Definitions
   3.2 Random-Access and Random Permutation

4 Random Access for CQs

5 Unions of ConjunctiveQueries
   5.1 Random Permutation with Expected Logarithmic Delay
   5.2 UCQs that Allow for Random Access

6 Implementation
   6.1 Naive Implementation of RENUM(CQ)
   6.2 Optimized Implementation of RENUM(CQ)
   6.3 RENUM(UCQ)
   6.4 RENUM(mCUCQ)

7 Experiments
   7.1 Experimental Setup
   7.2 Experimental Results
      7.2.1 Comparison to Naive Implementation
      7.2.2 CQ Running Time
      7.2.3 CQ Delay Analysis
      7.2.4 UCQ Enumeration
      7.2.5 Conclusions

8 Conclusions
A Additions to Chapter 7

A.1 Additional methods by Zhao et al. [ZCL+18] .......................... 53
  A.1.1 $EO$ ................................................................. 53
  A.1.2 $OE$ ................................................................. 53
  A.1.3 $RS$ ................................................................. 54

Hebrew Abstract .......................... i
# List of Figures

6.1 Implementation of REnum(CQ) .............................................. 32
6.2 Implementation of REnum(UCQ) ........................................... 35
6.3 Implementation of REnum(mcUCQ) ....................................... 36

7.1 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. ............ 43
7.2 Total enumeration time of CQs when requesting different percentages of answers. In each bar, the bottom (darker) part refers to the preprocessing phase and the top (lighter) part to the enumeration phase. ............ 44
7.3 The delay in a full enumeration. ........................................... 45
7.4 The mean and standard deviation of the delay in a full enumeration using each algorithm ............................................. 45
7.5 The delay when enumerating 50% of the answers. .................... 46
7.6 The mean and standard deviation of the delay when enumerating 50% of answers using each algorithm ................................. 47
7.7 The total time of UCQs with REnum(UCQ) vs. the total time of CQs comprising the union with REnum(CQ). .......................... 48
7.8 Time spent on producing answers vs. time spent on rejections across a full enumeration of $Q_S^7 \cup Q_C^7$ (log scale). .............................. 48

A.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. ......... 54
A.2 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. ......... 55
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 thesis, 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 intermediate results are representative of the whole result set in a statistically valuable manner. As we show in the thesis, 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 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.
Chapter 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 [BDG07] 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 pre-processing phase. Free-connex CQs are acyclic CQs that remain acyclic when the head of the query is added as another atom to the conjunction (see Section 2). Moreover, their work, combined with that of Brault-Baron [BB13], established that, in the absence of self-joins (i.e., when every relation occurs at most once in the query), 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) [CK19, BKS18] and the presence of constraints [CK18, BKS18].

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 [SK01]), computing approximate summaries that improve over time (e.g., as in online aggregation [HH99, LWYZ19]), and presenting the first pages of search results (e.g., as in keyword search over structured data [HP02, GKS08]). Yet, at least the latter two applications make the implicit assumption that the collection of intermediate results is a representative of the entire space of answers. In contrast, the aforementioned constant-delay algorithms enumerate in an order that is a merely an artifact of the tree selected to utilize free-connexity, and hence, intermediate answers may feature an extreme bias. Importantly, there has been a considerable recent progress in understanding the abil-
ity to enumerate the answers not just efficiently, but also with a guarantee on the order [DK19, TAG^+ 19].

Yet, to be a statistically meaningful representation of the space of answers, the enumeration order needs to be provably random. In this thesis, we investigate the task of enumerating answers in a uniformly random order. To be more precise, the goal is to enumerate the answers without repetitions, and the output induces a uniform distribution over the space of permutations of the answer set. We refer to this task as random permutation. Similarly to the recent work on ranked enumeration [DK19, TAG^+ 19], our focus here is on achieving a logarithmic 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 [AGPR99, CMN99] and recently gained a renewed attention [ZCL^+ 18].

One way of achieving a random permutation is via random access—a structure that is tied to some enumeration order and, given a position i, returns the ith answer in the order. 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 [Dur64] to design a random permutation with a negligible additive overhead over the preprocessing and enumeration phases.

So far, we have mentioned three tasks of an increasing demand: (a) enumeration, (b) random permutation, and (c) random access. We show that all three tasks can be performed efficiently (i.e., linear preprocessing time and evaluation with 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 [BB13]. Here, we devise our own random-access algorithm for free-connex CQs that is simpler and better lends itself to a practical implementation. Moreover, we design our algorithm in such a way that it is accompanied by an inverted access that is needed for our later results on UCQs.

The tractability of enumeration generalizes from free-connex CQs to unions of free-connex CQs [CK19, BKS18]. 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, with-
out 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 algorithm 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 algorithm to the approach of using a state-of-the-art random sampler [ZCL+18], 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 experiments show that Fisher-Yates over our random access for mc-UCQs can further accelerate the union enumeration (in addition to the deterministic guarantee on the delay), compared to our generic algorithm for UCQ random permutation; yet, this acceleration is not consistently evident in the experiments.

The thesis is structured as follows. The basic notation is fixed in Chapter 2. In Chapter 3 we introduce three classes of enumeration problems and discuss the relationships between them. Chapters 4 and 5 are devoted to our results concerning CQs and UCQs, respectively. Chapter 6 discusses the implementation of the algorithms we propose and Chapter 7 presents our experimental study. In Appendix A, we present minor additions to Chapter 7 that were deferred to the appendix for presentation sake.
Chapter 2

Preliminaries

In this section, we provide basic definitions and notation that we will use throughout this thesis.

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 $R_D$, 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 $R(c_1, \ldots, c_k)$, stating that the relation $R_D$ over the $k$-ary relation symbol $R$ contains the tuple $(c_1, \ldots, c_k)$.

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 $\varphi$ is a conjunction of atomic formulas of the form $R(\vec{t})$ with variables among those in $\vec{x}$ and $\vec{y}$. CQs are also known as select-project-join (SPJ) queries. 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 $\text{Vars}(Q)$ and $\text{Vars}(\alpha)$ to denote the sets of variables that occur in the CQ $Q$ and the atom $\alpha$, 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 $\bar{x}$ 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 $Q(\bar{x}) :- \alpha_1(\bar{x}, \bar{y}), \ldots, \alpha_k(\bar{x}, \bar{y})$ we associate a hypergraph $H_Q$ where the nodes are the variables in $\text{Vars}(Q)$, and the edges are $E = \{e_1, \ldots, e_k\}$ such that $e_i = \text{Vars}(\alpha_i)$. Hence, the nodes of $H_Q$ are $\bar{x} \cup \bar{y}$, and the hyperedge $e_i$ includes all the variables that appear in $\alpha_i$. 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 $\text{nodes}(T) = \text{edges}(H_Q)$, and for every $v \in \text{nodes}(H_Q)$, the nodes of $T$ that contain $v$ form a (connected) subtree of $T$. The CQ $Q$ is free-connex if $Q$ is acyclic and $H_Q$ remains acyclic when adding a hyperedge that consists of the head variables of $Q$.

A union of CQs (UCQ) is a query of the form $Q_1(\bar{x}) \cup \cdots \cup Q_m(\bar{x})$, where every $Q_i$ is a CQ with the sequence $\bar{x}$ of head variables. The set of answers to $Q_1(D) \cup \cdots \cup Q_m(D)$ over a database $D$ is, naturally, the union $Q_1(D) \cup \cdots \cup Q_m(D)$. UCQs are also known as select-project-join-union (SPJU) queries.

### 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 $A$ for an enumeration problem $P$ is an algorithm that consists of two phases: preprocessing and enumeration. During preprocessing, $A$ is given an input $I$, and it builds certain data structures. During the enumeration phase, $A$ 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 thesis, 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. [FFG02, AGM13, IUV17]. 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.

- **sparse-BMM** - 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 $O(m^{4/3})$ [AP09], which remains true even if the matrix multiplication exponent\(^1\) $\omega$ is equal to 2.

- **Triangle** - 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 $m^{2\omega/(\omega+1)+o(1)}$ [AYZ97], which is $\Omega(m^{4/3})$ even if $\omega = 2$. The Triangle hypothesis is also implied by a slightly stronger conjecture in [AW14].

- **Hyperclique** - A $(k+1, k)$-hyperclique is a set of $k+1$ vertices in a hypergraph such that every $k$-element subset is a hyperedge. By Hyperclique we denote the hypothesis that for every $k \geq 3$ there is no time $O(m)$ algorithm for deciding the existence of a $(k+1, k)$-hyperclique in a $k$-uniform hypergraph with $m$ hyperedges. This hypothesis is implied by the $(l, k)$-HYPERCLIQUE conjecture proposed in [LWW18].

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

---

1 The matrix multiplication exponent $\omega$ 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 $\omega$ is $\omega < 2.373$ and it is conjectured that $\omega = 2$ [Wil12, Gal14].
Chapter 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 $\text{Enum}(\text{lin}, d)$ to be the class of enumeration problems for which there exists an enumeration algorithm $A$ such that for every input $I$ it holds that $t_p \in O(|I|)$ and $t_d \in O(d(|I|))$. Furthermore, $\text{Enum}(\text{lin}, \text{polylog})$ is the union of $\text{Enum}(\text{lin}, \log^c)$ 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 tuple not yet emitted has equal probability of being emitted. As a result, if $|P(I)| = n$, every ordering of the answers $P(I)$ has probability $\frac{1}{n!}$ of representing the order in which $A$ prints the answers.

**Definition 3.2.** Let $d$ be a function from $\mathbb{N}_{\geq 1}$ to $\mathbb{R}_{\geq 0}$. We define $\text{REnum}(\text{lin}, d)$ to be the class of enumeration problems for which there exists a random permutation algorithm $A$ such that for every input $I$ it holds that $t_p \in O(|I|)$ and $t_d \in O(d(|I|))$. Furthermore, $\text{REnum}(\text{lin}, \text{polylog})$ is the union of $\text{REnum}(\text{lin}, \log^c)$ for all $c \geq 1$.

**Fact 3.3.** By definition, $\text{REnum}(\text{lin}, d) \subseteq \text{Enum}(\text{lin}, d)$ for all $d$.

A random-access algorithm for an enumeration problem $P$ is an algorithm $A$ 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.
More formally, for every input $I$, there exists an order of $P(I)$, denoted $t_1,\ldots,t_n$ and called the enumeration order of $A$ 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 $\text{RAccess}(\text{lin}, d)$ to be the class of enumeration problems for which there exists a random-access algorithm $A$ 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, $\text{RAccess}(\text{lin}, \text{polylog})$ is the union of $\text{RAccess}(\text{lin}, \log^c)$ for all $c \geq 1$.

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

**Fact 3.5.** By definition, $\text{RAccess}(\text{lin}, d) \subseteq \text{Enum}(\text{lin}, d)$ for all $d$.

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

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

**Proposition 3.6.** A random permutation of $0, \ldots, n-1$ can be generated with constant delay and constant preprocessing time.
Algorithm 3.1 Random Permutation

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

Proof. Algorithm 3.1 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 $k$.\(^1\) 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 $i$th iteration, the values $a_i, \ldots, a_{n−1}$ are exactly those that the procedure did not print yet. This can be shown by induction: the base case is the beginning of the first iteration (iteration 0), $a_0, \ldots, a_{n−1}$ represent 0, \ldots, $n−1$, and no numbers were printed. At the beginning of iteration $i−1$, by the induction hypothesis the algorithm has not yet printed $a_{i−1}, \ldots, a_{n−1}$. In that iteration the procedure stores in $a[i−1]$ the value that it prints (a value from those not yet printed), and moves the value that was there to a higher index. Therefore, at the beginning of iteration $i$ the element that was printed in iteration $i−1$ is in $a[i−1]$. As such, at the beginning of iteration $i$, $a_i, \ldots, a_{n−1}$ are the values not yet printed, as claimed.

Also, at each iteration, the algorithm chooses the value to print uniformly at random a value from those not yet printed, so the printed answer at every iteration has equal probability among all the values that have not yet been printed. That combined with our inductive reasoning proves that Algorithm 3.1 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 3.1 runs with constant delay, constant preprocessing time. Note that $O(n)$ space is used to generate a permutation of $n$ numbers. \(\blacksquare\)

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

\(^1\)To keep track of the array positions that are still uninitialized, one can use standard methods for lazy array initialization [MS91].
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 $P \in \text{RAccess}(\text{lin}, \log^c)$ and $P$ has polynomially many answers, then $P \in \text{REnum}(\text{lin}, \log^c)$, for all $c \geq 1$.

*Proof.* Let $P$ be an enumeration problem in $\text{RAccess}(\text{lin}, \log^c)$, 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 $k$ by trying to access the $k$th 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 $O(\log^c)$.

In this chapter we defined the problems of random-access and random permutation and delved into the relationship between random-access and random permutation. Now, with Algorithm 3.1 we know that random-access is sufficient to generate a random permutation (as shown in Theorem 3.7).
Chapter 4

Random Access for CQs

In this section, we discuss random access for CQs. For enumeration, the characterization of CQs with respect to $\text{Enum}(\text{lin}, \text{log})$ follows from known results of Bagan, Durand, Grandjean [BDG07], and Brault-Baron [BB13].

**Theorem 4.1** ([BDG07, BB13]). Let $Q$ be a CQ. If $Q$ is free-connex, then it is in $\text{Enum}(\text{lin}, \text{const})$. Otherwise, if it is also self-join-free, then it is not in $\text{Enum}(\text{lin}, \text{polylog})$ 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 [BB13]. Therefore, $Q$ it is not in $\text{Enum}(\text{lin}, \text{lin})$. Otherwise, if $Q$ is acyclic, the proof follows along the same lines as the one presented by Bagan et al. [BDG07].

According to Theorem 4.1, free-connex CQs can be answered with logarithmic delay. Brault-Baron [BB13] proved that there exists a random-access algorithm that works with linear preprocessing and logarithmic access time. Hence, we get a strengthening of Theorem 4.1: free-connex CQs belong to $\text{RAccess}(\text{lin}, \text{log})$. According to Theorem 3.7, this also shows the tractability of a random-order enumeration, that is, membership in $\text{REnum}(\text{lin}, \text{log})$.

In this section, we present a random-access algorithm for free-connex CQs that, compared to Brault-Baron [BB13], 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 Chapter 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 $j$th answer in the random-access is $e$; if $e$ is not an answer, then the inverted-access...
Algorithm 4.1 Preprocessing

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

indicates so by returning “not-a-member.”

**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., [IUV17, OZ15]). To prove it, the first step is performing a full reduction to remove dangling tuples (tuples that do not agree with any answer) from the database. This can be done in linear time as proposed by Yannakakis [Yan81] for acyclic join queries. Then, we utilize the fact that $Q$ is free-connex, which enables us to drop all atoms that contain quantified variables. This leaves us with a full acyclic join that has the same answers as the original free-connex CQ.

So, it is left to design a random-access algorithm for full acyclic CQs. We do so in the remainder of this section. Algorithm 4.1 describes the preprocessing phase that builds the data structure and computes the count (i.e., the number $|Q(D)|$ of answers). Then, Algorithm 4.2 provides random-access to the answers, and Algorithm 4.3 provides inverted-access. All three algorithms represent the query $Q$ via a suitable join tree of $Q$, and the notions “leaf”, “root”, “parent”, “child” refer to this tree.

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 $pAtts_R = \emptyset$. Given a relation $R$ and an assignment $a$, we denote by $bucket[S, a]$ 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 $S \bowtie a$).

We now elaborate on Algorithm 4.1 with an example for clarity. Consider the CQ

$$Q(v, w, x, y, z) :- R_1(v, w, x), R_2(v, y), R_3(w, z)$$
with the join tree with $R_1$ as root, and $R_2$ and $R_3$ are its children. Table 4.1 shows an input database for such a query.

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_2$</td>
</tr>
<tr>
<td></td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td></td>
<td>$b_1$</td>
<td>$d_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$b_1$</td>
<td>$d_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$d_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$d_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$d_3$</td>
<td>$c_2$</td>
</tr>
</tbody>
</table>

Table 4.1: Initial database instance

The preprocessing starts by partitioning every relation to buckets according to the different assignments to the attributes shared with the parent relation (the adhesion of the relation). Each bucket contains all tuples that agree with the assignment of that tuple. Thus, each tuple belongs to exactly one bucket. This can be done in linear time in the RAM model. The instance shown in Table 4.1 would be partitioned into the following buckets as depicted in Table 4.2.

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$a_1$</td>
<td>$b_1$</td>
<td>$c_2$</td>
</tr>
<tr>
<td></td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$a_2$</td>
<td>$b_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td></td>
<td>$b_1$</td>
<td>$d_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$b_1$</td>
<td>$d_2$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$d_1$</td>
<td>$c_1$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$d_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$d_2$</td>
<td>$c_2$</td>
</tr>
<tr>
<td></td>
<td>$b_2$</td>
<td>$d_3$</td>
<td>$c_2$</td>
</tr>
</tbody>
</table>

Table 4.2: Database instance after being partitioned to buckets

After partitioning the relations, we compute a weight $w(t)$ 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. Also, we define the weight of each bucket is the sum of the weights of the tuples it contains. In addition to assigning each tuple $t$ a weight, we assign it 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). These ranges are computed independently per bucket. If we denote the bucket $t$ belongs to as $B$ and the relation $t$ belongs to as $R$ then these ranges represent a partition of the answers to the query rooted in $R$ and restricted to the assignment $B$ represents. Each set in the partition belongs to a tuple and contain of all its answers ($w(t)$ answers). These sets of answers are ordered according to the order of the tuples in $B$. Therefore, startIndex($t$) is also the number of answers that precede the first answer of $t$ when enumerating the answers to the query rooted in $R$ and restricted to the assignment of $B$ in order of the tuples in $R$. 

17
Table 4.3: Database instance with weights computed

<table>
<thead>
<tr>
<th></th>
<th>R₁</th>
<th></th>
<th>R₂</th>
<th></th>
<th>R₃</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td></td>
<td>b₁</td>
<td>d₁</td>
</tr>
<tr>
<td></td>
<td>a₁</td>
<td>b₁</td>
<td>c₁</td>
<td></td>
<td>b₁</td>
<td>d₁</td>
</tr>
<tr>
<td></td>
<td>a₂</td>
<td>b₂</td>
<td>c₁</td>
<td></td>
<td>b₂</td>
<td>d₂</td>
</tr>
<tr>
<td></td>
<td>b₂</td>
<td>c₂</td>
<td>2</td>
<td></td>
<td>b₂</td>
<td>d₂</td>
</tr>
<tr>
<td></td>
<td>b₂</td>
<td>c₂</td>
<td>2</td>
<td></td>
<td>b₂</td>
<td>d₃</td>
</tr>
</tbody>
</table>

Algorithm 4.2 Random-Access

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

8: procedure SubtreeAccess(R, j)
9:   find t ∈ R s.t. startIndex(t) ≤ j < startIndex(t+1)
10:  answer = answer ∪ {AttsR → AttsR(t)}
11:  let R₁, . . . , Rₘ be the children of R
12:  j₁, . . . , jₘ = SplitIndex(j − startIndex(t),
13:                  w(bucket[R₁, t]), . . . , w(bucket[Rₘ, t]))
14:  for i in 1, . . . , m do
15:    SubtreeAccess(bucket[Rᵢ, t], jᵢ)

In Table 4.3, the weight of R₁(a₁ b₁ c₁) is 6 as it represents six different answers in the result of the query rooted in R₁. The bucket in R₂ that agrees with R₁(a₁ b₁ c₁) is of weight 2 and the corresponding bucket in R₃ is of weight 3. Each combination of answer from either bucket results in a different answer and that grants R₁(a₁ b₁ c₁) six answers and therefore a weight of 6. Also, these six answers take up indices 0, · · · , 5, hence startIndex(R₁(a₁ b₁ c₁)) = 6. Also note that at the end of preprocessing, the root relation has one bucket (since pAtts_root = ∅), and the weight of this bucket represents the number of answers to the query.

We now elaborate on Algorithm 4.2 via the same example used previously and demonstrate Access(13). 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. Note that in Line 10 we treat t as assignment to the variables and add the values of t to our answer. Finding t can be done in logarithmic time using binary search. The remaining index j′ = j − startIndex(t) is split into search tasks for the children using the method SplitIndex. The SplitIndex method receives the index of an answer and the weight.
of the bucket that agrees with \( t \) from each child. For each child, it returns an index to an answer to the query rooted in that child and restricted to the bucket given. These answers (one per child) put together result in the answer whose index was initially given to \texttt{SplitIndex}. Since answers were indexed (and tuple were weighted) based on combinations from child relations, this can be done in the same way as an index is split in standard multidimensional arrays: if the last bucket is of weight \( m \), its index would be \( j' \mod m \), and the other buckets will now recursively split between them the index \( \lfloor \frac{j'}{m} \rfloor \). For example, calling \texttt{Access}(13) finds \((a_2, b_2, c_1) \in R_1\) (the root). 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 \( R_2 \) via the \texttt{SplitIndex} method discussed earlier. 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 \((a_2, b_2, c_1, d_3, e_3)\).

Algorithm 4.3 is an inverse index of sorts. Given an assignment to the variables, it returns the index of that answer or an error if the assignment does not correspond to any answer. The procedure \texttt{InvertedSubtreeAccess}(\( R, a \)) can be defined as finding the index of \( a \) in the query rooted in \( R \) when restricted to the bucket that agrees with \( a \). Since the root relation only has one bucket calling \texttt{InvertedSubtreeAccess}(\texttt{root}, \( a \)) returns the index of \( a \) in the whole query. Algorithm 4.3 works similarly to Algorithm 4.2. But while the search down the tree in Algorithm 4.2 is guided by the index and the answer is the assignment, in Algorithm 4.3 the search is guided by the assignment and the answer is the index. The function \texttt{CombineIndex} is the reverse of \texttt{SplitIndex}, used in line 12 of Algorithm 4.2. \texttt{CombineIndex} receives as its input an index and weight from each child \( R_i \). The index refers to an answer to the query rooted by \( R_i \) and restricted to the assignment presented and the weight is the total number of answers in that query. Then, \texttt{CombineIndex} returns the index of an answer to the query rooted in \( R \) (the parent) and restricted to the tuple \( t \) found in Line 4. Therefore \texttt{startIndex}(\( t \)) is added for the final result. \texttt{CombineIndex} is implemented as the address resolution algorithm for multidimensional arrays. Inductively, \texttt{CombineIndex}(\( w_1, j_1, \ldots, w_m, j_m \)) is given by \( j_m + w_m \cdot \texttt{CombineIndex}(w_1, j_1, \ldots, w_{m-1}, j_{m-1}) \) with \texttt{CombineIndex}(\( \emptyset \)) = 0.

For example, calling \texttt{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 \texttt{InvertedSubtreeAccess} on \( R_2 \) returns the index startIndex(\( b_2, d_3 \)) = 1 from a bucket of weight 2, and calling \texttt{InvertedSubtreeAccess} on \( R_3 \) returns startIndex(\( c_1, e_3 \)) = 2 from a bucket of weight 3. The call for \texttt{CombineIndex}(\( 2, 1, 3, 2 \)) returns \( 2 + 3 \cdot 1 = 5 \), and the final result is \( 8 + 5 = 13 \).

Finally, Line 4 can be supported in constant time after an appropriate indexing of the buckets at preprocessing. In each bucket, simply prepare a mapping from the values tuple, \texttt{Attr}_R(\( t \)), to \( t \). With this mapping at hand, supporting Line 4 is easy, we simply find the bucket that matches the answer we were given and use the mapping to find the tuple \( t \). Since Algorithm 4.3 has a constant number of operations (in data complexity), inverted-access can be done in constant time (after the linear-time
Algorithm 4.3 Inverted-Access

1: procedure InvertedAccess(answer)
2: \hspace{1em} return InvertedSubtreeAccess(root, answer)

3: procedure InvertedSubtreeAccess(R, answer)
4: \hspace{1em} find \( t \in R \) s.t. \( \text{Atts}_R(t) = \text{Atts}_R(\text{answer}) \)
5: \hspace{1em} if \( t \) was not found then
6: \hspace{2em} return not-an-answer
7: \hspace{1em} let \( R_1, \ldots, R_m \) be the children of \( R \)
8: \hspace{1em} for \( i \) in \( 1, \ldots, m \) do
9: \hspace{2em} \( j_i = \text{InvertedSubtreeAccess}(R_i, \text{answer}) \)
10: \hspace{2em} \( \text{offset} = \text{CombineIndex}(\text{offset}(R_1, \text{answer}), j_1, \ldots, \text{offset}(R_m, \text{answer}), j_m) \)
11: \hspace{2em} return startIndex(t) + \text{offset}

preprocessing provided by Algorithm 4.1).

The next theorem, parts of which are already given by Brault-Baron [BB13], follows from 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.

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

**Corollary 4.4.** For every CQ \( Q \), the following holds: If \( Q \) is free-connex, then \( Q \) is in each of \( \text{RAccess} \langle \text{lin}, \text{log} \rangle \), \( \text{REnum} \langle \text{lin}, \text{log} \rangle \) and \( \text{Enum} \langle \text{lin}, \text{log} \rangle \). If \( Q \) is self-join-free and not free-connex, then it is not in any of \( \text{RAccess} \langle \text{lin}, \text{log} \rangle \), \( \text{REnum} \langle \text{lin}, \text{log} \rangle \), and \( \text{Enum} \langle \text{lin}, \text{log} \rangle \) assuming sparse-BMM, Triangle, and Hyperclique.
Chapter 5

Unions of Conjunctive Queries

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 $\text{Enum}(\text{lin}, d)$, for some $d$, then their union can also be enumerated within the same time bounds $[\text{DS11}, \text{CK19}]$. Since our goal is to answer queries in random order, a natural question arises: does the same apply to queries in $\text{RAccess}(\text{lin}, d)$ and $\text{REnum}(\text{lin}, d)$? We show that it does not apply to CQs in $\text{RAccess}(\text{lin}, d)$. 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 $\text{RAccess}(\text{lin}, \text{lin})$ under $\text{Triangle}$.

Example 5.1. Consider the following CQs:

$$Q_1(x, y, z) :- R(x, y), S(y, z), Q_2(x, y, z) :- S(y, z), T(x, z)$$

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 $Q_\cup \in \text{RAccess}(\text{lin}, \text{lin})$. 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 if and only 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 the $\text{Triangle}$ assumption. Thus, under the $\text{Triangle}$ assumption, the UCQ $Q_\cup$ does not belong to $\text{RAccess}(\text{lin}, \text{lin})$. ■
Algorithm 5.1 Random-Order Enumeration of $S_1 \cup \cdots \cup S_k$

\begin{algorithmic}[1]
\State \textbf{while} $\sum_{j=1}^k S_j.\text{COUNT}() > 0$ \textbf{do}
\State \hspace{1em} chosen = choose $i$ with probability $\frac{S_i.\text{COUNT}()}{\sum_{j=1}^k S_j.\text{COUNT}()}$
\State \hspace{1em} element = $S_{\text{chosen}}.\text{SAMPLE}()$
\State \hspace{1em} providers = \{ $S_j \mid S_j.\text{Test}(\text{element}) = \text{True}$ \}
\State \hspace{1em} owner = $\min\{ j \mid S_j \in \text{providers} \}$
\State \hspace{1em} \textbf{for} $S_j \in \text{providers} \setminus \{ S_{\text{owner}} \}$ \textbf{do}
\State \hspace{2em} $S_j.\text{DELETE}(\text{element})$
\State \hspace{1em} \textbf{if} $S_{\text{owner}} = S_{\text{chosen}}$ \textbf{then}
\State \hspace{2em} \hspace{1em} $S_{\text{chosen}}.\text{DELETE}(\text{element})$ \textbf{; output element}
\end{algorithmic}

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

\section{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 resembles the sampling algorithm by Karp and Luby \cite{Karp89}, but it allows for sampling without repetitions. We show how to enumerate a union in uniformly random order as stated by the following lemma. The lemma is proven by Algorithm 5.1 and the discussion following the lemma.

\textbf{Lemma 5.1.1.} 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 5.1 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 5.1 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 5.1 performs is bound by twice the number of answers. Therefore, Algorithm 5.1 guarantees logarithmic amortized delay. 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 in expectation and also a constant amortized number of operations. This proves Lemma 5.1.1.

In order to use Algorithm 5.1, the sets in question need to support counting, sampling, testing and deletion. The following lemma shows that counting, random-access and inverted-access can be used to support these operations. So, we may use this algorithm to answer UCQs.

**Lemma 5.1.2.** 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 \( O(t) \).

**Proof.** We next show how to support sampling, testing, deletion and counting using the shuffle mechanism provided in Algorithm 3.1, assuming that the sets in question support efficient counting, random-access and inverted-access.

We describe the construction of an extension to the Fisher-Yates shuffle described in Algorithm 3.1 to the aforementioned required operations. First, we count the number of answers \( n \). As in Algorithm 3.1, our data structure contains an array \( a \) of length \( n \).
and an integer $i$. Here, $i$ corresponds to the number of elements deleted. The values $a[0], \ldots, a[i-1]$ represent the indices of the deleted elements, while $a[i], \ldots, a[n-1]$ 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 3.1. When sampling, we generate a uniformly random number $k \in \{i, \ldots, n-1\}$. 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, through the inverted-access routine we find the index $m$ of the item to be deleted. We then use the reverse index $b$ to find the spot $k$ such that $a[k] = m$, swap $a[k]$ with $a[i]$, and increase $i$ by one. Since $a[k] = m$ is now no longer in $a[i], \ldots, a[n-1]$ it will not be returned in future sampling calls. Moreover, the item that used to be in $a[i]$ (and was not to be deleted) is found in $a[k]$ due to the swap and therefore may still be returned. When counting, we return $n - i$. The correctness of these procedures follows along the same lines of that of Algorithm 3.1.

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 5.1.1 and Lemma 5.1.2, we have an algorithm for answering UCQs with random order.

**Theorem 5.2.** Let $Q$ be a union of free-connex CQs. There exists a random permutation algorithm for answering $Q$ that uses linear preprocessing and every delay is logarithmic in expectation.

### 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 $\emptyset \neq I \subseteq \{1, \ldots, m\}$, the CQ $Q_I := \bigcap_{i \in I} Q_i$ is free-connex and, moreover, there are RAccess($\text{lin, log}$)-algorithms $A_I$ for $Q_I$ that: (a) provide inverted access in logarithmic time; (b) are compatible in the sense that on every database $D$ and $\emptyset \neq I \subseteq \{1, \ldots, m\}$ we have that $A_I$ is compatible with $A_{\{\min(I)\}}$. In this section we prove the following theorem:

**Theorem 5.3.** Every $mc$-UCQ $Q$ belongs to RAccess($\text{lin, log}^2$) and to REnum($\text{lin, log}^2$).

An example of an $mc$-UCQ is $Q_S \cup Q_C$ used in the experiments in Chapter 7. This UCQ is comprised of two acyclic CQs with the same structure, except they use dif-
Algorithm 5.2 Durand-Strozecki’s Union Trick for $A \cup B$

1: $a = A.\text{First}()$ ; $b = B.\text{First}()$
2: while $a \neq \text{EOE}$ do
3: \hspace{1em} if $a \notin B$ then
4: \hspace{2em} output $a$ ; $a = A.\text{Next}()$
5: \hspace{1em} else
6: \hspace{2em} output $b$ ; $b = B.\text{Next}()$ ; $a = A.\text{Next}()$
7: while $b \neq \text{EOE}$ do \{ output $b$ ; $b = B.\text{Next}()$ \}

A Union of Two Sets

We build upon Durand and Strozecki’s union trick [DS11], 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 5.2. Here, “$a = A.\text{First}()$” means that the enumeration algorithm for $A$ is started and $a$ shall be the first output element. Similarly, “$a = A.\text{Next}()$” 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, $A.\text{Next}()$ will return the end-of-enumeration message EOE; and in case that $A$ is the empty set, $A.\text{First}()$ 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 5.2 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.
B. \[ \text{where the routines} \quad A \quad \text{and} \quad B_S \]  
\[ \geq m \]  
\[ \text{is a sufficient condition. Assume we have available an algorithm that enumerates} \]
\[ \text{testing membership in} \quad B \]
\[ \text{Our next goal is to generalize this to the union of} \]
\[ \text{sets} \quad S_1, \ldots, S_m \]  
\[ \text{for an arbitrary} \quad m \geq 2. \]
\[ \text{We proceed by induction on} \quad m \]  
\[ \text{and have already established the basis for} \]
\[ m = 2. \]
\[ \text{Let us now consider the induction step from} \quad m - 1 \quad \text{to} \quad m. \]
\[ \text{We let} \quad A = S_1 \]
\[ \text{and} \quad B = S_2 \cup \cdots \cup S_m \]  
\[ \text{and use Algorithm 5.2 to enumerate} \quad A \cup B = S_1 \cup \cdots \cup S_m, \]
\[ \text{where the routines} \quad B.FIRST() \quad \text{and} \quad B.NEXT() \]  
\[ \text{are provided by the induction hypothesis.} \]
Algorithm 5.4 Workaround for line 7 of Algorithm 5.3 for $S_1 \cup \cdots \cup S_m$. We assume that $a \in S_1$.

1: procedure Compute-$k$ $(a)$
2: for each $I$ with $\emptyset \neq I \subseteq \{2, \ldots, m\}$ do
3: $b = T_{1,I}.LARGEST(a)$; $n_{1,I} = T_{1,I}.INVACC(b)$
4: $k = \sum_{\emptyset \neq I \subseteq \{2, \ldots, m\}} (-1)^{|I|+1}n_{1,I}$; output $k$

We would like to use Algorithm 5.3 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 5.3 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 $|A \cap B| = \left| \bigcup_{i=2}^{m} (S_i \cap S_1) \right| = \sum_{\emptyset \neq I \subseteq \{2, \ldots, m\}} (-1)^{|I|+1}\left| \bigcap_{i \in I} (S_i \cap S_1) \right|$.

Thus, we can compute $|A \cap B|$ provided that for each $I \subseteq \{2, \ldots, m\}$, we can compute the cardinality $|T_{1,I}|$ of the set $T_{1,I} := S_1 \cap \bigcap_{i \in I} S_i$.

Let us now discuss how to compute $k = \{|a_1, \ldots, a_j\} \cap B|$. Again using the inclusion-exclusion principle, we obtain that

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

We can compute this number if for each $\emptyset \neq I \subseteq \{2, \ldots, m\}$ 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 $i$th 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 5.3 by Algorithm 5.4.

To recap, we obtain the following for $S_1 \cup \cdots \cup S_m$. For $\ell \in \{1, \ldots, m\}$ and each $I$ with $\emptyset \neq I \subseteq \{\ell+1, \ldots, m\}$, let $T_{\ell,I} := S_\ell \cap \bigcap_{i \in I} S_i$. Assume that for every $\ell \in \{1, \ldots, m\}$ we have available an enumeration algorithm for $S_\ell$, and for every $\emptyset \neq
\( I \subseteq \{\ell+1, \cdots, m\} \) we have available an enumeration algorithm for \( T_{\ell,I} \), so that all of the following hold.

**Condition 1:** The enumeration for \( T_{\ell,I} \) is compatible with that for \( S_\ell \).

**Condition 2:** After having carried out the preprocessing phase for \( S_\ell \): (a) we know its cardinality \( |S_\ell| \), (b) given \( j \), the routine \( S_\ell.\text{Access}(j) \) returns in time \( t_{\text{acc}} \) the \( j \)th output element of the enumeration algorithm for \( S_\ell \), and (c) given \( u \), it takes time \( t_{\text{test}} \) to test whether \( u \in S_\ell \).

**Condition 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 \( T_{\ell,I}.\text{InvAcc}(c) \) can be computed in time \( t_{\text{inv-acc}} \), and (c) given \( a \in S_\ell \), it takes time \( t_{\text{lar}} \) to return largest element of \( T_{\ell,I} \) that does not succeed \( a \) in the enumeration order of \( S_\ell \).

Then, after having carried out the preprocessing phases for \( S_\ell \) and \( T_{\ell,I} \) for all \( \ell \in \{1, \cdots, m\} \) and all \( \emptyset \neq I \subseteq \{\ell+1, \cdots, m\} \), 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

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

(5.2)

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 5.2 (starting with \( A = S_1 \) and \( B = S_2 \cup \cdots \cup S_m \)). An iterated application of Algorithm 5.3 with Algorithm 5.4 produces the following recursion for the access time of \( S_1 \cup \cdots \cup S_m \):

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

(5.3)

Applying Equation (5.1) as the base case for Equation (5.3) \( f(2) \) results in the claimed time bound in Equation (5.2).

We can now prove Theorem 5.3 by applying our results over unions of sets.

**Proof. (of Theorem 5.3)**

Let \( Q = Q_1 \cup \cdots \cup Q_m \) be the given mc-UCQ, and let \( A_I \), for all \( \emptyset \neq I \subseteq \{1, \cdots, m\} \), be \( \text{RAccess} \) \((\text{lin}, \log)\)-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 \( A_I \) input \( D \). Now consider an arbitrary \( \ell \in \{1, \cdots, m\} \) and an \( I \subseteq \{\ell+1, \cdots, m\} \). For the sets \( S_\ell := Q_{\{\ell\}}(D) \) and \( T_{\ell,I} := Q_{\{\ell\},\cup}(D) \), we then immediately know that Condition 1 and Condition 2 are satisfied. Furthermore, we know that each of the time bounds \( t_{\text{test}}, t_{\text{acc}}, \) and \( t_{\text{inv-acc}} \) are at most logarithmic in the size \( |D| \) of \( D \). In order to satisfy Condition 3 we must show its last requirement is met. Therefore, to finish the proof it suffices to show the following for each \( \ell \in \{1, \cdots, m\} \) and each \( \emptyset \neq I \subseteq \{\ell+1, \cdots, m\} \):

28
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 $S_\ell$.

We can achieve this by doing a binary search on indexes w.r.t. the enumeration orders on $S_\ell$ and $T_{\ell,I}$ by using the routines $T_{\ell,I}.\text{Access}$ and $S_\ell.\text{InvAcc}$. More precisely, we start by letting $j = S_\ell.\text{InvAcc}(a)$, $c = T_{\ell,I}.\text{Access}(1)$, and $j_c = S_\ell.\text{InvAcc}(c)$. 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 $k_c = 1$, $k_d = |T_{\ell,I}|$, $d = T_{\ell,I}.\text{Access}(k_d)$, and $j_d = S_\ell.\text{InvAcc}(d)$. 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 $S_\ell$), $j_c, j_d$ are their indexes in $S_\ell$, 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 $c' = T_{\ell,I}.\text{Access}(k')$ and $j' = S_\ell.\text{InvAcc}(c')$. If $j' > j$, then we can safely terminate and return $c'$. If $j' < j$, then we proceed letting $(c, d, j_c, j_d, k_c, k_d) = (c', d, j', j_d, k', k_d)$. If $j' > j$, then 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 $T_{\ell,I}.\text{Access}$ and $S_\ell.\text{InvAcc}$. Since each such call is answered in time $O(\log |D|)$, we have achieved (*). Theorem 5.3 now follows from Equation (5.2) since Condition 1, Condition 2, and Condition 3 are all met and all time bounds ($t_{\text{test}}, t_{\text{acc}}, t_{\text{inv-acc}},$ and $t_{\text{lar}}$) are at most $\log^2$.

With this we conclude the proof of Theorem 5.3 and thus conclude the identification of a subclass of UCQs that presents random access with linear preprocessing and poly-logarithmic access time.
Chapter 6

Implementation

In this chapter, we present the implementation of the random-order enumeration algorithms presented in this thesis. Our algorithm for random-order CQ enumeration proposed in Section 4 is denoted as REnum(CQ), while 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).

All implementations we discuss use c++14 and its 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 4.1 and 4.2. Other than STL, REnum(CQ) uses Boost to hash complex types such as vectors. This chapter is structured as follows. We discuss a naive implementation of REnum(CQ) in Section 6.1, an optimized version of REnum(CQ) in Section 6.2, REnum(UCQ) in Section 6.3, and REnum(mcUCQ) in Section 6.4.

6.1 Naive Implementation of REnum(CQ)

The naive version of REnum(CQ) is a straight-forward implementation of the CQ algorithm given in Chapter 4. It differs from the source code only by minor implementation details such as caching values that are used often and choosing the appropriate and most efficient data structure for each task. As such, the algorithm does not distinguish between differently typed values and represents all relation tuples as string vectors.

6.2 Optimized Implementation of REnum(CQ)

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.
Example 6.1. To illustrate the process of code compilation we show a snippet of a code template and what it compiles to. The following snippet describes the preprocessing function and in it, placeholders are denoted by double curly brackets.

```cpp
double preprocessing(const unordered_map<string, string>& filenames,
                     const unordered_map<string, int>& numLines) {
    auto t1 = chrono::steady_clock::now();

    {{tables_local_vars_decl}}
    {{load_tables_from_files}}

    auto t2 = chrono::steady_clock::now();

    {{query_name}}YannakakisReducer().reduce(
        {{tables_local_vars_list}});

    #ifdef PROJECTION_{{query_name}}
    {{projection_and_split}}
    #else
    {{split_tables_from_tables}}
    #endif

    {{query_name}}Weighter().weight({{split_table_var_list}});
```

Figure 6.1: Implementation of REnum(CQ)
numAnswers = split_{{root}}tbl.getCummulativeWeight();

auto t3 = chrono::steady_clock::now();

return chrono::duration_cast<chrono::microseconds>(t3 - t2).count() / 1000.0;
}

The first two placeholders tables_local_vars_decl and load_tables_from_files read the input files into regular tables (before reduction, bucketing, and weighting). Then, we apply the Yannakakis full reduction [Yan81] with a function that receives those raw tables as arguments. Hence, we replace tables_local_vars_list with the table variables. Then, we apply the projection (if it exists) and then bucket the relations. For debugging purposes, projection can be discarded by removing the dedicated compilation symbol. Finally, we weight the bucketed tables and calculate the number of answers to the query.

double preprocessing(const unordered_map<string, string>& filenames, const unordered_map<string, int>& numLines) {
    auto t1 = chrono::steady_clock::now();

    Q3__C_Table _c_tbl;
    Q3__L_Table _l_tbl;
    Q3__O_Table _o_tbl;

    _c_tbl.loadFromFile(filenames.at("_C_"), numLines.at("_C_"));
    _l_tbl.loadFromFile(filenames.at("_L_"), numLines.at("_L_"));
    _o_tbl.loadFromFile(filenames.at("_O_"), numLines.at("_O_"));

    auto t2 = chrono::steady_clock::now();

    Q3YannakakisReducer().reduce(_c_tbl, _l_tbl, _o_tbl);

    #ifdef PROJECTION_Q3
    split__c_tbl.loadFromTable(_c_tbl);
    split__l_tbl.loadFromTable(_l_tbl);
    split__o_tbl.loadFromTable(_o_tbl);
    #else
    split__c_tbl.loadFromTable(_c_tbl);
    split__l_tbl.loadFromTable(_l_tbl);

    #endif

    return chrono::duration_cast<chrono::microseconds>(t3 - t2).count() / 1000.0;
}
As shown, `tables_local_vars_decl` was replaced with three variables, one for each table, and `load_tables_from_files` was replaced with function calls populating these tables from the input files. The populating function receives the path to the input file and optionally the number of lines in the file. Then, since Q3 is a full-join query, no projection is performed. Instead, we create a bucketed table directly out of the regular table (without projecting it beforehand). Afterwards, we replaced `split_table_var_list` with a list of the bucketed tables. This is a list of arguments to an object that weights the different relations according to the join-tree. Finally, we initialize the field `num_answers` with the cumulative weight of the root relation (O in this case) and return the total preprocessing time.

Figure 6.1 shows the different classes of the implementation and their relationships. The diagram is drawn for a hypothetical query we named QX. Hence, any class name that begins with QX marks a generated class. The main class, denoted `RandomOrderEnumeration`, is a very simple templated class that combines the random-access index with a Fisher-Yates shuffler (described in Algorithm 3.1 and denoted `FisherYatesShuffler` in the diagram) to create a random order enumerator. The class `QXRandomAccessIndex` implements Algorithm 4.1 and Algorithm 4.2 and is the random-access index used by `RandomOrderEnumeration`.

Implementing Algorithm 4.1 requires reducing, bucketing and weighting the different relations. To weight a relation, we must consider its children in the join tree. Hence, per relation we create a functional object that calls the weighting method on the relation and its children, that is denoted as `SingleTableWeighters` in the diagram. Then, we combine those into a single weighting object that uses the individual ones in leaf-to-root order, denoted `QXWeighter`. Similarly, to reduce we create a reducer functional object (`QXYannakakisReducer`) that calls on an implementation of the Yannakakis reduction (denoted `AbstractReducer`) in the order defined by the join tree. The bucketing process is what results in the objects denoted `BucketedRelations`,

```cpp
split__o_tbl.loadFromTable(_o_tbl);
#endif
Q3Weighter().weight(split__c_tbl, split__l_tbl, split__o_tbl);
numAnswers = split__o_tbl.getCumulativeWeight();
auto t3 = chrono::steady_clock::now();
return chrono::duration_cast<chrono::microseconds>
    (t3 - t2).count() / 1000.0;
}```
which conform to an interface called SplitTable. Finally, IndexUtils implements the methods SplitIndex and CombineIndex discussed in Chapter 4.

The implementation of Algorithm 4.2 is also very simple. Similarly, we just need to account for the order of the join tree.

### 6.3 REnum(UCQ)

![Diagram of REnum(UCQ)](image_url)

Figure 6.2: Implementation of REnum(UCQ)

As described in Algorithm 5.1, 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 4.3. Supporting inverted-access requires some additional preprocessing time (to support line 4 of Algorithm 4.3). Hence, REnum(CQ) meets the four requirements when the additional preprocessing required for inverted-access is performed and the shuffler capable of deletion is used. In this section, whenever we mention REnum(CQ), we refer to it with inverse-access included.

Figure 6.2 illustrates the implementation of REnum(UCQ) for a general hypothetical UCQ $Q_1 \cup \cdots \cup Q_m$ of $m$ queries. The implementation mainly uses the $m$ instances of REnum(CQ) and a more sophisticated shuffler that allows for deletions by index. When deleting an answer of index $k$, we swap $a[k]$ with $a[i]$ and increase $i$, mimicking what would have happened if $k$ was chosen as the next input. In order to find $k$ in the index $a$ we use a reverse-index $b$. In order to comply with c++ strict typing rules, instead of different answer classes we use a single answer class, denoted UCQAnswer, and wrap each CQ enumerator to return that answer type. Other than that, our im-
Implementation of Algorithm 5.1 is fairly straightforward and adheres to the pseudo-code listed in Section 7.2.4.

6.4 REnum(mcUCQ)

![Diagram](image)

Figure 6.3: Implementation of RENUM(mcUCQ)

RENUM(mcUCQ) uses the underlying index of RENUM(CQ) (which is denoted as $Q_1^{\text{RandomAccessIndex}}$ in Figure 6.1) 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 3.1 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 $|Q_1(I)| + |Q_2(I) \cup \ldots \cup Q_m(I)| - |Q_1(I) \cap (Q_2(I) \cup \ldots \cup Q_m(I))|$, for which we have all elements after linear time preprocessing. A minor difference between the implementation and the definition in Section 5.2 is that instead of computing the largest answer less than or equal to our current answer and then applying inverted-access on it, we compute that index directly (using the same binary-search concept as in the proof of Theorem 5.3).

Figure 6.3 illustrates the implementation of RENUM(mcUCQ) general hypothetical UCQ $Q_1 \cup \ldots \cup Q_m$ of $m$ queries. Once again, when discussing RENUM(CQ) in this section we refer to it with inverted-access included. As in RENUM(UCQ),
REnum(mcUCQ) also uses a single answer class.

Similarly to the implementation of REnum(CQ), REnum(mcUCQ) also has a main class that combines an index and a shuffler to achieve random-order enumeration. That class is denoted in Figure 6.3 as mcUCQRandomOrderEnumeration and it uses the implementation of random-access for mcUCQs along with the class FisherYatesShuffler to achieve random-order enumeration. Figure 6.3 also shows the implementation is REnum(mcUCQ) is recursive, like its definition. Meaning, to create an instance of random-access for $Q_1 \cup \cdots \cup Q_m$ ($m > 2$) we require an instance of random-access for $Q_2 \cup \cdots \cup Q_m$. Random-access can be achieved non-recursively for a union of size $m < 3$ (meaning, a union of 0,1, or 2 queries). It is best to use as large a base case as possible, so we use Algorithm 5.3 to create a random-access structure for $Q_{m-1} \cup Q_m$, as shown in Figure 6.3. Also seen, is the fact that REnum(mcUCQ) requires random-access for CQs defined by the intersection of different CQs of the union. These intersections CQs are those denoted as $Q_I$ in Section 5.2 and as Intersection CQs in Figure 6.3.
Chapter 7

Experiments

We now describe the setup of our experimental study. Our experiments have 3 goals. First, we examine the benefit of our optimizations by comparing between the naive version of REnum(CQ) and its optimized counterpart. Second, we examine the practical execution cost of REnum(CQ) compared to the alternative of repeatedly applying a state-of-the-art sampling algorithm (with replacement) [ZCL+18] and removing duplicates. Third, 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 discuss the experimental setup in Section 7.1, and the results in Section 7.2.

7.1 Experimental Setup

Algorithms

To the best of our knowledge, this thesis 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. [ZCL+18] 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). Zhao et al. [ZCL+18] suggest four different ways to initialize their algorithm, denoted RS, EO, OE, and EW. We compare our algorithm to EW as it consistently outperformed all other methods in our experiments (see Section A.1 in the Appendix). We denote this variant by Sample(EW). This sampling algorithm is also implemented in c++14. Hence, we consider four different algorithms: REnum(CQ), Sample(EW), REnum(UCQ), and REnum(mcUCQ).

1The application of this approach as an enumeration algorithm has also been discussed by Capelli and Strozecki [CS19].
Dataset

We used the TPC-H benchmark as the database for the experiments. We generated a
database using the TPC-H \textit{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 \texttt{REnum(CQ)} to \texttt{Sample(EW)} using six free-connex CQs on which \texttt{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. Also, in \texttt{Q3}, \texttt{Q7}, \texttt{Q9}, and \texttt{Q10}, we added attributes from the \textit{lineitem} relation to the query in order to achieve an equivalence between set semantics and bag semantics.

The following is a full description of our queries:

\textbf{Query \texttt{Q0}}: a chain join between the tables \texttt{partsupp}, \texttt{supplier}, \texttt{nation}, and \texttt{region}. It returns the suppliers that sell products (parts) along with their nation and region.

\begin{verbatim}
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
\end{verbatim}

\textbf{Query \texttt{Q2}}: similar to \texttt{Q0}, except for the addition of the \texttt{part} table with $ps\_partkey = p\_partkey$.

\begin{verbatim}
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
\end{verbatim}

\textbf{Query \texttt{Q3}}: the join of three tables: \texttt{customer}, \texttt{lineitem}, and \texttt{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.

\begin{verbatim}
SELECT DISTINCT o_orderkey, c_custkey, l_partkey, l_suppkey, l_linenumber
    FROM customer, orders, lineitem
    WHERE c_custkey = o_custkey AND o_orderkey = l_orderkey;
\end{verbatim}

\textbf{Query \texttt{Q7}}: similar to \texttt{Q3}, except it also joins \texttt{supplier} and \texttt{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, lineitem, partsupp, orders, and part. As in $Q_3$, we added the attributes $l_partkey$, $l_suppkey$, and $l_linenumber$ to the output to ensure an equivalence between bag and set semantics.

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

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

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

Query $Q_2^P$: similar to $Q_2$, except for the addition of the constraint $n_partkey \mod 2 = 0$. Meaning, the part identifier must be even.
Query $Q_3^S$: similar to $Q_2$, except for the addition of the constraint $n\_suppkey \mod 2 = 0$. Meaning, the supplier identifier must be even.

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 orders, lineitem, supplier, nation, and region.

```sql
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).

Each result is the average over three runs, except for Figures 7.3 and 7.5 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.

7.2 Experimental Results

We now describe the results of our experimentation with CQs and UCQs. The first CQ experiment compares the naive version of RENUM(CQ) to its optimized version (Section 7.2.1). Then, two CQ experiments analyze RENUM(CQ) in terms of the total enumeration time (Section 7.2.2) and delay (Section 7.2.3). Finally, three 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 7.2.4). We omit from all preprocessing times the portion devoted to reading the relations.

7.2.1 Comparison to Naive Implementation

We compare the total enumeration time of the different versions of RENUM(CQ) over the TPC-H CQs. To do so, 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
Figure 7.1: 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.

**Note:**
- **REnum(CQ) preprocessing**
- **REnum(CQ) enumeration**
- **preprocessing of naive REnum(CQ)**
- **enumeration of naive REnum(CQ)**

Figure 7.1 shows that REnum(CQ) is almost always significantly more efficient than

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 7.1 with a chart per query.
Figure 7.2: Total enumeration time of CQs when requesting different percentages of answers. In each bar, the bottom (darker) part refers to the preprocessing phase and the top (lighter) part to the enumeration phase.

The naive implementation. As such, we can conclude that the optimizations applied were indeed effective and that there is no reason to discuss the naive implementation in following experiments, as we have a better implementation of RENUM(CQ). Therefore, we omit the naive version from the rest of the experiments discussed in this chapter.
7.2.2 CQ Running Time

Figure 7.3: The delay in a full enumeration.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>query</th>
<th>mean ($\mu$)</th>
<th>SD ($\sigma$)</th>
<th>outliers [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>REnum(CQ)</td>
<td>$Q_0$</td>
<td>3.891264</td>
<td>18.9752</td>
<td>2.685475</td>
</tr>
<tr>
<td>Sample(EW)</td>
<td>$Q_0$</td>
<td>20.28385</td>
<td>2848.952</td>
<td>12.0188</td>
</tr>
<tr>
<td>REnum(CQ)</td>
<td>$Q_2$</td>
<td>4.319361</td>
<td>20.74831</td>
<td>3.662525</td>
</tr>
<tr>
<td>Sample(EW)</td>
<td>$Q_2$</td>
<td>38.02335</td>
<td>4815.667</td>
<td>12.412425</td>
</tr>
<tr>
<td>REnum(CQ)</td>
<td>$Q_3$</td>
<td>4.347431</td>
<td>140.2731</td>
<td>3.65655</td>
</tr>
<tr>
<td>Sample(EW)</td>
<td>$Q_3$</td>
<td>49.84528</td>
<td>20863.52</td>
<td>12.3539</td>
</tr>
<tr>
<td>REnum(CQ)</td>
<td>$Q_7$</td>
<td>5.254392</td>
<td>135.2184</td>
<td>3.47616</td>
</tr>
<tr>
<td>Sample(EW)</td>
<td>$Q_7$</td>
<td>72.04367</td>
<td>30804.44</td>
<td>12.50615</td>
</tr>
<tr>
<td>REnum(CQ)</td>
<td>$Q_9$</td>
<td>5.57028</td>
<td>142.0814</td>
<td>3.2938</td>
</tr>
<tr>
<td>Sample(EW)</td>
<td>$Q_9$</td>
<td>141.239102</td>
<td>56781.80</td>
<td>12.75</td>
</tr>
<tr>
<td>REnum(CQ)</td>
<td>$Q_{10}$</td>
<td>4.564015</td>
<td>138.4678</td>
<td>3.43488</td>
</tr>
<tr>
<td>Sample(EW)</td>
<td>$Q_{10}$</td>
<td>49.30842</td>
<td>11648.17</td>
<td>12.4268</td>
</tr>
</tbody>
</table>

Figure 7.4: The mean and standard deviation of the delay in a full enumeration using each algorithm

To characterize the total enumeration time of REnum(CQ), we compare it to that of Sample(EW) for the TPC-H CQs. In order to compare the two algorithms, we re-enact the experiment mentioned in Section 7.2.1. For each query, each algorithm enumerates $k$ distinct answers (chosen as a certain percentage of the full result) and we the total enumeration time. That process is repeated for increasing values of $k$.

The results of this experiment are presented in Figure 7.2 with a chart per query. The results indicate that, as $k$ grows, the total enumeration time of Sample(EW) grows more rapidly in comparison to 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 $\text{REnum(CQ)}$ 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. $\text{REnum(CQ)}$ performs better, relative to $\text{Sample(EW)}$, on queries with more relations ($Q_2, Q_7, Q_9$) than ones with fewer relations.

### 7.2.3 CQ Delay Analysis

To examine the delay of $\text{REnum(CQ)}$ and $\text{Sample(EW)}$, we record the delay of each answer and depict it in box-and-whisker plots. In addition, we list statistics of the answers’ delay in tables. Each query has two box plots and two table entries: enumeration of all answers (Figure 7.3 and Figure 7.4) and enumeration of 50% of the answers (Figure 7.5 and Figure 7.6). Outliers that fell outside the whiskers are not shown in the boxplots, since some are several orders of magnitude larger than the median. Therefore, the percent of samples that counted as outliers in each run is listed in the tables. The tables also show the mean and standard deviation (SD) observed in each run.

Figure 7.3 and Figure 7.5 show that in a full enumeration, $\text{REnum(CQ)}$ 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 $\text{Sample(EW)}$ actually exhibits a smaller median. Also, Figure 7.6 and Figure 7.4 show that $\text{REnum(CQ)}$ always possess a smaller mean than $\text{Sample(EW)}$, sometimes by an order of magnitude. As well as considerably smaller standard deviation and smaller number of outliers. This holds even when the two are close in median as is
<table>
<thead>
<tr>
<th>algorithm</th>
<th>query</th>
<th>mean (µ)</th>
<th>SD (σ)</th>
<th>outliers [%]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RENUM(CQ)</td>
<td>Q₀</td>
<td>3.964625</td>
<td>26.77761</td>
<td>2.3527</td>
</tr>
<tr>
<td>SAMPLE(EW)</td>
<td>Q₀</td>
<td>3.965105</td>
<td>155.8571</td>
<td>6.6181</td>
</tr>
<tr>
<td>RENUM(CQ)</td>
<td>Q₂</td>
<td>4.35985</td>
<td>29.14075</td>
<td>3.38665</td>
</tr>
<tr>
<td>SAMPLE(EW)</td>
<td>Q₂</td>
<td>5.455966</td>
<td>136.3711</td>
<td>6.1348</td>
</tr>
<tr>
<td>RENUM(CQ)</td>
<td>Q₃</td>
<td>4.443927</td>
<td>198.3520</td>
<td>3.07209</td>
</tr>
<tr>
<td>SAMPLE(EW)</td>
<td>Q₃</td>
<td>6.599028</td>
<td>519.8907</td>
<td>6.17242</td>
</tr>
<tr>
<td>RENUM(CQ)</td>
<td>Q₇</td>
<td>5.342141</td>
<td>191.1972</td>
<td>2.91181</td>
</tr>
<tr>
<td>SAMPLE(EW)</td>
<td>Q₇</td>
<td>8.471535</td>
<td>534.6212</td>
<td>7.12741</td>
</tr>
<tr>
<td>RENUM(CQ)</td>
<td>Q₉</td>
<td>5.664519</td>
<td>200.8911</td>
<td>2.8047</td>
</tr>
<tr>
<td>SAMPLE(EW)</td>
<td>Q₉</td>
<td>14.90882</td>
<td>537.2356</td>
<td>8.26721</td>
</tr>
<tr>
<td>RENUM(CQ)</td>
<td>Q₁₀</td>
<td>4.652109</td>
<td>195.7905</td>
<td>2.89414</td>
</tr>
<tr>
<td>SAMPLE(EW)</td>
<td>Q₁₀</td>
<td>6.866843</td>
<td>519.3847</td>
<td>6.330277</td>
</tr>
</tbody>
</table>

Figure 7.6: The mean and standard deviation of the delay when enumerating 50% of answers using each algorithm.

the case with $Q₀$. The smaller number of outliers and lower standard deviation indicates the predictability of the delay, as it does not grow rapidly. Finally, RENUM(CQ) usually shows better results on larger queries, in comparison to SAMPLE(EW). For instance, SAMPLE(EW) has a better median in $Q₀$ than in its larger counterpart $Q₂$.

### 7.2.4 UCQ Enumeration

This section analyzes the total enumeration time of the UCQ algorithms, RENUM(UCQ) and RENUM(mcUCQ), as well as the time spent on rejection of RENUM(UCQ) in three experiments, shown in Figures 7.7a, 7.7b, and 7.8, respectively. 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 7.2.2). 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 5.1. Figure 7.7a shows that this difference can be quite small, as seen in $Q₇^C \cup Q₇^F$ and $Q₄ \cup Q₃$. 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 \textit{REnum}(UCQ) compared to \textit{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 7.7a also shows that the slowdown between \textit{REnum}(CQ) and \textit{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.

Figure 7.7a shows that the difference in running time between \textit{REnum}(mcUCQ) and \textit{REnum}(UCQ) depends on the number of CQs in the union. For two CQs, \textit{REnum}(mcUCQ) outperforms \textit{REnum}(UCQ). \(Q_A \cup Q_E\) is a disjoint union, while \(Q_2^N \cup Q_2^P \cup Q_2^S\) is not. Both algorithms benefit from a disjoint union, but \textit{REnum}(mcUCQ) maintains its lead. If the union is disjoint, line 7 of Algorithm 5.3 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 $Q^N_2 \cup Q^P_2 \cup Q^S_2$ 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 7.7b shows the middle column of Figure 7.7a 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 7.8 shows that the time RENUM(UCQ) spends on producing rejected answers decays over time. A possible explanation 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.

### 7.2.5 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.
Chapter 8

Conclusions

We studied the problems of answering queries in a random permutation and via a random-access. We found that for CQs without self-joins it holds that $\text{Enum}(\text{lin}, \log) = \text{RAccess}(\text{lin}, \log) = \text{REnum}(\text{lin}, \log)$. We also studied the generalization to unions of free-connex CQs where, in contrast, we have $\text{Enum}(\text{lin}, \log) \neq \text{RAccess}(\text{lin}, \log)$ and random-access may be intractable even if tractable for each CQ in the union. We then studied two alternatives: (1) $\text{REnum}(\text{mcUCQ})$ uses the random-access approach for the restricted class of mc-UCQs and achieves guaranteed $\log^2$ delay; (2) $\text{REnum}(\text{UCQ})$ finds a random permutation directly for any UCQ comprising of free-connex CQs and achieves log delay in expectation.

We described an implementation of our algorithms, and presented an experimental study showing that our algorithms outperform the sampling-with-rejection alternatives. Also, we compare our expected algorithm for UCQs to our worst-case algorithm for mcUCQs over some mcUCQs. Our experimental study shows that the two solutions are comparable on a union of two CQs, but $\text{REnum}(\text{UCQ})$ preforms better on larger unions.

In this work we only examined the class of CQs that admit efficient random-order and random permutation. However, it still remains an open problem to see what can be done for a non-free-connex CQ. A non-free-connex CQ does not admit linear-preprocessing, polylog-delay random permutation. However, if it is a CQ of bounded treewidth we can use a hypertree decomposition and $\text{REnum}(\text{CQ})$ in order to generate a random permutation with some polynomial preprocessing overhead.

It is also an open problem to find which UCQs admit efficient fine-grained enumeration, even without order guarantees [CK19]. 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.
Appendix A

Additions to Chapter 7

A.1 Additional methods by Zhao et al. [ZCL+18]

As mentioned in Section 7.1, Zhao et al. [ZCL+18] 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 Q3. Here we review EO, OE, and RS (in sections A.1.1, A.1.2, and A.1.3 respectively) in order to explain our comparison to SAMPLE(EW) alone.

A.1.1 EO

As SAMPLE(EO) may reject, it possesses a much longer sampling time (as evident by our experiments). Figure A.1 repeats the experiment made in section 7.2.2 (depicted in Figure 7.2) with the addition of SAMPLE(EO). We omit the SAMPLE(EO) preprocessing, as Zhao et al. [ZCL+18] 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 A.1 is omitted. In addition, we omit Q10, as SAMPLE(EO) did not produce 1% of the answers within the time limit. Figure A.1 shows that with the exception of Q3 at 1%, EO is significantly slower than both REnum(CQ) and SAMPLE(EW).

A.1.2 OE

Out of our six queries, SAMPLE(OE) was implemented on Q3 alone. Figure A.2 shows the results of section 7.2.2 with SAMPLE(OE) added. In our experiments with Q3, SAMPLE(EW) has always out-preformed SAMPLE(OE).
Figure A.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.

### A.1.3 RS

SAMPLE(RS) was also implemented only on Q3. SAMPLE(RS) was unable to produce a sample of 1% of the answers to Q3 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.
Figure A.2: 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.
Bibliography


לבסוף, אנו מציגים מימוש של אלגוריתמי הגישה האקראית והפרמוטציה האקראית שלם ומצטברים עם תוצאות השיטות המופנות תпоיה (benchmark) TPC-H המשמש כמאגר מודרני (dbscale) והליחוש של האלגוריתמים. אנו משים לב שהפרמוטציה האקראית שלם,:]) נושאת תכונה של_algoritmית הממשיכה את הジョン אקפורד, אך [_18]", שערובuya על מתכון לו דגמה את вход האטום, ובעוד [_18]",,ZCL+"].

עקבות כך, מופיעו مضיים של_algoritmית המופנות תпоיה [18] או[_18], [ZCL+].

הנבט ניסיון מתוחכם, בשיטות המורзе, גם בשיטות מפורטות. ניסיון זה, בשיטות מפורטות,bellionוori כשל_algoritmית המופנות תпоיה [18] או[_18], [ZCL+].

עד יחר אט מיני התאמות (במסגרת המיקום והאקסטרים של הנטייה, והקיבוץ), בשיטות אלגוריתמיות הנבט של Воור פרמוטציה אקפורד שלם שאלות בחר.

iii
The enumeration of a finite set is a fundamental problem in computer science, with applications in various domains such as databases, algorithms, and artificial intelligence. In this thesis, we focus on the enumeration of conjunctive queries, which are a special class of queries that are widely used in database systems.

A conjunctive query is a logical expression that combines multiple simple queries using the AND operator. The enumeration of conjunctive queries refers to the process of generating all possible solutions to a given query, as efficiently as possible. This problem is known to be computationally hard, and there has been significant research into developing efficient algorithms to solve it.

One of the key contributions of this thesis is the development of a ranked enumeration algorithm for conjunctive queries. This algorithm leverages the fact that conjunctive queries can be effectively decomposed into smaller, simpler queries, which can then be solved independently and combined to form the final solution. This approach allows for efficient enumeration, even for queries that would otherwise be intractable.

Another important aspect of this work is the development of a theoretical framework for analyzing the complexity of enumeration algorithms. This framework provides a way to compare different enumeration strategies and to understand the trade-offs between efficiency and memory usage.

The results presented in this thesis have implications for a wide range of applications, from database management to artificial intelligence. They also contribute to the broader field of computational complexity, by providing new insights into the nature of enumeration problems and the limits of efficient computation.

In summary, this thesis presents a novel approach to the enumeration of conjunctive queries, and establishes a new theoretical foundation for understanding the complexity of such problems. The results are expected to have a significant impact on both theory and practice, and to open up new avenues of research in this exciting area of computer science.
תקציר

במאמץلزمعة המחירה החישוב יש לשאילות מסדי-נתונים למינימום האפשרי, חלה עלייה
(fine-grained complexity) המשמעות שלادات המחקר במחזורים לשיפוטי העדינות של מיני השחובות [BDG07] והאחתה כי סיפורי הפיקוד 
(free-connex acyclic conjunctive queries) והסידורי העדינות אי עיכוב קסמסתר לשאילות בעערת אלגריתמים מיניים 
(free-connex conjunctive queries) השזורים בחבר

הباحשה השזורים מסך לשאילות ברсты בשאילות המעלים вкус השאילתה. עבודה שלוב, עבורה בברשה קארה בקארה [BB13] קביעה כי ברשורי 
(כפלים, בברך כות מקובך כל חותר פוע אתח שיאלטה), א seçilית העדינות שלש

הנהיך והאחתה בעינו בשאילות העדינות התשובה. החסם הת蝸ן לשאילות ופונקצית הידיאומון בין חזרה בשס הת蝸ן

בחיסוכרתידג (למשל, הנקה האמורתי כלא נתח בצפיפות בשפיכות בוליאניות לולדה, חווית

מעגלים, אוזי-חרף-קלוקט בואכניס לינו.)andr. הרחבות נ朋友们对 כלולות הת蝸ן הת蝸ן לאידוי החששות [BK18] [CK18, BK18]

ירוח

[SK01] learning algorithms online aggregation [HH99, LWYZ19] [HP02, GKS08] [DK19, TAG+19]

במעי מתופעת שיאליטה, נ yOffset הת蝸ןฌית השיאליטה או ו resultat הת蝸ן הת蝸ן שיסופר הת蝸ית הת蝸ית בו

לפי היצורים בתשובה, שהוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של תしようと

streaming בשיפוטי הפריסטים שיאליטה לכל שיפוטי התשובות [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]

לפי היצורים בתשובה, הוא הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית הת蝸ית של ת.showToast [SK01] learning algorithms online aggregation [HH99, LWYZ19]
המחקר בוצע בתנחיתו של פרופסור הרב בנימין קימלפלד, בפקולטה להנדסת מחשבים. חלקי מחקרי החיבור הזה פורסמו כמאמרים מעת מחבריו ושוחפים המחקר במותונים שונים ובעיתונים. המחבר נתקף מחקר הדוג辂ייני של המחברים, אשר גרסאותיו העדכניות יותר הן:

Nofar Carmeli, Shai Zeevi, Christoph Berkholz, Benny Kimelfeld, and Nicole Schweikardt. Answering (unions of) conjunctive queries using random access and random-order enumeration, 2020. To appear in PODS.

תודה

ברצוני להודות לנחיתו של נח, פרופ' בנימין קימלפלד, על בדיקת הנשמות ולдорך העדות. ברצוני להודות לכל הנשים של נח הבכירים ו.',' mấy שנים לפני שהקימו וקימו וברחובות הערמון.יונתן הזיתון, בברית לחדות שלפחתים על תחומי וארטיקות והאנטומיה, הבעיות והמשותפויות.ابل רוני ויה Caught.אי bluetooth להڅو החכם של הכנסת וה다가 יסוד לחית.
מענה על (איהוד של)シアレハート צירוף על
ידי גישה אקריאית ומייניה בסטור אקריאי

היבר על מחקר

לשם مليי חלקי של הדרישות נקבבל התואר
metisル למדים במדעי המחשב

שוי זאבי

הון כלני הרכזון – מכון טכנולוגיה הישראלית
ינס חדש'פ海滨 התאורה
אפריל 2020
מענה על (אינדוץ של)シアワシネクライウィキון על
ידי גישה הארץיאית ומייניה הבסייד אקראי

шин ז所提供