Threshold Queries in Theory and in the Wild

Angela Bonifati
Lyon 1 Univ., Liris CNRS
angela.bonifati@univ-lyon1.fr

Stefania Dumbrava
ENSIIE & Inst. Polytechnique de Paris
stefania.dumbrava@ensiie.fr

George Fletcher
Eindhoven Univ. of Technology
g.h.l.fletcher@tue.nl

Jan Hidders
Univ. of London, Birkbeck
jan@dcs.bbk.ac.uk

Matthias Hofer
University of Bayreuth
matthias.hofer@uni-bayreuth.de

Wim Martens
University of Bayreuth
wim.martens@uni-bayreuth.de

Filip Murlak
Univ. of Warsaw
fmurlak@mimuw.edu.pl

Joshua Shinavier
LinkedIn
jshinavier@linkedin.com

Sławek Staworko
Univ. Lille, INRIA, CNRS, UMR 9189 - CRIStAL
slawomir.staworko@inria.fr

Dominik Tomaszuk
Univ. of Bialystok
d.tomaszuk@uwb.edu.pl

ABSTRACT

Threshold queries are an important class of queries that only require computing or counting answers up to a specified threshold value. To the best of our knowledge, threshold queries have been largely disregarded in the research literature, which is surprising considering how common they are in practice. In this paper, we present a deep theoretical analysis of threshold query evaluation and show that thresholds can be used to significantly improve the asymptotic bounds of state-of-the-art query evaluation algorithms. We also empirically show that threshold queries are significant in practice. In surprising contrast to conventional wisdom, we found important scenarios in real-world data sets in which users are interested in computing the results of queries up to a certain threshold, independent of a ranking function that orders the query results.

PVLDB Reference Format:
Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Sławek Staworko, and Dominik Tomaszuk. Threshold Queries in Theory and in the Wild. PVLDB, 15(5): 1105–1118, 2022.
doi:10.14778/3510397.3510407

1 INTRODUCTION

Queries encountered in a wide range of data management applications, such as data interaction, data visualization, data exploration, data curation and data monitoring [8, 41, 45], often require computing or counting answers only up to a given threshold value. We call such queries *threshold queries*.

Threshold queries for data exploration. Querying voluminous rich data sources, such as Wikidata [67], may return more results than are needed during exploratory analytics. Thus, users may specify a threshold on the number of answers they want to see. Consider the following LIMIT query which lists up to a threshold businesses, total assets, and headquarter locations in the Wikidata dataset.

```
SELECT ?business ?assets ?city ?country
{ ?business <total_assets> ?assets .
  ?business <headquarters_location> ?city .
  ?city <country> ?country . }
LIMIT 10;
```

Proceedings of the VLDB Endowment, Vol. 15, No. 5 ISSN 2150-8097.
doi:10.14778/3510397.3510407

Without LIMIT, this query would return 2,684,138 objects, amounting to about 413MB in size, which is too large for human consumption. Note that here the user is interested in unranked output (i.e., there is no ORDER BY clause), which is typical for data exploration [8]. As we will see later in a deep-dive into real query logs (Section 5), such queries on Wikidata are very common in practice.

Threshold queries for data curation. Threshold queries are also useful in detecting violations of database constraints and identifying data items requiring curation actions. As an example, consider the following threshold query using *grouping* and aggregation in the Nobel Prize database, requiring that every Nobel prize has at most three laureates [4].

```
SELECT P.ID FROM NobelPrize P, Laureate L
WHERE P.ID = L.Prize_ID
GROUP BY P.ID HAVING COUNT(*) > 3;
```

Differences with other query answering paradigms. Although threshold queries might look similar in spirit to top-k queries [35, 43, 59, 60], they are inherently different because they do not assume that the results are ranked. They are also different from counting queries [32, 63], since these aim at computing an exact value, rather than only desiring exact values up to a given threshold. Therefore, prior problems are either more specific or have different objectives. We examine these differences in depth in Section 6.

Our contributions. We are motivated by the following question: Can we exploit the fact that we only need to count or compute the answers of a query *up to a threshold*? In this paper, we answer this question positively. The starting point for our work is the observation that evaluating some queries with a threshold k requires storing not more than $k + 1$ intermediate results [18, 51]. We show that this idea can be fully integrated with state-of-the-art complex join algorithms, leading to significant savings in the size of the intermediate results as typically computed in query processing, leading for some queries to improvements from $\tilde{O}(n^f)$ to $\tilde{O}(n \cdot k)$, where f is the *free-connex treewidth* of the query and k is the value of the threshold. Here, the free-connex treewith of a query measures its treewidth in connection with its output variables. It can be large even if the query is acyclic.

In detail, the key contributions of our paper are as follows:

(1) New results explaining the interplay between different structural properties of conjunctive queries (i.e., select-project-join queries) used in sophisticated evaluation algorithms (Lemma 4.5), and the consequences for threshold query processing (Theorem 4.7).

(2) New evaluation algorithms for threshold queries (for computing answers, counting answers, and sampling answers) with improved asymptotic guarantees (Theorem 4.10).
(3) A comprehensive empirical study of threshold queries found in the wild, which highlights their characteristics and shows that they are quite common in important practical scenarios.
(4) An experimental evaluation of a proof-of-concept SQL implementation of our algorithm against the current query optimizer in PostgreSQL, showing speedups of several orders of magnitude.

Our work provides the first in-depth theoretical treatment of threshold queries. In addition, it also shows that these queries are important in practical settings by means of an in-depth empirical study.

The paper is organized as follows. Section 2 contains basic definitions. In Section 3, we identify the problems of interest, explain the complexity-theoretic limits, and explain some of our main ideas in an example. Section 4 presents the algorithms for the problems of interest. In Section 5, we present experiments and findings on threshold queries that can be found in practice. In Section 6, we discuss related work. Finally, in Section 7 we summarize our findings and outline further research directions. Because of space limitation, detailed proofs as well as additional material are omitted and can be found in an extended technical report [13].

2 PRELIMINARIES

Our results apply in both a relational database setting and a graph database setting. We first focus on the relational database setting, for which we loosely follow the preliminaries as presented in [1]. We assume that we have countably infinite and disjoint sets Rel of *relation names* and Const of *values*. Furthermore, when $(a_1, \ldots, a_k)$ is a Cartesian tuple, we may abbreviate it as $\bar{a}$. A k-ary *database tuple* (henceforth abbreviated as *tuple*) is an element of Const^k for some $k \in \mathbb{N}$. A *relation* is a finite set S of tuples of the same arity. We denote the set of all such relations by $\mathcal{R}$. A *database* is a partial function $D : \mathsf{Rel} \to \mathcal{R}$ such that $\mathsf{Dom}(D)$ is finite. If D is a database and $R \in \mathsf{Rel}$, we write $R(a_1, \ldots, a_k) \in D$ to denote that $(a_1, \ldots, a_k) \in D(R)$. By $\mathsf{Adom}(D)$ we denote the set of constants appearing in tuples of D, also known as the *active domain* of D.

For defining conjunctive queries, we assume a countably infinite set Var of variables, disjoint from Rel and Const. An *atom* is an expression of the form $R(u_1, \ldots, u_k)$, where $u_i \in \mathsf{Var} \cup \mathsf{Const}$ for each $i \in \{1, \ldots, k\}$. A *conjunctive query (CQ)* is an expression q of the form

$$\exists \bar{y}\ A_1(\bar{u}_1) \land \ldots \land A_n(\bar{u}_n)\ ,$$

where $\bar{y} = (y_1, \ldots, y_m)$ consists of *existentially quantified variables* and each $A_i(\bar{u}_i)$, with $i \in \{1, \ldots, n\}$ is an atom. Each variable that appears in $\bar{y}$ should also appear in $\bar{u}_1, \ldots, \bar{u}_n$. On the other hand, $\bar{u}_1, \ldots, \bar{u}_n$ can contain variables not present in $\bar{y}$. For a tuple $\bar{x}$, we write $q(\bar{x})$ to emphasize that q is a CQ such that all variables in $\bar{u}_1, \ldots, \bar{u}_n$ appear in either $\bar{x}$ or $\bar{y}$. Unless we say otherwise, we assume that the variables in $\bar{u}_1, \ldots, \bar{u}_n$ are *precisely* the variables in $\bar{x}$ and $\bar{y}$. The *arity* of $q(\bar{x})$ is defined as the arity of the tuple $\bar{x}$. We denote by $\mathsf{Var}(q)$ the set of all variables appearing in q and by $\mathsf{FVar}(q)$ the set of so-called *free variables* of q, which are the variables in $\mathsf{Var}(q)$ that are not existentially quantified. A *full CQ* is a CQ without existentially quantified variables.

Query Answers and Relational Algebra. We consider queries under *set semantics*, i.e., each answer occurs at most once in the result. A *binding of* $X \subseteq \mathsf{Var}$ is a function $\eta : X \to \mathsf{Const}$. We say that bindings η and η' are *compatible* if $\eta(x) = \eta'(x)$ for all $x \in \mathsf{Dom}(\eta) \cap \mathsf{Dom}(\eta')$. For compatible bindings η_1 and η_2, the *join of* η_1 *and* η_2, is the binding $\eta_1 \bowtie \eta_2$ such that $\mathsf{Dom}(\eta_1 \bowtie \eta_2) = \mathsf{Dom}(\eta_1) \cup \mathsf{Dom}(\eta_2)$ and $(\eta_1 \bowtie \eta_2)(x) = \eta_i(x)$ for all $x \in \mathsf{Dom}(\eta_i)$ and $i \in \{1, 2\}$. If P_1 and P_2 are sets of bindings, then the *join of* P_1 *and* P_2 is $P_1 \bowtie P_2 = \{\eta_1 \bowtie \eta_2 \mid \eta_1 \in P_1 \text{ and } \eta_2 \in P_2 \text{ are compatible}\}$. For $X \subseteq \mathsf{Dom}(\eta)$, the *projection of* η *on* X, written as $\pi_X(\eta)$, is the binding η' with $\mathsf{Dom}(\eta') = X \cap \mathsf{Dom}(\eta)$ and $\eta'(x) = \eta(x)$, for every $x \in \mathsf{Dom}(\eta')$. For a set P of bindings, the *projection of* P *on* X is $\pi_X(P) = \{\pi_X(\eta) \mid \eta \in P\}$.

A *match for* q *in* D is a binding of $\mathsf{Var}(q)$ such that $A_i(\eta(\bar{u}_i)) \in D$ for every $i \in \{1, \ldots, n\}$.[1] The set of *answers to* q *on* D is $q(D) = \{\pi_{\mathsf{FVar}(q)}(\eta) \mid \eta \text{ is a match for } q \text{ in } D\}$. We define answers as functions instead of database tuples, as this simplifies the presentation and as reasoning about their underlying domains is useful in further sections, when dealing with query decompositions.

Threshold Queries. A *threshold query (TQ)* is an expression t of the form

$$q(\bar{x}) \land \exists^{a,b} \bar{y}\ p(\bar{x}, \bar{y})$$

where, from left to right, $q(\bar{x})$ is a CQ, $a \in \mathbb{N}$, $b \in \mathbb{N} \cup \{\infty\}$, and $p(\bar{x}, \bar{y})$ is a CQ in which we do not require that every variable in $\bar{x}$ appears in one of its atoms. Notice that a TQ only has a single counting quantifier $\exists^{a,b}$, although further ordinary existential quantifiers may occur inside q and p. We use $\exists^{\geq a}$ and $\exists^{\leq b}$ as shorthands for $\exists^{a,\infty}$ and $\exists^{0,b}$, respectively, and the corresponding threshold queries will be called *at-least* and *at-most* queries. Similar to CQs, we usually denote the entire query as $t(\bar{x})$ or even as t when $\bar{x}$ is clear from the context. When representing $t(\bar{x})$ for decision problems, we assume that the numbers a and b are given in binary.

As an example, recall the Nobel Prize threshold query in Section 1, and suppose that the schema is *NobelPrize*(<u>*id*</u>, *year*, *category*) and *Laureate*(<u>*nid*</u>, <u>*name*</u>, *country*) with the foreign key constraint *Laureate*[*nid*] $\subseteq$ *NobelPrize*[*id*]. This threshold query can be formalized as follows.

$$t(x) = \exists x_1, x_2.\ \mathit{NobelPrize}(x, x_1, x_2) \land \exists^{\geq 4} y.\ \exists z.\ \mathit{Laureate}(x, y, z).$$

The set of answers of t on D, written $t(D)$, is the set of answers η of $q(\bar{x})$ on D that have between a and b compatible answers of $p(\bar{x}, \bar{y})$. Formally, $\eta \in t(D)$ iff $\eta \in q(D)$ and $a \leq |p(D, \eta)| \leq b$, where $p(D, \eta) = \{\eta' \in p(D) \mid \eta' \text{ is } \textit{compatible} \text{ with } \eta\}$.

If t is a threshold query of the form above, we call $\bar{x}$ the *free variables* (or *answer variables*) of the query t and we write $\mathsf{FVar}(t)$ for the set of these variables. We call $\bar{y}$ the *tally variables* of the query t and we write $\mathsf{TVar}(t)$ for the set of these variables. For the ease of presentation we shall assume that the sets of existentially quantified variables in q and p are disjoint; we shall write $\mathsf{QVar}(t)$ for the union of these sets. Thus, $\mathsf{Var}(t)$ is the union of three disjoints sets: $\mathsf{FVar}(t)$, $\mathsf{TVar}(t)$, and $\mathsf{QVar}(t)$.

A threshold query t can be intuitively expressed as a SQL query defining $q(\bar{x})$ such that in the WHERE clause we additionally check

[1]Notice that here we also denote by η the extension of η that is the identity on Const, and its extension thereof to tuples of variables and constants.

if the number of tuples returned by a correlated SELECT-FROM-WHERE subquery defining $p(\bar{x}, \bar{y})$ is at least a and at most b.

Graph Databases. For the purposes of this paper, graph databases can be abstracted as relational databases with unary and binary relations. That is, a graph database can be seen as a database G with a unary relation Node and binary relations $A, B, \ldots$ where

- $\text{Node}(a) \in G$ if a is a node of the graph database and
- $A(a_1, a_2) \in G$ if (a_1, a_2) is an edge with label A in the graph database.

An important feature that distinguishes graph database queries from relational database queries are *regular path queries (RPQs)* [14]. In a nutshell, a regular path query is an expression of the form $r(x, y)$, where r is a regular expression over the set of edge labels in the graph database. When evaluated over a graph database G, the query returns all node pairs (u, v) such that there exists a path from u to v that is labeled with some word in the language of r.

All results in the paper extend naturally to RPQs and therefore to so-called conjunctive regular path queries (CRPQs) [14]. This means that we can generalize CQs to CRPQs, which are defined exactly the same as CQs, but we additionally allow RPQs at the level of atoms. Generalizing threshold queries is analogous. If we want to evaluate a threshold query with RPQs, we can pre-evaluate all RPQs in the query and treat their result as a *materialized view.* We can then treat the query as an ordinary threshold query in which each RPQ becomes an atom, which is evaluated over the corresponding materialized view.

3 EXPLOITING THRESHOLDS AT A GLANCE

Query evaluation is arguably the most fundamental problem in databases and comes in many variants, such as:

(E1) Boolean evaluation, i.e., testing existence of an answer;
(E2) returning a given number of answers;
(E3) counting the total number of answers;
(E4) sampling answers with uniform probability; and
(E5) enumerating the answers with small delay.

An important reason why all these variants are considered is that the set of answers to a query can be very large and one is not always interested in the set of all answers.

The computational cost of these variants tends to increase as we go down in the list, but already for CQs even the simplest problem (E1) is intractable [1, Chapter 15]. Triggered by Yannakakis' seminal result on efficient evaluation of *acyclic* CQs [71], the literature developed a deep understanding that teaches us that, essentially, *low tree-width* is not only helpful but even *necessary* for polynomial-time Boolean evaluation of CQs [39]. Intuitively, the tree-width of a CQ measures the likeness of its graph structure to a tree. In essence, this graph structure is obtained by taking the queries' variables as nodes and connecting variables with an edge iff they appear in a common atom. Queries with low tree-width are tree-like and queries with high tree-width are highly cyclic.

Example 3.1. Consider the following variant of the first query from the introduction:

$$q(x, y, z) \leftarrow \mathit{Assets}(x, y), \mathit{Subsidiary}(w, x), \mathit{Shareholder}(w, z)\,.$$

For the purpose of Boolean evaluation we can rewrite q as

$$\begin{aligned} q'() &\leftarrow \mathit{Assets}(x, y), U(x)\,; \\ U(x) &\leftarrow \mathit{Subsidiary}(w, x), V(w)\,; \\ V(w) &\leftarrow \mathit{Shareholder}(w, z) \end{aligned}$$

using views U and V. It is clear that one can materialize the views and answer q' in time $O(n \cdot \log n)$ over databases of size n. The tree-width of q manifests itself as the number of variables used in the definitions of the views. For CQs of tree-width d, views in the optimal rewriting will use up to d variables and the data complexity will be $O(n^d \cdot \log n)$. $\triangleleft$

For threshold queries, however, low tree-width is not sufficient. The reason is that it is already hard to decide if the number of results of an acyclic CQ is above a threshold (represented in binary).

Proposition 3.2. *Given an acyclic conjunctive query q, a threshold k in binary representation, and a database D, testing if q returns at least k tuples on D is* coNP*-hard.*

So, we cannot have a polynomial-time algorithm even for evaluating Boolean acyclic threshold queries of the form $\exists^{a,b}\bar{y}\, p(\bar{y})$, unless P = NP. This is why one focus in the paper is on

pseudopolynomial-time algorithms
for threshold queries of low tree-width.

We call an algorithm *pseudopolynomial*, if it is a polynomial-time algorithm assuming that the numerical values a and b in "$\exists^{a,b}\bar{y}$" are represented in unary (instead of binary). For instance, a pseudopolynomial algorithm can evaluate threshold queries of the form $\exists^{a,b}\bar{y}\, p(\bar{y})$ by keeping $b + 1$ intermediate results in memory, which is not possible in a polynomial-time algorithm.

Let us revisit Example 3.1 and rewrite the query in a suitable way to produce its output. The next rewriting is inspired by research on constant-delay enumeration and answer counting for CQs.

Example 3.3. The rewriting in Example 3.1 is not suitable for non-Boolean evaluation because it projects out answer variables. The only way to rewrite q while keeping track of all answer variables is

$$\begin{aligned} q''(x, y, z) &\leftarrow \mathit{Assets}(x, y), U(x, z)\,; \\ U(x, z) &\leftarrow \mathit{Subsidiary}(w, x), \mathit{Shareholder}(w, z)\,. \end{aligned}$$

The standard approach for constant-delay enumeration algorithms [5] first has a preprocessing phase, in which it materializes U and groups *Assets* and U by x. In the enumeration phase it iterates over possible values of x and, for each value of x, over the contents of the corresponding groups of *Assets* and U. The complexity of the preprocessing phase is then affected by the cost of materializing U, which can be quadratic in the worst case. Overall, the complexity is $O(n^2 \cdot \log n)$ over databases of size n. $\triangleleft$

Evaluating a threshold query is closely related to constant-delay enumeration and counting answers to CQs. Indeed, a threshold query of the form $\exists^{a,b}\bar{y}\, p(\bar{y})$ can be evaluated by enumerating answers to p up to threshold $b + 1$ or by counting all answers to p. For both these tasks, however, tractability relies on more restrictive parameters of the query. For enumeration, tree-width needs to be replaced with its *free-connex* variant [5], which treats answer variables in a special way. For counting, the additional parameter is the *star-size* [32]. Intuitively, it measures how many answer variables are maximally connected to a non-answer variable. The query q has star-size 2 because the existentially quantified variable w is connected to two answer variables, x and z.

Thus, in the general approaches to constant-delay enumeration and counting, complexity is very sensitive to the interaction between answer and non-answer variables. Our key insight is that in the presence of a threshold this is no longer the case and low tree-width is sufficient.

Example 3.4. Consider again query q from Example 3.1 and suppose that we should return up to c answers. We can rely on the rewriting in Example 3.1, but we need to store additional information when materializing the views. For each w in V we store up to c witnessing values of z such that $\mathit{Shareholder}(w, z)$ holds. Similarly, for each x in U we store up to c values of z that were stored as witnesses for some w in V with $\mathit{Subsidiary}(w, x)$. Now, we can obtain up to c answers to q by taking the join of $\mathit{Assets}(x, y)$ with $U(x)$ and iterating through the witnessing values of z for each x. Both extended materialization steps, as well as the final computation of answers, can be realized in time $O\big(c \cdot n \cdot \log(c \cdot n)\big)$. If we are to count answers up to threshold c, we can just count the ones returned by the algorithm above. $\lhd$

This idea allows evaluating low tree-width TQs of the form $\exists^{a,b}\bar{y}\, p(\bar{y})$ in pseudopolynomial-time. It is also crucial in our treatment of general TQs of the form $q(\bar{x}) \wedge \exists^{a,b}\bar{y}\, p(\bar{x}, \bar{y})$ but, as the following proposition shows, we cannot expect pseudopolynomial evaluation algorithms even for *acyclic* threshold queries. Our proof uses a reduction from MINSAT [52].

Proposition 3.5. *Boolean evaluation of acyclic at-least and at-most threshold queries is* NP*-hard, even if thresholds are given in unary.*

The reason is that acyclic queries can have arbitrarily high *free-connex tree-width*, which is the actual source of hardness. For TQs of *bounded free-connex tree-width* our approach will yield pseudopolynomial evaluation algorithms for all variants (E1)–(E5). That is, our results are tight in terms of combined complexity.

In the remainder of the paper, we will analyze algorithms using $\tilde{O}$-notation. We will use this notation to reflect the *data complexity* of the algorithms and to hide *logarithmic factors*. Essentially, using $\tilde{O}$ allows us to freely use sorting and indexes such as B-trees. For instance, if we say that something can be done in time $\tilde{O}(n^2)$ we mean that its data complexity is in time $O(n^2 \log n)$.

4 THRESHOLD QUERIES IN THEORY

In this section we define the notions of widths and decompositions informally discussed in Section 3, explore in depth the approach to threshold queries via exact counting, develop the idea illustrated in Example 3.4 to cover arbitrary CQs in a slightly more general setting involving grouping, and employ the obtained algorithm to construct a single data structure that supports constant-delay enumeration, counting, and sampling answers to TQs.

4.1 Tree Decompositions and How to Find Them

The rewritings discussed in Section 3 are guided by tree decompositions of queries. A *tree decomposition* of a conjunctive query q is a finite tree T with a set $X_v \subseteq \mathsf{Var}(q)$, called a *bag*, assigned to each node v of T, satisfying the following conditions:

(1) for each atom A of q there exists a node v of T such that $\mathsf{Var}(A) \subseteq X_v$ (we say that v *covers* A);
(2) for each variable $x \in \mathsf{Var}(q)$, the set of nodes v of T such that $x \in X_v$ forms a connected subgraph of T.

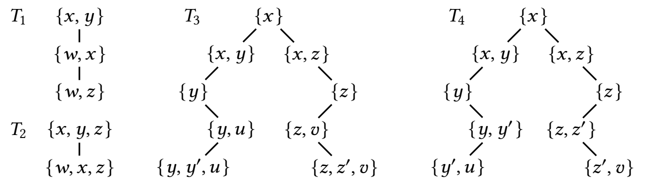

Figure 1: Tree decompositions.

By the *width of* T we shall understand $\max_{v \in T} |X_v|$.[2] The *tree-width of query* q, written as $\mathrm{tw}(q)$, is the minimal width of a tree decomposition of q. For example, T_1 in Figure 1 is a tree decomposition of width 2 for the query q in Example 3.1. Since q contains atoms involving two variables, it does not admit a tree decomposition of width 1. Hence, $\mathrm{tw}(q) = 2$.

A tree decomposition T of a query q is X*-connex* for a set $X \subseteq \mathsf{Var}(q)$, if there exists a connected subset U of nodes of T, containing the root of T, such that the union of bags associated to nodes in U is precisely X. Note that there is exactly one such U that is maximal: it is the one that includes all nodes u with $X_u \subseteq X$. We shall refer to it as *the maximal* X*-connex set in* T. The X*-connex tree-width* of q is the minimal width of an X-connex tree decomposition of q. If $X = \mathsf{FVar}(q)$, we speak of *free-connex decompositions* and *free-connex tree-width*; we write $\text{fc-tw}(q)$ for the free-connex tree-width of q. For instance, T_2 in Figure 1 is a free-connex tree decomposition of width 3 for the query q of Example 3.1. It is not hard to see that q has no free-connex decomposition of width 2. That is, $\mathrm{tw}(q) = 2$ but $\text{fc-tw}(q) = 3$. This difference can be arbitrarily large, e.g., for the CQs we used in the proof of Proposition 3.2.

A tree decomposition T of q is X*-rooted*, for $X \subseteq \mathsf{Var}(q)$, if the root bag of T is exactly X. The X*-rooted tree-width of* q is the minimal tree-width of an X-rooted tree decomposition. By analogy to free-connex, if $X = \mathsf{FVar}(q)$, we speak of *free-rooted decompositions* and *free-rooted tree-width*.

It is convenient to work with tree decompositions T of a special shape. A node u of T is: a *project* node if it has exactly one child v and $X_u \subsetneq X_v$; a *join* node if it has exactly two children, v_1 and v_2, and $X_u = X_{v_1} \cup X_{v_2}$. We say that u is *safe* if each variable in X_u occurs in an atom of q that is covered either by u or by a descendant of u. We say that T is *nice* if each node of T is either a leaf or a project node or a join node, and all nodes of T are safe.

Lemma 4.1. *Each tree decomposition T of a conjunctive query q can be transformed in polynomial time into a nice tree decomposition T' of the same width and linear size. Moreover, if T is X-rooted or X-connex, so is T'.*

[2]It is customary to define the width of decomposition T as $\max_{v \in T} |X_v| - 1$, to ensure that tree-shaped queries have tree-width 1. For the purpose of this paper we prefer not to do it, thus avoiding adjustments by 1 in multiple formulas.

We now introduce some notions that will be useful later. With each node u in a tree decomposition T of q we associate the subquery q_u obtained by taking all atoms of q over the variables appearing in the subtree of T rooted at u, and quantifying existentially all used variables except those in $X_u \cup \mathsf{FVar}(q)$. Let $\check{q}_u$ be the full CQ obtained by taking all atoms of q_u that do not occur in $q_{v_1} \wedge q_{v_2} \wedge \cdots \wedge q_{v_k}$, where $v_1, v_2, \ldots, v_k$ are the children of u; we then have $\mathsf{Var}(\check{q}_u) = \mathsf{FVar}(\check{q}_u) \subseteq X_u$. For instance, for the query q discussed in Section 3 and its tree decomposition T_1 shown in Figure 1 with nodes numbered 0, 1, 2 starting from the root, we have $q_0 = q = \exists w\, \mathit{Assets}(x,y) \wedge \mathit{Subsidiary}(w,x) \wedge \mathit{Shareholder}(w,z)$, $\check{q}_0 = \mathit{Assets}(x,y)$, $q_1 = \exists x\, \mathit{Subsidiary}(w,x) \wedge \mathit{Shareholder}(w,z)$, $\check{q}_1 = \mathit{Subsidiary}(w,x)$, and $q_2 = \check{q}_2 = \mathit{Shareholder}(w,z)$. In general, if u is a leaf then $q_u = \check{q}_u$ and if u is the root then $q_u = q$. One can evaluate q on a database D by computing $q_u(D)$ bottom up, as follows. Assuming that T is nice, if u is a leaf then

$$q_u(D) = \check{q}_u(D)\,,$$

if u is a project node with child v then

$$q_u(D) = \pi_{\mathsf{FVar}(q_u)}\big(q_v(D)\big)\,,$$

and if u is a join node with children v_1 and v_2 then

$$q_u(D) = \check{q}_u(D) \bowtie q_{v_1}(D) \bowtie q_{v_2}(D)\,.$$

If T is free-rooted then $\mathsf{FVar}(q_u) \subseteq X_u$ for each u and the above computation can be performed in time $\tilde{O}(|D|^d)$, where d is the width of T. The following simple fact will also be useful.

Lemma 4.2. *Let T be a nice tree decomposition of a CQ q.*

(1) For each node u in T it holds that $X_u \subseteq \mathsf{FVar}(q_u)$.
(2) If T is X-connex and U is the maximal X-connex set in T, then each node in U either has all its children in U or it is a U-leaf, that is, it has no children in U.

Tree decompositions are not easy to find. Indeed, determining if an arbitrary graph admits a tree decomposition of width at most d is NP-hard [3]. However, the problem has been studied in great depth and there is an ongoing effort of making these approaches practical at large scale. For instance, computing tree decompositions of large graphs was the topic of the PACE Challenge [29, 30] twice.

However, in the present context, we only want to compute *tree decompositions of queries*, which are very small in practice. There are libraries available [31, 36] that can find optimal tree decompositions of queries very efficiently. Indeed, DetkDecomp [31] was used to compute the tree-width of more than 800 million real-world queries [15–17] and worked very efficiently. Importantly for us, the analysis in [15–17] showed that real-life queries have very low tree-width.

Each algorithm for computing tree decompositions can be used also to compute X-rooted and X-connex tree decompositions, with quadratic overhead [5]. In our complexity estimations we rely on Bodlaender's algorithm [11], which allows computing optimal tree decompositions in linear time (assuming bounded tree-width).

4.2 Threshold Queries via Exact Counting

Processing a threshold query

$$t(\bar{x}) = q(\bar{x}) \wedge \exists^{a,b}\bar{y}\, p(\bar{x},\bar{y})$$

involves counting, for each answer η in $q(D)$, how many answers in $p(D)$ are compatible to η. Formally, this means that we need to determine the size of $p(D,\eta)$ for each $\eta \in q(D)$, where $p(D,\eta)$ is the set of answers in $p(D)$ that are compatible to η. So we need to solve the following computational problem for p.

Counting answers to p grouped by $X \subseteq \mathsf{FVar}(p)$ over database D consists in computing all pairs (η, k) such that $\eta : X \to \mathsf{Adom}(D)$ and $k = |p(D,\eta)|$.

Counting answers can leverage low *star-size* [32]. While the original notion is designed to fit hypertree decompositions, we shall work with a slightly faster growing variant that fits tree decompositions better and is much easier to define; the two variants coincide for queries using at most binary atoms. The *star-size* of a conjunctive query q, written $\mathsf{ss}(q)$, is the least positive integer f such that by grouping atoms and pushing quantifiers down, we can rewrite q as $q_1 \wedge q_2 \wedge \cdots \wedge q_\ell$ with $|\mathsf{FVar}(q_i)| \leq f$ or $\mathsf{QVar}(q_i) = \emptyset$ for all i. The following is a routine generalization of the result on counting answers [32].

Proposition 4.3. *Counting answers grouped by X for conjunctive queries of X-rooted tree-width d and star-size f over databases of size n can be done in time $\tilde{O}\big(n^{d\cdot f}\big)$.*

When we consider (constant-delay) *enumeration algorithms*, we see that the state-of-the-art approaches, e.g., [5, 42], use a different parameter of the query. Instead of bounded *star-size*, these rely on bounded *free-connex tree-width.* At first sight, this difference is not surprising, because in the absence of a threshold, counting and enumeration cannot be reduced to each other. But a closer look reveals that both parameters play a similar role: limiting them allows to reduce the problem to the much simpler case of full CQs by rewriting the input query as a join of views of bounded arity. This is readily visible for the star-size method, where the views correspond to the queries q_i in the definition of star-size of q, but it is also true for the free-connex method: there, the views correspond to subtrees of the decomposition rooted at the shallowest nodes holding an existentially quantified variable. Hence, the methods can be used interchangeably and we can replace star-size with free-connex tree-width in Proposition 4.3. Below, the *X-rooted free-connex tree-width* of a query is the minimal width of a tree decomposition of the query that is both X-rooted and free-connex.

Proposition 4.4. *Counting answers grouped by X for conjunctive queries of X-rooted free-connex tree-width d over databases of size n can be done in time $\tilde{O}\big(n^{d}\big)$.*

In particular, all answers to p can be counted in $\tilde{O}\big(n^{\mathsf{tw}(p)\cdot\mathsf{ss}(p)}\big)$ by Proposition 4.3 or in $\tilde{O}\big(n^{\mathsf{fc\text{-}tw}(p)}\big)$ by Proposition 4.4. The following lemma shows that the latter bound is tighter, so we shall rely on Proposition 4.4.

Lemma 4.5. *For each conjunctive query p,*

$$\mathsf{ss}(p) \leq \mathsf{fc\text{-}tw}(p) \leq \mathsf{tw}(p) \cdot \max\big(1, \mathsf{ss}(p)\big)\,.$$

The same holds for the X-rooted variant and the X-connex variant, for every $X \subseteq \mathsf{FVar}(p)$. Moreover, there exist CQs p with arbitrarily large $\mathsf{fc\text{-}tw}(p)$ that satisfy $\mathsf{fc\text{-}tw}(p) \leq \sqrt{\mathsf{tw}(p)\cdot\mathsf{ss}(p)} + 1$.

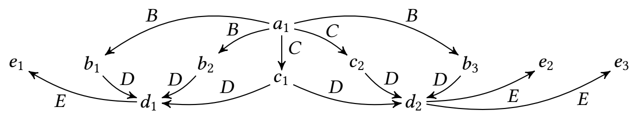

Figure 2: Database from Example 4.6.

Let us now come back to the threshold query t. By a tree decomposition of t we shall mean a tree decomposition of the associated CQ $q \wedge p$. Based on this, we define all variants of tree-width for threshold queries just like for CQs. Importantly, free-connex refers to $\mathsf{FVar}(t)$-connex, not $\mathsf{FVar}(q \wedge p)$-connex. Applying Proposition 4.4 requires an $\mathsf{FVar}(p)$-connex decomposition; that is, $\mathsf{FTVar}(t)$-connex in terms of t, where $\mathsf{FTVar}(t) = \mathsf{FVar}(t) \cup \mathsf{TVar}(t)$. Moreover, to avoid manipulating full answers, we need to factorize the evaluation of q and counting answers to p grouped by $\mathsf{FVar}(t)$ in a compatible way. This can be done if our decomposition is also $\mathsf{FVar}(t)$-connex. Putting it together, we arrive at free-connex $\mathsf{FTVar}(t)$-connex decompositions of t.

Example 4.6. Consider an at-least query

$$t(x,y,z) = B(x,y) \wedge C(x,z) \wedge \exists^{\geq 3}(u,v)\ r(y,u) \wedge s(z,v)$$

for $r(y,u) = \exists y'\, D(y,y') \wedge E(y',u)$, $s(z,v) = \exists z'\, D(z,z') \wedge E(z',v)$. Figure 1 shows a free-connex $\mathsf{FTVar}(t)$-connex tree decomposition T_3 of t of minimal width, which is 3. The subqueries $r(y,u)$ and $s(z,v)$ correspond to bags $\{y\}$ and $\{z\}$, respectively, and the subtrees rooted at these bags are $\{y\}$-rooted (resp. $\{z\}$-rooted) free-connex tree decompositions for these subqueries. Consider the input database in Figure 2. Let us count the answers to $r(y,u)$ grouped by y and the answers to $s(z,v)$ grouped by z (using Proposition 4.4) and store the resulting pairs in sets $R_{\{y\}}$ and $R_{\{z\}}$, respectively. We get $R_{\{y\}} = R_{\{z\}} = \{(b_1,1),(b_2,1),(b_3,2),(c_1,3),(c_2,2)\}$, where a pair (b,k) in $R_{\{y\}}$ means that for $y = b$ there are k witnessing values of u, and similarly for $R_{\{z\}}$. This is the initial information that we shall now propagate up the tree. In the set $R_{\{x,y\}}$ we put triples (a,b,k) such that $B(a,b)$ and for $y = b$ there are k witnessing values of u; analogously for $R_{\{x,z\}}$. We get $R_{\{x,y\}} = \{(a_1,b_1,1),(a_1,b_2,1),(a_1,b_3,2)\}$, $R_{\{x,z\}} = \{(a_1,c_1,3),(a_1,c_2,2)\}$. These sets can be computed based on $R_{\{y\}}$ and $R_{\{z\}}$, respectively. The set $R_{\{x\}}$ stores pairs (a,k) such that k is the maximal number of witnessing pairs of values for u and v that any combination of $y = b$ and $z = c$ with $B(a,b)$ and $C(a,c)$ can provide. In our case, $R_{\{x\}} = \{(a_1,6)\}$. Because b and c can be chosen independently from each other, we have $k = m \cdot n$, where $m = \max_a \{\ell \mid (a,b,\ell) \in R_{\{x,y\}}\}$ and $n = \max_b \{\ell \mid (a,c,\ell) \in R_{\{x,z\}}\}$. That is, the only information that needs to be passed from $\{x,y\}$ to $\{x\}$ is the set $R'_{\{x,y\}}$ of pairs (a,m) with m defined as above, and similarly for $\{x,z\}$. In our case, $R'_{\{x,y\}} = \{(a,2)\}$ and $R'_{\{x,z\}} = \{(a,3)\}$. We return YES iff $R_{\{x\}}$ contains a pair (a,k) with $k \geq 3$. In our case, it is so. ◁

Theorem 4.7. *Boolean evaluation of an at-most or at-least query t of free-connex $\mathsf{FTVar}(t)$-connex tree-width d over databases of size n can be done in time $\tilde{O}(n^d)$. The combined complexity of the algorithm is polynomial, assuming d is constant.*

In terms of combined complexity, Theorem 4.7 is optimal as both conditions imposed on tree decompositions are needed. Indeed, the hard TQs used in Proposition 3.5 have $\mathsf{FTVar}(t)$-connex tree-width 2; and the hard Boolean TQs stemming from Proposition 3.2 have free-connex tree-width 2.

Algorithm 1 Answers to p grouped by X up to threshold c

```
1: $T \longleftarrow$ a nice $X$-rooted tree decomposition of $p$
2: loop through nodes $u$ of $T$ bottom-up
3:     if $u$ is a leaf then $A_u \longleftarrow p_u(D)$
4:     if $u$ is a project node with child $v$ then
5:         $A_u \longleftarrow prune^c_{X_u}\big(\pi_{\mathsf{FVar}(p_u)}(A_v)\big)$
6:     if $u$ is a join node with children $v_1, v_2$ then
7:         $A_u \longleftarrow prune^c_{X_u}\big(A_{v_1} \bowtie A_{v_2} \bowtie \check{p}_u(D)\big)$
```

4.3 Threshold Problems for CQs

In terms of data complexity, however, we can improve Theorem 4.7. Here we exploit the presence of thresholds to handle queries with unbounded $\mathsf{FTVar}(t)$-connex tree-width. We will cover not only at-least and at-most queries, but arbitrary TQs, and we will solve not only Boolean evaluation, but also constant-delay enumeration, counting answers, and sampling, all based on a single data structure. The small price we have to pay is pseudopolynomial combined complexity. Our starting point is again the problem of counting grouped answers, but this time up to a threshold.

Counting answers to p grouped by $X \subseteq \mathsf{FVar}(p)$ up to a threshold c over database D consists in computing the set of pairs (η, k) such that $\eta : X \longrightarrow \mathsf{Adom}(D)$ and $k = \min(c, |p(D,\eta)|)$.

In the presence of a threshold, it is not impractical to solve counting by enumerating answers. This is what we shall do.

Computing answers to query p grouped by $X \subseteq \mathsf{FVar}(p)$ up to a threshold c over database D consists in computing a subset A of $p(D)$ that is *complete for X and c*; i.e., for each $\eta : X \longrightarrow \mathsf{Adom}(D)$ either $p(D,\eta) \subseteq A$ and $|p(D,\eta)| \leq c$, or $|p(D,\eta) \cap A| = c$.

Consider a CQ p, a set $X \subseteq \mathsf{FVar}(p)$ of grouping variables, and a database D. We will use the term *group* to refer to the set of answers to p that agree on the grouping variables X; that is, a subset of $p(D)$ of the form $p(D,\eta)$ for some $\eta : X \longrightarrow \mathsf{Adom}(D)$. If the X-rooted tree-width of p is d, then $|X| \leq d$ and the number of groups is $O(|D|^d)$. Consequently, if we can compute answers to p grouped by X up to threshold c in time $\tilde{O}(c \cdot |D|^d)$, then we can also count them within the same time, because we will get at most c answers per group. Additionally, for each $\eta : X \longrightarrow \mathsf{Adom}(D)$ with $p(D,\eta) = \emptyset$, we need to include $(\eta, 0)$ into the result; this does not affect the complexity bound. Hence, it suffices to show how to compute answers grouped by X up to threshold c. Having seen an illustrating example in Section 3, we are ready for the full solution.

Theorem 4.8. *Computing answers grouped by X up to a threshold c for conjunctive queries of X-rooted tree-width d over databases of size n can be done in time $\tilde{O}(c \cdot n^d)$.*

Proof sketch. Consider Algorithm 1. We begin by computing a nice X-rooted tree decomposition T of minimal width d, as described in Section 4.1. That is, each bag of T has size at most d and the root bag is X. Consider the queries p_u associated to nodes u

of T. By Lemma 4.2, $X_u \subseteq \mathsf{FVar}(p_u)$. By analogy to the evaluation algorithm described in Section 4.1, for each u we solve the problem of computing answers to p_u grouped by X_u up to threshold c over D; that is, we compute a subset $A_u \subseteq p_u(D)$ that is complete for X_u and c. The final answer is then the set obtained in the root.

If u is a leaf, then $X_u = \mathsf{FVar}(p_u) = \mathsf{Var}(p_u)$ and $p_u(D)$ itself is the only subset of $p_u(D)$ complete for X_u and c. Because $|\mathsf{Var}(p_u)| \le d$, one can compute $p_u(D)$ in time $\tilde{O}\big(|D|^d\big)$.

Consider a project node u and its unique child v. Then $X_u \subsetneq X_v$. To compute A_u the algorithm projects A_v on $\mathsf{FVar}(p_u)$ and then, using the operation $prune^c_{X_u}$, it groups the resulting set of bindings by X_u and keeps only c bindings from each group. It is clear that this can be done in time $\tilde{O}\big(c \cdot |D|^d\big)$.

Finally, consider a join node u with children v_1, v_2. As explained in Section 4.1, the set $p_u(D)$ is then the join of $p_{v_1}(D)$, $p_{v_2}(D)$, and $\check{p}_u(D)$. We compute A_u in exactly the same way, except that for each binding η of variables in X_u we only keep the first c bindings extending η and discard the remaining ones. Naive implementation takes time $\tilde{O}\big(c^2 \cdot |D|^d\big)$, but it is easy to modify the standard merge-join algorithm to achieve this in time $\tilde{O}\big(c \cdot |D|^d\big)$.

It is routine to check that each A_u is complete for X_u and c. □

Recall that d is very small in practice and most queries can be expected to be acyclic [16]. If we are just interested in returning answers up to a threshold c (no grouping), then the algorithm underlying the present theorem improves the state-of-the-art algorithm from $\tilde{O}(n^f)$ to $\tilde{O}(c \cdot n)$ for acyclic queries, where f is the *free-connex treewidth* of the query, which can be large even for acyclic queries.

4.4 Processing Threshold Queries

We now turn to processing general threshold queries. We give a unified approach to constant-delay enumeration, counting, and sampling, based on a single data structure that can be computed using the methods presented in Section 4.3.

Example 4.9. Consider again the database from Figure 2 and the query t from Example 4.6. This time we shall work with the free-connex tree decomposition T_4 of t shown in Figure 1; it has smaller width than T_3. Like before, the subqueries $r(y, u)$ and $s(z, v)$ correspond to bags $\{y\}$ and $\{z\}$, respectively, and the subtrees rooted at these bags are $\{y\}$-rooted (resp. $\{z\}$-rooted) tree decompositions for these subqueries, but they are not free-connex. We count the answers to r grouped by y and the answers to s grouped by z, up to threshold 3, as described in Section 4.3, and store the results in sets $R_{\{y\}}$ and $R_{\{z\}}$, respectively. Because the counts obtained in Example 4.6 were all below 3, $R_{\{y\}}$ and $R_{\{z\}}$ are just like before. Sets $R_{\{x,y\}}$ and $R_{\{x,z\}}$ are also computed like before. For Boolean evaluation we would now put into $R'_{\{x,y\}}$ all pairs (a, m) such that m is the maximal number of witnessing values u that any $y = b$ with $B(a, b)$ can provide. To support constant-delay enumeration we need to pass more information up the tree: we include all pairs (a, k) such that some $y = b$ with $B(a, b)$ can provide k witnesses (up to threshold 3); that is, we forget the values b, but we keep all values k (up to 3), not only the greatest of them. In our case, $R'_{\{x,y\}} = \big\{(a_1, 1), (a_1, 2)\big\}$ and $R'_{\{x,y\}} = \big\{(a_1, 2), (a_1, 3)\big\}$. In the root we store the set $R_{\{x\}}$ of values a such that some combination of $y = b$ and $z = c$ with $B(a, b)$ and $C(a, c)$ can provide at least 3 witnessing pairs of values for u and v. The set $R_{\{x\}}$ can be obtained by taking all a such that there exist $(a, m) \in R'_{\{x,y\}}$ and $(a, n) \in R'_{\{x,z\}}$ with $m \cdot n \ge 3$. In our case, $R_{\{x\}} = \{a_1\}$.

Algorithm 2 Record sets R_u for nodes $u \in U$ of decomposition T

```
 1: loop through nodes $u \in U$ of $T$ bottom-up
 2:     if $u$ is a $U$-leaf of $T$ then
 3:         $R_u \leftarrow \big(q_u(D) \times \{1\}\big) \bowtie p_u(D, \mathsf{FVar}(p_u) \cap \mathsf{FVar}(t), c)$
 4:     if $u$ is a project node of $T$ with child $v \in U$ then
 5:         $R_u \leftarrow \gamma_{witnessCount, X_u; \mathrm{sum}(multiplicity)}\big(R_v\big)$
 6:     if $u$ is a join node of $T$ with children $v_1, v_2 \in U$ then
 7:         $R_{v_1,v_2,u} \leftarrow R_{v_1} \overset{c}{\bowtie} R_{v_2} \ltimes \check{q}_u(D)$
 8:         $R'_{v_1,v_2,u} \leftarrow R_{v_1,v_2,u} \ltimes \check{p}_u(D) \uplus R_{v_1,v_2,u} \triangleright \check{p}_u(D)$
 9:         $R_u \leftarrow \gamma_{witnessCount, X_u; \mathrm{sum}(multiplicity)}\big(R'_{v_1,v_2,u}\big)$
10:     if $u$ is the root of $T$ then $R_u \leftarrow \sigma_{a \le witnessCount \le b}\big(R_u\big)$
```

In the enumeration phase, we iterate over all combinations of $(a, m) \in R'_{\{x,y\}}$ and $(a, n) \in R'_{\{x,z\}}$ with $a \in R_{\{x\}}$. For each such combination we iterate over corresponding $(a, b, m) \in R_{\{x,y\}}$ and $(a, c, n) \in R_{\{x,z\}}$ and return (a, b, c). In our case this gives (a_1, b_1, c_1), (a_1, b_2, c_1), (a_1, b_3, c_1), and (a_1, b_3, c_2). To access relevant (a, b, m) and (a, c, n) directly, we use an index that can be built while constructing $R'_{\{x,y\}}$ and $R'_{\{x,z\}}$ from $R_{\{x,y\}}$ and $R_{\{x,z\}}$

To support counting and sampling, we add multiplicities to the pairs stored in the nodes of the decomposition. Specifically, for each $(a, m) \in R'_{\{x,y\}}$ we also store the number of values b with $B(a, b)$ that provide n witnesses (up to 3). In our case, the multiplicity of $(a_1, 1)$ in $R'_{\{x,y\}}$ is 2, and all other multiplicities in $R'_{\{x,y\}}$ and $R'_{\{x,z\}}$ are 1. Similarly, for each $a \in R_{\{x\}}$ we store the number of combinations of b and c with $B(a, b)$ and $C(a, c)$ that provide at least 3 pairs of witnesses. In our case, the multiplicity of a_1 in $R_{\{x\}}$ is 4: the witnessing combinations are (b_1, c_1), (b_2, c_1), (b_3, c_1), and (b_3, c_2). The number of answers to t is the the sum of all multiplicities in the root. In our case it is 4. To sample an answer, we first sample $a \in R_{\{x\}}$ with weights given by the multiplicities. In our case we choose a_1 with probability 1. Then we sample $\big((a, m), (a, n)\big) \in R_{\{x,y\}} \times R_{\{x,z\}}$ such that $m \cdot n \ge 3$ with weights given by the product of the multiplicities of (a, m) in $R'_{\{x,y\}}$ and (a, n) in $R'_{\{x,z\}}$. In our case, $\big((a_1, 1), (a_1, 3)\big)$ is chosen with probability $\frac{1}{2}$, and both $\big((a_1, 2), (a_1, 3)\big)$ and $\big((a_1, 2), (a_1, 2)\big)$ with probability $\frac{1}{4}$. Finally, we sample (a, b, m) and (a, c, n) uniformly among relevant triples in $R_{\{x,y\}}$ and $R_{\{x,z\}}$, respectively, and we return (a, b, c). In our case, for $\big((a_1, 1), (a_1, 3)\big)$ we choose either $\big((a_1, b_1, 1), (a_1, c_1, 3)\big)$ or $\big((a_1, b_2, 1), (a_1, c_1, 3)\big)$ with probability $\frac{1}{2}$, and for $\big((a_1, 2), (a_1, 3)\big)$ and $\big((a_1, 2), (a_1, 2)\big)$ there is only one choice; overall, each answer is returned with probability $\frac{1}{4}$. ◁

Theorem 4.10. *For TQs of free-connex tree-width d, over databases of size n, one can: count answers in time $\tilde{O}(n^d)$; enumerate answers with constant delay after $\tilde{O}(n^d)$ preprocessing; and sample answers uniformly at random in constant time after $\tilde{O}(n^d)$ preprocessing. Assuming d is constant, the combined complexity of each of these algorithms is pseudopolynomial.*

Compared to [5, 32], this theorem provides the same complexity guarantees as for counting answers and enumerating answers for *conjunctive queries*, but we are able to generalize these to *threshold queries* (and add sampling). This generalization comes at the small cost of dependence on the value of the threshold. The results in [5, 32] can be strengthened to more refined measures such as *hypertreewidth*, but we believe that this is also true here.

Proof sketch. Consider a database D and a threshold query

$$t(\bar{x}) = q(\bar{x}) \wedge \exists^{a,b}\, \bar{y}\; p(\bar{x}, \bar{y})\,.$$

We write c for the threshold up to which we will be counting witnesses: if $b < \infty$, we let $c = b + 1$; if $b = \infty$, we let $c = a$. Let T be a nice free-connex tree decomposition of t, of width d, and let U be the maximal free-connex set in T.

Being a tree decomposition of $r(\bar{x}, \bar{y}) = q(\bar{x}) \wedge p(\bar{x}, \bar{y})$, up to dropping unused variables, T is a tree decomposition of both $p(\bar{x})$ and $q(\bar{x}, \bar{y})$. Let q_u and p_u be the corresponding subqueries associated to node u. With each node $u \in U$ we associate the set R_u of records that will provide information necessary to enumerate, count, and sample answers to t. A *record for* $u \in U$ is a triple (η, n, k) consisting of a binding $\eta : X_u \longrightarrow \mathsf{Adom}(D)$, a *multiplicity* $n > 0$, and a *witness count* $k \in \{0, 1, \ldots, c\}$ such that there are exactly n extensions η' of η to $\mathsf{FVar}(q_u) \cup \big(\mathsf{FVar}(p_u) \cap \mathsf{FVar}(t)\big)$ such that $\pi_{\mathsf{FVar}(q_u)}(\eta') \in q_u(D)$ and $k = \min\big(c, |p_u(D, \eta')|\big)$. We can interpret a record (η, n, k) as a binding extending η to two fresh variables, *multiplicity* and *witnessCount*, with values n and k.

By Lemma 4.2, each node in u either is a U-leaf or has all its children in U. Consequently, we can compute R_u for $u \in U$ bottom-up, as in Algorithm 2. In U-leaves, we use Theorem 4.8 to get a solution $p_u\big(D, \mathsf{FVar}(p_u) \cap \mathsf{FVar}(t), c\big)$ to the problem of counting answers to p_u grouped by $\mathsf{FVar}(p_u) \cap \mathsf{FVar}(t)$ up to threshold c over D; the set $q_u(D)$ of all answers to q_u over D is computed as explained in Section 4.1. Higher up the tree, we compute R_u based on the values obtained for the children of u, by means of standard relational operators with multiset semantics, including grouping (γ) and antijoin ($\triangleright$), as well as the c-*join* $R_{v_1} \overset{c}{\bowtie} R_{v_2}$ defined as the multiset of records $\big(\eta_1 \bowtie \eta_2, n_1 \cdot n_2, \min(c, k_1 \cdot k_2)\big)$ such that $(\eta_1, n_1, k_1) \in R_{v_1}$, $(\eta_2, n_2, k_2) \in R_{v_2}$, and η_1 is compatible with η_2. Each R_u is computed in $\tilde{O}(c \cdot |D|^d)$. With all R_u at hand, we can enumerate, count, and sample answers to t as in Example 4.9. ◻

5 THRESHOLD QUERIES IN THE WILD

In this section we give evidence that threshold queries are indeed common and useful in practice. Furthermore, we present an experimental evaluation of our algorithm.

5.1 Quantitative Study on Query Logs

We first present some analytical results on large-scale real-world query logs from Wikidata's SPARQL query service [69]. Our study considers a corpus of more than 560M queries. (Previous work has considered a subset in the order of 200M queries [15].) These logs contain a massive amount of real-life queries, which are classified into (1) *robotic* (high-volume, single-source bursts) and *organic* (human-in-the-loop) [58]. Furthermore, the logs distinguish between successful ("OK") and timeout ("TO") requests.

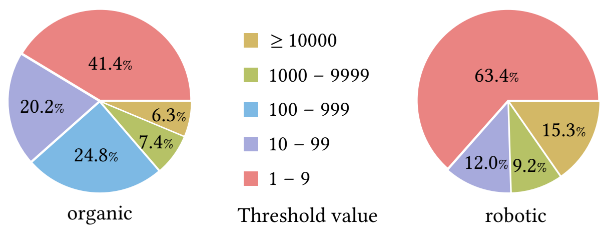

Figure 3: Threshold value occurrence ratio in *organic* and *robotic* CRPQ logs.

Occurrences of Threshold Queries. We first report on the usage of the keywords LIMIT and ORDER BY in the Wikidata logs, which contain ~563M well-formed queries, among which ~74M are unique (Table 1). Since our algorithms apply to CRPQs, we focus on those. As can be seen in the table, these still constitute 45.2% of the queries and 56.4% of the unique ones. In the remainder of this part, whenever we write a percentage as X% (Y%), then X refers to *all* and Y to the *unique* queries i.e., the set of queries after removing duplicates.

If we simply investigate how many CRPQs use LIMIT (columns *All* and *Unique*), these numbers are not so spectacular. Indeed, around 6% (4.5%) of the CRPQs use the LIMIT operator. What is remarkable though, is that almost all these queries that use LIMIT, *do not* use an ORDER BY operation (bottom line of Table 1).

By looking at the data more closely, we discovered that many of these queries are rather trivial in the sense that they only use one atom (or, equivalently, just one RDF triple pattern), which means that they do not even perform a join. For this reason, we decided to zoom in on the queries with *at least two* atoms, see the columns *All* (≥ 2) and *Unique* (≥ 2). It turns out that the numbers change significantly: around 14.9% (45.3%) of the CRPQs with at least two triples use LIMIT, which is a significant amount. Again, we see that almost all the queries that use LIMIT, do not use ORDER BY.

This is remarkable, because a commonly held belief is that LIMIT is most often used in combination with ORDER BY, i.e., as a means to express *top-k queries*. But, in this major real-world query log, this is not the case. Indeed, almost all non-trivial queries using LIMIT are threshold queries, seeking to return just an arbitrary *unranked* sample of results. In fact, we have run the same analysis on a broader class of queries (CRPQ$_f$, which extend CRPQ with unary filter conditions) and the percentages were very similar.

LIMIT Values. We now investigate the *values* of the thresholds of queries in the logs. To this end, we considered the subset of CRPQs in the raw logs that use the keyword LIMIT (~15.2M queries, including duplicates). To gain deeper insight, we break down the logs into robotic (~15.2M) and organic (~44k) queries. Fig. 3 shows the relative shares of threshold values with varying numbers of digits. The figure shows that threshold values between 1 and 9 are the most common. Still, in both organic and robotic queries, we see that large threshold values (≥ 10K) are not uncommon. Since robotic logs can have large bursts of similar queries, we see that the distribution is not as smooth as for organic logs. For instance, only 0.08% of queries in the robotic logs have three-digit limit values (depicted in blue), whereas there are much more (9.2%) with four-digit values (depicted in green). The largest value we found in

Table 1: Statistics of Wikidata query logs. Percentages in the top half are relative to the total number of queries. Percentages in the bottom half are relative to the number of CRPQs.

	Query Class	*All*		*Unique*		*All* (≥ 2)		*Unique* (≥ 2)	
	All Queries	563,066,025	100.0%	74,060,492	100.0%	254,802,446	100.0%	30,502,206	100.0%
	CRPQ	254,241,512	45.2%	41,778,884	56.4%	88,390,479	34.7%	3,346,280	11.0%
With *Limit*	CRPQ	15,225,933	6.0%	1,896,811	4.5%	13,205,975	14.9%	1,514,221	45.3%
With *Limit*, no *Order*	CRPQ	15,214,527	6.0%	1,894,879	4.5%	13,195,363	14.9%	1,512,629	45.2%

robotic logs had 9 digits. In organic logs, however, we found three limit values containing 11, 14, and 17 digits, respectively.

Conclusion of Quantitative Study. Threshold queries are indeed quite common, e.g., in querying knowledge bases such as Wikidata. Since the actual values of the thresholds are typically small, our empirical study confirms the utility in practice of our pseudopolynomial algorithm (Theorem 4.10) that, in order to evaluate queries with a threshold k, solely needs to maintain up to $k+1$ intermediate results per each candidate answer tuple. This is in contrast with traditional query plans where the number of intermediate results per candidate answer tuple is determined by the input data and therefore potentially orders of magnitude larger.

5.2 Qualitative Study

To demonstrate further the usefulness of threshold queries in practice across diverse contemporary domains, we also performed a qualitative study on two real-world graph datasets.

Covid-19 Dataset. The Covid-19 Knowledge Graph [26] is a continuously evolving dataset, with more than 10M nodes and 25M edges, obtained by integrating various data sources, including gene expression repositories (e.g., the Genotype Tissue Expression (GTEx) and the COVID-19 Disease Map genes), as well as article collections from different scientific domains (ArXiv, BioRxiv, MedRxiv, PubMed, and PubMed Central). The inferred schema of this graph exhibits more than 60 distinct node labels and more than 70 distinct edge labels [54]. Such typing information is, however, not sufficient to express the domain-specific constraints that can be found in these real-life graph datasets. Non-trivial constraints expressible with threshold queries can be naturally crafted in order to complement the schema, as we showcase in our study.

Wikidata Dataset. Wikidata is a collaborative knowledge base launched in 2012 and hosted by the Wikimedia Foundation [67]. By the efforts of thousands of volunteers, the project has produced a large, open knowledge base with numerous applications. Wikidata can be seen as a graph database with a SPARQL endpoint that lets users and APIs run queries on its massive knowledge graph. The query logs collected along the years on this endpoint [53] constitute a useful resource for the qualitative analysis of threshold queries.

Working Method. We have manually inspected the Covid-19 Knowledge Graph schema in search of constraints that can be validated with threshold queries. We have also found a number of threshold queries by sieving through a large sample of Wikidata query logs. We analyzed the structural properties of the collected threshold queries. Below we discuss our findings with the help of

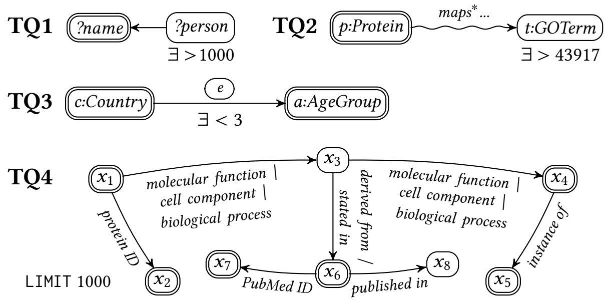

Figure 4: Structural diagrams of selected threshold queries.

five selected threshold queries from the two datasets. The queries are depicted in Figure 4. We focus on the structure of the underlying graph patters, omitting most of the labels. Constants are in quotes, variables are in rounded rectangles, and output variables have a double edge. Wavy edges represent path expressions in the original query. For readability, we sometimes change the labels of nodes.

Selected Queries. The first example comes from the Wikidata logs and appears to be a *data exploration* query.

TQ1 *Return all given names with more than 1000 occurrences.*

```
SELECT ?name WHERE { ?person <given_name> ?name }
GROUP BY ?name  HAVING ( ( COUNT (*) > 1000 ) );
```

A structurally similar query can help detect *integrity violations* in the Covid-19 Knowledge Graph. Namely, the latest release (June 2021) of the Gene Ontology (GO) contains 43917 valid terms [24, 25]. Therefore, in the Covid-19 Knowledge Graph one can check whether a protein is suspicious if it exhibits more than this number of GO terms associated to it.

TQ2 *Find each protein that has more than* 43917 *associated gene ontology terms.*

A formulation of this query in Cypher-like syntax follows.[3]

```
MATCH (p:Protein)-[:MAPS* . :HAS_ASSOCIATION .
                   (:IS_A*|PART_OF*)]->(t:GOTerm)
WITH p, COUNT(DISTINCT t) AS count_go
WHERE count_go > 43917 RETURN p;
```

Notice the large threshold and the complex regular expression.

As another example, reporting coverage in the Covid-19 Knowledge Graph demands that for each age group, in each country, there should be at least three reports for the current number of Covid cases (one for females, one for males, and one for the total). Deviations can be identified with the following threshold query.

[3]Note that Cypher does not currently support concatenation (`:A . :B`).

TQ3 *Find each country that does not have three reports for some age group.*

This query can be expressed in Cypher-like syntax as follows.

```
MATCH (c:Country)-[e:CURRENT_FEMALE|CURRENT_MALE
                   |CURRENT_TOTAL]->(a:AgeGroup)
WITH c, a, COUNT(type(e)) AS ecount
WHERE ecount < 3 RETURN c, a;
```

We point out that this query although acyclic is not free-connex.

We also discovered that graph patterns in limit-k queries can be quite large, as in the following query from the Wikidata logs.

TQ4 *Return 1000 tuples containing names (x1) and UniProt IDs (x2) for genes with domains (x3) referenced in "Science", "Nature," or "Cell" articles, as well as their indirect domain label (x4) and class (x5), the article name (x6), and its PubMed ID (x7).*

Conclusion of Qualitative Study. Our qualitative study discovered several interesting and realistic uses of threshold queries that we illustrated here. Connecting these to our algorithms, it is interesting to note that all these queries are acyclic, but only TQ1-TQ2 are free-connex (Fig. 4). For instance, TQ4 has star size 3 (due to x_3, which has 3 neighboring nodes that propagate to the output) and free-connex tree-width 4. Evaluation of such queries with existing querying approaches may incur significant cost while our algorithms handle them very efficiently. For instance, TQ4 can be evaluated in linear time by Algorithm 1 (up to logarithmic factors), while the approach via enumeration [5] requires at least cubic time.

5.3 Experimental Study

As a proof-of-concept, we implemented the algorithm in Theorem 4.8 in SQL and compared it with the optimizer of a popular DBMS engine, namely the PostgreSQL 13.4 optimizer. Using SQL windowing functions, our implementation can mimic some important aspects of our query evaluation algorithm (like internal information passing up to a threshold), but we note that SQL does not allow to capture our algorithm precisely. As shown in the remainder, the results of this comparison already show the superiority of our algorithm for threshold queries compared to naive evaluation.

All the experiments were executed on an Intel Core i7-4770K CPU @ 3.50GHz, 16GB of RAM, and an SSD. We used PostgreSQL 13.4 in Linux Mint 20.2 and built our own micro-benchmark [12] consisting of the following three types of query templates:

(q1) k-`path` selects up to 10 pairs of nodes linked by a k-hop path;
(q2) k-`neigh` selects all nodes with $\geq$ 10 k-hop neighbors;
(q3) k-`conn` selects all pairs of nodes linked by $\geq$ 10 k-hop paths.

Assuming that the database consists of a single binary relation R, then these queries for $k = 2$ can be naturally formulated in SQL as:

```
SELECT DISTINCT R1.A, R2.B FROM R AS R1, R AS R2
WHERE R1.B = R2.A LIMIT 10;

SELECT R1.A FROM R AS R1, R AS R2 WHERE R1.B = R2.A
GROUP BY R1.A HAVING COUNT(DISTINCT R2.B) >= 10;

SELECT X0, X2 FROM
   (SELECT DISTINCT R1.A AS X0, R1.B AS X1, R2.B AS X2
   FROM R AS R1, R AS R2 WHERE R1.B = R2.A) AS SUB
GROUP BY X0, X2 HAVING COUNT(*) >= 10;
```

In the following, we will refer to these formulations as the *baseline* formulations of the queries.

We compared these queries with alternative formulations in SQL that mimic crucial aspects of our evaluation algorithm and that can be understood as follows. A simple decomposition for k-`path` is a tree with a single branch and one node per each joined copy of table R. For this decomposition and $k = 2$ the algorithm in Theorem 4.8 amounts to evaluating the following query:

```
WITH J1 AS (SELECT DISTINCT A AS X1, B AS X2 FROM R),
     W1 AS (SELECT X1, X2, ROW_NUMBER() OVER
                 (PARTITION BY X1) AS RK FROM J1),
     S1 AS (SELECT X1, X2 FROM W1 WHERE RK <= 10),
     J2 AS (SELECT DISTINCT R.A AS X0, X2
           FROM R, S1 WHERE R.B = S1.X1),
     W2 AS (SELECT X0, X2, ROW_NUMBER() OVER
                 (PARTITION BY X0) AS RK FROM J2),
     S2 AS (SELECT X0, X2 FROM W2 WHERE RK <= 10)
SELECT X0, X2 FROM S2 LIMIT 10;
```

For k-`neigh` we can use the same decomposition and the corresponding query is the same except for the last line which is

```
SELECT X0 FROM S1 GROUP BY X0 HAVING count(*)>=10;
```

For k-`conn` we can also use the same decomposition but the corresponding query is a bit different, as we need to collect the whole paths, not just their endpoints. We refer to these alternative implementations as the *windowed* versions of the queries. Natural query plans for the windowed versions have worst-case complexity $\tilde{O}(k \cdot n)$ for k-`path` and k-`neigh`, and $\tilde{O}(k \cdot n^2)$ for k-`conn`.

Datasets. We considered two kinds of data sets:

(1) The real-world IMDb data set used in Join Order Benchmark [55], which contains information about movies and related facts about actors, directors, production companies, etc. In our experiments, we focused on the *movie_link* relation, which has a graph-like structure, so that finding paths is meaningful.
(2) Barabási-Albert graphs [7], which are synthetic data sets that model the structure of social networks, with varying parameters of n (total number of nodes to add) and m_0 (the number of edges to attach from newly added nodes to existing nodes).

We repeated each experiment several times and report the median.

First Experiment. We compared the running time of the baseline and windowed versions of (q1–q3) for varying values of k on both the IMDb and synthetic data sets, both on fully indexed and non-indexed databases. Table 2 shows the results for the non-indexed case. We see that our approach (windowed) outperforms the baseline with speedups up to three orders of magnitude, while the baseline times out (T/O) for higher values of k. The running times of the windowed versions reflect the good theoretical bounds, while the baseline clearly shows exponential dependence on k. For the indexed case, leveraging the DBMS's default indexes, the baseline ran only marginally faster (~5%), remaining three orders of magnitude slower than our algorithm.

Second Experiment. In this second experiment, we wanted to assess the impact of the structure and size of the data on the runtimes of our algorithm. To this purpose, we have employed the Barabási-Albert synthetic graphs with varying outdegree (m_0) and number of nodes (n) and considered query q2 with $k = 3$. Table 3 shows the different speedups that the windowed approach offers, when compared to the baseline. We varied m_0 from 5 to 25, using increments of 5, and varied n from 32 to 1M in a logarithmic

Table 2: Experimental evaluation for the baseline (b) and windowed (w) versions of (q1–q3) on the IMDb database (left) and on Barabási-Albert graphs (right) with $m_0 = 10$ and $n = 3000$. Time is measured in ms. Better running time is depicted in bold.

k	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10
q1/b	**39**	277	5,157	103,449	T/O	T/O	T/O	T/O	T/O	T/O	**74**	549	3,364	19,161	106,741	601,279	T/O	T/O	T/O	T/O
q1/w	**39**	**58**	**69**	**88**	**97**	**115**	**132**	**148**	**162**	**176**	80	**196**	**298**	**404**	**537**	**665**	**1,260**	**1,478**	**1,547**	**1,675**
q2/b	50	349	5,742	134,970	T/O	T/O	T/O	T/O	T/O	T/O	**138**	1,296	11,207	90,199	644,860	T/O	T/O	T/O	T/O	T/O
q2/w	**44**	**63**	**74**	**93**	**104**	**119**	**137**	**150**	**167**	**180**	152	**282**	**399**	**506**	**579**	**653**	**1,235**	**1,390**	**1,540**	**1,667**
q3/b	**94**	**652**	11,568	267,176	T/O	T/O	T/O	T/O	T/O	T/O	**273**	**2,423**	**18,670**	122,894	T/O	T/O	T/O	T/O	T/O	T/O
q3/w	146	690	**2,695**	**5,266**	**10,679**	**18,400**	**31,020**	**48,942**	**74,795**	**103,054**	420	4,201	**23,903**	**57,555**	**109,764**	**189,893**	**301,510**	**390,968**	**510,171**	**576,009**

Table 3: Experimental evaluation of the speedup factor (the ratio of baseline to windowed) for q2 on Barabási-Albert graphs for varying n and m_0.

$m_0 \backslash n$	32	100	316	1k	3.2k	10k	32k	100k	316k	1M
5	0.6	1.8	3.0	3.6	4.6	5.7	6.7	7.7	8.7	10.0
10	1.4	6.8	13.8	22.8	29.7	37.1	40.0	73.5	T/O	T/O
15	2.1	16.3	46.6	66.2	90.5	120.4	T/O	T/O	T/O	T/O
20	1.1	35.4	96.0	149.1	192.8	404.3	T/O	T/O	T/O	T/O
25	0.6	55.2	184.7	272.5	351.7	T/O	T/O	T/O	T/O	T/O

scale with increment factor of $\sqrt{10} \approx 3.16$. In the table, we see that the speedup factor of the windowed approach increases by up to three orders of magnitude as the size of the data and out-degree m_0 increase. T/O means that the baseline approach timed out (> 30 minutes). For all entries in Table 3, the windowed algorithm terminated under 15 minutes. These results show the robustness of our algorithm to variations of dataset size and outdegree as well as its superiority with respect to the baseline.

6 RELATED WORK

Top-k and Any-k. The potential of predefined thresholds to speed up query processing was first noticed by Carey and Kossmann [18], who explored ways of propagating thresholds down query plans, dubbed *LIMIT pushing*. This early study only considered applying the thresholds directly to subplans, which made joins a formidable obstacle. In contrast, we first group the answers to subqueries by variables determined based on the structure of the whole query, and then apply thresholds within groups; this way we can push thresholds down through multi-way joins, guided by a tree decomposition of the query. Most follow-up work concerns the ranked scenario, where the goal is to compute *top-k* answers according to a specified preference order. The celebrated Threshold Algorithm [33] solves the top-k selection problem: it operates on a single vertically-partitioned table, with each partition being managed by a different external service that only knows the scores of base tuples in its own partition, and produces k tuples with the highest score while minimizing the number of reads. There are also multiple approaches to the more general top-k join problem. J* [60] is based on the A* search algorithm: it maintains a priority queue of partial and complete join combinations ordered by the upper bounds of their scores. Rank-Join [43] maintains scores for complete join combinations only, and stops when new combinations cannot improve the current top-k. LARA-J* [59] offers improved handling of multi-way joins. FRPA [35] keeps the number of reads within a constant factor of the optimal. Overall, the focus and the main challenge in top-k processing is ordering the answers according to their ranking scores [44]. In the unranked case, when this challenge is absent, the rich body of work on top-k processing does not go beyond the initial observations made by Carey and Kossmann [18]. NeedleTail [50, 51] specifically focuses on providing *any-k* answers to queries without ORDER BY clauses, but it only handles key-foreign key joins, which dominate in the OLAP scenarios. In contrast, we support arbitrary CQs (i.e., select-project-join queries), allowing the complexity to grow with the tree-width (Theorem 4.8). Moreover, any-k evaluation of CQs is just a building block of the processing of much more general threshold queries.

Runtime optimization. A large body of research on query processing led to powerful optimization techniques, such as *aggregate pushing* [20, 21, 40, 70] and *sideways information passing* [10, 22, 46, 57, 62]. These techniques aim to speed up the execution of a given join plan and rely on a cost model to heuristically approximate instance-optimal plans. Our focus is on reducing the search space of the heuristic methods by identifying plans with good worst-case guarantees. Such plans can be further improved towards instance-optimality, using classical techniques. For LIMIT queries the combination of sideways information passing and LIMIT pushing might be beneficial. Indeed, if we can ensure that each tuple produced by the subplan extends to a full answer, then we can stop the execution of the subplan when the desired number of tuples is output. For general threshold queries, the potential for such optimization is less clear. There, instead of a global limit on the number of answers we have a per-group limit. Consequently, the execution of the subplan can be stopped only when each group has sufficiently many tuples. The level of savings depends on the order in which the subplan produces its results. Such optimization goes beyond the scope of our paper, but is a promising direction for future work.

Quantified Graph Patterns. Fan et al. [34] introduced *quantified graph patterns* (QGP) that allow expressing nested counting properties like having at least 5 friends, each with at least 2 pets. In contrast to threshold queries, QGPs are unable to count k-hop neighbours for $k \geq 2$, nor can they count tuples of variables. Moreover, QGPs adopt isomorphism matching, while threshold queries follow the standard semantics of database queries.

Aggregate Queries. In the context of factorized databases, Bakibayev et al. [6] observed that pushing aggregation down through joins can speed up evaluating queries. These results can be reinterpreted in the context of tree decompositions [61], but they optimize different aggregates in isolation and do not investigate the interplay between counting and existential quantification. AJAR (aggregations and joins over annotated relations) [47] and FAQs (functional aggregate queries) [49] are two very general sister formalisms capturing, among others, CQs enriched with multiple aggregate functions. Because different aggregate functions are never commutative, the evaluation algorithms for both these formalisms require decompositions consistent with the order of aggregating operations. For example, when applied to counting answers to CQs, this amounts to free-connex decompositions, as in our Proposition 4.4. In contrast, we remove the free-connex assumption by showing that counting up to a threshold and existential quantification can be reordered at the cost of keeping additional information of limited size.

Counting Answers. For *projection-free* CQs, the complexity of counting answers is tightly bound to tree-width [27, 37], just like in the case of Boolean evaluation [38, 39], but the presence of projection makes counting answers intractable even for *acyclic* CQs [63]. Efficient algorithms for counting answers require not only low tree-width but also low star-size [32]. However, when the problem is relaxed to randomized approximate counting, low tree-width is enough [2], just like for Boolean evaluation. Our results imply that for a different relaxation — counting exactly, but only up to a given threshold — CQs of low tree-width can also be processed efficiently. However, we go far beyond CQs and show how to count answers to threshold queries (which themselves generalize counting answers to CQs up to a threshold).

Enumerating Answers. Also in the context of constant-delay enumeration low tree-width is not enough to guarantee efficient algorithms: the query needs to have low *free-connex tree-width* [5]. Importantly, even acyclic queries can have very large free-connex tree-width. Tree-width with can be replaced with fractional hypertree-width [28, 48, 61] or submodular width [9] but always in the restrictive free-connex variant. Tziavelis et al. [64, 65] partially lift these results to the setting of *ranked enumeration*, where query answers must be enumerated according to a predefined order; the lifted results allow enumeration with *logarithmic* delay and handle projection-free CQs of low submodular width as well as free-connex acyclic CQs (but not general CQs). In this work, we show that if the number of needed answers is known beforehand, general CQs of low tree-width can be processed efficiently even if they have large free-connex tree width. Moreover, this result is only the starting point for processing general threshold queries, for which we also provide constant-delay enumeration algorithms.

Sampling Answers. Sampling query answers was identified as an important data management task by Chaudhuri at al. [19], who proposed a simple algorithm for sampling the join $S \bowtie T$ by sampling a tuple $s \in S$ with weight $|T \ltimes \{s\}|$ and then uniformly sampling a tuple $t \in T \ltimes \{s\}$. Using the *alias method* for weighted sampling [66, 68], this algorithm can be implemented in such a way that after a linear preprocessing phase, independent samples can be obtained in constant time. This approach was generalized to acyclic projection-free CQs [72]. We extend the latter result in three ways: we handle non-acyclic CQs, allowing the complexity to grow with the tree-width; we can allow projection, at the cost of replacing tree-width with its faster growing free-connex variant; and we handle threshold queries, rather than just CQs. A different approach to non-acyclic projection-free CQs [23] provides a uniform sampling algorithm with guarantees on the *expected* running time; this is incomparable to constant-time sampling after polynomial-time preprocessing, offered by our approach. Finally, Arenas et al. [2] show that efficient *almost uniform* sampling is possible for CQs of low tree-width. Here, *almost uniform* means that the algorithm approximates the uniform distribution up to a multiplicative error; this is a weaker notion than uniform sampling. Let us also reiterate that the all these papers only consider CQs, not threshold queries.

7 CONCLUSIONS AND FUTURE WORK

In this paper, we have embarked on a deep theoretical study of a newly identified class of *threshold queries.* Our extensive empirical study shows that threshold queries are highly relevant in practice as witnessed by their utility in real-world knowledge graphs and their presence in massive query logs. Our theoretical investigation shows that threshold queries occupy a distinctive spot in the landscape of database querying problems. Indeed, our complexity analysis proves that threshold queries allow for a more efficient evaluation than solutions for closely related problems of counting query answers, constant-delay query answer enumeration, and top-k querying.

As one of the first future steps, we intend to gauge thoroughly the performance of the proposed algorithms. Designing adequate protocols for experimental evaluation requires a deep understanding of the relationships between evaluation of threshold queries and the related querying problems, which we have already accomplished in the present paper. We intend to carry out a comprehensive implementation of threshold queries in an existing database system similarly to how algorithms of top-k queries have been implemented and evaluated [56]. More precisely, we will implement dedicated threshold-aware variants of relational operators and then we will introduce them in the query planning stage.

Our work has led us to identify remarkable similarities in the query answering methods tackling counting and enumerating answers: they rely on various techniques for compiling out existentially quantified variables. We plan to pursue this discovery further and give full treatment to the emerging question: is there a unifying framework for assessing threshold queries and the related problems? Further theoretical results about threshold queries can be envisioned, such as establishing a dichotomy of evaluation complexity and identifying a condition under which evaluation is strongly polynomial, rather than pseudopolynomial.

ACKNOWLEDGMENTS

This work stems from a larger effort on developing schemas for property graph databases, including cardinality constraints, led by the LDBC Property Graph Schema Working Group. We thank Andrea Calì, Leonid Libkin, Victor Lee, Juan Sequeda, Bryon Jacob, and Michael Schmidt for their useful comments and early feedback.

This work was partially supported by DFG grants 369116833 and 431183758 (Martens); NCN grant 2018/30/E/ST6/00042 (Murlak); ANR-21-CE48-0015 VeriGraph (Bonifati); EU Horizon 2020 research and innovation programme under grant agreement No 825041 (Fletcher).

REFERENCES

[1] Marcelo Arenas, Pablo Barceló, Leonid Libkin, Wim Martens, and Andreas Pieris. 2021. *Principles of Databases*. Open source at https://github.com/pdm-book/community (visited: 2022-01).

[2] Marcelo Arenas, Luis Alberto Croquevielle, Rajesh Jayaram, and Cristian Riveros. 2021. When is Approximate Counting for Conjunctive Queries Tractable?. In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing* (Virtual, Italy) *(STOC 2021)*. Association for Computing Machinery, New York, NY, USA, 1015–1027. https://doi.org/10.1145/3406325.3451014

[3] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. 1987. Complexity of finding embeddings in a k-tree. *SIAM JOURNAL OF DISCRETE MATHEMATICS* 8, 2 (1987), 277–284. https://doi.org/10.1137/0608024

[4] Raabia Asif and Mohammad Abdul Qadir. 2017. Enhancing the Nobel Prize schema. In *2017 International Conference on Communication, Computing and Digital Systems (C-CODE)*. IEEE, Islamabad,Pakistan, 193–198. https://doi.org/10.1109/C-CODE.2017.7918927

[5] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Proc. CSL 2007 (LNCS)*, Vol. 4646. Springer, Berlin, Heidelberg, 208–222. https://doi.org/10.1007/978-3-540-74915-8_18

[6] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. 2013. Aggregation and Ordering in Factorised Databases. *Proc. VLDB Endow.* 6, 14 (Sept. 2013), 1990–2001. https://doi.org/10.14778/2556549.2556579

[7] Albert-László Barabási and Réka Albert. 1999. Emergence of scaling in random networks. *Science* 286, 5439 (1999), 509–512. https://doi.org/10.1126/science.286.5439.509

[8] Leilani Battle and Carlos Scheidegger. 2021. A Structured Review of Data Management Technology for Interactive Visualization and Analysis. *IEEE Trans. Vis. Comput. Graph.* 27, 2 (2021), 1128–1138. https://doi.org/10.1109/TVCG.2020.3028891

[9] Christoph Berkholz and Nicole Schweikardt. 2019. Constant Delay Enumeration with FPT-Preprocessing for Conjunctive Queries of Bounded Submodular Width. In *Proc. MFCS 2019 (LIPIcs)*, Vol. 138. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 58:1–58:15. https://doi.org/10.4230/LIPIcs.MFCS.2019.58

[10] Philip A. Bernstein and Dah-Ming W. Chiu. 1981. Using Semi-Joins to Solve Relational Queries. *J. ACM* 28, 1 (1981), 25––40. https://doi.org/10.1145/322234.322238

[11] Hans L. Bodlaender. 1996. A Linear-Time Algorithm for Finding Tree-Decompositions of Small Treewidth. *SIAM J. Comput.* 25, 6 (1996), 1305–1317.

[12] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Sławek Staworko, and Dominik Tomaszuk. 2021. *Threshold queries in Python.* Zenodo. https://doi.org/10.5281/zenodo.5658389

[13] Angela Bonifati, Stefania Dumbrava, George Fletcher, Jan Hidders, Matthias Hofer, Wim Martens, Filip Murlak, Joshua Shinavier, Sławek Staworko, and Dominik Tomaszuk. 2021. Threshold Queries in Theory and in the Wild. arXiv:2106.15703 [cs.DB]

[14] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs.* Morgan & Claypool Publishers, Williston. https://doi.org/10.2200/S00873ED1V01Y201808DTM051

[15] Angela Bonifati, Wim Martens, and Thomas Timm. 2019. Navigating the maze of Wikidata query logs. In *The World Wide Web Conference*. Association for Computing Machinery, New York, NY, USA, 127–138. https://doi.org/10.1145/3308558.3313472

[16] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. An analytical study of large SPARQL query logs. *VLDB J.* 29, 2-3 (2020), 655–679. https://doi.org/10.1007/s00778-019-00558-9

[17] Angela Bonifati, Wim Martens, and Thomas Timm. 2020. SHARQL: Shape Analysis of Recursive SPARQL Queries. In *International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, New York, NY, USA, 2701–2704. https://doi.org/10.1145/3318464.3384684

[18] Michael J. Carey and Donald Kossmann. 1997. On Saying "Enough Already!" in SQL. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data* (Tucson, Arizona, USA) *(SIGMOD '97)*. Association for Computing Machinery, New York, NY, USA, 219–230. https://doi.org/10.1145/253260.253302

[19] Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. 1999. On Random Sampling over Joins. In *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, Alex Delis, Christos Faloutsos, and Shahram Ghandeharizadeh (Eds.), Vol. 28. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/304181.304206

[20] Surajit Chaudhuri and Kyuseok Shim. 1994. Including Group-By in Query Optimization. In *VLDB*. Morgan Kaufmann, San Francisco, CA, USA, 354–366.

[21] Surajit Chaudhuri and Kyuseok Shim. 1996. Optimizing Queries with Aggregate Views. In *EDBT (Lecture Notes in Computer Science)*, Vol. 1057. Springer, 167––182. https://doi.org/10.1007/BFb0014151

[22] Ming-Syan Chen, Hui-I Hsiao, and Philip S. Yu. 1997. On Applying Hash Filters to Improving the Execution of Multi-Join Queries. *VLDB J.* 6, 2 (1997), 121–131. https://doi.org/10.1007/s007780050036

[23] Yu Chen and Ke Yi. 2020. Random Sampling and Size Estimation Over Cyclic Joins. In *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020 (LIPIcs)*, Carsten Lutz and Jean Christoph Jung (Eds.), Vol. 155. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Copenhagen, Denmark, 7:1–7:18.

[24] The Gene Ontology Consortium. 2018. The Gene Ontology Resource: 20 years and still GOing strong. *Nucleic Acids Research* 47, D1 (11 2018), D330–D338. https://doi.org/10.1093/nar/gky1055

[25] The Gene Ontology Consortium. 2021. Gene Ontology Resource. http://geneontology.org/stats.html (visited: 2021-06).

[26] CovidGraph. 2021. COVID-19 Knowledge Graph. https://covidgraph.org/.

[27] Víctor Dalmau and Peter Jonsson. 2004. The complexity of counting homomorphisms seen from the other side. *Theor. Comput. Sci.* 329, 1-3 (2004), 315–323.

[28] Shaleen Deep and Paraschos Koutris. 2018. Compressed Representations of Conjunctive Query Results. In *Proc. PODS 2018*, Jan Van den Bussche and Marcelo Arenas (Eds.). ACM, New York, NY, USA, 307–322. https://doi.org/10.1145/3196959.3196979

[29] Holger Dell, Thore Husfeldt, Bart M. P. Jansen, Petteri Kaski, Christian Komusiewicz, and Frances A. Rosamond. 2016. The First Parameterized Algorithms and Computational Experiments Challenge. In *Proc. IPEC 2016 (LIPIcs)*, Jiong Guo and Danny Hermelin (Eds.), Vol. 63. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Aarhus, Denmark, 30:1–30:9. https://doi.org/10.4230/LIPIcs.IPEC.2016.30

[30] Holger Dell, Christian Komusiewicz, Nimrod Talmon, and Mathias Weller. 2017. The PACE 2017 Parameterized Algorithms and Computational Experiments Challenge: The Second Iteration. In *Proc. IPEC 2017 (LIPIcs)*, Daniel Lokshtanov and Naomi Nishimura (Eds.), Vol. 89. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Vienna, Austria, 30:1–30:12.

[31] DetkDecomp. 2021. detkdecomp. https://github.com/daajoe/detkdecomp (visited: 2021-06).

[32] Arnaud Durand and Stefan Mengel. 2015. Structural Tractability of Counting of Solutions to Conjunctive Queries. *Theory Comput. Syst.* 57, 4 (2015), 1202–1249. https://doi.org/10.1007/s00224-014-9543-y

[33] Ronald Fagin, Amnon Lotem, and Moni Naor. 2003. Optimal aggregation algorithms for middleware. *J. Comput. System Sci.* 66, 4 (2003), 614–656. https://doi.org/10.1016/S0022-0000(03)00026-6 Special Issue on PODS 2001.

[34] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Adding Counting Quantifiers to Graph Patterns. In *SIGMOD Conference*. Association for Computing Machinery, New York, NY, USA, 1215–1230. https://doi.org/10.1145/2882903.2882937

[35] Jonathan Finger and Neoklis Polyzotis. 2009. Robust and efficient algorithms for rank join evaluation. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*. Association for Computing Machinery, New York, NY, USA, 415–428. https://doi.org/10.1145/1559845.1559890

[36] Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. 2019. HyperBench: A Benchmark and Tool for Hypergraphs and Empirical Findings. In *Symposium on Principles of Database Systems (PODS)*. Association for Computing Machinery, New York, NY, USA, 464–480. https://doi.org/10.1145/3294052.3319683

[37] Jörg Flum and Martin Grohe. 2004. The Parameterized Complexity of Counting Problems. *SIAM J. Comput.* 33, 4 (2004), 892–922. https://doi.org/10.1137/S0097539703427203

[38] Martin Grohe. 2007. The Complexity of Homomorphism and Constraint Satisfaction Problems Seen from the Other Side. *J. ACM* 54, 1 (2007), 1:1–1:24. https://doi.org/10.1145/1206035.1206036

[39] Martin Grohe, Thomas Schwentick, and Luc Segoufin. 2001. When is the evaluation of conjunctive queries tractable?. In *ACM Symposium on Theory of Computing (STOC)*. Association for Computing Machinery, New York, NY, USA, 657–666. https://doi.org/10.1145/380752.380867

[40] Venky Harinarayan and Ashish Gupta. 1994. *Generalized Projections: A Powerful Query-Optimization Technique.* Technical Report. Stanford University, Stanford, CA, USA.

[41] Stratos Idreos, Olga Papaemmanouil, and Surajit Chaudhuri. 2015. Overview of Data Exploration Techniques. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*. Association for Computing Machinery, New York, NY, USA, 277–281. https://doi.org/10.1145/2723372.2731084

[42] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. 2017. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *International Conference on Management of Data (SIGMOD)*. Association for Computing Machinery, New York, NY, USA, 1259–1274. https://doi.org/10.1145/3035918.3064027

[43] Ihab F Ilyas, Walid G Aref, and Ahmed K Elmagarmid. 2004. Supporting top-k join queries in relational databases. *The VLDB journal* 13, 3 (2004), 207–221. https://doi.org/10.1007/s00778-004-0128-2

[44] Ihab F. Ilyas, George Beskales, and Mohamed A. Soliman. 2008. A survey of top-k query processing techniques in relational database systems. *Comput. Surveys* 40, 4 (2008), 11:1–11:58.

[45] Ihab F. Ilyas and Xu Chu. 2019. *Data Cleaning*. ACM, New York, NY, USA. https://doi.org/10.1145/3310205

[46] Zachary G. Ives and Nicholas E. Taylor. 2008. Sideways Information Passing for Push-Style Query Processing. In *ICDE*. IEEE Computer Society, 774–783. https://doi.org/10.1109/ICDE.2008.4497486

[47] Manas R Joglekar, Rohan Puttagunta, and Christopher Ré. 2016. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. Association for Computing Machinery, New York, NY, USA, 91–106. https://doi.org/10.1145/2902251.2902293

[48] Ahmet Kara and Dan Olteanu. 2018. Covers of Query Results. In *21st International Conference on Database Theory (LIPIcs)*, Benny Kimelfeld and Yael Amsterdamer (Eds.), Vol. 98. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Vienna, Austria, 16:1–16:22.

[49] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proc. PODS 2016*. Association for Computing Machinery, New York, NY, USA, 13–28. https://doi.org/10.1145/2902251.2902280

[50] Albert Kim, Liqi Xu, Tarique Siddiqui, Silu Huang, Samuel Madden, and Aditya Parameswaran. 2016. *Optimally Leveraging Density and Locality for Exploratory Browsing and Sampling*. Technical Report. Univeristy of Illinois. https://data-people.cs.illinois.edu/needletail.pdf (visited: 2021-06).

[51] Albert Kim, Liqi Xu, Tarique Siddiqui, Silu Huang, Samuel Madden, and Aditya Parameswaran. 2018. Optimally Leveraging Density and Locality for Exploratory Browsing and Sampling. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (Houston, TX, USA) *(HILDA'18)*. Association for Computing Machinery, New York, NY, USA, 7. https://doi.org/10.1145/3209900.3209903

[52] Rajeev Kohli, Ramesh Krishnamurti, and Prakash Mirchandani. 1994. The Minimum Satisfiability Problem. *SIAM J. Discret. Math.* 7, 2 (1994), 275–283. https://doi.org/10.1137/S0895480191220836

[53] Markus Krötzsch. 2018. Practical Linked Data Access via SPARQL: The Case of Wikidata. In *LDOW@ WWW*. CEUR Workshop Proceedings, Lyon, France, 1–10.

[54] Hanâ Lbath, Angela Bonifati, and Russ Harmer. 2021. Schema Inference for Property Graphs. In *Proceedings of the 24th International Conference on Extending Database Technology, EDBT 2021*. OpenProceedings.org, Nicosia, Cyprus, 499–504. https://doi.org/10.5441/002/edbt.2021.58

[55] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal* 27, 5 (2018), 643–668. https://doi.org/10.1007/s00778-017-0480-7

[56] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. 2005. RankSQL: Query Algebra and Optimization for Relational Top-k Queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. Association for Computing Machinery, New York, NY, USA, 131–142. https://doi.org/10.1145/1066157.1066173

[57] Lothar F. Mackert and Guy M. Lohman. 1986. R* Optimizer Validation and Performance Evaluation for Distributed Queries. In *VLDB*. Morgan Kaufmann, New York, NY, USA, 149—-159. https://doi.org/10.1145/16894.16863

[58] Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. 2018. Getting the Most Out of Wikidata: Semantic Technology Usage in Wikipedia's Knowledge Graph. In *International Semantic Web Conference (ISWC)*. Springer, Cham, 376–394. https://doi.org/10.1007/978-3-030-00668-6_23

[59] Nikos Mamoulis, Man Lung Yiu, Kit Hung Cheng, and David W Cheung. 2007. Efficient top-k aggregation of ranked inputs. *ACM Transactions on Database Systems (TODS)* 32, 3 (2007), 19–es.

[60] Apostol Natsev, Yuan-Chi Chang, John R Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. 2001. Supporting incremental join queries on ranked inputs. In *VLDB*, Vol. 1. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 281–290.

[61] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Trans. Database Syst.* 40, 1 (2015), 2:1–2:44. https://doi.org/10.1145/2656335

[62] Laurel J. Orr, Srikanth Kandula, and Surajit Chaudhuri. 2019. Pushing Data-Induced Predicates Through Joins in Big-Data Clusters. *Proc. VLDB Endow.* 13, 3 (2019), 252—-265. https://doi.org/10.14778/3368289.3368292

[63] Reinhard Pichler and Sebastian Skritek. 2013. Tractable counting of the answers to conjunctive queries. *J. Comput. System Sci.* 79, 6 (Sep 2013), 984–1001. https://doi.org/10.1016/j.jcss.2013.01.012

[64] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *Proc. VLDB Endow.* 13, 9 (2020), 1582–1597. https://doi.org/10.14778/3397230.3397250

[65] Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. arXiv:1911.05582 [cs.DB]

[66] Michael D. Vose. 1991. A Linear Algorithm For Generating Random Numbers With a Given Distribution. *IEEE Transactions on software engineering* 17, 9 (1991), 972–975.

[67] Denny Vrandečić. 2012. Wikidata: A New Platform for Collaborative Data Collection. In *Proceedings of the 21st International Conference on World Wide Web* (Lyon, France) *(WWW '12 Companion)*. Association for Computing Machinery, New York, NY, USA, 1063–1064. https://doi.org/10.1145/2187980.2188242

[68] Alastair J. Walker. 1977. An Efficient Method for Generating Discrete Random Variables with General Distributions. *ACM Trans. Math. Softw.* 3, 3 (1977), 253–256.

[69] WikiData. 2021. WikiData Query Service. http://query.wikidata.org/.

[70] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *VLDB*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 345-–357.

[71] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Proc. VLDB 1981* (Cannes, France). IEEE Computer Society, Cannes, France, 82–94.

[72] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). Association for Computing Machinery, New York, NY, USA, 1525–1539. https://doi.org/10.1145/3183713.3183739