Threshold Queries

Angela Bonifati
Lyon 1 Univ., Liris CNRS

Stefania Dumbrava
ENSIIE & IP Paris

George Fletcher
Eindhoven Univ. of Technology

Jan Hidders
Birkbeck, Univ. of London

Matthias Hofer
Univ. of Bayreuth

Wim Martens
Univ. of Bayreuth

Filip Murlak
Univ. of Warsaw

Joshua Shinavier
LinkedIn

Sławek Staworko
RelationalAI & Univ. of Lille

Dominik Tomaszuk
Univ. of Bialystok

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. We explore how such queries appear in practice and present a method that can be used to significantly improve the asymptotic bounds of their state-of-the-art evaluation algorithms. Our experimental evaluation of these methods shows order-of-magnitude performance improvements.

1. INTRODUCTION

Top-k evaluation of queries asks the database engine to return the k most relevant answers to the query. In Web search engines, it is the main mode of evaluating keyword search queries and it has been studied in depth in the wider context of database query answering [22, 29, 17, 27]. Internally, this mode of evaluation typically uses two parameters: a *limit* k on the number of answers to return, and a *ranking function* that determines which answers are more relevant than others and allows to order them by importance.

Top-k evaluation is indeed a very useful querying paradigm, but it is also easy to imagine scenarios, such as data exploration, in which the ranking function is less important or may even be absent. For instance, upon investigating a log file of queries for Wikidata's SPARQL endpoint [26, 8], containing over 254M pattern matching queries, we saw the following. While ∼15M (6%) pattern matching queries use `LIMIT`, only 11,406 (0.0045%) use both `LIMIT` and `ORDER BY`. Figure 1 (left) visualises these ratios. The picture is even more pronounced in the non-trivial pattern matching queries, that is, queries that use at least one join. The logs

This is a minor revision of the paper "Threshold queries in theory and in the wild", published in PVLDB, Vol. 15, No. 5, ISSN 2150-8097. DOI: https://doi.org/10.14778/3510397.35104072150-8097/15/12.

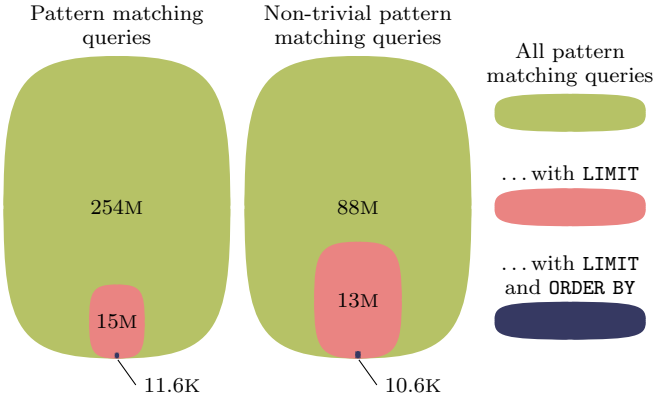


Figure 1: Fractions of pattern matching queries with `LIMIT` and `ORDER BY` in Wikidata query logs.

contain ∼88M such non-trivial pattern matching queries, out of which 13M (14.9%) use `LIMIT`, which is quite a large chunk. However, again, only 10,612 (0.0121%) use both `LIMIT` and `ORDER BY`; see Figure 1 (right).

We therefore believe that *threshold queries*, which only ask for a limited number of answers irrespective of any ranking, are worthy of study. In this paper, we present several scenarios in which threshold queries occur (Section 2), ranging from data exploration to checking cardinality constraints on data. We then discuss how thresholds can be exploited to speed up query evaluation (Sections 3 and 4). Notice that there is room for more aggressive query optimization than for top-k query answering, since threshold queries do not care about the ranking of results. As a proof of concept, we implemented the main ideas of our algorithm using SQL window functions. Although these only have limited flexibility and we cannot fully exploit our algorithmic ideas, we already see order of magnitude performance improvements (Section 5). Finally, we discuss other modes of query evaluation (Section 6) and possible next steps (Section 7).

2. THRESHOLD QUERIES IN THE WILD

Our goal is to understand how thresholds are used in real-life queries. Therefore, we begin with a handful of examples from various domains and then distill a formal definition.

Wikidata is a collaborative knowledge base launched in 2012 and hosted by the Wikimedia Foundation [34]. By the efforts of thousands of volunteers, the project has produced a large open knowledge base with numerous applications. Query logs [24] collected along the years on the **Wikidata** SPARQL endpoint offer an invaluable glimpse into real-life workloads and constitute a useful resource for the qualitative analysis of threshold queries. In the logs we found the following *data exploration* query.

TQ1 *Return up to 10 journal/article pairs for journals with ISSN 1175-5326.*

```
SELECT * WHERE {
  ?journal  <ISSN>  "1175-5326" .
  ?article  <published_in>  ?journal
} LIMIT 10
```

The query illustrates the simplest way in which a threshold can be used: to limit the number of returned answers. Without `LIMIT`, it would return 32,222 objects, which is not comfortable for human consumption. The `LIMIT` clause ensures that the output is human-consumable and potentially reduces the amount of work required from the query engine. As is common in data exploration, the user is happy with any 10 answers: they need not be the best, or random, or persistent. Any 10 answers will do. (Note that we do not consider features like `OFFSET` or `SKIP`.)

The next query comes from the **Offshore Leaks Database** [21], which records interconnections between offshore entities involved in the Pandora Papers, Paradise Papers, Bahamas Leaks, and Offshore Leaks investigations. It contains information regarding over 800K people or companies, spanning 80 years up to 2020 and over 200 jurisdictions. The dataset was published by the International Consortium of Investigative Journalists (ICIJ), a network of 280 journalists and over 140 media organizations from more than 100 countries, and is curated as a Neo4j graph with 840,000 nodes and 1.3 million relationships. The very first example in the documentation is the following query:

TQ2 *Return up to 20 entities, intermediaries, or addresses that can be reached from an officer called* `$name` *in at most 10 steps.*

```
MATCH (a:Officer {name:$name})-[r:officer_of
      |intermediary_of|registered_address*..10]-(b)
RETURN b.name as name LIMIT 20
```

This is again a limit query, but this time it involves a path of varying length (at most 10). While we present out methods over conjunctive queries (project-join queries), one can apply them also to queries involving path expressions, such as *conjunctive regular path queries*.

A more complex way of using thresholds is to express *cardinality constrains*. For example, the SQL query

TQ3 *Find Nobel prizes with more than 3 laureates.*

```
SELECT NobelPrize.id
FROM NobelPrize, Laureate
WHERE NobelPrize.id = Laureate.id
GROUP BY NobelPrize.id
HAVING COUNT(DISTINCT Laureate.name) >= 4
```

detects violations of a cardinality constraint arising in data curation of the **Nobel Prize data** [3]. This is a `GROUP BY` query that uses a threshold to specify the desired size of the groups. This a common query pattern; we give two more examples below in different query languages.

In the **Wikidata** logs we also found the following *data monitoring* query, very similar to TQ3:

TQ4 *Are there at least 67 language versions of Wikipedia pages for the movie* The Matrix*?*

```
SELECT( ?var2 ) WHERE {
  VALUES ( ?var2 ) { ( <The_Matrix> ) }
  ?var2 <instance_of> <film> .
  ?var3 <about> ?var2 .
  ?var3 ( <is_Part_Of> / <belongs_to> ) "wikipedia" .
} GROUP BY ?var2 HAVING (COUNT (*) >= 67);
```

The **COVID-19 Knowledge Graph** [13] 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) and article collections from different scientific domains (ArXiv, BioRxiv, MedRxiv, PubMed, and PubMed Central). Reporting coverage in the COVID-19 Knowledge Graph can be monitored using the following query, expressed in a Cypher-like syntax:

TQ5 *Find each country that does not have three reports for some age group.*

```
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
```

In the Covid-19 Knowledge Graph, 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. The query finds countries and age groups for which this is not the case. Note the use of Cypher's notorious *implicit group-by* [28]: the grouping variabes `c` and `a` are derived automatically.

Limit and threshold queries. The above examples illustrate two different query templates: TQ1–TQ2 are simple limit queries and TQ3–TQ5 are grouping queries with cardinality constraints over group sizes. We shall reserve the term *threshold queries* for the second template, but let us stress that both of them use thresholds to limit the numbers of sought tuples (answers in the case of limit queries and witnesses for answers in the case of threshold queries) without any preference over sought tuples. Indeed, generalising limit queries will be the key to efficient evaluation of threshold queries.

Typical threshold values. To get a better grasp of the parameters of the task ahead of us, we had another look at the pattern matching queries in the Wikidata logs to find out what the typical *threshold values* are (see Figure 2). It turned out that, in both organic (human-generated) and robotic queries, more than half of the pattern matching queries use threshold values not larger than 100. Values of 10k or more do occur, but they are much less common than small values.

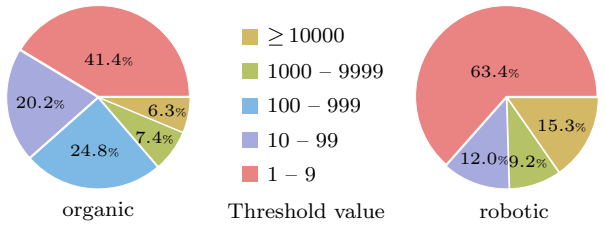


Figure 2: Threshold value occurrence ratio in *organic* and *robotic* pattern matching queries in our logs.

Formal definition. Formally, we can define a threshold query (TQ) in terms of first-order logic as an expression of the form

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

which can be read as "Return $\bar{x}$ that match q and for which there are between a and b many $\bar{y}$ that match p together with $\bar{x}$." From left to right, $q(\bar{x})$ is a conjunctive query, $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$ specify the lower and upper bounds for the groups, and $p(\bar{x}, \bar{y})$ is a conjunctive query which defines the elements of the groups (it may not use all variables in $\bar{x}$). Here the quantifier $\exists^{a,b}$ is a so-called *counting quantifier* which states that there are between a and b distinct instances of $\bar{y}$ for which the subsequent formula holds.

Notice that a threshold query only has a single counting quantifier $\exists^{a,b}$, but further ordinary existential quantifiers may occur inside q and p, as is usual for conjunctive queries.

As an example, consider the Nobel Prize threshold query (TQ3), and suppose that the schema is

$$\mathit{NobelPrize}(\underline{\mathit{id}}, \mathit{year}, \mathit{category}), \mathit{Laureate}(\underline{\mathit{id}}, \underline{\mathit{name}}, \mathit{country})$$

with the foreign key constraint $\mathit{Laureate}[\mathit{id}] \subseteq \mathit{NobelPrize}[\mathit{id}]$. This threshold query can be formalized as follows.

$$\begin{aligned}\mathrm{TQ3}(x) = \exists x_1, x_2.\ \mathit{NobelPrize}(x, x_1, x_2) \wedge \\ \exists^{4,\infty} y.\ \exists z.\ \mathit{Laureate}(x, y, z).\end{aligned}$$

In the remainder of this paper we represent threshold queries in this logic-based notation, a specific query language, or relational algebra, depending on what is discussed.

3. EXPLOITING THRESHOLDS

Let us see a simple example illustrating how thresholds can be exploited to speed up query evaluation. Let $G = (V, E)$ be a graph. For positive integers t and k, consider the following threshold query, dubbed (t, k)-neighbour:

Find all nodes with at least t k-hop neighbours.

Assuming that the graph is stored in a single table `Edge` with columns `source` and `target`, the query (t, k)-neighbour can be expressed in SQL as a k-fold join followed by aggregation. For instance, for $t = 10$ and $k = 3$, we have

```
SELECT X.source
FROM Edge AS X, Edge AS Y, Edge AS Z
WHERE X.target = Y.source AND Y.target = Z.source
GROUP BY X.source
HAVING COUNT(DISTINCT Z.target) >= 10
```

If we compute the joins naively and then aggregate, we obtain a cubic algorithm (modulo log factors) in the example above, and $\tilde{O}(m^k)$ in general over any graph with m edges.

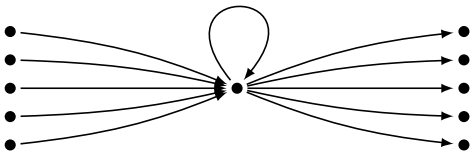


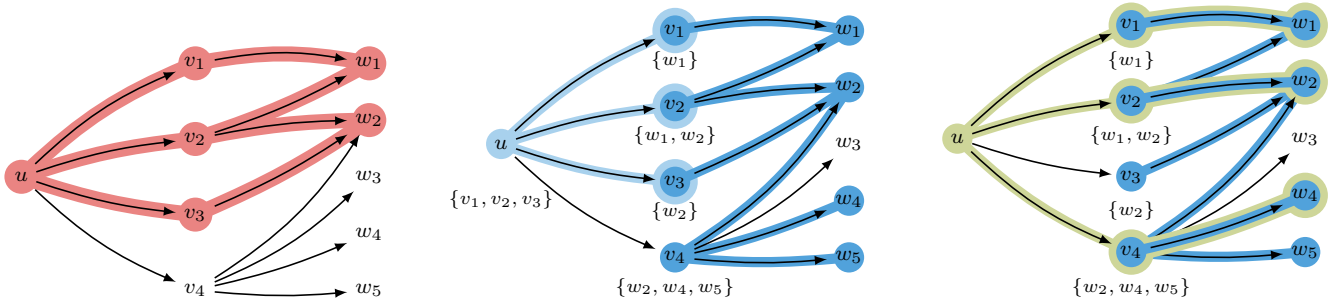
Figure 3: Quadratically many pairs of k-hop neighbours for all $k \geq 2$.

Here, we use the $\tilde{O}$ notation to indicate that logarithmic factors are supressed.

A less naïve approach is to project out irrelevant columns as early as possible. This amounts to computing all the i-hop neighbours for all nodes iteratively for $i = 1, 2, \ldots, k$ and returning nodes with at least t k-hop neighbours. This optimization decreases the time cost drastically, leading to a quadratic algorithm for every k. Can this be improved further? The bottleneck here is simply the size of the intermediate results representing i-hop neighbours for all nodes: in a graph with m edges there can be as many as $(\frac{m}{2})^2$ pairs of nodes connected by an i-hop path, for any $i \geq 2$; see Figure 3. Can we avoid computing all i-hop neighbours?

We claim that it is enough to compute up to t many i-hop neighbours for each node in the graph. More precisely, we will compute *i-hop neighbours up to threshold t*: that is, if a node has at most t i-hop neighbours, then we store all of them, and if it has more, then we discard all but t arbitrary ones. One has to be careful, though, with the order in which the joins are processed. It might seem natural to go forward, or "left to right": in order to compute $(i+1)$-hop neighbours of node u we would take all (1-hop) neighbours of the computed i-hop neighbours of u. This works if we have all i-hop neighbours of u. If we only keep them up to threshold t, this will not work in general. For instance, let $t = 3$, $i = 1$ and consider the graph in Figure 4a. Suppose that we have only kept v_1, v_2, v_3 out of u's four 1-hop neighbours. Then, even if we consider all their 1-hop neighbours, we only recover two of u's 2-hop neighbours, w_1 and w_2. That is, we have not found the correct number of 2-hop neighbours up to the threshold value $t = 3$.

To obtain correct results we have to go the opposite way: in order to compute $(i + 1)$-hop neighbours of node u we consider the computed i-hop neighbours of *all* neighbours of u. In our example in Figure 4b, this means that we first compute up to three 1-hop neighbours of $v_1, \ldots, v_4$, which are annotated under the nodes. (The three 1-hop neighbours v_1, v_2, v_3 for u are computed in the first iteration, but will be discarded in the next.) For the next iteration, that is, $i = 2$, consider Figure 4c. We proceed through the neighbours of u in some arbitrary order (here we choose top to bottom), while remembering the number of 2-hop neighbours we see. When we visit v_1, we have the set $\{w_1\}$ of 2-hop neighbours. After v_2, we have $\{w_1\} \cup \{w_1, w_2\} = \{w_1, w_2\}$. Visiting v_3 does not add any 2-hop neighbours to this set, and we finally find our threshold of three 2-hop neighbours after visiting v_4. This means that u is an answer to $(3, 2)$-neighbour. At this point, we could annotate u with any subset of $\{w_1, w_2, w_4, w_5\}$ consisting of three elements, witnessing three 2-hop neighbours. In the picture, we highlighted $\{w_1, w_2, w_4\}$. Notice that w_3 was already discarded when computing witnesses for 1-hop neighbours of v_4.



(a) How not to do it. (b) How to do it: first iteration. (c) How to do it: second iteration.

Figure 4: Computing 2-hop neighbours up to threshold 3.

In general, if we have correctly computed i-hop neighbours up to threshold t for all nodes, then we are guaranteed to have $(i+1)$-hop neighbours up to threshold t for all nodes after the next round.

We can prove this by contradiction. Suppose that we have not retained sufficiently many $(i+1)$-hop neighbours of u. Let v be one of the missing ones. If v has been retained as an i-hop neighbour of some neighbour u' of u, then the only reason why it would not be retained as a $(i+1)$-hop neighbour for u can be that u had t $(1+1)$-hop neighbours without it. That contradicts the initial assumption. Hence, v must have not been retained among i-hop neighbours of some neighbour u' of u. Since we have assumed that i-hop neighbours were computed correctly up to threshold t, it follows that u' has at least t i-hop neighbours retained. But all these nodes are $(i+1)$-hop neighbours of u, which also contradicts our initial assumption.

4. EVALUATION IN THEORY

Discussing query evaluation is easier for queries expressed in relational algebra. We assume the named perspective. A conjunctive query is then an expression of the form

$$\pi_X\left(R_1 \bowtie \cdots \bowtie R_n\right)$$

where $\bowtie$ is the natural join and π_X is the projection on the set X of columns. In order to capture limit and threshold queries we introduce two new relational algebra operators:

- one that corresponds to a grouping combined with a selection based on group sizes and
- one that prunes a relation so that the maximum group size is below a specified upper bound.

The first new operator is *bounded (counting) aggregation*, which combines ordinary counting aggregation with selection and is a direct relational algebra counterpart of the counting quantifier. We write it as

$$\gamma_X^{a,b}(R)\,,$$

where X is a set of columns and $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$. The meaning of $\gamma_X^{a,b}(R)$ is to select tuples r in $\pi_X(R)$ such that the number of tuples in R that agree with r on X is between a and b. As an example, consider the relation R and the result of $\gamma_{\mathrm{A,B}}^{2,3}(R)$ in Figure 5. This operator allows us for example to express TQ3 as

$$\pi_{id}(\mathit{NobelPrize}) \bowtie \gamma_{id}^{4,\infty}(\pi_{id,name}(\mathit{Laureate}))\,.$$

R

A	B	C	D
1	1	1	1
1	2	1	1
1	2	1	2
2	1	1	1
2	1	1	2
2	1	2	1
2	2	1	1
2	2	1	2
2	2	2	1
2	2	2	2

$\gamma_{\mathrm{A,B}}^{2,3}(R)$

A	B
1	2
2	1

$\pi_{\mathrm{A,B}}^{2}(R)$

A	B	C	D
1	1	1	1
1	2	1	1
1	2	1	2
2	1	1	2
2	1	2	1
2	2	1	2
2	2	2	1

Figure 5: The bounded aggregation and pruning operators.

The second new relational operation is a nondeterministic *pruning* operator written as

$$\pi_X^b(R)\,,$$

where X is a set of columns and b is a natural number. The result of $\pi_X^b(R)$ is a subset of R such that if it were grouped on X, then the resulting groups would be of size at most b and the groups that were already of size at most b in the grouping of R should remain the same in the grouping of $\pi_X^b(R)$. An example is given in Figure 5, where a possible result of $\pi_{\mathrm{A,B}}^2(R)$ is shown. It is possible to use this pruning operator to express limit queries. For example, the query TQ2 can be expressed by $\pi_\emptyset^{20}(Q)$ where Q is the expression for the underlying basic graph pattern of TQ2.

The method described in Section 3 can be applied to all threshold queries of the form

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

where $p(\bar{x},\bar{y})$ is a conjunctive query and $a \geq 1$. Note that (t,k)-neighbour is an example of such query: it can be expressed as

$$\exists^{t,\infty} x_k\, \exists x_1,\ldots,x_{k-1}\, E(x_0,x_1) \wedge \cdots \wedge E(x_{k-1},x_k)\,,$$

where $E = \texttt{Edge}$ and one should read $\exists^{t,\infty} x_k$ as "there are at least t values of x_k such that". The output of the query is the set of nodes that can be matched to the variable x_0 (which is not quantified). Using the bounded aggregation

operator, threshold queries of this form can be written in relation algebra as

$$\gamma_X^{a,b}\big(\pi_{X\cup Y}\big(R_1 \bowtie \cdots \bowtie R_n\big)\big)\,.$$

For example, for (t,k)-neighbour we get

$$\gamma_{A_0}^{t,\infty}\big(\pi_{A_0,A_n}(E_1 \bowtie \cdots \bowtie E_k)\big)$$

where $E_i = \texttt{Edge}(A_{i-1}, A_i)$.

The classical evaluation method for conjunctive queries aims to optimize performance by choosing an order of joins and projections that minimizes the size of intermediate results. That is, it seeks an equivalent project-join expression whose subexpressions evaluate to relations as small as possible. For example, for the internal conjunctive query $\pi_{A_0,A_n}(E_1 \bowtie \cdots \bowtie E_k)$ of (t,k)-neighbour, one could take

$$\begin{aligned}\gamma_{A_0}^{t,\infty}\big(\pi_{A_0,A_k}(E_1 \bowtie\\ \pi_{A_1,A_k}(E_2 \bowtie\\ \cdots\\ \pi_{A_{k-2},A_k}(E_{k-1} \bowtie\\ E_k)\ldots)))\end{aligned}$$

as in the "less naïve" evaluation method mentioned in Section 3. The rationale behind it is that if each subexpression yields an intermediate result of size at most K, then one can evaluate the whole expression in time $\tilde{O}(K)$.

A simple proxy for the size of an intermediate result is the arity of the corresponding subexpression. Indeed, a subexpression of arity d yields at most N^d tuples on a database of size N. We shall refer to the maximal arity of a subexpression of a project-join expression E as the *width of* E. If E has width d, it can be evaluated in time $\tilde{O}(N^d)$. The *width of a conjunctive query* Q is the minimal width of an equivalent project-join expression. This notion can be equivalently defined in terms of *tree decompositions*, and coincides (up to 1) with a variant of *tree-width* suitable for non-boolean conjunctive queries (some bag must contain all answer columns) [7]. In the running example, the chosen expression has width 3 (and it is optimal), so we only get a cubic bound. Yet, all intermediate results are in fact at most quadratic, because all computed tuples of arity 3 are obtained from an edge and a node. The notion of width could be refined to capture this: instead of measuring the arity of subexpressions, one could measure the number of relations needed to capture the set of columns of each subexpression. This would align it with hypertree-width [19], one of many tightenings of tree-width. For simplicity, however, we stick to the basic width measure.

One way to execute a threshold query is to evaluate the underlying conjunctive query $Q = \pi_{X\cup Y}\big(R_1 \bowtie \cdots \bowtie R_n\big)$, and then perform the aggregation $\gamma_X^{a,b}$. We can use the optimal project-join expression for Q, but we are still limited by its width, which is at least $|X \cup Y|$. In order to break this barrier we need to avoid evaluating Q in full. The key insight is that a pruned result of Q is sufficient to evaluate the threshold query, because

$$\gamma_X^{a,b}(Q) = \gamma_X^{a,b}(\pi_X^t Q)$$

for $t = t(a,b)$ where $t(a,b) = b+1$ if $b < \infty$ and $t(a,\infty) = a$. How do we compute the pruned result efficiently?

The technique of pushing down projections is commonly used to reduce the width of project-join expressions. It amounts to exhaustively applying the identity

$$\pi_X\big(F \bowtie G\big) = \pi_X\big(\pi_{(X\cup Z_G)\cap Z_F}(F) \bowtie \pi_{(X\cup Z_F)\cap Z_G}(G)\big)\,,$$

where Z_F and Z_G are the sets of columns of the expressions F and G, respectively. For example, the expression chosen in the running example is obtained this way from $\pi_{A_1,A_k}(E_1 \bowtie (E_2 \bowtie \cdots (E_{k-1} \bowtie E_k)))$. We can do the same with pruning.

In their seminal paper [10], Carey and Kossmann explored ways of pushing the `LIMIT` operator down query plans. This, however, is close to impossible. Indeed, `LIMIT` cannot be pushed down through a projection, because different answers may become identical after dropping some columns. It is also very hard to push `LIMIT` down through a join, because we need to ensure that matching answers are kept in both subplans. This is why Carey and Kossmann end up recomputing additional tuples often.

The pruning operator π_X^t, however, offers significantly more control over which answers are kept, as it limits each group independently. This makes it much easier to propagate. In fact, it behaves very much like the projection π_X and satisfies the following identities

$$\begin{aligned}\pi_X^t\big(F \bowtie G\big) &\equiv_X^{a,b} \pi_X^t\big(\pi^t_{(X\cup Z_G)\cap Z_F}(F) \bowtie \pi^t_{(X\cup Z_F)\cap Z_G}(G)\big)\,,\\ \pi_X^t\big(\pi_Y F\big) &\equiv_X^{a,b} \pi_X^t\big(\pi_Y(\pi^t_{X\cup(Z_F\setminus Y)}(F))\big)\,.\end{aligned}$$

Note that we cannot hope for equality here, because we have no control over the actual tuples that the operation π_X^t keeps. Instead, we have an equivalence $\equiv_X^{a,b}$ defined as follows:

$$E_1 \equiv_X^{a,b} E_2 \iff \gamma_X^{a,b}(E_1) = \gamma_X^{a,b}(E_2)\,.$$

Now, starting from any project-join expression E for Q we can propagate π_X^t down in $\pi_X^t(E)$ by applying the identities above exhaustively (replacing left-hand sides by right-hand sides). Let E' be the resulting expression.

Unlike projection π_X, pruning π_X^t does not reduce the arity of relations. However, it does reduce the size of the relation in a similar fashion: $\pi_X^t(E)$ returns at most $t \cdot N^{|X|}$ tuples over every database instance of size N. That is, what actually matters for the complexity are the columns that have not been affected by pruning. Indeed, the expression E' can be evaluated in time $\tilde{O}(t \cdot N^d)$ where d is the width of E' computed as if each π_Z^t were replaced by π_Z. For example, for (t,k)-neighbour we get the width-2 expression

$$\begin{aligned}\gamma_{A_0}^{t,\infty}\big(\pi^t_{A_0}\pi_{A_0,A_k}(E_1 \bowtie\\ \pi^t_{A_1}\pi_{A_1,A_k}(E_2 \bowtie\\ \cdots\\ \pi^t_{A_{k-2}}\pi_{A_{k-2},A_k}(E_{k-1} \bowtie\\ \pi^t_{A_{k-1}}(E_k))\ldots)))\end{aligned}$$

corresponding precisely to the algorithm for (t,k)-neighbour from Section 3. (The gap between the theoretical quadratic bound and the actual linear complexity is again caused by relying on tree-width rather than hypertree-width.)

The best project-join expression E for Q, as obtained in the query optimization process, need not result in the best E'. For example, consider the query

$$\gamma_{A_1,A_3}^{t,\infty}\big(E_1 \bowtie E_2 \bowtie E_3 \bowtie E_4'\big)$$

for $E_4' = \texttt{Edge}(A_3, A_0)$, selecting pairs of nodes lying on at least t 4-hop cycles as non-consecutive nodes. One of the

minimal-width project-join expressions for the inner conjunctive query is $(E_1 \bowtie E_2) \bowtie (E_3 \bowtie E_4')$. This expression leads to a width-4 prune-project-join expression. We can do better. Instead of a minimal-width expression for Q, we start with a minimal-width expression for $\pi_X Q$. For example, for the cycle query above we can use

$$\pi_{A_1,A_3}(E_2 \bowtie E_3) \bowtie \pi_{A_1,A_3}(E_4' \bowtie E_1).$$

Each such expression can be turned into a prune-project-join expression for $\pi_X^t(Q)$ by replacing projections that remove columns from Y with appropriate pruning operators, bottom up: each $\pi_U(G)$ should be replaced with $\pi_U^t(\pi_{U\cup(Z_G\cap Y)}(G))$; additionally, we apply π_X^t on top. Then, we propagate pruning operators down the expression, using the identities. In the cycle example we get a width-3 expression

$$\pi_{A_1,A_3}^t(\pi_{A_1,A_3}^t(E_2 \bowtie E_3) \bowtie \pi_{A_1,A_3}^t(E_4' \bowtie E_1)).$$

This shows that $\pi_X^t(Q)$ can be evaluated in time $\tilde{O}(t \cdot N^d)$ where d is the width of $\pi_X(Q)$ rather than of Q itself. (In particular, for limit queries of the form $\pi_\emptyset^t(Q)$, d is the width of $\pi_\emptyset(Q)$ which equals one plus the tree-width of Q.) To get $\gamma_X^{a,b}(Q)$ we simply apply $\gamma_X^{a,b}$ to the result of $\pi_X^t(Q)$.

Evaluating a general threshold query $q(\bar{x}) \wedge \exists^{a,b} p(\bar{x},\bar{y})$ with $a \geq 1$, amounts to evaluating an expression of the form

$$\pi_{X\cup Z}(S_1 \bowtie \cdots \bowtie S_m) \bowtie \gamma_X^{a,b}(\pi_{X\cup Y}(R_1 \bowtie \cdots \bowtie R_n))$$

where Z corresponds to variables in $\bar{x}$ that are not actually used in p. We can do this by evaluating the two subexpressions independently and computing the join. If $a = 0$, we need to return

$$\pi_{X\cup Z}(S_1 \bowtie \cdots \bowtie S_m) \bowtie \gamma_X^{1,b}(\pi_{X\cup Y}(R_1 \bowtie \cdots \bowtie R_n)) \cup$$
$$\cup\, \pi_{X\cup Z}(S_1 \bowtie \cdots \bowtie S_m) \triangleright \gamma_X^{1,\infty}(\pi_{X\cup Y}(R_1 \bowtie \cdots \bowtie R_n))$$

where $\triangleright$ is the antijoin operator. With $\pi_X^t(\pi_{X\cup Y}(R_1 \bowtie \cdots \bowtie R_n))$ computed for $t = t(1,b)$, this can be done with minimal overhead. Overall, threshold queries of width d with thresholds a, b can be evaluated in time $\tilde{O}(t(a,b) \cdot N^d)$ over databases of size N.

5. GETTING PRACTICAL

Interestingly, our query evaluation algorithm can be mimicked with the use of SQL window functions, which allows us to evaluate its performance in practice. *SQL window functions* are a modification of the standard grouping and aggregation functionality. A window function uses values from one or multiple rows in a group to return a value *for each row*. Unlike aggregation, using window functions does not replace the rows of a group with a single output row, but rather the input rows retain their separate identities. Below is an example of a query, over the same schema as TQ3, that lists Nobel prizes with their laureates, and adds a column with the laureate count of each Nobel.

```
SELECT NobelPrize.ID, Laureate.Name, COUNT(*)
       OVER (PARTITION BY NoblePrize.ID) AS LCount
FROM NobelPrize, Laureate
WHERE NobelPrize.ID = Laureate.Prize_ID
```

We implement the generalized projection operator π_X^t with the help of the window function `ROW_NUMBER()` that assigns consecutive natural numbers to rows in each group. We illustrate how to use window functions to implement the pruning operator on the example of $\pi_{\texttt{source}}^{10}(\texttt{Edge})$, conveniently broken up into subexpressions.

```
WITH
  S AS (SELECT DISTINCT source, target FROM Edge)
  C AS (SELECT source, target, ROW_NUMBER() OVER
               (PARTITION BY source) AS i FROM S)
SELECT source, target FROM C WHERE i <= 10;
```

As a complete example, consider the $(10,3)$-neighbour query from Section 3 that identifies all nodes with at least 10 3-hop neighbours. The rewriting using window functions follows.

```
WITH
  S1 AS (SELECT DISTINCT source AS x2, target AS x3
         FROM Edge),
  C1 AS (SELECT x2, x3, ROW_NUMBER() OVER
                (PARTITION BY x2) AS i FROM S1),
  E1 AS (SELECT x2, x3 FROM C1 WHERE i <= 10),
  S2 AS (SELECT DISTINCT source AS x1, x3
         FROM Edge JOIN E1 ON Edge.target = T3.x2),
  C2 AS (SELECT x1, x3, ROW_NUMBER() OVER
                (PARTITION BY x1) AS i FROM S2),
  E2 AS (SELECT x1, x3 FROM C2 WHERE i <= 10)
  S3 AS (SELECT DISTINCT source AS x0, x3
         FROM Edge JOIN E2 ON Edge.target = T2.x1),
  C3 AS (SELECT x0, x3, ROW_NUMBER() OVER
                (PARTITION BY x0) AS i FROM S2),
  E3 AS (SELECT x0, x3 FROM C3 WHERE i <= 10),
SELECT x0 FROM E3 GROUP BY x0 HAVING COUNT(*)>=10;
```

We have used window functions, as illustrated above, to asses the performance of our algorithm on the following three types of queries, parameterized by path length k:

(q_1) $(10,k)$-reach selects up to 10 pairs of nodes linked by a k-hop path;

(q_2) $(10,k)$-neighbour selects all nodes with at least 10 k-hop neighbours;

(q_3) $(10,k)$-path selects all pairs of nodes linked by at least 10 k-hop paths.

In our experiments we have used two kinds of data sets:

(1) The real-world IMDb data set used in Join Order Benchmark [25], which contains information about movies and related facts about actors, directors, production companies, etc. We used the *movie_link* relation, which exhibits small-world network structure, and finding paths in it is meaningful.

(2) Barabási-Albert graphs [5], which are synthetic data sets that model the structure of scale-free 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 have used PostgreSQL 13.4 powered by a machine with Intel Core i7-4770K CPU @ 3.50GHz, 16GB of RAM, and an SSD. PostgreSQL, and other RDBMSs, implements window functions using sorting, to handle an optional but frequently used `ORDER BY` clause in the partition definitions. Sorting is unnecessary for our purposes and leads to non-optimal implementations of threshold queries. Still, the use of window functions allows to greatly reduce the number of intermediate results, which is an essential aspect of our algorithms. Consequently, the rewriting offers a significant overall performance boost despite the overhead cost of sorting.

In the first experiment we compare the running time of the baseline and windowed versions of (q_1–q_3) for varying values of k on both the IMDB and synthetic data sets (Table 1). We see that our approach (windowed) outperforms the baseline with speedups of 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

Table 1: Experimental evaluation for the baseline (b) and windowed (w) versions of (q_1–q_3). All measurements are ms.

	IMDB database										Barabási-Albert graphs with $m_0 = 10$ and $n = 3000$									
k	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7	8	9	10
q_1/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
q_1/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**
q_2/b	50	349	5.742	134.97	T/O	T/O	T/O	T/O	T/O	T/O	**138**	1.296	11.207	90.199	644.86	T/O	T/O	T/O	T/O	T/O
q_2/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**
q_3/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
q_3/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 2: Experimental evaluation of the speedup factor, the ratio of baseline to windowed, for q_2 with $k = 3$ on Barabási-Albert graphs.

$m_0 \backslash n$	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

good theoretical bounds, while the baseline clearly shows exponential dependence on k.

In the second experiment, we wish to assess the impact of the structure and size of the data on the running time of our algorithm. We use Barabási-Albert synthetic graphs with varying outdegree (m_0) and varying number of nodes (n), and we compare the running times of the two versions of query q_2 with $k = 3$ (Table 2) We varied m_0 from 5 to 25, using increments of 5, and varied n from 32 to 1M in a logarithmic 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 2, the windowed algorithm terminated in 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. EVALUATION IN ALL ITS FLAVOURS

In Section 4, we discussed methods to evaluate threshold queries more efficiently. However, when the arity of queries is large, the complexity of the algorithm is high, that is, exponential in the arity. So, can we do better?

In order to deal with queries potentially returning very large number of answers, researchers have considered several variants of the query evaluation problem. We briefly discuss these variants — and their relationship to our work — before explaining our main theorem. Standard query evaluation asks, for a query q and a database D, to compute

$$q(D) \ ,$$

which is the *set of answers of q on D*, also called the *output of q on D*. Variants of query evaluation studied in literature include at least the following.

(E1) Boolean evaluation; that is, testing existence of an answer. This variant tests if $q(D)$ is non-empty.

(E2) Counting all answers. Here, the task is to compute $|q(D)|$, the number of elements of $q(D)$.

(E3) Sampling answers with uniform probability. Here, the task is to return a single random answer $a \in q(D)$ with the probability of each a being $1/|q(D)|$.

(E4) Enumerating all the answers of the query, that is, generating $q(D)$ element by element, without repetition.

The computational cost of these variants tends to increase as we go down in the list. Indeed, (E1) is the easiest problem since it just aims at testing if a query has at least one result, and (E4) is the most difficult one since it is aims at computing all answers. For each of the variants, we briefly discuss some results that are important to us.

Boolean Evaluation. Even the simplest problem (E1) is intractable already for *conjunctive queries* [1, Chapter 15]. One reason is that conjunctive queries can look for k-clique-shaped pattern in the data, which means that testing if the answer of such queries is non-empty amounts to solving the NP-hard k-clique problem. However, queries in practice very often have a simple structure and are often *acyclic* [8, 9]. The large body of work triggered by Yannakakis' seminal result on efficient evaluation of *acyclic* conjunctive queries [36] teaches us that Boolean evaluation for queries whose topological structure is tree-like can be done efficiently. Even stronger, we know that tree-likeness of the query is not only helpful but even *necessary* for polynomial-time Boolean evaluation of conjunctive queries [20]. Technically, the notion of *tree-width* is used to measure how tree-like a query is: queries with low tree-width are tree-like and queries with high tree-width are highly cyclic. For instance, a tree-pattern query has tree-width one, whereas a query asking for the existence of a k-clique has tree-width $k - 1$. Recall that tree-width is one less than the *width* discussed in Section 4 (for Boolean queries).

Counting. One way to generalize (E1) is by computing the exact number of answers of a query (E2). For projection-free conjunctive queries (join queries), counting all answers is, just like Boolean evaluation, tightly connected to their tree-width [14, 18]. In the presence of projection, however, counting query answers is intractable even for *acyclic* conjunctive queries [31]. Efficient algorithms for counting answers to general conjunctive queries (project-join queries) require not low tree-width but low *free-connex tree-width* [16]. Here, free-connex tree-width is, just like tree-width, usually defined in terms of *tree decompositions*, but it restricts the kind of decompositions that are allowed: there needs to be a connected set of nodes including the root that contain exactly the output variables. This allows projecting out all remaining variables before beginning to count. In fact, the original result [16] was stated in terms of low tree-width and low *star size* but we have proved that this is equivalent to low free-connex tree-width [7]. However, when the problem

is relaxed to randomized approximate counting, low free-connex tree-width is not necessary for efficient algorithms to exist. That is, it is sufficient for queries to have low tree-width [2], just like for Boolean evaluation. Our methods in Section 4 show that for a different relaxation—exact counting, but only up to a given threshold—conjunctive queries of low tree-width can also be processed efficiently, simply by evaluating the associated limit query (with the same threshold) and counting the answers. However, in our main result (Theorem 1) we go far beyond conjunctive queries and show how to count answers to threshold queries (which themselves generalize the problem of counting answers to conjunctive queries up to a threshold).

Sampling an Answer with Uniform Probability. Sampling query answers was identified as an important data management task by Chaudhuri at al. [12], 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 [33, 35], this algorithm can be implemented in such a way that after a linear time preprocessing phase, independent samples can be obtained in constant time. This approach was generalized to acyclic projection-free conjunctive queries [37]. Our Theorem 1 extends the latter result in three ways: we handle non-acyclic conjunctive queries, 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 conjunctive queries. Finally, Arenas et al. [2] show that efficient *almost uniform* sampling is possible for conjunctive queries 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.

Enumerating Answers. In the context of enumerating all the answers to a query, much work has concentrated on guarantees on the *delay* between successive answers. Such query answering algorithms start with a preprocessing phase, which is followed by an enumeration phase that generates the answers one by one. A significant focus was on *constant-delay* enumeration which, after a polynomial-time preprocessing phase, aims at generating the answers to the query such that the time between successive answers is constant. In this setting, again, low tree-width is not enough to guarantee the existence of constant-delay algorithms: the query needs to have low free-connex tree-width [4]. Importantly, even acyclic queries can have large free-connex tree-width. Tree-width can be replaced with fractional hypertree-width [30, 15, 23] or submodular width [6] but always in the restrictive free-connex variant. In Section 4 we have seen that if the number of needed answers is known beforehand, conjunctive queries of low tree-width can be processed efficiently even if they have large free-connex tree-width. This result is only the starting point for processing threshold queries, for which we provide a general constant-delay enumeration algorithm (Theorem 1).

Our Results. Our paper gives two main insights. The first insight, which was explained in Section 4, is that the threshold variants of (E2) and (E4) for conjunctive queries are easier than their general variants: we show that the threshold variants of these problems are tractable if the (unrestricted) tree-width of queries is small whereas, in the general variants, restricted forms of their tree-width need to be small. The point here is that, in general, the restricted tree-width of queries is never smaller than the unrestricted tree-width; and it may be larger.

The second insight is about threshold queries, which are more general than conjunctive queries, and whose evaluation problem generalizes the threshold evaluation problems for conjunctive queries. Our main theorem about threshold queries is about counting answers (E2), sampling answers (E3), and constant-delay enumeration (E4); and its proof relies on our first insight. The result shows that, for such threshold queries of free-connex tree-width d (which is one of the restricted variants we just mentioned), these problems can be done in time $\tilde{O}(n^d)$. Here, we need free-connex tree-width because for these problems it is needed already for conjunctive queries, which are a special case of threshold queries. Recall that the value d is usually very small in practice.

THEOREM 1. *For threshold queries of free-connex tree-width d, over databases of size n, one can*

(a) count answers in time $\tilde{O}(n^d)$;

(b) enumerate answers with constant delay after $\tilde{O}(n^d)$ preprocessing; and

(c) sample answers uniformly in constant time after $\tilde{O}(n^d)$ preprocessing.

Assuming d is constant, the combined complexity of each of these algorithms is pseudopolynomial; that is, it is polynomial in the values *of the finite thresholds, rather than the number of bits in their binary representations.*

7. CONCLUSIONS

In this paper, we have shown that thresholds are commonplace in real-life queries and that with dedicated optimization techniques, we can speed up the evaluation of threshold queries, in all its flavours.

We argue that threshold queries are a category on their own and deserve separate treatment alongside other classical query classes. As future directions of investigation, we intend to explore the role of threshold queries in algebraic components of query engines, we wish to develop and extend optimization methods, and expand benchmarking efforts to broader classes of queries and more diverse data sets. In particular, we envision a comprehensive study of algebraic equivalence and optimization rules, as well as cost models and selectivity estimation methods for the new operators and their use in query optimizers. For example, the existing state-of-the-art benchmarks, such as the TPC benchmarks [32] and the Join Order Benchmark [25] could be extended to cover this important class of queries. Classical query optimization and selectivity estimation methods [11], would need to be redesigned in order to cover queries with thresholds.

8. REFERENCES

[1] M. Arenas, P. Barceló, L. Libkin, W. Martens, and A. Pieris. *Database Theory*. Open source at `https://github.com/pdm-book/community`, 2022.

[2] M. Arenas, L. A. Croquevielle, R. Jayaram, and C. Riveros. When is approximate counting for conjunctive queries tractable? In *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*, STOC 2021, pages 1015–1027, New York, NY, USA, 2021. Association for Computing Machinery.

[3] R. Asif and M. A. Qadir. Enhancing the Nobel Prize schema. In *2017 International Conference on Communication, Computing and Digital Systems (C-CODE)*, pages 193–198, Islamabad,Pakistan, 2017. IEEE.

[4] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proc. CSL 2007*, volume 4646 of *LNCS*, pages 208–222, Berlin, Heidelberg, 2007. Springer.

[5] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[6] C. Berkholz and N. Schweikardt. Constant delay enumeration with fpt-preprocessing for conjunctive queries of bounded submodular width. In *Proc. MFCS 2019*, volume 138 of *LIPIcs*, pages 58:1–58:15, Dagstuhl, Germany, 2019. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[7] A. Bonifati, S. Dumbrava, G. Fletcher, J. Hidders, M. Hofer, W. Martens, F. Murlak, J. Shinavier, S. Staworko, and D. Tomaszuk. Threshold queries in theory and in the wild. *Proc. VLDB Endow.*, 15(5):1105–1118, 2022.

[8] A. Bonifati, W. Martens, and T. Timm. Navigating the maze of wikidata query logs. In *The World Wide Web Conference*, pages 127–138, New York, NY, USA, 2019. Association for Computing Machinery.

[9] A. Bonifati, W. Martens, and T. Timm. An analytical study of large SPARQL query logs. *VLDB J.*, 29(2-3):655–679, 2020.

[10] M. J. Carey and D. Kossmann. On saying "enough already!" in sql. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, SIGMOD '97, pages 219–230, New York, NY, USA, 1997. Association for Computing Machinery.

[11] D. D. Chamberlin, M. M. Astrahan, M. W. Blasgen, J. N. Gray, W. F. King, B. G. Lindsay, R. Lorie, J. W. Mehl, T. G. Price, F. Putzolu, P. G. Selinger, M. Schkolnick, D. R. Slutz, I. L. Traiger, B. W. Wade, and R. A. Yost. A history and evaluation of system r. *Commun. ACM*, 24(10):632–646, oct 1981.

[12] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In A. Delis, C. Faloutsos, and S. Ghandeharizadeh, editors, *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*, volume 28, pages 263–274, New York, NY, USA, 1999. Association for Computing Machinery.

[13] CovidGraph. COVID-19 Knowledge Graph, 2021. `https://covidgraph.org/`.

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

[15] S. Deep and P. Koutris. Compressed representations of conjunctive query results. In J. V. den Bussche and M. Arenas, editors, *Proc. PODS 2018*, pages 307–322, New York, NY, USA, 2018. ACM.

[16] A. Durand and S. Mengel. Structural tractability of counting of solutions to conjunctive queries. *Theory Comput. Syst.*, 57(4):1202–1249, 2015.

[17] J. Finger and N. Polyzotis. Robust and efficient algorithms for rank join evaluation. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 415–428, New York, NY, USA, 2009. Association for Computing Machinery.

[18] J. Flum and M. Grohe. The parameterized complexity of counting problems. *SIAM J. Comput.*, 33(4):892–922, 2004.

[19] G. Gottlob, G. Greco, and F. Scarcello. Treewidth and hypertree width. In L. Bordeaux, Y. Hamadi, and P. Kohli, editors, *Tractability: Practical Approaches to Hard Problems*, pages 3–38. Cambridge University Press, 2014.

[20] M. Grohe, T. Schwentick, and L. Segoufin. When is the evaluation of conjunctive queries tractable? In *ACM Symposium on Theory of Computing (STOC)*, pages 657–666, New York, NY, USA, 2001. Association for Computing Machinery.

[21] ICIJ. The Offshore Leaks Database, 2022. `https://github.com/ICIJ/offshoreleaks-data-packages`.

[22] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB journal*, 13(3):207–221, 2004.

[23] A. Kara and D. Olteanu. Covers of query results. In B. Kimelfeld and Y. Amsterdamer, editors, *21st International Conference on Database Theory*, volume 98 of *LIPIcs*, pages 16:1–16:22, Vienna, Austria, 2018. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[24] M. Krötzsch. Practical linked data access via SPARQL: The case of wikidata. In *LDOW@ WWW*, pages 1–10, Lyon, France, 2018. CEUR Workshop Proceedings.

[25] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5):643–668, 2018.

[26] S. Malyshev, M. Krötzsch, L. González, J. Gonsior, and A. Bielefeldt. Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In *International Semantic Web Conference (ISWC)*, pages 376–394, Cham, 2018. Springer.

[27] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-k aggregation of ranked inputs. *ACM Transactions on Database Systems (TODS)*, 32(3):19–es, 2007.

[28] F. Murlak, J. Posiadala, and P. Susicki. On the semantics of Cypher's implicit group-by. In A. Cheung and K. Nguyen, editors, *Proceedings of the 17th ACM SIGPLAN International Symposium on Database Programming Languages, DBPL 2019, Phoenix, AZ, USA, June 23, 2019*, pages 59–69. ACM, 2019.

[29] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, volume 1, pages 281–290, San Francisco, CA, USA, 2001. Morgan Kaufmann

Publishers Inc.

[30] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *ACM Trans. Database Syst.*, 40(1):2:1–2:44, 2015.

[31] R. Pichler and S. Skritek. Tractable counting of the answers to conjunctive queries. *Journal of Computer and System Sciences*, 79(6):984–1001, Sep 2013.

[32] K. Shanley. TPC releases benchmark results on 65 systems. *SIGMETRICS Perform. Evaluation Rev.*, 19(2):19–23, 1991.

[33] M. D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on software engineering*, 17(9):972–975, 1991.

[34] D. Vrandečić. Wikidata: A new platform for collaborative data collection. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12 Companion, pages 1063–1064, New York, NY, USA, 2012. Association for Computing Machinery.

[35] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Softw.*, 3(3):253–256, 1977.

[36] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. VLDB 1981*, pages 82–94, Cannes, France, 1981. IEEE Computer Society.

[37] Z. Zhao, R. Christensen, F. Li, X. Hu, and K. Yi. Random sampling over joins revisited. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1525–1539, New York, NY, USA, 2018. Association for Computing Machinery.