Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries

Nikolaos Tziavelis
Northeastern University
Boston, MA, USA
ntziavelis@ccs.neu.edu

Deepak Ajwani
University College Dublin
Dublin, Ireland
deepak.ajwani@ucd.ie

Wolfgang Gatterbauer
Northeastern University
Boston, MA, USA
w.gatterbauer@northeastern.edu

Mirek Riedewald
Northeastern University
Boston, MA, USA
m.riedewald@northeastern.edu

Xiaofeng Yang*
VMware
Palo Alto, CA, USA
alice.xiaofeng.yang@gmail.com

ABSTRACT

We study ranked enumeration of join-query results according to very general orders defined by selective dioids. Our main contribution is a framework for ranked enumeration over a class of dynamic programming problems that generalizes seemingly different problems that had been studied in isolation. To this end, we extend classic algorithms that find the k-shortest paths in a weighted graph. For full conjunctive queries, including cyclic ones, our approach is optimal in terms of the time to return the top result and the delay between results. These optimality properties are derived for the widely used notion of data complexity, which treats query size as a constant. By performing a careful cost analysis, we are able to uncover a previously unknown trade-off between two incomparable enumeration approaches: one has lower complexity when the number of returned results is small, the other when the number is very large. We theoretically and empirically demonstrate the superiority of our techniques over batch algorithms, which produce the full result and then sort it. Our technique is not only faster for returning the first few results, but on some inputs beats the batch algorithm even when all results are produced.

PVLDB Reference Format:
Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald and Xiaofeng Yang. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. *PVLDB*, 13(9): 1582-1597, 2020.
DOI: https://doi.org/10.14778/3397230.3397250

1. INTRODUCTION

Joins are an essential building block of queries in relational and graph databases, and recent work on worst-case optimal joins for cyclic queries renewed interest in their efficient evaluation [73]. Part of the excitement stems from the fact that conjunctive query (CQ) evaluation is equivalent to other key problems such as constraint satisfaction [61] and hypergraph homomorphism [38]. Similar to [73], we consider *full conjunctive queries*, yet we are interested in *ranked enumeration*, recently identified as an important open problem [22]: return output tuples in the order determined by a given ranking function. Here success is measured not only in the time for total result computation, but the main challenge lies in *returning the top-ranked result(s) as quickly as possible.*

We share this motivation with top-k query evaluation [55], which defines the importance of an output tuple based on the *weights* of its participating input tuples. However, many top-k approaches, including the famous Threshold Algorithm [36], were developed for a middleware-centric cost model that charges an algorithm only for accesses to external data sources, but does not take into account the cost associated with potentially huge intermediate results. We want optimality guarantees in the standard RAM-model of computation for (1) the time until the first result is returned and (2) the delay between results.

EXAMPLE 1 (4-CYCLE QUERY). *Let w be a function that returns the real-valued weight of a tuple and consider 4-cycle query Q_{C4} over $R_1(A_1, A_2)$, $R_2(A_2, A_3)$, $R_3(A_3, A_4)$, and $R_4(A_4, A_1)$ with at most n tuples each:*

```
SELECT   R1.A1, R2.A2, R3.A3, R4.A4
FROM     R1, R2, R3, R4
WHERE    R1.A2=R2.A2 AND R2.A3=R3.A3 AND
         R3.A4=R4.A4 AND R4.A1=R1.A1
ORDER BY w(R1) + w(R2) + w(R3) + w(R4) ASC
LIMIT    k
```

One can compute the full output with a worst-case optimal join algorithm such as NPRR *[73] or* GENERIC-JOIN *[74] and then sort it. Since the fractional edge cover number ρ^* of Q_{C4} is 2, it takes $\mathcal{O}(n^2)$ just to produce the full output [9].*

On the other hand, the Boolean version of this query ("Is there any 4-cycle?") can be answered in $\mathcal{O}(n^{1.5})$ [69]. Our approach returns the top-ranked 4-cycle in $\mathcal{O}(n^{1.5})$ as well. This is remarkable, given that determining the existence of a 4-cycle appears easier than finding the top-ranked 4-cycle (we can use the latter to answer the former). After the top-

Proceedings of the VLDB Endowment, Vol. 13, No. 9
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3397230.3397250

*Work performed while PhD student at Northeastern University.

ranked 4-cycle is found, our approach continues to return the remaining results in ranked order with "minimal" delay.

We develop a theory of *optimal ranked enumeration over full CQs*. It reveals deeper relationships between recent work that only partially addresses the problem we are considering: Putting aside the focus on twig patterns [26] and subgraph isomorphism [90], graph-pattern ranking techniques can in principle be applied to conjunctive queries. An unpublished paper [33] that was developed concurrently with our work offers a recursive solution for ranked enumeration. All this prior work raises the question of how the approaches are related and whether they can be improved: Can time complexity of the top-k algorithm by Chang et al. [26] be improved for large k and is it possible to extend it to give optimality guarantees for cyclic queries? For [60, 90], how can the worst-case delay be reduced? Is it possible to reduce the complexity of [33] for returning the first few results and can one close the asymptotic gap between the time complexity for returning the top-ranked result and the complexity of the corresponding Boolean query for simple cycles?

It is non-trivial to answer the above questions, because those approaches blend various elements into monolithic solutions, sometimes reinventing the wheel in the process.

Key contributions. We identify and formally model *the underlying structure of the ranked enumeration problem* for conjunctive queries and then solve it in a principled way:

(1) For CQs that are paths, we *identify and formalize the deeper common foundations of problems that had been studied in isolation*: k-shortest path, top-k graph-pattern retrieval, and ranked enumeration over joins. While interesting in its own right, uncovering those relationships enables us to propose the first algorithms with *optimal time complexity for ranked enumeration* of the results of both cyclic and acyclic full CQs. In particular, the top-ranked output tuple of an acyclic join query is returned in time linear in input size. For cyclic queries this complexity increases with the submodular width (subw) of the query [69], which is currently the best known for Boolean queries. Delay between consecutive output tuples is logarithmic in k.

(2) To achieve optimality, we make several technical contributions. First, for path CQs we propose a new algorithm Take2 that has lower time complexity for returning the top-k results than all previous work but Eppstein's algorithm [35], whose practical performance is unknown. Take2 matches the latter and has the added benefit that it can be generalized to arbitrary acyclic queries.[1] Second, to generalize k-shortest path algorithms to arbitrary acyclic CQs, we introduce *ranked enumeration over Tree-based Dynamic Programming* (T-DP), a variant of Non-Serial Dynamic Programming (NSDP) [20]. Third, we propose *Union of T-DP problems* (UT-DP), a framework for *optimally incorporating in our approach all existing decompositions* of a cyclic CQ into a union of trees. Thereby, any decomposition of a full CQ that achieves optimality for the Boolean version of the query will result in an optimal algorithm for ranked enumeration over full CQs in our framework.

(3) Ranked enumeration over path CQs forms the backbone of our approach, therefore we analyze all techniques for this problem in terms of both *data and query complexity*. This is complemented by the *first empirical study* that directly compares landmark results on ranked enumeration from diverse domains such as k-shortest paths, graph-pattern search, and CQs. Our analysis reveals several interesting insights: (*i*) In terms of time complexity the best *Lawler-type* [65] approaches are asymptotically optimal for general inputs and dominate the *Recursive Enumeration Algorithm (REA)* [33, 57]. (*ii*) Since REA smartly reuses comparisons, there exist inputs for which it produces the *full ordered output* with lower time complexity than Lawler; it is even faster than sorting! Our experiments verify this behavior and suggest that Lawler-type approaches should be preferred for small k, but REA for large k. Thus we are the first to not only propose different approaches, but also reveal that *neither dominates all others*, both in terms of asymptotic complexity and measured running time. (*iii*) Even though our new Take2 algorithm has lower complexity than Lazy [26], in our environment it is often not the winner because it suffers from higher constant factors.

Further information is available on the project web page at https://northeastern-datalab.github.io/anyk/.

[1] This generalization is unknown for Eppstein and it would be challenging due to the complex nature of that algorithm.

2. FORMAL SETUP

We use $\mathbb{N}_i^j$ to denote the set of natural numbers $\{i, \dots, j\}$. For complexity results we use the standard *RAM-model of computation* that charges $\mathcal{O}(1)$ per data-element access. Reading or storing a vector of i elements therefore costs $\mathcal{O}(i)$. In line with previous work [19, 43, 73], we also assume the existence of a data structure that can be built in linear time to support tuple lookups in constant time. In practice, this is virtually guaranteed by hashing, though formally speaking, only in an expected, amortized sense.

2.1 Conjunctive Queries

Our approach can be applied to any join query, including those with theta-join conditions and projections, but we provide optimality results only for *full CQs* [73] and hence focus on them. A full CQ is a first-order formula $Q(\mathbf{x}) = (g_1 \wedge \dots \wedge g_\ell)$, written $Q(\mathbf{x}) :- g_1(\mathbf{x}_1), \dots, g_\ell(\mathbf{x}_\ell)$ in Datalog notation, where each atom g_i represents a relation $R_i(\mathbf{x}_i)$ with different atoms possibly referring to the same physical relation, and $\mathbf{x} = \bigcup_i \mathbf{x}_i$ is a set of m attributes. The *size* of the query $|Q|$ is the size of the formula. We use n to refer to the maximal cardinality of any input relation referenced in Q. Occurrence of the same variable in different atoms encodes an *equi-join* condition. A CQ can be represented by a hypergraph with the variables as the nodes and the atoms as the hyperedges; acyclicity of the query is defined in terms of the acyclicity of the associated hypergraph [42]. A *Boolean* conjunctive query just asks for the satisfiability of the formula. We use Q^B to denote the Boolean version of Q. To avoid notational clutter and without loss of generality, we assume that there are no selection conditions on individual relations.

Example 2 (ℓ-path and ℓ-cycle queries). *Let $R_i(A, B), i \in \mathbb{N}_1^\ell$, be tables containing directed graph edges from A to B. A length-ℓ path and a length-ℓ cycle can respectively be expressed as:*

$$Q_{P\ell}(\mathbf{x}) :- R_1(x_1, x_2), R_2(x_2, x_3), \dots, R_\ell(x_\ell, x_{\ell+1})$$
$$Q_{C\ell}(\mathbf{x}) :- R_1(x_1, x_2), R_2(x_2, x_3), \dots, R_\ell(x_\ell, x_1).$$

We often represent an output tuple as a vector of those input tuples that joined to produce it, e.g., $(r_1, r_2, r_3, r_4) \in$

$R_1 \times R_2 \times R_3 \times R_4$ for 4-path query Q_{P4}. We refer to this vector as the *result witness*.

2.2 Ranked Enumeration Problem

We want to order the results of a full CQ based on the weights of their corresponding witnesses. For maximal generality, we define ordering based on *selective dioids* [41], which are semirings with an ordering property:

Definition 3 (Semiring). *A* monoid *is a 3-tuple* $(W, \oplus, \bar{0})$ *where* W *is a non-empty set,* $\oplus : W \times W \to W$ *is an associative operation, and* $\bar{0}$ *is the identity element, i.e.,* $\forall x \in W : x \oplus \bar{0} = \bar{0} \oplus x = x$. *In a commutative monoid,* $\oplus$ *is also commutative. A* semiring *is a 5-tuple* $(W, \oplus, \otimes, \bar{0}, \bar{1})$*, where* $(W, \oplus, \bar{0})$ *is a commutative monoid,* $(W, \otimes, \bar{1})$ *is a monoid,* $\otimes$ *distributes over* $\oplus$*, i.e.,* $\forall x, y, z \in W : (x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$*, and* $\bar{0}$ *is absorbing for* $\otimes$*, i.e.,* $\forall a \in W : a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$.

Definition 4 (Selective dioid). *A* selective dioid *is a semiring for which* $\oplus$ *is* selective*, i.e., it always returns one of the inputs:* $\forall x, y \in W : (x \oplus y = x) \vee (x \oplus y = y)$.

Note that $\oplus$ being selective induces a total order on W by setting $x \leq y$ iff $x \oplus y = x$. We define result weight as an aggregate of input-tuple weights using $\otimes$:

Definition 5 (Tuple Weights). *Let* w *be a weight function that maps each input tuple to some value in* W *and let* $Q(\mathbf{x}) :- R_1(\mathbf{x}_1), \ldots, R_\ell(\mathbf{x}_\ell)$ *be a full CQ. The* weight *of a result tuple* r *is the weight of its witness* $(r_1, \ldots, r_\ell)$*,* $r_i \in R_i$*,* $i \in \mathbb{N}_1^\ell$*, defined as* $w(r) = w(r_1) \otimes \cdots \otimes w(r_\ell)$.

Recall Example 1 where we rank output tuples by the sum of the weights of the corresponding input tuples, i.e., the weight of $(r_1, \ldots, r_\ell)$ is $\sum_{i=1}^{\ell} w(r_i)$. We achieve this by using the selective dioid $(\mathbb{R}^\infty, \min, +, \infty, 0)$ with $\mathbb{R}^\infty = \mathbb{R} \cup \{\infty\}$ that is also called the *tropical semiring*.

The central problem in this paper is the following:

Definition 6 (Ranked enumeration). *Given a query* Q *over an input database* D*, selective dioid* $(W, \oplus, \otimes, \bar{0}, \bar{1})$*, and weight function* w *as defined above, a* ranked enumeration *algorithm returns the output of* Q *on* D *according to the total order induced by* $\oplus$.

We refer to algorithms for ranked enumeration over the results of a CQ as *any-k* algorithms. This conforms to our previous work [90] and reflects the fact that the number of returned results need not be set apriori.

Generality. Our approach supports any selective dioid, including less obvious cases such as *lexicographic ordering* where two output tuples are first compared on their R_1 component, and if equal then on their R_2 component, and so on. For this to be well-defined, there must be a total order on the tuples within each relation. Without loss of generality, assume this total order is represented by the natural numbers, such that input tuple r has weight $w'(r) \in \mathbb{N}$. For the selective dioid, we set $W = \mathbb{N}^\ell$, i.e., each tuple weight is an ℓ-dimensional vector of integers. Input tuple $r_j \in R_j$ has weight $w(r_j) = (0, \ldots, 0, w'(r_j), 0, \ldots, 0)$ with zeros except for position j that stores the "local" weight value in R_j. Operator $\otimes$ is standard element-wise vector addition, therefore the weight of a result tuple with witness $(r_1, \ldots, r_\ell)$ is $(w'(r_1), \ldots, w'(r_\ell))$. To order two such vectors, $\oplus$ implements element-wise minimum, e.g., for $\ell = 2$, $(a, b) \oplus (c, d) = (a, b)$ iff $w'(a) < w'(c)$ or $w'(a) = w'(c)$ and $w'(b) < w'(d)$. The $\bar{0}$ and $\bar{1}$ elements of the dioid are ℓ-dimensional vectors $(\infty, \ldots, \infty)$ and $(0, \ldots, 0)$, respectively.

We will present our approach for the tropical semiring $(\mathbb{R}^\infty, \min, +, \infty, 0)$. Generalization to other selective dioids follows immediately from the fact that the only algebraic properties that are used in our derivations are those mentioned in Definitions 3 and 4. Note that addition over real numbers has an *inverse*, hence $(\mathbb{R}^\infty, +, 0)$ is a group, not just a monoid. This simplifies the algorithm somewhat, but our main result (Theorem 13) holds even without the inverse with some minor subtleties (see the full version [87]).

2.3 Determining Optimality

An any-k algorithm should return the first result as soon as possible, then the next results with minimal delay. We therefore ask "how long does it take to return the k top-ranked results, *for any value of* k?" and use $\mathrm{TT}(k)$ and $\mathrm{MEM}(k)$ to denote time and space complexity, respectively, to produce the k top results. We will pay particular attention to the special cases of *time-to-first* ($\mathrm{TTF} = \mathrm{TT}(1)$) and *time-to-last* ($\mathrm{TTL} = \mathrm{TT}(|\mathrm{out}|)$), where out denotes the output of the query. In line with most previous work on worst-case optimal join algorithms and decompositions of cyclic queries, we measure asymptotic cost in terms of data complexity [88], i.e., treat query size $|Q|$ as a constant. The exception is the in-depth analysis of ranked enumeration algorithms for path CQs (Section 4.3), where including query complexity reveals interesting differences.

Consider full CQ Q over input relations with at most n tuples. It takes $\mathcal{O}(n)$ just to look at each input tuple and $\mathcal{O}(k)$ to output k result tuples, establishing $\Omega(n + k)$ as a *lower bound* for $\mathrm{TT}(k)$. Since we also require the output to be sorted and sorting k items has complexity $\Omega(k \log k)$, we consider a ranked enumeration algorithm to be *optimal* if it satisfies $\mathrm{TT}(k) = \mathcal{O}(n + k \log k)$.[2] For *acyclic* CQs, this $\mathrm{TT}(k)$ optimality target is realistic, because the well-known Yannakakis algorithm [92] computes the full (unsorted) output in time $\mathcal{O}(n + |\mathrm{out}|)$.

For *cyclic* CQs, Ngo et al. [73] argue that the join result cannot be computed in $\mathcal{O}(n + |\mathrm{out}|)$ and propose the notion of *worst-case optimal* (WCO) join algorithms, whose computation time is $\mathcal{O}(n + |\mathrm{out}_{\mathrm{WC}}|)$. Here, $|\mathrm{out}_{\mathrm{WC}}|$ is the maximum output size of query Q over *any possible* database instance, which is determined by the AGM bound [9]. WCO join algorithms are thus *not sensitive* to the actual output size of the query on a given database instance. Abo Khamis et al. [5] argue for a stronger, output-sensitive notion of optimality based on the *width* ω of a decomposition of a cyclic CQ Q into a set $\overline{Q}$ of acyclic CQs covering Q.[3] The input relations of the acyclic CQs in $\overline{Q}$ are derived from the original input and have cardinality $\mathcal{O}(n^\omega)$ for $\omega \geq 1$ ideally as small as possible. Let $\mathcal{A}$ be such a decomposition-based algorithm and let $T(\mathcal{A})$ denote its time complexity for creating decomposition $\overline{Q}$. By applying the Yannakakis algorithm to the acyclic queries in $\overline{Q}$, cyclic query Q can be evaluated in time $\mathcal{O}(T(\mathcal{A}) + |\mathrm{out}|)$ and its Boolean version Q^B in

[2] To be precise, sorting may add less than $k \log k$ overhead if one can replace generic comparison-based sorting with an algorithm that exploits structural relationships between weights of input and output tuples. However, this is not possible for all inputs and k values.

[3] The union of their output equals the output of Q.

$\mathcal{O}(T(\mathcal{A}))$. The current frontier are decompositions based on the *submodular* width $\omega = \mathsf{subw}(Q)$ [69], which is considered a yardstick of optimality for full and Boolean CQs [5].

We adopt this notion of optimality and, arguing similar to the acyclic case, we say that ranked enumeration over a full CQ is optimal if $\mathrm{TT}(k) = \mathcal{O}(T(\mathcal{A}) + k \log k)$. Intuitively, this ensures that ranked enumeration adds "almost no overhead" compared to unranked enumeration, because outputting k results would take at least $\Omega(k)$.

3. PATH QUERY AND ITS CONNECTION TO DYNAMIC PROGRAMMING (DP)

We formulate optimal ranked enumeration for path queries as a Dynamic Programming (DP) problem, then generalize to trees and cyclic queries. Following common terminology, we use DP to denote what would more precisely be called *deterministic serial DP* with a finite fixed number of decisions [21, 29, 30]. These problems have a *unique minimum* of the cost function and DP constructs a *single* solution that realizes it. Formally, a DP problem has a set of *states* S, which contain local information for decision-making [21]. We focus on what we will refer to as *multi-stage DP*. Here each state belongs to exactly one of $\ell > 0$ stages, where S_i denotes the set of states in stage i, $i \in \mathbb{N}_0^\ell$. The *start* stage has a single state $S_0 = \{s_0\}$ and there is a *terminal* state $s_{\ell+1}$ which we also denote by t for convenience. At each state s of stage i, we have to make a *decision* that leads to a state $s' \in S_{i+1}$. We use $E \subseteq \bigcup_{i=0}^{\ell}(S_i \times S_{i+1})$ for the set of possible decisions.

DP is equivalent to a shortest-path problem on a corresponding weighted graph, in our case a $(\ell+2)$-partite directed acyclic graph (DAG) [21, 30], where states correspond to nodes and decisions define the corresponding edges. Each decision (s, s') is associated with a *cost* $w(s, s')$, which defines the weight of the corresponding edge in the shortest-path problem.[4] By convention, an edge exists iff its weight is less than ∞.

We now generalize the path definition from Example 2 and show that ranked enumeration over this query can be modeled as an instance of DP. Consider

$$Q'_{P\ell}(\mathbf{x}, \mathbf{y}) :- R_1(\mathbf{y}_1, \mathbf{x}_2), R_2(\mathbf{x}_2, \mathbf{y}_2, \mathbf{x}_3), \ldots, R_\ell(\mathbf{x}_\ell, \mathbf{y}_\ell, \mathbf{x}_{\ell+1}),$$

allowing multiple attributes in the equi-join conditions and additional attribute sets $\mathbf{y}_i$ that do not participate in joins. This query can be mapped to a DP instance as follows: (1) atom R_i corresponds to stage S_i and each tuple in R_i maps to a unique state in S_i, (2) there is an edge between $s \in S_i$ and $s' \in S_{i+1}$ iff the corresponding input tuples join and the edge's weight is the weight of the tuple corresponding to s', (3) there is an edge from s_0 to each state in S_1 whose weight is the weight of the corresponding R_1-tuple, and (4) each state in S_ℓ has an edge to t of weight 0. Clearly, there is a 1:1 correspondence between paths from s_0 to t and output tuples of $Q'_{P\ell}$, and path "length" (weight) equals output-tuple weight. Hence the k-th heaviest output tuple corresponds to the k-shortest path in the DP instance.

EXAMPLE 7 (CARTESIAN PRODUCT). *We use the problem of finding the minimum-weight output of Cartesian product $R_1 \times R_2 \times R_3$ as the running example. Let $R_1 = \{$"1", "2", "3"$\}$, $R_2 = \{$"10", "20", "30"$\}$ and $R_3 = \{$"100", "200", "300"$\}$ and set tuple weight equal to tuple label, e.g., tuple "20" in R_2 has weight $w($"20"$) = 20$. Fig. 1 depicts how this problem translates into our framework.*

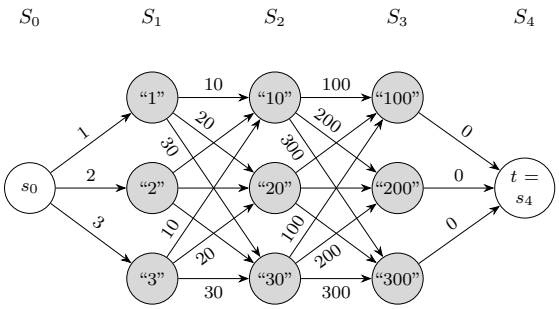


Figure 1: DP instance for Example 7.

A *solution* to the DP problem is a sequence of ℓ states $\Pi = \langle s_1 \ldots s_\ell \rangle$ that is *admissible*, i.e. $(s_i, s_{i+1}) \in E, \forall i \in \mathbb{N}_0^\ell$. The *objective function* is the total cost of a solution,

$$w(\Pi) = \sum_{i=0}^{\ell} w(s_i, s_{i+1}), \tag{1}$$

and DP finds the minimal-cost solution Π_1. The index denotes the rank, i.e., Π_k is the k-th best solution.

Principle of optimality. [15, 16] The core property of DP is that a solution can be efficiently derived from solutions to subproblems. In the shortest-path view of DP, the subproblem at *any* state $s \in S_i$ is the problem of finding the shortest path from s to t. With $\Pi_1(s)$ and $\pi_1(s)$ denoting the shortest path from s and its weight respectively, DP is recursively defined for all states $s \in S_i, i \in \mathbb{N}_0^{\ell+1}$ by

$$\begin{aligned} &\pi_1(s) = 0 \text{ for terminal } s \in S_{\ell+1} \\ &\pi_1(s) = \min_{(s,s') \in E} \{w(s, s') + \pi_1(s')\}, \text{ for } s \in S_i, i \in \mathbb{N}_0^\ell. \end{aligned} \tag{2}$$

The optimal DP solution is $\pi_1(s_0)$, i.e., the weight of the lightest path from s_0 to t. For convenience we define the set of optimal paths reachable from s according to Eq. (2) as $\texttt{Choices}_1(s) = \{s \circ \Pi_1(s') \mid (s, s') \in E\}$. Here $\circ$ denotes concatenation, i.e., $s_i \circ \langle s_{i+1} \ldots s_\ell \rangle = \langle s_i \; s_{i+1} \ldots s_\ell \rangle$.

EXAMPLE 8 (CONTINUED). *Consider state "2" in Fig. 2. It has three outgoing edges and $\pi_1($"2"$)$ is computed as the minimum over these three choices. The winner is path "2" $\circ$ $\Pi_1($"10"$)$ of weight 112. Similarly, $\Pi_1($"10"$)$ is found as "10" $\circ$ $\Pi_1($"100"$)$, and so on.*

Equation (2) can be computed for all states in time $\mathcal{O}(|S| + |E|)$ bottom-up, i.e., in decreasing stage order from $\ell + 1$ to 0. Consider stage S_i: To compute $\texttt{Choices}_1(s)$ for state $s \in S_i$, the algorithm retrieves all edges $(s, s') \in E$ from s to any state $s' \in S_{i+1}$, looks up $\pi_1(s')$, and keeps track of the minimal total weight $w(s, s') + \pi_1(s')$ on-the-fly. (If no such edge is found, then the weight is set to ∞.) When computing $\pi_1(s)$, the algorithm also adds pointers to keep track of optimal solutions. E.g., in Fig. 2 entry "2" $\circ$ $\Pi_1($"30"$)$ at state "2" would point to the minimum-weight choice "30" $\circ$ $\Pi_1($"100"$)$ at state "30". This way the corresponding paths can be reconstructed by tracing the pointers back "top-down" from $\pi_1(s_0)$ [21]. Notice that *DP needs only the pointer from the top choice at each state*, but

[4] We use *cost* and *weight* interchangeably. Cost is more common in optimization problems, weight in shortest-path problems. We sometimes use "lightest path" in order to emphasize that all paths have the same number of nodes, but differ in their weights.

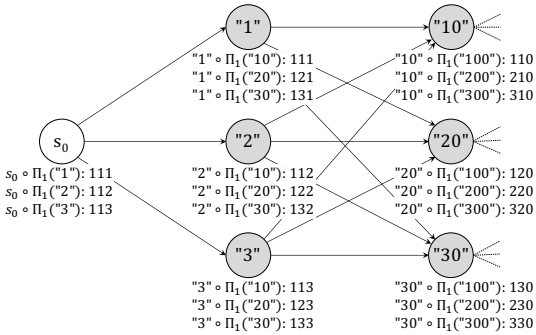


Figure 2: Excerpt from Fig. 1, showing $\texttt{Choices}_1(s)$ for some states s. Term $s \circ \Pi_1(s') : w$ denotes a choice, which is a path from s, and its weight $w = w(s, s') + \pi_1(s')$.

adding the others is "free" complexity-wise, which we later use for ranked enumeration.

Whenever the bottom-up phase determines $\pi_1(s) = \infty$ during the evaluation of Eq. (2), then that state s and all its adjacent edges can be removed without affecting the space of solutions. We use $\mathbb{S}_i \subseteq S_i$ and $\mathbb{E} \subseteq E$ to denote the *remaining sets of states and decisions*, respectively. This DP algorithm corresponds to variable elimination [31] on the *tropical semiring* [39, 78] and is reminiscent of the *semi-join reductions by Yannakakis* [92], which corresponds to DP with variable elimination on the Boolean semiring [2].

Encoding equi-joins efficiently. For an equi-join, the shortest-path problem has $\mathcal{O}(\ell n)$ states and $\mathcal{O}(\ell n^2)$ edges, therefore the DP algorithm has quadratic time complexity in the number of tuples. We reduce this to $\mathcal{O}(\ell n)$ by an *equi-join specific graph transformation* illustrated in Fig. 3. Consider the join between R_1 and R_2, representing stages S_1 and S_2, respectively. For each join-attribute value, the corresponding states in R_1 and R_2 form a fully connected bipartite graph. For each state, all *incoming* edges have the same weight, as edge weight is determined by tuple weight. Hence we can represent the subgraph equivalently with a single node "in-between" the matching states in S_1 and S_2, assigning zero weight to the edges adjacent to states in S_1 and the corresponding tuple weight to those adjacent to a state in S_2. The transformed representation has only $\mathcal{O}(\ell n)$ edges. At its core, our encoding relies on the conditional independence of the non-joining attributes given the join attribute value, a property also exploited in factorized databases [75]. Here we provide a different perspective on it as a graph transformation that preserves all paths.

4. ANY-K ALGORITHMS FOR DP

We defined a class of DP problems that can be described in terms of a multi-stage DAG, where every solution is equivalent to a path from s_0 to t in graph $(\mathbb{S} = \bigcup_{i=0}^{\ell+1} \mathbb{S}_i, \mathbb{E})$. Hence we use terminology from DP (solution, state, decision) and graphs (path, node, edge) interchangeably.

In addition to the minimum-cost path, ranked enumeration must retrieve *all paths* in cost order. Let $\Pi_k(s)$ be the k^{th}-shortest path from s to t and $\pi_k(s)$ its cost. The asymptotically best k-shortest-paths algorithm was proposed by Eppstein [35], yet it is not the best choice for our use case. In the words of its author, it is "rather complicated", thus it is unclear how to extend it from path to *tree* queries. Since our DP problems are only concerned with *multi-stage DAGs*

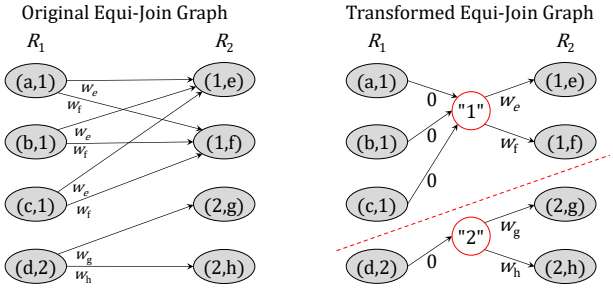


Figure 3: *Equi-join* from $\mathcal{O}(n^2)$ representation to $\mathcal{O}(n)$.

(Eppstein targets more general graphs), we propose a simpler and easier-to-extend algorithm, TAKE2, that guarantees the same complexity as Eppstein.[5]

Below we explore algorithms that fall into two categories. The first appeared in various optimization contexts as methods that *partition* the solution space and trace their roots to Lawler [65] and Murty [70], including recent work on subgraph isomorphism [26]. We call this family ANYK-PART; it includes TAKE2. The second finds the k-shortest paths in a graph via *recursive equations* [34, 57]. We refer to the application of this idea to our framework as ANYK-REC.

4.1 Repeated Partitioning DP (ANYK-PART)

4.1.1 The Lawler Procedure and DP

Lawler [65] proposed a procedure for ranked enumeration by repeatedly *partitioning the solution space*, which can be applied to any optimization problem over a fixed set of variables, not only DP. In our problem, there is one variable per stage and it can take any state in that stage as a value. Lawler only assumes the existence of a method `best` that returns the optimal variable assignment over any space $S'_1 \times \cdots \times S'_\ell$, where $\forall i : S'_i \subseteq S_i$.

The top-ranked solution $\langle s^*_1 \dots s^*_\ell \rangle$ is obtained by executing `best` on the unconstrained space $S_1 \times \cdots \times S_\ell$. To find the second-best solution, Lawler creates ℓ *disjoint subspaces* such that subspace i has the first $i-1$ variables fixed to the top-ranked solution's prefix $\langle s^*_1 \dots s^*_{i-1} \rangle$ and the i-th variable restricted to $S_i - \{s^*_i\}$. Then it applies `best` to each of these subspaces to find the top solution in each. The second-best overall solution is the best of these ℓ subspace solutions. The procedure continues analogously by generating the corresponding subspaces for the second-best solution, adding them to a priority queue of candidates.

Chang et al. [26] showed that the k^{th}-ranked solution $\langle s_1 \dots s_\ell \rangle$ is the output of `best` on some subspace

$$P = \{s_1\} \times \cdots \times \{s_{r-1}\} \times (S_r - U_r) \times S_{r+1} \times \cdots \times S_\ell, \quad (3)$$

with U_r being a set of states excluded from S_r. The new candidates to be added to the candidate set for the $(k+1)^{\text{st}}$ result are the results obtained by executing `best` on the following $\ell - r + 1$ subspaces:

$$\begin{aligned} P_r &= \{s_1\} \times \cdots \times \{s_{r-1}\} \times (S_r - U_r - \{s_r\}) \times S_{r+1} \times \cdots \times S_\ell \\ P_{r+1} &= \{s_1\} \times \cdots \times \{s_{r-1}\} \times \{s_r\} \times (S_{r+1} - \{s_{r+1}\}) \times \cdots \times S_\ell \\ &\vdots \\ P_\ell &= \{s_1\} \times \cdots \times \{s_{r-1}\} \times \cdots \times \{s_{\ell-1}\} \times (S_\ell - \{s_\ell\}). \end{aligned}$$

[5] Implementations of "Eppstein's algorithm" exist, but they seem to implement a simpler variant with weaker asymptotic guarantees that was also introduced in [35].

Efficient computation. Instead of calling `best` from scratch on each subspace, we propose to exploit the structure of DP. Consider any subspace P as defined in Eq. (3). Since prefix $\langle s_1 \dots s_{r-1} \rangle$ is fixed, we need to find the best suffix starting from state s_{r-1}. In the next stage S_r, only states that are *not* in exclusion set U_r can be selected, i.e., the set of choices at s_{r-1} is restricted by U_r. Formally,

$$\texttt{best}(P) = \langle s_1 \dots s_{r-1} s \rangle \circ \Pi_1(s), \text{ where} \tag{4}$$

$$s = \arg \min_{s' \in S_r - U_r} \{ w(s_{r-1}, s') + \pi_1(s') |$$

$$s_{r-1} \circ \Pi_1(s') \in \texttt{Choices}_1(s_{r-1}) \}, \tag{5}$$

therefore Eq. (5) can be solved *using only information that was already computed by the standard DP algorithm.* Note that all elements in a choice set other than the minimum-weight element are often referred to as *deviations* from the optimal path.

EXAMPLE 9 (CONTINUED). *After returning* $\Pi_1(s_0) = \langle$"1" "10" "100"$\rangle$, *Lawler would solve three new optimization problems to find the second-best result. The first subspace is the set of paths that start at* s_0, *but cannot use state* "1". *The second has prefix* $\langle$"1"$\rangle$ *and cannot use state "10". The third has prefix* $\langle$"1" "10"$\rangle$ *and cannot use state* "100". *The best solution to the first subproblem is* $\langle$"2" "10" "100"$\rangle$, *corresponding to deviation* $s_0 \circ \pi_1($"2"$)$ *of weight 112. For the second subproblem, the best result is found similarly as the second-best option* "1" $\circ \pi_1($"20"$) = \langle$"1" "20" "100"$\rangle$. *For the third subproblem, the best subspace solution* $\langle$"1" "10" "200"$\rangle$ *is obtained analogously at state* "10".

4.1.2 The ANYK-PART family of algorithms

We propose a generic template for ANYK-PART algorithms and show how all existing approaches and our novel TAKE2 algorithm are obtained as specializations based on how the Lawler-created subspace candidates are managed. All ANYK-PART algorithms first execute standard DP, which produces for each state s the shortest path $\Pi_1(s)$, its weight $\pi_1(s)$, and set of choices $\texttt{Choices}_1(s)$. The main feature of ANYK-PART is a set `Cand` of *candidates*: it manages the best solution(s) found in each of the subspaces explored so far. To produce the next result, the ANYK-PART algorithm (Algorithm 1) (1) removes the lightest candidate from the candidate set `Cand`, (2) expands it into a complete solution, and (3) adds all new candidates found in the corresponding subspaces to `Cand`. We implement `Cand` using a priority queue with combined logarithmic time for removing the top element and inserting a batch of new candidates.

EXAMPLE 10 (CONTINUED). *The standard DP algorithm identifies* $\langle$"1" "10" "100"$\rangle$ *as the shortest path and generates the choice sets as shown in Fig. 2. Hence* `Cand` *initially contains only candidate* $(\langle s_0 \rangle, \text{"1"}, 0, 1 + 110 = 111)$ *(Line 6), which is popped in the first iteration of the repeat-loop (Line 7), leaving* `Cand` *empty for now. The for-loop (Line 11) is executed for stages 1 to* $\ell = 3$. *For stage 1, we have* tail $= s_0$ *and* last $=$ "1". *For the successor function (Line 15), there are different choices as we discuss in more detail in Section 4.1.3. For now, assume* $\texttt{Succ}(x, y)$ *returns the* next-best choice at state x after the previous choice y. *Hence the successor of* "1" *at state* s_0 *is* "2". *As a result, newCandidate is set to* $(\langle s_0 \rangle, \text{"2"}, 0, 2 + 110)$*—it is the winner for the first subspace—and added to* `Cand`. *Then the solution is expanded (Line 19) to* $(\langle s_0 \rangle$ "1"$\rangle$, "10", $1, 10 + 100)$, *because* "10" *is the best choice from* "1". *The next iteration of the outer for-loop (Line 11) adds candidate* $(\langle s_0 \rangle$ "1"$\rangle$, "20", $1, 20 + 100)$ *to* `Cand` *and updates the solution to* $(\langle s_0 \rangle$ "1" "10"$\rangle$, "100", $11, 100)$. *The third and final iteration adds candidate* $(\langle s_0 \rangle$ "1" "10"$\rangle$, "200", $11, 200)$ *and updates the solution to* $(\langle s_0 \rangle$ "1" "10" "100"$\rangle, t, 111, 0)$, *which is returned as the top-1 result.*

At this time, `Cand` *contains entries* $(\langle s_0 \rangle$, "2", $0, 112)$, $(\langle s_0 \rangle$ "1"$\rangle$, "20", $1, 120)$, *and* $(\langle s_0 \rangle$ "1" "10"$\rangle$, "200", $11, 200)$. *Note that each is the shortest path in the corresponding subspace as defined by the Lawler procedure. Among the three,* $(\langle s_0 \rangle$, "2", $0, 112)$ *is popped next, because it has the lowest sum of prefix-weight (0) and choice-weight (112). The first new candidate created for it is* $(\langle s_0 \rangle$, "3", $0, 113)$, *followed by* $(\langle s_0 \rangle$ "2"$\rangle$, "20", $2, 120)$, *and* $(\langle s_0 \rangle$ "2" "10"$\rangle$, "200", $12, 200)$. *At the same time, the solution is expanded to* $(\langle s_0 \rangle$ "2" "10" "100"$\rangle, t, 112, 0)$.

Algorithm 1: ANYK-PART

```
1  Input: DP problem with stages S_1, ..., S_ℓ
2  Output: solutions in increasing order of weight
3  Execute standard DP algorithm to produce for each state s:
     Π_1(s), π_1(s), and Choices_1(s)
4  //Initialize candidate set with top-1 result ⟨s*_1 ... s*_ℓ⟩
5  //A candidate consists of 4 fields: prefix ⟨s_1 ... s_{r-1}⟩,
     lastState s_r, prefixWeight w(⟨s_1 ... s_{r-1}⟩), and
     choiceWeight w(s_{r-1}, s_r) + π_1(s_r).
6  Cand.add([⟨s*_0⟩, s*_1, 0, w(s*_0, s*_1) + π_1(s*_1)])
7  repeat
8      //Pop the candidate with the lowest sum of prefixWeight
         and choiceWeight. Let that be
         [⟨s_1 ... s_{r-1}⟩, s_r, w(⟨s_1 ... s_{r-1}⟩), w(s_{r-1}, s_r) + π_1(s_r)]
9      solution = Cand.popMin()
10     //Complete the partial solution with the optimal suffix
         and generate new candidates in all subspaces.
11     for stages from r to ℓ do
12         //Expand the prefix to the next stage. The tail of a
             prefix is its last element. Succ(x, y) returns an
             appropriate subset of Choices_1(x).
13         tail = solution.prefix.tail
14         last = solution.lastState
15         for s ∈ Succ(tail, last) do
16             newCandidate = (solution.prefix, s,
                 solution.prefixWeight, w(tail, s) + π_1(s))
17             Cand.add(newCandidate)
18         //Update solution by appending the last state to the
             prefix.
19         solution.prefix.append(last)
20         solution.prefixWeight.add(w(tail, last))
21         s' = arg min_{s''} {w(last, s'') + π_1(s'') | last ∘ Π_1(s'') ∈
             Choices_1(last)}
22         solution.lastState = s'
23         solution.choiceWeight = w(last, s') + π_1(s')
24     output solution
25 until query is interrupted or Cand is empty
```

4.1.3 Instantiations of ANYK-PART

The main design decision in Algorithm 1 is how to manage the choices at each state and how to implement successor-finding (Line 15) over these choices.

Strict approaches. A natural implementation of the successor function returns precisely the next-best choice.

Eager Sort (EAGER): Since a state might be reached repeatedly through different prefixes, it may pay off to pre-sort all choice sets by weight and add pointers from each choice to the next one in sort order. Then $\texttt{Succ}(x, y)$ returns the

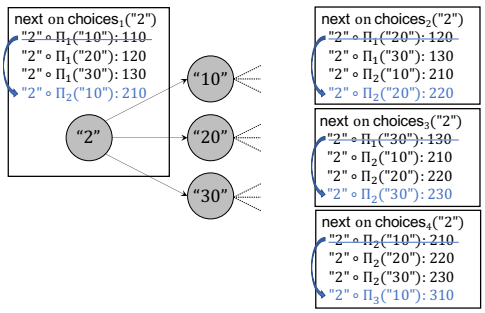


Figure 4: Example 11: Recursive enumeration at state "2".

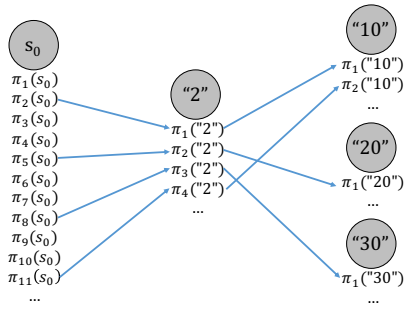


Figure 5: Pointers between solutions from and to "2".

next-best choice at x in constant time by following the next-pointer from y.

Lazy Sort (Lazy): For lower pre-processing cost, we can leverage the approach Chang et al. [26] proposed in the context of graph-pattern search. Instead of sorting a choice set, it constructs a binary heap in linear time. Since all but one of the successor requests in a single repeat-loop execution are looking for the second-best choice[6], the algorithm already pops the top two choices off the heap and moves them into a sorted list. For all other choices, the first access popping them from the heap will append them to the sorted list that was initialized with the top-2 choices. As the algorithm progresses, the heap of choices gradually empties out, filling the sorted list and thereby converging to Eager.

Relaxed approaches. Instead of finding the *single true successor* of a choice, what if the algorithm could return a set of *potential successors*? Correctness is guaranteed, as long as the true successor is contained in this set or is already in `Cand`. (Adding potential successors early to `Cand` does not affect correctness, because they have higher weight and would not be popped from `Cand` until it is "their turn.") This relaxation may enable faster successor finding, but inserts candidates earlier into `Cand`.

All choices (All): This approach is based on a construction that Yang et al. [90] proposed for any-k queries in the context of graph-pattern search. Instead of trying to find the true successor of a choice, *all* but the top choice are returned by `Succ`. While this avoids any kind of pre-processing overhead, it inserts $\mathcal{O}(n)$ potential successors into `Cand`.

Take2: We propose a new approach that has better asymptotic complexity than any of the above. Intuitively, we want to keep pre-processing at a minimum (like All), but also return a few successors fast (like Eager). To this end, we organize each choice set as a binary heap. In this tree structure, the root node is the minimum-weight choice and the weight of a child is always greater than its parent. Function `Succ`(x, y) (Line 15) returns the two children of y in the tree. Unlike Lazy, we never perform a pop operation and the heap stays intact for the entire operation of the algorithm; it only serves as a partial order on the choice set, pointing to two successors every time it is accessed. Also note that the true successor does not necessarily have to be a child of node y. Overall, returning two successors is asymptotically the same as returning one and heap construction time is linear [29], hence this approach asymptotically dominates the others.

[6] During each execution of the repeat-loop, only the first iteration of Line 11 looks for a lower choice.

4.2 Recursive Enumeration DP (ANYK-REC)

anyK-rec relies on a generalized principle of optimality [68]: if the k-th path from start node s_0 goes through $s \in \mathbb{S}_1$ and takes the j_s-lightest path $\Pi_{j_s}(s)$ from there, then the next lightest path from s_0 that goes through s will take the (j_s+1)-lightest path $\Pi_{j_s+1}(s)$ from there. We will refer to the prototypical algorithm in this space as Recursive [57]. Recall that lightest path $\Pi_1(s_0)$ from start node s_0 is found as the minimum-weight path in $\texttt{Choices}_1(s_0)$. Assume it goes through $s \in \mathbb{S}_1$. Through which node does the 2nd-lightest path $\Pi_2(s_0)$ go? It has to be either the 2nd-lightest path through s, of weight $w(s_0, s) + \pi_2(s)$, or the lightest path through any of the other nodes adjacent to s_0. In general, the k-th lightest path $\Pi_k(s_0)$ is determined as the lightest path in some later version $\texttt{Choices}_k(s_0) = \{s_0 \circ \Pi_{j_s}(s) \mid (s_0, s) \in \mathbb{E}\}$ of the set of choices, for appropriate values of j_s. Let $\Pi_k(s_0) = s_0 \circ \Pi_{j_{s'}}(s')$. Then the $(k+1)^{\text{st}}$ solution $\Pi_{k+1}(s_0)$ is found as the minimum over the same set of choices, except that $s_0 \circ \Pi_{j_{s'}+1}(s')$ replaces $s_0 \circ \Pi_{j_{s'}}(s')$. To find $\Pi_{j_{s'}+1}(s')$, the same procedure is applied recursively at s' *top-down*. Intuitively, an iterator-style `next` call at start node s_0 triggers a chain of ℓ such `next` calls along the path that was found in the previous iteration.

Example 11 (continued). *Consider node "2" in Fig. 1. Since it has adjacent states "10", "20", and "30" in the next stage, the lightest path* Π_1("2") *is selected from* $\texttt{Choices}_1$("2") = {"2" ∘ Π_1("10"), "2" ∘ Π_1("20"), "2" ∘ Π_1("30")} *as shown in Fig. 2. The first* `next` *call on state "2" returns* "2" ∘ Π_1("10"), *updating the set of choices for* Π_2("2") *to* {"2" ∘ Π_2("10"), "2" ∘ Π_1("20"), "2" ∘ Π_1("30")} *as shown in the left box in Fig. 4. The subsequent* `next` *call on state "2" then returns* "2" ∘ Π_1("20") *for* Π_2("2"), *causing* "2" ∘ Π_1("20") *in* $\texttt{Choices}_2$("2") *to be replaced by* "2" ∘ Π_2("20") *for* $\texttt{Choices}_3$("2"); *and so on.*

As the lower-ranked paths starting at various nodes in the graph are computed, each node keeps track of them for producing the results as shown in Fig. 5. For example, the pointer from Π_1("2") to Π_1("10") at node "10" was created by the first `next` call on "2", which found "2" ∘ Π_1("10") as the lightest path in the choice set. For details see [87].

4.3 Any-k DP Algorithm Complexity

In contrast to the discussion in Section 2.3, which focused on data complexity and treated query size as a constant, we now include query size in the analysis to uncover more subtle performance tradeoffs between the different any-k approaches. Since each input relation has at most n tuples, the DP problem has $\mathcal{O}(\ell n)$ nodes, each with at most n outgoing

Algorithm	TT(k)	TTL for $\|\text{out}\| = \Omega(\ell n)$	TTL for $\|\text{out}\| = \Theta(n^\ell)$	MEM(k)
Recursive	$\mathcal{O}(\ell n + k\ell \log n)$	$\mathcal{O}(\|\text{out}\|\ell \log n)$	$\mathcal{O}(n^\ell(\log n + \ell))$	$\mathcal{O}(\ell n + k\ell)$
Take2	$\mathcal{O}(\ell n + k(\log k + \ell))$	$\mathcal{O}(\|\text{out}\|(\log \|\text{out}\| + \ell))$	$\mathcal{O}(n^\ell \cdot \ell \log n)$	$\mathcal{O}(\ell n + k\ell)$
Lazy	$\mathcal{O}(\ell n + k(\log k + \ell + \log n))$	$\mathcal{O}(\|\text{out}\|(\log \|\text{out}\| + \ell))$	$\mathcal{O}(n^\ell \cdot \ell \log n)$	$\mathcal{O}(\ell n + k\ell)$
Eager	$\mathcal{O}(\ell n \log n + k(\log k + \ell))$	$\mathcal{O}(\|\text{out}\|(\log \|\text{out}\| + \ell))$	$\mathcal{O}(n^\ell \cdot \ell \log n)$	$\mathcal{O}(\ell n + k\ell)$
All	$\mathcal{O}(\ell n + k(\log k + \ell n))$	$\mathcal{O}(\|\text{out}\|(\log \|\text{out}\| + \ell))$	$\mathcal{O}(n^\ell \cdot \ell \log n)$	$\mathcal{O}(\ell n + \min\{kn, \|\text{out}\|\}\ell)$
Batch	$\mathcal{O}(\ell n + \|\text{out}\|(\log \|\text{out}\| + \ell))$	$\mathcal{O}(\|\text{out}\|(\log \|\text{out}\| + \ell))$	$\mathcal{O}(n^\ell \cdot \ell \log n)$	$\mathcal{O}(\ell n + \|\text{out}\|\ell)$

Figure 6: Complexity of ranked-enumeration algorithms for equi-joins. Best performers in each column are colored in green.

edges. Based on our equi-join construction (Fig. 3), it is easy to see that the total number of edges is $|E| = \mathcal{O}(\ell n)$. For simplicity we make the following assumptions: (1) the maximum arity of a relation is bounded by a constant, thus $|Q| = \ell$, and (2) the operations $\oplus$ and $\otimes$ of the selective dioid over which the ranking function is defined take $\gamma = \mathcal{O}(1)$ time to execute. It is straightforward to extend our analysis to scenarios where those assumptions do not hold. Note that (2) holds for many practical problems, e.g., tropical semiring $(\mathbb{R}^\infty, \min, +, \infty, 0)$, but not for lexicographic ordering where weights are ℓ-dimensional vectors and hence $\gamma = \mathcal{O}(\ell)$. With Batch, we refer to an algorithm that sorts the full output produced by the Yannakakis algorithm [92].

4.3.1 Time to First

All any-k algorithms first execute DP to find the top result and create all choice sets in time $\mathcal{O}(\ell n)$. Eager requires $\mathcal{O}(\ell n \log n)$ for sorting of choice sets. Heap construction for Lazy and Take2 takes time linear in input size.

4.3.2 Delay

Each algorithm requires $\mathcal{O}(\ell)$ to assemble an output tuple. In addition, the following costs are incurred:

anyK-rec. In Recursive each `next` call on s_0 triggers $\mathcal{O}(\ell)$ `next` calls in later stages—at most one per stage. The call deletes the top choice at the state and replaces it with the next-heavier path through the same child node in the next stage (see Fig. 4). With a priority queue, these operations together take time $\mathcal{O}(\log n)$ per state accessed, for a total delay of $\mathcal{O}(\ell \log n)$ between consecutive results. In total, it takes $\mathcal{O}(\ell n + k\ell \log n)$ to produce the top k results. The resulting TTL bound of $\mathcal{O}(\ell n + |\text{out}| \cdot \ell \log n)$ can be loose because it does not take into account that in later iterations many `next` calls will stop early because the corresponding suffixes Π_i had already been computed by an earlier call:

Theorem 12. *There exist DP problems where* Recursive *has strictly lower* TTL *complexity than* Batch.

An example are problems with near-worst-case output size $\Theta(n^\ell)$ such as the Cartesian product [87]. The lower TTL of Recursive is at first surprising, given that Batch is optimized for bulk-computing and bulk-sorting the entire output. Intuitively, Recursive wins because it exploits the multi-stage structure of the graph—which enables the re-use of shared path suffixes—while Batch uses a general-purpose comparison-based sort algorithm. We leave as future work a more precise characterization of graph properties that ensure better TTL for Recursive over Batch.

anyK-part. For all anyK-part algorithms, popMin and bulk-insertion of all new candidates during result expansion take $\mathcal{O}(\log |\texttt{Cand}|)$. For efficient candidate generation (Line 15 in Algorithm 1) the new candidates do not copy the solution prefix, but simply create a pointer to it. Therefore, a new candidate is created in $\mathcal{O}(1)$.

Eager finds each successor in constant time. Since $|\texttt{Cand}| \leq k\ell$, its total delay is $\mathcal{O}(\log(k\ell) + \ell) = \mathcal{O}(\log k + \ell)$. For Lazy, in the first iteration of the main for-loop (Algorithm 1, Line 11), finding the successor (Line 15) requires at most one pop on a heap storing $\mathcal{O}(n)$ choices. All later iterations find the successor in constant time. Hence total delay is $\mathcal{O}(\log k + \ell + \log n)$. The All algorithm might insert up to ℓn new candidates to `Cand` for each result produced. Hence access to `Cand` after producing k results takes a total of $\mathcal{O}(\log(k\ell n))$. All together, delay is $\mathcal{O}(\log k + \log \ell + \log n + \ell n) = \mathcal{O}(\log k + \ell n)$. Finally, Take2 finds up to two successor candidates of a choice in constant time. Delay therefore is $\mathcal{O}(\log k + \ell)$. It is easy to see that all these algorithms have worst-case TTL of $\mathcal{O}(n^\ell \cdot \ell \log n)$, the same as Batch (refer to [90] for All).

4.3.3 Memory

All algorithms need $\mathcal{O}(\ell n)$ memory for storing the input. The memory consumption of anyK-part approaches depends on the size of `Cand`. All grows `Cand` by $\mathcal{O}(\ell n)$ elements in each iteration, but creates at most $|\text{out}|$ candidates in total. The others create only $\mathcal{O}(\ell)$ new candidates per iteration, thus $\text{MEM}(k) = \mathcal{O}(\ell n + k\ell)$. For Recursive, size of a choice set $\texttt{Choices}_k(s)$ is bounded by the out-degree of s, hence cannot exceed n. However, we need to store the suffixes $\Pi_i(s)$ produced by the algorithm, whose number is $\mathcal{O}(\ell)$ per iteration, thus $\text{MEM}(k) = \ell n + k\ell$. Batch first materializes the output and then sorts it in-place, therefore has $\text{MEM}(k) = \mathcal{O}(\ell n + |\text{out}|\ell)$, regardless of k.

4.3.4 Summary

Figure 6 summarizes the analysis for TT(k), for TTL where the output is sufficiently big (so that result-enumeration time dominates pre-processing time), for TTL on worst-case outputs where we can see the advantage of Recursive, and for memory MEM(k). Setting $k = 1$ in the TT(k) column, we observe that all any-k algorithms except Eager have optimal TTF $= \mathcal{O}(\ell n)$. In contrast, Batch has to sort the full output in $\mathcal{O}(|\text{out}| \log |\text{out}|)$. Eager and Take2 have the lowest delay $\mathcal{O}(\log k + \ell)$. Only our new algorithm Take2 achieves optimal TT(k) (Section 2.3).

While Recursive has higher delay than Take2, Lazy, and Eager, it has the lowest TTL for a worst-case-size output. This seemingly paradoxical result stems from the fact that as Recursive outputs results, it builds up state (ranking of suffixes) that speeds up computation for later results. Hence even though its delay complexity is tight for small k, our amortized accounting showed that it ultimately must achieve lower delay for large k.

All any-k algorithms but All require minimal space, depending only on input size and the number of iterations k times query size ℓ. All has higher memory demand because it overloads the candidate set early, while Batch materializes the complete output.

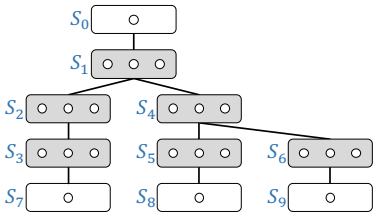


Figure 7: *Tree-Based DP* (T-DP) problem structure. Rounded rectangles are *stages*, small circles are *states*.

5. EXTENSION TO GENERAL CQS

We extend our ranked enumeration framework from serial to Tree-Based DP (T-DP), and then to a Union of T-DPs (UT-DP). This enables optimal ranked enumeration of *arbitrary conjunctive queries*.

5.1 Tree-Based DP (T-DP)

We first consider problems where the stages are organized in a *rooted tree* with $S_0 = \{s_0\}$ as the root stage. In these problems, there is a distinct set of decisions for each parent-child pair. Figure 7 depicts an example with 10 stages. We assume that all leaf stages contain only one (terminal) state[7], thus every root-to-leaf path represents an instance of serial DP as discussed in Sections 3 and 4. We now extend our approach to Tree-based DP problems (T-DP) and adapt all any-k algorithms accordingly. Due to space constraints, only the main ideas are discussed; for details see [87].

A T-DP solution is a tree with one state per stage. For the bottom-up phase, we serialize the stages by assigning a *tree order* that places every parent before its children, e.g., by a topological sorting of the tree. The optimal solution is then computed bottom-up by following the serial order of the stages in reverse. Each subproblem corresponds to finding an optimal subtree and it is solved by computing the optimal decision in each branch *independently*.

To enumerate lower-ranked results, we need to extend the path-based any-k algorithms. This is straightforward for ANYK-PART algorithms by simply following the serialized order of the stages. Intuitively, the i^{th} stage in this tree order is treated like the i^{th} stage in the path problem, except that the sets of choices are determined by the actual parent-child edges in the tree. For illustration, assume a tree order as indicated by the stage indices in Figure 7. Given a prefix $\langle s_1 s_2 s_3 \rangle$, the choices for $s_4 \in S_4$ are not determined by s_3 (as they would be for a path with stages S_1, S_2,...), but by $s_1 \in S_1$, because S_1 is the parent of S_4 in the tree. This means that we can run Algorithm 1 unchanged as long as we define the successor function `Succ` based on the parent-child relationships in the tree. Hence the complexity analysis in Section 4.3 still applies as summarized in Figure 6.

Unfortunately, for ANYK-REC the situation appears more challenging, because each state processes a `next` call by recursively calling `next` on its children. The challenge is to combine the lower-ranked solutions from the children and to rank these combinations efficiently. For example, consider a state $s_1 \in S_1$ with children S_2 and S_4. A solution rooted at s_1 consists of two parts: one solution rooted at the first child S_2 and the other at S_4. Suppose this solution contains the 2^{nd}-best path from S_2 and the 3^{rd}-best path from S_4—$[\Pi_2, \Pi_3]$ for short. Then the next-best solution from s_1 could be either $[\Pi_3, \Pi_3]$ or $[\Pi_2, \Pi_4]$. Since any combination of child solutions $[\Pi_{j_1}, \Pi_{j_2}]$ is valid for the parent, the problem is essentially to rank the Cartesian product space of subtree solutions. This produces duplicates when directly applying the recursive algorithm [33], or requires a different approach such as ANYK-PART for this Cartesian product problem to avoid duplicates. We adopt the latter approach. As a result, ANYK-REC behaves similar to the (path) DP case for nodes with a single child, but similar to ANYK-PART when encountering branches. In the extreme case of star queries (where a root stage is directly connected to all leaves), RECURSIVE degenerates to an ANYK-PART variant.

[7] Artificial stages can be introduced to meet this assumption.

5.2 DP over a Union of Trees (UT-DP)

We define a *union of T-DP problems* as a set of T-DP problems where a solution to any of the T-DP problems is a valid solution to the UT-DP problem. Thus, we are given a set of u functions $F = \{f^{(i)}\}$, each defined over a solution space $\Pi^{(i)}, i \in \mathbb{N}^u$. The UT-DP problem is then to *find the minimum solution across all T-DP instances*.

Changes to ranked enumeration. The necessary changes to any of our any-k algorithms are now straightforward: We add one more top-level data structure `Union` that maintains the last returned solution of each separate T-DP algorithm in a single priority queue. Whenever a solution is popped from `Union`, it gets replaced by the next best solution of the corresponding T-DP problem.

5.3 Cyclic Queries

Recent work on cyclic join queries indicates that a promising approach is to reduce the problem to the acyclic case via a *decomposition algorithm* [43]. Extending the notion of tree decompositions for graphs [79], hypertree decompositions [46] organize the relations into "bags" and arrange the bags into a tree [80]. Each decomposition is associated with a width parameter that captures the degree of acyclicity in the query and affects the complexity of subsequent evaluation: smaller width implies lower time complexity. *Our approach is orthogonal to the decomposition algorithm used and it adds ranked enumeration capability virtually "for free."*

The state-of-the-art decomposition algorithms rely on the submodular width $\mathsf{subw}(Q)$ of a query Q. Marx [69] describes an algorithm that runs in $\mathcal{O}(f(|Q|)n^{(2+\delta)\mathsf{subw}(Q)})$ for $\delta > 0$ and a function f that depends only on query size. PANDA [5] runs in $\mathcal{O}(f_1(|Q|)n^{\mathsf{subw}(Q)}(\log n)^{f_2(|Q|)})$ for query-dependent functions f_1 and f_2. Since this is an active research area, we expect these algorithms to be improved and we believe our framework is general enough to accommodate future decomposition algorithms. Sufficient conditions for applicability of our approach and for achieving optimal delay are, respectively, (1) the full output of Q is the union of the output produced by the trees in the decomposition and (2) the number of trees depends only on query size $|Q|$. Both are satisfied by current decompositions and it is hard to imagine how this would change in the future.

We can execute any decomposition algorithm almost as a blackbox to create a union of acyclic queries to which we then apply our UT-DP framework. *However, there are subtle challenges:* For correctness, we have to (1) properly compute the weights of tuples in the bags (i.e., tree nodes) and (2) deal with possible output duplicates when a decomposition creates multiple trees. For (1), we slightly modify the decomposition algorithm to track the lineage for bags at

the schema level: We only need to know from which input relation a tuple originates and if that relation's weight values had already been accounted for by another bag that is a descendent in the tree structure.

For (2), note that if all output tuples have distinct weights, then an output tuple's duplicates will be produced by our any-k algorithm one right after the other, making it trivial to eliminate them on-the-fly. Since the number of trees depends only query size $|Q|$, total delay induced by duplicate filtering is $\mathcal{O}(1)$ (data complexity). When different output tuples can have the same weight, we break ties using lexicographic ordering on their witnesses [87].

Simple cycles. For ℓ-cycle queries $Q_{C\ell}$ we use the standard decomposition [4, 80], which was pioneered by Alon et al. [8] in the context of graph-pattern queries. It does not produce output duplicates and achieves $\mathcal{O}(n^{2-1/\lceil \ell/2 \rceil})$ for TTF. On the other hand, for a worst-case optimal join algorithm such as NPRR [73] or Generic-Join [74], TTF is $\mathcal{O}(n^{\ell/2})$. We show in [87] that those algorithms can indeed not be modified to overcome this problem.

5.4 Putting everything together

Our main result follows from the above analysis when using Take2 for the acyclic CQ base case:

Theorem 13. *Given a decomposition algorithm $\mathcal{A}$ that takes time $T(\mathcal{A})$ and space $S(\mathcal{A})$, ranked enumeration of the results of a full conjunctive query can be performed with* $\text{TT}(k) = \mathcal{O}(T(\mathcal{A}) + k \log k)$ *and* $\text{MEM}(k) = \mathcal{O}(S(\mathcal{A}) + k)$ *in data complexity.*

6. EXPERIMENTS

Since asymptotic complexity only tells part of the story, we compare all algorithms in terms of actual running time.

Algorithms. All algorithms are implemented in the same Java environment and use the same data structures for the same functionality. We compare: (1) Recursive representing the anyK-rec approach, (2) Take2, (3) Lazy [26], (4) Eager, (5) All [90] representing the anyK-part approach, and (6) Batch, which computes the full result using the Yannakakis algorithm [92] for acyclic queries and NPRR [73] for cyclic queries, both followed by sorting.

Queries. We explore *paths*, *stars*, and *simple cycles* over binary relations. The SQL queries are listed in the full version [87]. A path is the simplest acyclic query, making it ideal for studying core differences between the algorithms. The star represents a typical join in a data warehouse and by treating it as a single root (the center) with many children, we can study the impact of node degree. The simple cycles apply our decomposition method as described in Section 5.3.

Synthetic data. Our goal for experiments with synthetic data is to create input with regular structure that allows us to identify and explain the core differences between the algorithms. For path and star queries, we create tuples with values uniformly sampled from the domain $\mathbb{N}_1^{n/10}$. That way, tuples join with 10 others in the next relation, on average. For cycles, we follow a construction by [73] that creates a worst-case output: every relation consists of $n/2$ tuples of the form $(0, i)$ and $n/2$ of the form $(i, 0)$ where i takes all the values in $\mathbb{N}_1^{n/2}$. Tuple weights are real numbers uniformly drawn from $[0, 10000]$.

Real Data. We use two real networks. In Bitcoin OTC [62, 63], edges have weights representing the degree of trust between users. Twitter [94] edges model followership among users. Edge weight is set to the sum of the PageRanks [23] of both endpoints. To control input size, we only retain edges between users whose IDs are below a given threshold. Since the cycle queries are more expensive, we run them on a smaller sample (TwitterS) than the path queries (TwitterL). Figure 8 summarizes relevant statistics. Note that the size of our relations n is equal to the number of edges.

Dataset	Nodes	Edges	Max/Avg Degree	Weights
Bitcoin [62, 63]	5,881	35,592	1,298 / 12.1	Provided
TwitterS [94]	8,000	87,687	6,093 / 21.9	PageRank
TwitterL [94]	80,000	2,250,298	22,072 / 56.3	PageRank

Figure 8: Datasets used for experiments with real data.

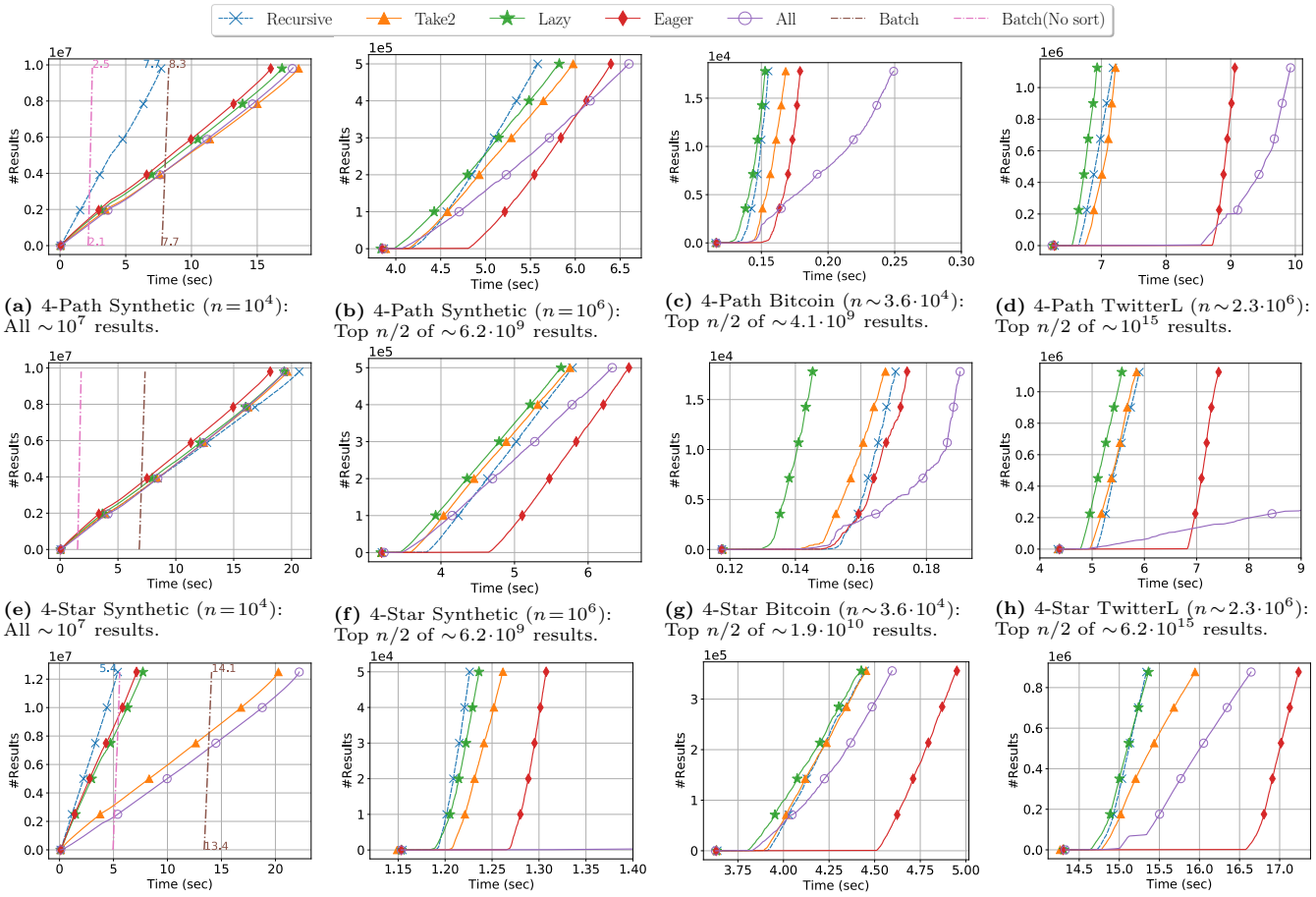
Implementation details. All algorithms are implemented in Java and run on an Intel Xeon E5-2643 CPU with 3.3Ghz and 128 GB RAM with Ubuntu Linux. Each data point is the median of 200 runs. We initialize all data structures lazily when they are accessed for the first time. For example, in Eager, we do not sort the `Choices` set of a node until it is visited. This can significantly reduce $\text{TT}(k)$ for small k, and we apply this optimization to all algorithms. Notice that our complexity analysis in Section 4.3 assumes constant-time inserts for priority queues, which is important for algorithms that push more elements than they pop per iteration. This bound is achieved by data structures that are well-known to perform poorly in practice [28, 64]. To address this issue in the experiments, we use "bulk inserts" which heapify the inserted elements [26] or standard binary heaps when query size is small.

6.1 Experimental results

Figure 9 reports the number of output tuples returned in ranking order over time for queries of size 4. On the larger input, Batch runs out of memory or we terminate it after 2 hours. This clearly demonstrates the need for our approach. We then set a limit on the number of returned results and compare our various any-k algorithms for relatively small k. We also use a fairly small synthetic input to be able to compare TTL performances against Batch.

Results. *For* TTL, Recursive *is fastest on paths and cycles, finishing even before* Batch. *This advantage disappears in star queries due to the small depth of the tree. For small k,* Lazy *is consistently the top-performer and is even faster than the asymptotically best* Take2. Batch *is impractical for real-world data since it attempts to compute the full result, which is extremely large.*

For path and cycle queries on the small synthetic data, Recursive is faster than Batch (Figs. 9a and 9i) due to the large number of suffixes shared between different output tuples. It returns the *full sorted result faster (7.7 sec and 5.4 sec) than* Batch *(8.3 sec and 14.1 sec).* Especially for cycles, our decomposition method really pays off compared to Batch [73], as Recursive terminates around the same time Batch starts to sort. For star queries, Recursive behaves like an anyK-part approach because of the shallowness of the tree (Fig. 9e). When many results are returned, the strict anyK-part variants (Eager, Lazy) have an advantage over the relaxed ones (Take2, All) as they produce fewer candidates per iteration and maintain a smaller priority queue. Eager is slightly better than Lazy because sorting is faster than incrementally converting a heap to a



(a) 4-Path Synthetic ($n=10^4$): All $\sim 10^7$ results.
(b) 4-Path Synthetic ($n=10^6$): Top $n/2$ of $\sim 6.2 \cdot 10^9$ results.
(c) 4-Path Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $n/2$ of $\sim 4.1 \cdot 10^9$ results.
(d) 4-Path TwitterL ($n \sim 2.3 \cdot 10^6$): Top $n/2$ of $\sim 10^{15}$ results.
(e) 4-Star Synthetic ($n=10^4$): All $\sim 10^7$ results.
(f) 4-Star Synthetic ($n=10^6$): Top $n/2$ of $\sim 6.2 \cdot 10^9$ results.
(g) 4-Star Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $n/2$ of $\sim 1.9 \cdot 10^{10}$ results.
(h) 4-Star TwitterL ($n \sim 2.3 \cdot 10^6$): Top $n/2$ of $\sim 6.2 \cdot 10^{15}$ results.
(i) 4-Cycle Synthetic ($n=5 \cdot 10^3$): All $\sim 1.2 \cdot 10^7$ results.
(j) 4-Cycle Synthetic ($n=10^5$): Top $n/2$ of $\sim 5 \cdot 10^9$ results.
(k) 4-Cycle Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $10n$ of $\sim 7.3 \cdot 10^6$ results.
(l) 4-Cycle TwitterS ($n \sim 8.8 \cdot 10^4$): Top $10n$ of $\sim 3.1 \cdot 10^8$ results.

Figure 9: Experiments on queries of size 4 (Section 6.1).

sorted list. This situation is reversed for small k where *initialization time* becomes a crucial factor: Then EAGER and RECURSIVE lose their edge, while LAZY shines (Figs. 9c, 9g, 9h, 9k and 9l). RECURSIVE starts off slower, but often overtakes the others for sufficiently large k (Figs. 9b and 9j). EAGER is also slow in the beginning because it has to sort each time it accesses a new choice set. TAKE2 showed mixed results, performing near the top (Fig. 9f) or near the bottom (Fig. 9l). ALL performs poorly overall due to the large number of successors it inserts into its priority queue.

6.2 More results for different query sizes

We performed the same experiments for different query sizes: 3-Path, 6-Path, 3-Star, 6-Star, and 6-Cycle [87]. We summarize our findings below:

Results. RECURSIVE*'s* TTL *advantage over* BATCH *is more evident in longer queries since there are more opportunities of reusing computation.* LAZY *again dominates for the first results (small k) for all query sizes.*

6.3 Comparison against PostgreSQL

To validate our BATCH implementation, we compare it against PostgreSQL 9.5.20. Following standard methodology [13], we remove the system overhead as much as possible and make sure that the input relations are cached in memory. On our synthetic datasets, our implementation is 12% to 54% faster [87]. Although the two implementations are not directly comparable since they are written in different languages and PostgreSQL is a full-fledged database system, this result shows that our BATCH implementation is competitive with existing batch algorithms.

7. RELATED WORK

Top-k. Top-k queries received significant attention in the database community [6, 7, 14, 24, 27, 55, 85, 86]. Much of that work relies on the value of k given in advance in order to *prune the search space*. Besides, the cost model introduced by the seminal Threshold Algorithm (TA) [36] only accounts for the *cost of fetching* input tuples from external sources. Later work such as J* [71], Rank-Join [54], LARA-J* [66], and a-FRPA [37] generalizes TA to more complex join patterns, yet also focuses on minimizing the number of accessed input tuples. While some try to find a balance between the cost of accessing tuples and the cost of detecting termination, previous work on top-k queries is *sub-optimal when accounting for all steps of the computation*, including intermediate result size (see the full version [87]).

Optimality in Join Processing. Acyclic Boolean queries can be evaluated optimally in $\mathcal{O}(n)$ data complexity by the Yannakakis algorithm [92]. The AGM bound [9], a tight bound on the worst-case output size for full conjunctive queries, motivated worst-case optimal algorithms [72,

73, 74, 89] and was extended to more general scenarios, such as the presence of functional dependencies [45] or degree constraints [3, 5]. The upper bound for cyclic Boolean CQs was improved over the years with decomposition methods that achieve ever smaller width-measures, such as treewidth [79], (generalized) hypertree width (ghw) [46, 47, 48, 49, 51], fractional hypertree width (fhw) [52], and submodular width (subw) [69]. Current understanding suggests that achieving the improvements of subw over fhw requires decomposing a cyclic query into a union of acyclic queries. Our method can leverage this prior work on subw [5, 69] to match the subw bound of Boolean CQs for TTF. We also show that it is possible to achieve better complexity for TTL than sorting the output of any of these batch computation algorithms.

Unranked enumeration of query results. Enumerating the answers to CQs *with projections* in no particular order can be achieved only for some classes of CQs with constant delay, and much effort has focused on identifying those classes [11, 18, 25, 83, 84]. If the ranking function is defined over the Boolean semiring, our technique achieves constant delay if we replace the priority queues with simple unsorted lists. However, we consider only *full* CQs, eschewing the difficulties introduced by projections and focusing instead on the challenges of ranking. A recent paper by Berkholz and Schweikard [19] also uses a union of tree decompositions based on subw. Our focus is on the issues arising from imposing a rank on the output tuples, which requires solutions for *pushing sorting into such enumeration algorithms*.

Factorization and Aggregation. Factorized databases [13, 75, 76, 82] exploit the distributivity of product over union to represent query results compactly and generalize the results on bounded fhw to the non-Boolean case [77]. Our encoding as a DP graph leverages the same principles and is at least as efficient space-wise. Finding the top-1 result is a case of aggregation that is supported by both factorized databases, as well as the FAQ framework [1, 2] that captures a wide range of aggregation problems over semirings. Factorized representations can also enumerate the query results with constant delay according to lexicographic orders of the variables [12], which is a special case of the ranking that we support (Section 2.2). For that to work, *the desired lexicographic order has to agree with the factorization order*; a different order requires a restructuring operation that could result in a quadratic blowup even for a simple binary join (see [87] for the full example). Related to this line of work are different representation schemes [58] and the exploration of the continuum between representation size and enumeration delay [32].

Ranked enumeration. Both [26] and [90] provide any-k algorithms for *graph queries* instead of the more general CQs; they describe the ideas behind Lazy and All respectively. [60] gives an any-k algorithm for acyclic queries with polynomial delay. Similar algorithms have appeared for the equivalent Constraint Satisfaction Problem (CSP) [44, 50]. These algorithms fit into our family anyK-part, yet do not exploit common structure between sub-problems hence have weaker asymptotic guarantees for delay than any of the any-k algorithms discussed here. After we introduced the general idea of ranked enumeration over *cyclic* CQs based on multiple tree decompositions [91], an unpublished paper [33] on arXiv proposed an algorithm for it. Without realizing it, the authors reinvented the REA algorithm [57], which corresponds to Recursive, for that specific context. We are the first to *guarantee optimal time-to-first result and optimal delay for both acyclic and cyclic* queries. For instance, we return the top-ranked result of a 4-cycle in $\mathcal{O}(n^{1.5})$, while [33] requires $\mathcal{O}(n^2)$. Furthermore, our work (1) addresses the more general problem of ranked enumeration for DP over a union of trees, (2) unifies several approaches that have appeared in the past, from graph-pattern search to k-shortest path, and shows that neither dominates all others, (3) provides a theoretical and experimental evaluation of trade-offs including algorithms that perform best for small k, and (4) is the first to prove that it is possible to achieve a time-to-last that asymptotically improves over batch processing by exploiting the stage-wise structure of the DP problem.

k-shortest paths. The literature is rich in algorithms for finding the k-shortest paths in general graphs [10, 17, 34, 35, 53, 56, 57, 59, 65, 68, 67, 93]. Many of the subtleties of the variants arise from issues caused by cyclic graphs whose structure is more general than the acyclic multi-stage graphs in our DP problems. Hoffman and Pavley [53] introduces the concept of "deviations" as a sufficient condition for finding the k^{th} shortest path. Building on that idea, Dreyfus [34] proposes an algorithm that can be seen as a modification to the procedure of Bellman and Kalaba [17]. The *Recursive Enumeration Algorithm* (REA) [57] uses the same set of equations as Dreyfus, but applies them in a top-down recursive manner. Our anyK-rec builds upon REA. To the best of our knowledge, prior work has ignored the fact that this algorithm reuses computation in a way that can asymptotically outperform sorting in some cases. In another line of research, Lawler [65] generalizes an earlier algorithm of Murty [70] and applies it to k-shortest paths. Aside from k-shortest paths, the Lawler procedure has been widely used for a variety of problems in the database community [40]. Along with the Hoffman-Pavley deviations, they are one of the main ingredients of our anyK-part approach. Eppstein's algorithm [35, 56] achieves the best known asymptotical complexity, albeit with a complicated construction *whose practical performance is unknown*. His "basic" version of the algorithm has the same complexity as Eager, while our Take2 algorithm matches the complexity of the "advanced" version for our problem setting where output tuples are materialized explicitly.

8. CONCLUSIONS AND FUTURE WORK

We proposed a framework for ranked enumeration over a class of dynamic programming problems that generalizes seemingly different problems that to date had been studied in isolation. Uncovering those relationships enabled us to propose the first algorithms with *optimal time complexity for ranked enumeration* of the results of both cyclic and acyclic full CQs. In particular, our technique returns the top result in a time that meets the currently best known bounds for Boolean queries, and even beats the batch algorithm on some inputs when all results are produced. It will be interesting to go beyond our worst-case analysis and study the *average-case* behavior [81] of our algorithms.

Acknowledgements. This work was supported in part by the National Institutes of Health (NIH) under award number R01 NS091421 and by the National Science Foundation (NSF) under award number CAREER IIS-1762268. The content is solely the responsibility of the authors and does not necessarily represent the official views of NIH or NSF.

References

[1] M. Abo Khamis, R. R. Curtin, B. Moseley, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. On functional aggregate queries with additive inequalities. In *PODS*, pages 414–431, 2019. DOI: 10.1145/3294052.3319694.

[2] M. Abo Khamis, H. Q. Ngo, and A. Rudra. Faq: questions asked frequently. In *PODS*, pages 13–28, 2016. DOI: 10.1145/2902251.2902280.

[3] M. Abo Khamis, H. Q. Ngo, and D. Suciu. Computing join queries with functional dependencies. In *PODS*, pages 327–342, 2016. DOI: 10.1145/2902251.2902289.

[4] M. Abo Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? *CoRR*, abs/1612.02503, 2016. URL: http://arxiv.org/abs/1612.02503.

[5] M. Abo Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *PODS*, pages 429–444, 2017. DOI: 10.1145/3034786.3056105.

[6] P. Agrawal and J. Widom. Confidence-aware join algorithms. In *ICDE*, pages 628–639, 2009. DOI: 10.1109/ICDE.2009.141.

[7] R. Akbarinia, E. Pacitti, and P. Valduriez. Best position algorithms for efficient top-k query processing. *Information Systems*, 36(6):973–989, 2011. DOI: 10.1016/j.is.2011.03.010.

[8] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997. DOI: 10.1007/BF02523189.

[9] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013. DOI: 10.1137/110859440.

[10] J. Azevedo, M. E. O. S. Costa, J. J. E. S. Madeira, and E. Q. V. Martins. An algorithm for the ranking of shortest paths. *European Journal of Operational Research*, 69(1):97–106, 1993. DOI: 10.1016/0377-2217(93)90095-5.

[11] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic (CSL)*, pages 208–222, 2007. DOI: 10.1007/978-3-540-74915-8_18.

[12] N. Bakibayev, T. Kočiský, D. Olteanu, and J. Závodný. Aggregation and ordering in factorised databases. *PVLDB*, 6(14):1990–2001, 2013. DOI: 10.14778/2556549.2556579.

[13] N. Bakibayev, D. Olteanu, and J. Závodný. Fdb: a query engine for factorised relational databases. *PVLDB*, 5(11):1232–1243, 2012. DOI: 10.14778/2350229.2350242.

[14] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. IO-top-k: index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006. URL: https://dl.acm.org/doi/10.5555/1182635.1164169.

[15] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958. DOI: 10.1090/qam/102435.

[16] R. Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515, Nov. 1954. URL: https://projecteuclid.org:443/euclid.bams/1183519147.

[17] R. Bellman and R. Kalaba. On kth best policies. *Journal of the Society for Industrial and Applied Mathematics*, 8(4):582–588, 1960. DOI: 10.1137/0108044.

[18] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *PODS*, pages 303–318, 2017. DOI: 10.1145/3034786.3034789.

[19] C. Berkholz and N. Schweikardt. Constant delay enumeration with FPT-preprocessing for conjunctive queries of bounded submodular width. In *44th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, volume 138 of *LIPIcs*, 58:1–58:15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019. DOI: 10.4230/LIPIcs.MFCS.2019.58.

[20] U. Bertele and F. Brioschi. *Nonserial dynamic programming*. Academic Press, 1972. URL: https://dl.acm.org/doi/book/10.5555/578817.

[21] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume I. Athena Scientific, 3rd edition, 2005. URL: http://www.athenasc.com/dpbook.html.

[22] E. Boros, B. Kimelfeld, R. Pichler, and N. Schweikardt. Enumeration in Data Management (Dagstuhl Seminar 19211). *Dagstuhl Reports*, 9(5):89–109, 2019. DOI: 10.4230/DagRep.9.5.89.

[23] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998. DOI: 10.1016/S0169-7552(98)00110-X.

[24] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: mapping strategies and performance evaluation. *TODS*, 27(2):153–187, 2002. DOI: 10.1145/568518.568519.

[25] N. Carmeli and M. Kröll. On the enumeration complexity of unions of conjunctive queries. In *PODS*, pages 134–148, 2019. DOI: 10.1145/3294052.3319700.

[26] L. Chang, X. Lin, W. Zhang, J. X. Yu, Y. Zhang, and L. Qin. Optimal enumeration: efficient top-k tree matching. *PVLDB*, 8(5):533–544, 2015. DOI: 10.14778/2735479.2735486.

[27] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB*, volume 99, pages 397–410, 1999. URL: https://dl.acm.org/doi/10.5555/645925.671359.

[28] B. V. Cherkassky, A. V. Goldberg, and T. Radzik. Shortest paths algorithms: theory and experimental evaluation. *Mathematical programming*, 73(2):129–174, 1996. DOI: 10.1007/BF02592101.

[29] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009. URL: https://dl.acm.org/doi/book/10.5555/1614191.

[30] S. Dasgupta, C. H. Papadimitriou, and U. V. Vazirani. *Algorithms*. McGraw-Hill Higher Education, 2008. URL: https://dl.acm.org/doi/book/10.5555/1177299.

[31] R. Dechter. Bucket elimination: A unifying framework for reasoning. *Artif. Intell.*, 113(1-2):41–85, 1999. DOI: 10.1016/S0004-3702(99)00059-4.

[32] S. Deep and P. Koutris. Compressed representations of conjunctive query results. In *PODS*, pages 307–322, 2018. DOI: 10.1145/3196959.3196979.

[33] S. Deep and P. Koutris. Ranked enumeration of conjunctive query results. *CoRR*, abs/1902.02698, 2019. URL: http://arxiv.org/abs/1902.02698.

[34] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *Operations research*, 17(3):395–412, 1969. DOI: 10.1287/opre.17.3.395.

[35] D. Eppstein. Finding the k shortest paths. *SIAM J. Comput.*, 28(2):652–673, 1998. DOI: 10.1137/S0097539795290477.

[36] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003. DOI: 10.1016/S0022-0000(03)00026-6.

[37] J. Finger and N. Polyzotis. Robust and efficient algorithms for rank join evaluation. In *SIGMOD*, pages 415–428, 2009. DOI: 10.1145/1559845.1559890.

[38] E. Friedgut and J. Kahn. On the number of copies of one hypergraph in another. *Israel Journal of Mathematics*, 105(1):251–256, 1998. DOI: 10.1007/BF02780332.

[39] J. S. Golan. *Semirings and their applications*. Kluwer Academic Publishers, Dordrecht, 1999. URL: https://www.springer.com/gp/book/9780792357865.

[40] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Optimizing and parallelizing ranked enumeration. *PVLDB*, 4(11), 2011. URL: http://www.vldb.org/pvldb/vol4/p1028-golenberg.pdf.

[41] M. Gondran and M. Minoux. *Graphs, Dioids and Semirings: New Models and Algorithms (Operations Research/Computer Science Interfaces Series)*. Springer, 2008. DOI: 10.1007/978-0-387-75450-5.

[42] N. Goodman, O. Shmueli, and Y. C. Tay. GYO reductions, canonical connections, tree and cyclic schemas and tree projections. In *PODS*, pages 267–278, 1983. DOI: 10.1145/588058.588089.

[43] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: questions and answers. In *PODS*, pages 57–74, 2016. DOI: 10.1145/2902251.2902309.

[44] G. Gottlob, G. Greco, and F. Scarcello. Tree projections and constraint optimization problems: fixed-parameter tractability and parallel algorithms. *Journal of Computer and System Sciences*, 94:11–40, 2018. DOI: 10.1016/j.jcss.2017.11.005. URL: https://doi.org/10.1016/j.jcss.2017.11.005.

[45] G. Gottlob, S. T. Lee, G. Valiant, and P. Valiant. Size and treewidth bounds for conjunctive queries. *J. ACM*, 59(3):1–35, 2012. DOI: 10.1145/2220357.2220363.

[46] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences*, 64(3):579 –627, 2002. DOI: https://doi.org/10.1006/jcss.2001.1809.

[47] G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *Journal of Computer and System Sciences*, 66(4):775–808, 2003. DOI: 10.1145/375551.375579.

[48] G. Gottlob, Z. Miklós, and T. Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM*, 56(6):30, 2009. DOI: 10.1145/1568318.1568320.

[49] G. Greco and F. Scarcello. Greedy strategies and larger islands of tractability for conjunctive queries and constraint satisfaction problems. *Inf. Comput.*, 252:201–220, 2017. DOI: 10.1016/j.ic.2016.11.004.

[50] G. Greco and F. Scarcello. Structural tractability of constraint optimization. In *International Conference on Principles and Practice of Constraint Programming (CP)*, pages 340–355, 2011. DOI: 10.1007/978-3-642-23786-7_27. URL: https://doi.org/10.1007/978-3-642-23786-7_27.

[51] G. Greco and F. Scarcello. The power of local consistency in conjunctive queries and constraint satisfaction problems. *SIAM Journal on Computing*, 46(3):1111–1145, 2017. DOI: 10.1137/16M1090272.

[52] M. Grohe and D. Marx. Constraint solving via fractional edge covers. *ACM TALG*, 11(1):4, 2014. DOI: 10.1145/2636918.

[53] W. Hoffman and R. Pavley. A method for the solution of the nth best path problem. *J. ACM*, 6(4):506–514, 1959. DOI: 10.1145/320998.321004.

[54] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB J.*, 13(3):207–221, 2004. DOI: 10.1007/s00778-004-0128-2.

[55] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys*, 40(4):11, 2008. DOI: 10.1145/1391729.1391730.

[56] V. M. Jiménez and A. Marzal. A lazy version of eppstein's K shortest paths algorithm. In *International Workshop on Experimental and Efficient Algorithms (WEA)*, pages 179–191. Springer, 2003. DOI: 10.1007/3-540-44867-5_14.

[57] V. M. Jiménez and A. Marzal. Computing the K shortest paths: a new algorithm and an experimental comparison. In *International Workshop on Algorithm Engineering (WAE)*, pages 15–29. Springer, 1999. DOI: 10.1007/3-540-48318-7_4.

[58] A. Kara and D. Olteanu. Covers of query results. In *ICDT*, 16:1–16:22, 2018. DOI: 10.4230/LIPIcs.ICDT.2018.16.

[59] N. Katoh, T. Ibaraki, and H. Mine. An efficient algorithm for K shortest simple paths. *Networks*, 12(4):411–427, 1982. DOI: 10.1002/net.3230120406.

[60] B. Kimelfeld and Y. Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In *International Workshop on Next Generation Information Technologies and Systems (NGITS)*, pages 141–152, 2006. DOI: 10.1007/11780991_13.

[61] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, 61(2):302 –332, 2000. DOI: 10.1006/jcss.2000.1713.

[62] S. Kumar, B. Hooi, D. Makhija, M. Kumar, C. Faloutsos, and V. Subrahmanian. Rev2: fraudulent user prediction in rating platforms. In *WSDM*, pages 333–341, 2018. DOI: 10.1145/3159652.3159729.

[63] S. Kumar, F. Spezzano, V. Subrahmanian, and C. Faloutsos. Edge weight prediction in weighted signed networks. In *ICDM*, pages 221–230, 2016. DOI: 10.1109/ICDM.2016.0033.

[64] D. H. Larkin, S. Sen, and R. E. Tarjan. *A back-to-basics empirical study of priority queues*. In *2014 Proceedings of the Sixteenth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 61–72. DOI: 10.1137/1.9781611973198.7.

[65] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management science*, 18(7):401–405, 1972. DOI: 10.1287/mnsc.18.7.401.

[66] N. Mamoulis, M. L. Yiu, K. H. Cheng, and D. W. Cheung. Efficient top-k aggregation of ranked inputs. *TODS*, 32(3):19, 2007. DOI: 10.1145/1272743.1272749.

[67] E. Q. V. Martins and M. M. B. Pascoal. A new implementation of Yen's ranking loopless paths algorithm. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 1(2):121–133, 2003. DOI: 10.1007/s10288-002-0010-2.

[68] E. Q. V. Martins, M. M. B. Pascoal, and J. L. E. Santos. A new improvement for a K shortest paths algorithm. *Investigação Operacional*, 21(1):47–60, 2001. URL: http://apdio.pt/documents/10180/15407/IOvol21n1.pdf.

[69] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *J. ACM*, 60(6):42:1–42:51, 2013. DOI: 10.1145/2535926.

[70] K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968. DOI: 10.1287/opre.16.3.682.

[71] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB*, pages 281–290, 2001. URL: http://www.vldb.org/conf/2001/P281.pdf.

[72] G. Navarro, J. L. Reutter, and J. Rojas-Ledesma. Optimal joins using compact data structures. In *ICDT*, 2020. URL: https://arxiv.org/abs/1908.01812.

[73] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16, 2018. DOI: https://doi.org/10.1145/3180143.

[74] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, Feb. 2014. DOI: 10.1145/2590989.2590991.

[75] D. Olteanu and M. Schleich. Factorized databases. *SIGMOD Record*, 45(2), 2016. DOI: 10.1145/3003665.3003667.

[76] D. Olteanu and J. Závodnỳ. Factorised representations of query results: size bounds and readability. In *ICDT*, pages 285–298, 2012. DOI: 10.1145/2274576.2274607.

[77] D. Olteanu and J. Závodnỳ. Size bounds for factorised representations of query results. *TODS*, 40(1):2, 2015. DOI: 10.1145/2656335.

[78] J.-E. Pin, J. M. Taylor, and M. Atiyah. *Tropical semirings*. In *Idempotency*. J. Gunawardena, editor. Publications of the Newton Institute. Cambridge University Press, 1998, pages 50–69. DOI: 10.1017/CBO9780511662508.004.

[79] N. Robertson and P. Seymour. Graph minors. II. algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309 –322, 1986. DOI: https://doi.org/10.1016/0196-6774(86)90023-4.

[80] F. Scarcello. From hypertree width to submodular width and data-dependent structural decompositions. In *26th Italian Symposium on Advanced Database Systems*, 2018. URL: http://ceur-ws.org/Vol-2161/paper24.pdf.

[81] A. Schickedanz, D. Ajwani, U. Meyer, and P. Gawrychowski. Average-case behavior of k-shortest path algorithms. In *7th International Conference on Complex Networks and Their Applications*, volume 812 of *Studies in Computational Intelligence*, pages 28–40. Springer, 2018. DOI: 10.1007/978-3-030-05411-3_3.

[82] M. Schleich, D. Olteanu, M. A. Khamis, H. Q. Ngo, and X. Nguyen. Learning models over relational data: A brief tutorial. In *International Conference on Scalable Uncertainty Management (SUM)*, volume 11940 of *LNCS*, pages 423–432. Springer, 2019. DOI: 10.1007/978-3-030-35514-2_32.

[83] L. Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Record*, 44(1):10–17, 2015. DOI: 10.1145/2783888.2783894.

[84] L. Segoufin and A. Vigny. Constant Delay Enumeration for FO Queries over Databases with Local Bounded Expansion. In *ICDT*, volume 68, 20:1–20:16, 2017. DOI: 10.4230/LIPIcs.ICDT.2017.20.

[85] M. Theobald, H. Bast, D. Majumdar, R. Schenkel, and G. Weikum. TopX: efficient and versatile top-k query processing for semistructured data. *VLDB J.*, 17(1):81–115, 2008. DOI: 10.1007/s00778-007-0072-z.

[86] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *ICDE*, pages 277–288. IEEE, 2003. DOI: 10.1109/ICDE.2003.1260799.

[87] N. Tziavelis, D. Ajwani, W. Gatterbauer, M. Riedewald, and X. Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *CoRR*, abs/1911.05582, 2019. URL: https://arxiv.org/abs/1911.05582.

[88] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *STOC*, pages 137–146, 1982. DOI: 10.1145/800070.802186.

[89] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014. DOI: 10.5441/002/icdt.2014.13.

[90] X. Yang, D. Ajwani, W. Gatterbauer, P. K. Nicholson, M. Riedewald, and A. Sala. Any-k: anytime top-k tree pattern retrieval in labeled graphs. In *WWW*, pages 489–498, 2018. DOI: 10.1145/3178876.3186115.

[91] X. Yang, M. Riedewald, R. Li, and W. Gatterbauer. Any-k algorithms for exploratory analysis with conjunctive queries. In *International Workshop on Exploratory Search in Databases and the Web (ExploreDB)*, pages 1–3, 2018. DOI: 10.1145/3214708.3214711.

[92] M. Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981. URL: https://dl.acm.org/doi/10.5555/1286831.1286840.

[93] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17(11):712–716, 1971. DOI: 10.1287/mnsc.17.11.712.

[94] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009. URL: http://socialcomputing.asu.edu. (accessed on 09/2019).