Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries

NIKOLAOS TZIAVELIS, Northeastern University, USA
DEEPAK AJWANI, University College Dublin, Ireland
WOLFGANG GATTERBAUER, Northeastern University, USA
MIREK RIEDEWALD, Northeastern University, USA
XIAOFENG YANG, Northeastern University, USA

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.

ACM Reference Format:
Nikolaos Tziavelis, Deepak Ajwani, Wolfgang Gatterbauer, Mirek Riedewald, and Xiaofeng Yang. 2020. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries. 1, 1 (September 2020), 50 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [82]. Part of the excitement stems from the fact that conjunctive query (CQ) evaluation is equivalent to other key problems such as constraint satisfaction [70] and hypergraph homomorphism [46]. Similar to [82], we consider *full conjunctive queries*, yet we are interested in *ranked enumeration*, recently identified as an important open problem [24]: 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 [64], 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 [43], were developed for a middleware-centric cost model that charges an algorithm only for accesses to external data sources, but does not take

Authors' addresses: Nikolaos Tziavelis, ntziavelis@ccs.neu.edu, Northeastern University, Boston, Massachusetts, USA; Deepak Ajwani, deepak.ajwani@ucd.ie, University College Dublin, Dublin, Ireland; Wolfgang Gatterbauer, w.gatterbauer@northeastern.edu, Northeastern University, Boston, Massachusetts, USA; Mirek Riedewald, m.riedewald@northeastern.edu, Northeastern University, Boston, Massachusetts, USA; Xiaofeng Yang, alice.xiaofeng.yang@gmail.com, Northeastern University, Boston, Massachusetts, USA.

XXXX-XXXX/2020/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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, R1.W + R2.W + R3.W + R4.W as Weight
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 Weight ASC
LIMIT    k
```

One can compute the full output with a worst-case optimal join algorithm such as NPRR [82] or Generic-Join [83] and then sort it. Since the fractional edge cover number ρ^* of Q_{C4} is 2, it takes $O(n^2)$ 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 $O(n^{1.5})$ [78]. Our approach returns the top-ranked 4-cycle in $O(n^{1.5})$ 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-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 [31] and subgraph isomorphism [101], graph-pattern ranking techniques can in principle be applied to conjunctive queries. An unpublished paper [38] 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. [31] be improved for large k and is it possible to extend it to give optimality guarantees for cyclic queries? For [69, 101], how can the worst-case delay be reduced? Is it possible to reduce the complexity of [38] 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 [78], 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 with lower delay given linear-time pre-processing than all previous work but Eppstein's algorithm [42], 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) [21]. 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* [74] approaches are asymptotically optimal for general inputs and dominate the *Recursive Enumeration Algorithm (REA)* [38, 66]. (*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 needs asymptotically the same pre-processing and has lower delay than Lazy [31], its overall *time-to-the-k'th result is the same* and we do not find it winning in our experiments.

This is the extended version of a paper appearing in VLDB'20 [97]. Further information is available on the project web page at https://northeastern-datalab.github.io/anyk/.

2 FORMAL SETUP

We use $\mathbb{N}_i^j$ to denote the set of natural numbers $\{i, \ldots, j\}$.

2.1 Conjunctive Queries (CQs)

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 conjunctive queries (CQs)* with equi-joins [82] and hence focus on them. A full CQ is a first-order formula $Q(\mathbf{x}) = (g_1 \wedge \cdots \wedge g_\ell)$, written $Q(\mathbf{x}) :- g_1(\mathbf{x}_1), \ldots, g_\ell(\mathbf{x}_\ell)$ 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. An *answer* or *query result* or *output tuple* is an assignment of the variables $\mathbf{x}$ to values from the domain of the database such that the formula is satisfied. 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 [50]. In particular, we say that a query is acyclic when its hypergraph is alpha-acyclic[26]. This property can be verified efficiently in $O(|Q|)$ by the well-known GYO reduction [84, 105] which constructs a join tree. A *Boolean* CQ just asks for the satisfiability of the formula. We use Q^B to denote the Boolean version of Q. A query with *self-joins* has at least one relation appearing in more than one atom and a *self-join-free* query has no self-joins. To avoid notational clutter and without loss of generality, we assume that there are no selection conditions on individual relations (like $R(x, 1)$ or

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

$R(x, x)$): Tables can be copied, and selection conditions can always be applied directly to the tables in a preprocessing step that takes $O(n)$.

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:*

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

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 *witness* of a result.

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 the order of query results based on algebraic structures called *selective dioids* [49], which are semirings with an ordering property.

A *monoid* is a 3-tuple $(W, \oplus, \bar{0})$ where W is a set and $\oplus : W \times W \to W$ is a closed binary operation such that:

(1) $(x \oplus y) \oplus z = x \oplus (y \oplus z)$ (associativity),
(2) $\bar{0} \in W$ satisfies $x \oplus \bar{0} = \bar{0} \oplus x = x, \forall x \in W$ (neutral element).

If additionally it holds that

(3) $x \oplus y = y \oplus x, \forall x, y \in W$ (commutativity),

then the monoid is called a commutative monoid.

A *semiring* is a 5-tuple $(W, \oplus, \otimes, \bar{0}, \bar{1})$, where

(1) $(W, \oplus, \bar{0})$ is a commutative monoid,
(2) $(W, \otimes, \bar{1})$ is a monoid,
(3) $\forall x, y, z \in W : (x \oplus y) \otimes z = (x \otimes z) \oplus (y \otimes z)$ (distributivity of $\otimes$ over $\oplus$),
(4) $\forall a \in W : a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$ ($\bar{0}$ is absorbing for $\otimes$).

Definition 3 (Selective dioid). *A* selective dioid *is a semiring for which $\oplus$ is* selective*, i.e., it always returns one of the operands: $\forall x, y \in W : (x \oplus y = x) \vee (x \oplus y = y)$.*

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

Definition 4 (Result 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 $\bigotimes_{i=1}^{\ell} w(r_i)$.*

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*. Notice the correspondence of $\otimes$ to $+$ and $\oplus$ to min. In general, we use the term *ranking function* to refer to a function that maps the query results to a domain equipped with a total order $\preceq$. In this paper, the ranking function is captured by a selective dioid: we use the $\otimes$ operator to aggregate the input weights into a result weight and then we use the $\oplus$ operator on the result weights to compare (or *rank*) them.

The central problem in this paper is the following:

Definition 5 (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 [101] and reflects the fact that the number of returned results need not be set apriori. Thus, any-k algorithms can be seen as a fusion of top-k and anytime algorithms [107] that gradually improve their result over time.

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, the selective dioid addition $\oplus$ returns the operand that comes first according to the lexicographic order e.g., for $\ell = 2$, $(a, b) \oplus (c, d) = (a, b)$ if $w'(a) < w'(c)$, or $w'(a) = w'(c)$ and $w'(b) < w'(d)$, and (c, d) otherwise. 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 and proofs are those that imply the algebraic structure of a selective dioid Definition 3.

Notice that addition over real numbers has an *inverse*, hence $(\mathbb{R}^\infty, +, 0)$ is a group, not just a monoid. This simplifies the algorithms to a certain extend. Our main result (Theorem 15) still holds even without the inverse with some minor subtleties that we explain in Section 6.2.

2.3 Complexity Measures

For complexity results we use the standard *RAM-model of computation* that charges $O(1)$ per data-element access. Reading or storing a vector of i elements therefore costs $O(i)$. In line with previous work [20, 51, 82], 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.

We measure success with respect to three measures: (i) the pre-processing time or *time-to-first* denoted by TTF, (ii) the delay between the $k-1$'th and k'th results for any value of k denoted by Delay(k) and (iii) the space requirement until the k'th result denoted by MEM(k). We will also look at TT(k) which is the overall time to get the k'th result and the special case of the *time-to-last* (TTL = TT(|out|)), where out denotes the output of the query. Notice that TTF = TT(1).

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 [99], 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.

2.4 Determining Optimality

Consider full CQ Q over input relations with at most n tuples. It takes $\Omega(n)$ just to look at each input tuple and $\Omega(k)$ to output k result tuples. 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 TTF = $O(n)$ and Delay(k) = $O(\log k)$. [2] For *acyclic* CQs, this optimality target is realistic, because the well-known Yannakakis algorithm [103] computes the full (unsorted) output in time $O(n + |\text{out}|)$.

For *cyclic* CQs, Ngo et al. [82] argue that the join result cannot be computed in $O(n + |\text{out}|)$ and propose the notion of *worst-case optimal* (WCO) join algorithms, whose computation time is $O(n + |\text{out}_{\text{WC}}|)$. Here, $|\text{out}_{\text{WC}}|$ 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 $\boldsymbol{Q}$ of acyclic CQs covering Q.[3] The input relations of the acyclic CQs in $\boldsymbol{Q}$ are derived from the original input and have cardinality $O(n^{\omega})$ 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 $\boldsymbol{Q}$. By applying the Yannakakis algorithm to the acyclic queries in $\boldsymbol{Q}$, cyclic query Q can be evaluated in time $O(T(\mathcal{A}) + |\text{out}|)$ and its Boolean version Q^B in $O(T(\mathcal{A}))$. The current frontier are decompositions based on the *submodular* width $\omega = \text{subw}(Q)$ [78], 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 TTF = $O(T(\mathcal{A}))$ and Delay(k) = $O(\log k)$. 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 [22, 34, 35]. 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 [22]. 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) [22, 35], 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

[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.

[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.

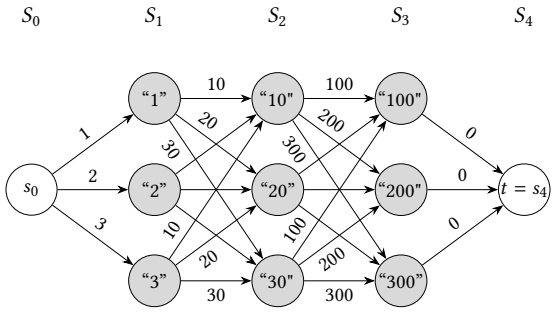


Fig. 1. DP instance for Example 6.

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 6 (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.*

A *solution* to the DP problem is a sequence of ℓ states $\Pi = \langle s_1 \dots s_\ell \rangle$ 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., $s_i \circ \langle s_{i+1} \dots s_\ell \rangle = \langle s_i \; s_{i+1} \dots s_\ell \rangle$.

Example 7 (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 $O(|S| + |E|)$ 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

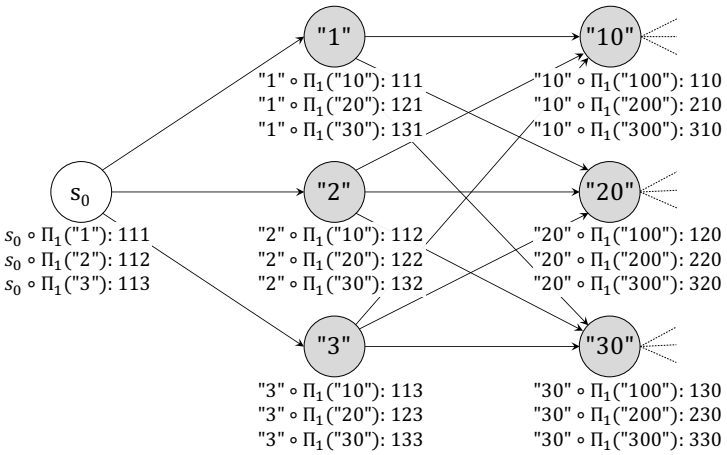


Fig. 2. Excerpt from Fig. 1, showing $\texttt{Choices}_1(s)$ for some states s. Term $s \circ \Pi_1(s') : w$ denotes a choice, which is a path from s, and its weight $w = w(s, s') + \pi_1(s')$.

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 $\pi_1(s_0)$ [22]. Notice that *DP needs only the pointer from the top choice at each state*, but 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 [36] on the *tropical semiring* [47, 88] and is reminiscent of the *semi-join reductions by Yannakakis* [103], 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 $O(\ell n)$ states and $O(\ell n^2)$ edges, therefore the DP algorithm has quadratic time complexity in the number of tuples. We reduce this to $O(\ell n)$ 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 $O(\ell n)$ 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 [85]. 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.

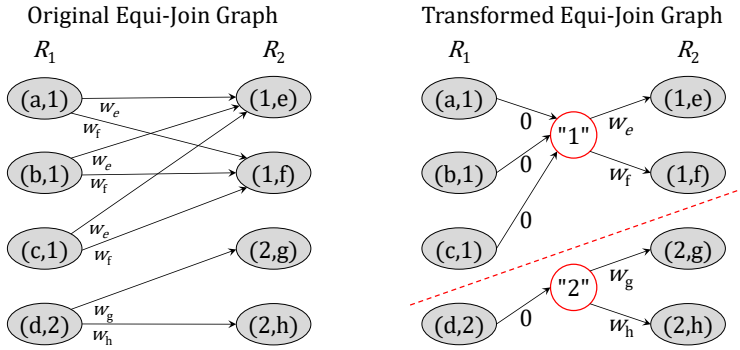


Fig. 3. *Equi-join* from $O(n^2)$ representation to $O(n)$.

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 [42], 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* (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 [74] and Murty [79], including recent work on subgraph isomorphism [31]. We call this family anyK-part; it includes Take2. The second finds the k-shortest paths in a graph via *recursive equations* [40, 66]. 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 [74] 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. [31] showed that the k^{th}-ranked solution $\langle s_1 \dots s_\ell \rangle$ is the output of `best` on some subspace

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

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:

[5] Implementations of "Eppstein's algorithm" exist, but they seem to implement a simpler variant with weaker asymptotic guarantees that was also introduced in [42].

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

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 8 (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 9 (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* $(\langle s_0 \rangle$, "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* `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* $(\langle s_0 \rangle$, "2", $0, 2 + 110)$*—it is the winner for the first subspace—and added to* `Cand`. *Then the solution is expanded (Line 19) to* $(\langle s_0$ "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* $(\langle s_0$ "1"$\rangle$, "20", $1, 20 + 100)$ *to* `Cand` *and updates the solution to* $(\langle s_0$ "1" "10"$\rangle$, "100", $11, 100)$. *The third and final iteration adds*

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
```

candidate (⟨s_0 "1" "10"⟩, "200", 11, 200) *and updates the solution to* (⟨s_0 "1" "10" "100"⟩, t, 111, 0)*, which is returned as the top-1 result.*

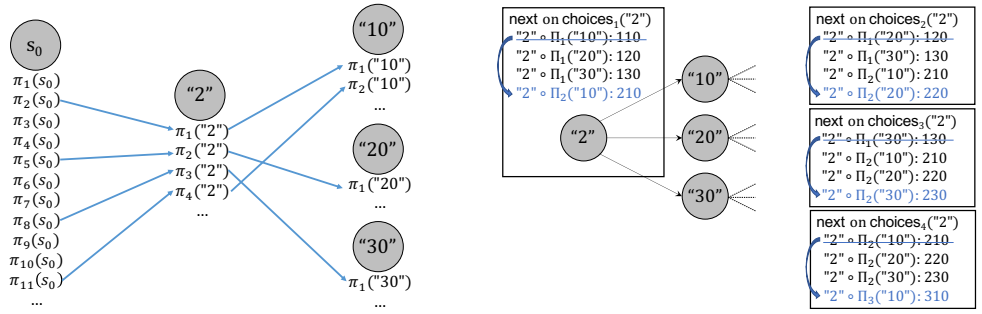
At this time, Cand *contains entries* (⟨s_0⟩, "2", 0, 112)*,* (⟨s_0 "1"⟩, "20", 1, 120)*, and* (⟨s_0 "1" "10"⟩, "200", 11, 200)*. Note that each is the shortest path in the corresponding subspace as defined by the Lawler procedure. Among the three,* (⟨s_0⟩, "2", 0, 112) *is popped next, because it has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

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 Succ(x, y) returns the 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. [31] 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



(a) Pointers between solutions from and to "2". **(b)** Recursive enumeration at state "2".

Fig. 4. Example 10: Recursive enumeration

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. [101] 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 $O(n)$ 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 [34], hence this approach asymptotically dominates the others.

4.2 Recursive Enumeration DP (ANYK-REC)

anyK-rec relies on a generalized principle of optimality [77]: 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 [66]. 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 2$^{\text{nd}}$-lightest path $\Pi_2(s_0)$ go? It has to be either the 2$^{\text{nd}}$-lightest path

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

Algorithm 2: RECURSIVE

Input: DP problem with stages $S_1, \ldots, S_\ell$
Output: solutions in increasing order of weight
Execute standard DP algorithm to produce for each state s: $\Pi_1(s)$, $\pi_1(s)$, and $\texttt{Choices}_1(s)$
//Initialization phase
for *stages* i *from* $\ell - 1$ *to* 0 **do**
  for *states* $s \in S_i$ **do**
    $\texttt{Choices}_1(s) = \{s \circ \Pi_1(s') \mid (s, s') \in \mathbb{E}\}$
    $\Pi_1(s) = \texttt{Choices}_1(s).\texttt{peek}()$
//Enumeration phase
$k = 1$
repeat
  $\Pi_k(s_0) = \texttt{Choices}_k(s_0).\texttt{popMin}()$
  Output $\Pi_k(s_0)$
  $\texttt{next}(\Pi_k(s_0))$
  $k = k + 1$
until *User stop process* $\vee$ $\texttt{Choices}_k(s_0)$ *is empty*

//Returns the next best solution starting from s
Function $\texttt{next}(\Pi_{j_s}(s))$**:**
  //Base case: Last stage
  if $s \in \mathbb{S}_\ell$ **then**
    return null
  //If $\Pi_{j_s+1}(s)$ has been computed by some previous call, it has been stored at state s
  if $\Pi_{j_s+1}(s)$ *has not been computed* **then**
    //$\Pi_{j_s}(s)$ is at the top of the priority queue, pop it so that we can construct $\texttt{Choices}_{j_s+1}(s)$
    $\texttt{Choices}_{j_s}(s).\texttt{popMin}()$
    //Assume $\Pi_{j_s}(s) = s \circ \Pi_{j_{s'}}(s')$.
    //Compute $\Pi_{j_{s'}+1}(s')$ recursively.
    $\Pi_{j_{s'}+1}(s') = \texttt{next}(\Pi_{j_s}(s'))$
    if $\Pi_{j_{s'}+1}(s') \neq$ *null* **then**
      $\texttt{Choices}_{j_s}(s).\texttt{insert}(s \circ \Pi_{j_{s'}+1}(s'))$
    $\texttt{Choices}_{j_s+1}(s) = \texttt{Choices}_{j_s}$
    //To get $\Pi_{j_s+1}(s)$, peek instead of popping. The pop will happen in the following call for $\texttt{next}(\Pi_{j_s+1}(s))$.
    $\Pi_{j_s+1}(s) = \texttt{Choices}_{j_s+1}(s).\texttt{peek}()$
  return $\Pi_{j_s+1}(s)$

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 10 (CONTINUED). *Consider node "2" in Fig. 1. Since it has adjacent states "10", "20", and "30" in the next stage, the lightest path* $\Pi_1(\text{"2"})$ *is selected from* $\texttt{Choices}_1(\text{"2"}) = \{\text{"2"} \circ \Pi_1(\text{"10"}), \text{"2"} \circ \Pi_1(\text{"20"}), \text{"2"} \circ \Pi_1(\text{"30"})\}$ *as shown in Fig. 2. The first* `next` *call on state "2" returns* $\text{"2"} \circ \Pi_1(\text{"10"})$*, updating the set of choices for* $\Pi_2(\text{"2"})$ *to* $\{\text{"2"} \circ \Pi_2(\text{"10"}), \text{"2"} \circ \Pi_1(\text{"20"}), \text{"2"} \circ \Pi_1(\text{"30"})\}$ *as shown in the left box in Fig. 4b. The subsequent* `next` *call on state "2" then returns* $\text{"2"} \circ \Pi_1(\text{"20"})$ *for* $\Pi_2(\text{"2"})$*,*

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

Fig. 5. Complexity of ranked-enumeration algorithms for equi-joins. Best performing any-k algorithms with linear TTF $O(\ell n)$ in each column are colored in green).

causing "2" $\circ$ Π_1("20") *in* $\texttt{Choices}_2$("2") *to be replaced by* "2" $\circ$ 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. 4a. 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" $\circ$ Π_1("10") as the lightest path in the choice set. Algorithm 2 contains the detailed pseudocode.

4.3 Any-k DP Algorithm Complexity

In contrast to the discussion in Section 2.4, 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 $O(\ell n)$ nodes, each with at most n outgoing edges. Based on our equi-join construction (Fig. 3), it is easy to see that the total number of edges is $|E| = O(\ell n)$. For simplicity we make the following assumptions: (1) the maximum arity of a relation is bounded by a constant [60] , thus $|Q| = \ell$, and (2) the operations $\oplus$ and $\otimes$ of the selective dioid over which the ranking function is defined take $\gamma = O(1)$ 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 $\gamma = O(\ell)$. With Batch, we refer to an algorithm that sorts the full output produced by the Yannakakis algorithm [103].

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 $O(\ell n)$. Eager requires $O(\ell n \log n)$ for sorting of choice sets. Heap construction for Lazy and Take2 takes time linear in input size.

4.3.2 Delay and TTL. Each algorithm requires $O(\ell)$ to assemble an output tuple. In addition, the following costs are incurred:

anyK-part. For all anyK-part algorithms, popMin and bulk-insertion of all new candidates during result expansion take $O(\log |\texttt{Cand}|)$. 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 $O(1)$.

Eager finds each successor in constant time. Since $|\texttt{Cand}| \leq k\ell$, its total delay is $O(\log(k\ell) + \ell) = O(\log k + \ell)$. 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 $O(n)$ choices. All later iterations find the successor in constant time. Hence total delay is $O(\log k + \ell + \log n)$. 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 $O(\log(k\ell n))$. All together, delay is $O(\log k + \log \ell + \log n + \ell n) = O(\log k + \ell n)$. Finally, Take2 finds up to two successor candidates of a choice in constant time. Delay therefore is $O(\log k + \ell)$. It is easy to see that all these algorithms have worst-case TTL of $O(n^\ell \cdot \ell \log n)$, the same as Batch (refer to [101] for All).

anyK-rec. In Recursive each `next` call on s_0 triggers $O(\ell)$ `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. 4b). With a priority queue, these operations together take time $O(\log n)$ per state accessed, for a total delay of $O(\ell \log n)$ between consecutive results. In total, it takes $O(\ell n + k\ell \log n)$ to produce the top k results. The resulting TTL bound of $O(\ell n + |\text{out}| \cdot \ell \log n)$ 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 11. *There exist DP problems where Recursive has strictly lower* TTL *complexity than Batch.*

Proof. Regardless of the implementation of Batch, before it terminates it has to (i) process the input in $\Omega(n\ell)$, (ii) enumerate all results in $\Omega(|\text{output}| \cdot \ell)$ and (iii) use a standard comparison-based sort algorithm to batch-rank the entire output in $(|\text{out}| \log |\text{out}|)$. In total, it needs $\Omega(n\ell + |\text{out}|(\log |\text{out}| + \ell))$.

For Recursive, when computing the full result, for each suffix $\pi_i(s)$ of any state s, it holds that the suffix is *exactly once* inserted into and removed from the priority queue managing `Choices` at s. Hence the total number of priority queue operations, each costing $O(\log n)$, equals the number of suffixes. Let $\Pi_*(i)$ denote the number of suffixes in stage i, i.e., the total number of paths starting from any node in $\mathbb{S}_i$. Then the total cost for all priority-queue operations is $O(\log n \sum_{i=1}^{\ell} \Pi_*(i))$. If $\sum_{i=1}^{\ell} \Pi_*(i) = O(\Pi_*(1))$, then this cost is $O(|\text{output}| \cdot \log n)$. (To see this, note that the set of paths starting at nodes in stage 1 is the set of all possible paths, i.e., the full output.) Together with pre-processing time and time to assemble each output tuple, total TTL complexity of Recursive then adds up to $O(\ell n + |\text{output}|(\log n + \ell))$. To complete the proof, we show instances where the condition $\sum_{i=1}^{\ell} \Pi_*(i) = O(\Pi_*(1))$ holds and in which the running time of Batch is strictly worse.

Consider the instances with worst-case output $\Theta(n^\ell)$ such as a Cartesian product. Recall that the size of the output is the same as the number of suffixes in the first stage, thus $\Pi_*(1) = \Theta(n^\ell)$. Now consider the ratio between $\Pi_*(i)$ and $\Pi_*(i+1)$ for some stage $i \in \mathbb{N}_1^{\ell-1}$. That ratio can't be more than n which occurs when i and $i+1$ are fully connected. It follows that in order to get that many suffixes in the first stage, *every stage* i has to increase the number of suffixes of stage $i+1$ by a factor of $\Theta(n)$. Therefore, $\Pi_*(1)$ asymptotically dominates the sum $\sum_{i=1}^{\ell} \Pi_*(i)$, similarly to a geometric series. Also note that the running time of Batch in these instances is $\Omega(n^\ell \cdot \ell \log n)$, which is higher than $O(n^\ell(\log n + \ell))$ of Recursive.

□

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.

4.3.3 TT(k). For all algorithms, TT$(k) = O(\text{TTF} + k \cdot \text{Delay}(k))$. Thus for Take2, TT$(k) = O(\ell n + k(\log k + \ell))$, while for Lazy, TT$(k) = O(\ell n + k(\log k + \ell + \log n))$. However, a more careful analysis for the anyK-part variants, gives us the following result:

Proposition 12. *Lazy achieves* TT$(k) = O(\ell n + k(\log k + \ell))$, *the same as Take2.*

Proof. We will show for all values of k that $\ell n + k(\log k + \ell) = \Omega(\ell n + k(\log k + \ell + \log n))$, thus the seemingly lower TT(k) complexity of Take2 is lower bounded by the seemingly higher TT(k)

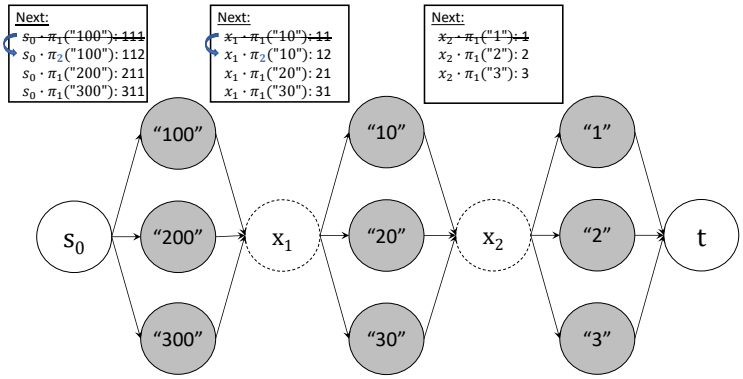


Fig. 6. A worst-case example for Recursive and $k = n$. Notice the sharing of data structures between tuples due to our special equi-join encoding (Fig. 3). Each returned query result entails a sequence of ℓ priority queue operations.

complexity Lazy. Since $\ell \geq 1$ and $\log n$ is dominated by $\log k$ for $k \geq n$, it suffices to show that $n + k \log k = \Omega(n + k \log n)$ for $k < n$.

For any $1 \leq k \leq n$, it holds that $n/k \geq \log(n/k)$ and $\log k \geq 0$ and therefore

$$\begin{aligned} &\frac{n}{k} \geq \log n - \log k \geq \log n - 2 \log k \\ \Rightarrow &n \geq k \log n - 2k \log k \\ \Rightarrow &(\frac{1}{0.5} - 1)n \geq k \log n - \frac{1}{0.5} k \log k \\ \Rightarrow &\frac{1}{0.5}(n + k \log k) \geq n + k \log n \end{aligned}$$

This means that there exists an $a > 0$ ($a = 0.5$ here) for which $n + k \log k \geq a(n + k \log n)$ for all values of n, which completes the proof. □

For Recursive, our analysis shows that when the number of the k returned results is not "too large", the best anyK-part approaches are asymptotically faster. For instance, when $k = O(n)$, Take2 achieves $O(n \log n + n \cdot \ell)$ compared to $O(n \cdot \ell \log n)$ of Recursive. One might be inclined to think that this gap is an just an artifact of our analysis and it can potentially be closed with arguments similar to the proof of Theorem 11. However this is not the case, as we now show that the aforementioned bound is *tight*, i.e., there exists an instance for which Recursive needs $\Theta(n \cdot \ell \log n)$ time to return $k = n$ results.

Proposition 13. *Recursive is strictly slower than the best implementation of anyK-part for* $\mathrm{TT}(n)$ *in the worst case.*

Proof. To achieve worst-case behavior for Recursive, we need to (i) create large priority queues and (ii) minimize the sharing of common suffixes between different results. Figure 6 depicts the simplest such example, corresponding to a Cartesian product between $\ell = 3$ relations. As before, tuple weight is equal to tuple value. Notice that each of the first $k = n$ results uses a different tuple from R_ℓ. It is straightforward to set the weights appropriately in order to achieve the same for arbitrary values of n, ℓ. To retrieve the k'th result, a `next` call at s_0 will trigger a chain of $\ell - 1$ other recursive `next` calls, each one computing Π_k for a different stage. Every `next` call (except maybe

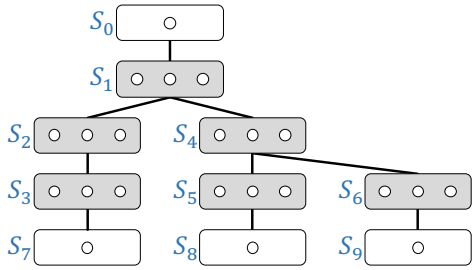


Fig. 7. *Tree-Based DP* (T-DP) problem structure. Rounded rectangles are *stages*, small circles are *states*.

the last one) involves a pop and a push from a priority queue of size $\Theta(n)$, hence $\Theta(n \cdot \ell \log n)$ in total. At the same time, the worst-case bound for Take2 is $O(n \log n + n \cdot \ell)$. □

4.3.4 Memory. All algorithms need $O(\ell n)$ memory for storing the input. The memory consumption of anyK-part approaches depends on the size of Cand. All grows Cand by $O(\ell n)$ elements in each iteration, but creates at most |out| candidates in total. The others create only $O(\ell)$ new candidates per iteration, thus $\text{MEM}(k) = O(\ell n + k\ell)$. 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 $O(\ell)$ per iteration, thus $\text{MEM}(k) = \ell n + k\ell$. Batch first materializes the output and then sorts it in-place, therefore has $\text{MEM}(k) = O(\ell n + |\text{out}|\ell)$, regardless of k.

4.3.5 Summary. Figure 5 summarizes the analysis for TTF, for Delay(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). All any-k algorithms except Eager have optimal TTF = $O(\ell n)$. In contrast, Batch has to sort the full output in $O(|\text{out}| \log |\text{out}|)$. Eager and Take2 have the lowest delay $O(\log k + \ell)$. Only our new algorithm Take2 achieves optimal Delay(k) after linear TTF (Section 2.4).

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.

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 E_{pc} for each parent-child pair $p - c$. 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

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

Sections 3 and 4. We now extend our approach to Tree-based DP problems (T-DP) and adapt all any-k algorithms accordingly.

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. To simplify the notation we force the t leaf nodes to be numbered last, i.e., $S_i = \{s_i\}$ for $i \in \{\ell+1, \dots, \ell+t\}$. We define $C(S_i)$ to be the set of indices of child stages and $pr(S_i)$ to be the index of the parent stage of S_i. In our example, $C(S_4) = \{5, 6\}$ and $pr(S_4) = 1$. By $\lceil S_v \rceil$ we denote the stages of the subtree rooted at S_v, while $\lVert S_v \rVert := \lceil S_v \rceil \setminus \{v\}$. In our example, $\lVert S_4 \rVert := \{5, 6, 8, 9\}$. Slightly overloading the notation, we also use $C(s_i) := C(S_i)$ for a state $s_i \in S_i$. Analogously for $pr(s_i)$, $\lceil s_i \rceil$, and $\lVert s_i \rVert$.

A T-DP *solution* $\Pi = \langle s_1 \dots s_\ell \rangle$ [8] is a tree with one state per stage and is admissible, i.e., $\forall c \in \mathbb{N}_1^{\ell+t}$, if $pr(c) = p$ then $(s_p, s_c) \in E_{pc}$. The *objective function* aggregates the weights of decisions across the entire tree structure:

$$w(\Pi) = \sum_{c=1}^{\ell+t} w(s_{pr(s_c)}, s_c) \tag{6}$$

T-DP Bottom-up. The optimal solution is then computed bottom-up by following the serial order of the stages in reverse. A bottom-up step for a state s solves a *subproblem* which corresponds to finding an optimal subtree $\Pi(s)$. If $C(s) = \{i_1, \dots, i_\lambda\}$, then that subtree consists of s and a list of other subtrees rooted at its children $s_{i_1}, \dots, s_{i_\lambda}$. To solve a subproblem, we *independently* choose the best decision for each child stage. The equations describing the bottom-up phase in T-DP are recursively defined for all states and stages by

$$\begin{aligned} \pi_1(s) &= 0, \text{ for the } t \text{ terminals with } C(s) = \emptyset \\ \pi_1(s) &= \sum_{c \in C(s)} \min_{(s,s_c) \in E_{pc}} \{w(s, s_c) + \pi_1(s_c)\}, \\ &\qquad\qquad \text{for } s \in S_p, p \in \mathbb{N}_0^\ell \end{aligned} \tag{7}$$

T-DP top-down. Similarly to serial DP, after the bottom-up phase we get reduced sets of states $\mathbb{S}_i \subseteq S_i, \mathbb{E}_{pc} \subseteq E_{pc}$ and the top-1 solution $\Pi_1(s_0)$ is found by a top-down phase that follows optimal decisions.

T-DP principle of optimality. Comparing the above formulation to serial DP, we now may have multiple terminals (i.e. leaves in the tree) that are initialized with 0 cost, but we still have only one single root node. Comparing the objective functions of T-DP Eq. (6) with DP Eq. (1), we changed the indexing to reflect the fact that each state has exactly one parent (but not the other way around). Consider Fig. 7 after removing the subtree rooted at S_4; then our problem degenerates to standard serial DP and we are back at Fig. 1. Contrasting the new *principle of optimality* formulation in Eq. (7) against Eq. (2), we now have that a minimum-cost solution contains other subtree solutions that achieve themselves minimum cost for their respective subproblems.

Theorem 14 (T-DP). *Equation (7) finds an optimal solution to the problem of minimizing Eq. (6).*

Proof. We will show by an induction on the tree stages in reverse serial order that for all states $s \in S$:

$$\min_{\Pi(s)} \Big\{ \sum_{i \in \lVert s \rVert} w(s_{pr(s_i)}, s_i) \Big\} = \pi_1(s) \tag{8}$$

The base case for the (terminal) leaf states follows by definition from Equation (7). For the inductive step, assume that the above holds for all descendant states $s_d \in S_d, d \in \lVert s \rVert$ of a state s. In particular

[8] Notice that as in DP, we do not include the unique root state and the t terminal states in the t leaf nodes (i.e. stages with unique states) in the solution.

for any "child state" s_c with $s \in S_{pr(s_c)}$:

$$\min_{\Pi(s_c)} \Big\{ \sum_{i \in [\![s_c]\!]} w(s_{pr(s_i)}, s_i) \Big\} = \sum_{g \in C(s_c)} \min_{(s_c, s_g) \in E_{cg}} \big\{ w(s_c, s_g) + \pi_1(s_g) \big\} \tag{9}$$

Then for any state $s \in S_p$:

$$\begin{aligned}
\pi_1(s) &= \sum_{c \in C(s)} \min_{(s, s_c) \in E_{pc}} \big\{ w(s, s_c) + \pi_1(s_c) \big\} \\
&= \sum_{c \in C(s)} \min_{(s, s_c) \in E_{pc}} \Big\{ w(s, s_c) + \min_{\Pi(s_c)} \Big\{ \sum_{g \in [\![s_c]\!]} w(s_{pr(g)}, s_g) \Big\} \Big\} \\
&= \min_{s_c} \Big\{ \min_{\Pi(s_c)} \Big\{ \sum_{c \in C(s)} \Big\{ w(s, s_c) + \sum_{g \in [\![s_c]\!]} w((s_{pr(g)}, s_g) \Big\} \Big\} \Big\} \\
&= \min_{\Pi(s)} \Big\{ \sum_{i \in [\![s]\!]} w(s_{pr(s_i)}, s_i) \Big\}
\end{aligned}$$

Since Equation (8) hold for any state s, it also holds for the starting state s_0, thus the theorem follows. □

To enumerate lower-ranked results for T-DP, we need to extend the path-based any-k algorithms.

Changes to ANYK-PART. All ANYK-PART algorithms are straightforward to extend to the tree case by 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. In general, at stage S_c, we have to find the successors $\mathsf{Succ}(s_p, s_c)$ where $p = pr(s_c)$. Similarly, to optimally expand a prefix $\langle s_1 \dots s_{c-1} \rangle$ by one stage, we append s_c such that $\Pi_1(s_c)$ is a subtree of $\Pi_1(s_p)$. Thus, we can run Algorithm 1 unchanged as long as we define the choice sets based on the parent-child relationships in the tree. Hence the complexity analysis in Section 4.3 still applies as summarized in Figure 5.

Changes to ANYK-REC. 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. First, we give a high-level overview: 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 [38], or requires a different approach such as ANYK-PART for this Cartesian product problem to avoid duplicates. We adopt the latter approach.

In more detail, let $\Pi_j(s, c)$ be the j-th best solution that starts from state s but is restricted only to a single branch $c \in C(s)$. Thus, $\Pi_j(s, c)$ consists of state s, then a state from stage S_c and from there, a list of pointers to other solutions (i.e., subtrees) that have their own rank. We write that as $\Pi_j(s, c) = s \circ [\Pi_{j_1}(s_c, i_1), \dots, \Pi_{j_\lambda}(s_c, i_\lambda)]$ for $s_c \in S_c$, $C(s_c) = \{i_1, \dots, i_\lambda\}$ and appropriate values $j_1, \dots, j_\lambda$. For example, in Figure 7, $\Pi_k(s_1, 4) = s_1 \circ [\Pi_{j_1}(s_4, 5), \Pi_{j_2}(s_4, 6)]$ for some values j, j_1, j_2. Notice that this definition matches the one in Section 4.2 for $|C(s_c)| = 1$ and since S_0 always has a single child S_1, we have that $\Pi_k(s_0) = \Pi_k(s_0, 1)$ for all values of k.

A state $s \in S_p$ maintains one data structure per branch $c \in C(s)$ for storing and comparing solutions $\Pi_j(s, c)$. At the beginning of the algorithm, we initialize it as $\texttt{Choices}_1(s, c) = \{s \circ [\Pi_1(s_c, i_1), \ldots, \Pi_1(s_c, i_\lambda)] \mid (s, s_c) \in \mathbb{E}_{pc}\}$. To process a `next` call, we pop the best solution from the data structure but unlike DP, we now have to replace it with more than one new candidates. To compute the `next` of $\Pi_j(s, c) = s \circ [\Pi_{j_1}(s_c, i_1), \ldots, \Pi_{j_\lambda}(s_c, i_\lambda)]$, we have to consider as new candidates all the following solutions:

$$s \circ [\Pi_{j_1+1}(s_c, i_1), \ldots, \Pi_{j_\lambda}(s_c, i_\lambda)]$$
$$\vdots$$
$$s \circ [\Pi_{j_1}(s_c, i_1), \ldots, \Pi_{j_\lambda+1}(s_c, i_\lambda)]$$

There are two problems associated with this. First, we could have up to ℓ children, hence up to ℓ new candidates and each one could have size (i.e., number of pointers) up to ℓ. Thus, in order to create them we would have to pay $O(\ell^2)$ delay. Second, it is easy to see that this process creates duplicates.

An elegant way to address both issues is to notice that from a specific parent node (e.g. s_4) all possible solutions belong to the Cartesian Product space formed by the sub-solutions of its children. Therefore, we need to apply a ranked enumeration algorithm that enumerates this space of solutions with low delay and without duplicates. We apply ANYK-PART which does not need to precompute all the elements from the beginning of the algorithm (we do not want to materialize all the children solutions from the start) given that they are accessed in a sorted order. This is equivalent to running EAGER over the Cartesian product of children solutions, except that we do not need to sort to find the successors – the sorted order is guaranteed by how RECURSIVE pulls lower ranked solutions in-order.

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.

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* [51]. Extending the notion of tree decompositions for graphs [89], hypertree decompositions [54] organize the relations into "bags" and arrange the bags into a tree [90]. 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."*

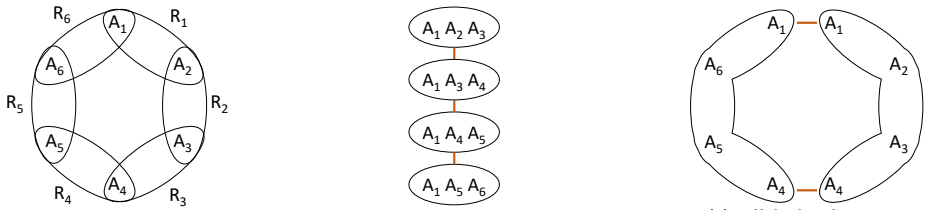
The state-of-the-art decomposition algorithms rely on the submodular width $\text{subw}(Q)$ of a query Q. Marx [78] describes an algorithm that runs in $O(f(|Q|)n^{(2+\delta)\text{subw}(Q)})$ for $\delta > 0$ and a function

f that depends only on query size. Panda [5] runs in $O(f_1(|Q|)n^{\text{subw}(Q)}(\log n)^{f_2(|Q|)})$ 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 $O(1)$ (data complexity). When different output tuples can have the same weight, we break ties using lexicographic ordering on their witnesses , as we describe in Section 6.3.

5.3.1 Simple Cycle Decomposition. For ℓ-cycle queries $Q_{C\ell}$ we use the standard decomposition [4, 90], which was pioneered by Alon et al. [8] in the context of graph-pattern queries. It does not produce output duplicates and achieves $O(n^{2-1/\lceil \ell/2 \rceil})$ for TTF. On the other hand, for a worst-case optimal join algorithm such as NPRR [82] or Generic-Join [83], TTF is $O(n^{\ell/2})$. We show in Section 9.1.1 that those algorithms can indeed not be modified to overcome this problem.



(a) Query Q_{C6} **(b)** Heavy decomp. **(c)** All-light decomp.

Fig. 8. Simple cycle of length 6 and two decompositions.

We illustrate with 6-cycle query Q_{C6}, depicted in Fig. 8a. (Here x_i in Example 2 is replaced by A_i to better distinguish the concrete 6-cycle from the general ℓ-cycle case.) First, we horizontally partition each relation R_i into R_{iH} and R_{iL} according to whether the tuples are *heavy* or *light*: R_{iH} receives all heavy tuples; R_{iL} the others (light ones). A tuple t in relation R_i is heavy [8] iff value $t.A_i$ occurs at least $n^{2/\ell} = n^{1/3}$ times in column $R_i.A_i$. Then the maximum number of distinct heavy values in a column is at most $n^{1-2/\ell} = n^{2/3}$. We create $\ell + 1 = 7$ database partitions:

$$
\begin{aligned}
T_1 &= \{R_{1H},\ R_2,\ R_3,\ R_4,\ R_5,\ R_6\} \\
T_2 &= \{R_{1L},\ R_{2H},\ R_3,\ R_4,\ R_5,\ R_6\} \\
&\vdots \\
T_6 &= \{R_{1L},\ R_{2L},\ R_{3L},\ R_{4L},\ R_{5L},\ R_{6H}\} \\
T_7 &= \{R_{1L},\ R_{2L},\ R_{3L},\ R_{4L},\ R_{5L},\ R_{6L}\}
\end{aligned}
$$

It is easy to verify that each output tuple will be produced by exactly one partition. The first $\ell = 6$ partitions use a "heavy" tree decomposition where the cycle is "broken" at the heavy attribute.

For instance, A_1 is the heavy attribute for R_{1H}; the resulting tree is shown in Fig. 8b. Each tree node is a bag whose content is materialized in time $O(n^{2-2/6}) = O(n^{5/3})$ by appropriately joining the corresponding relations. Consider top bag (A_1, A_2, A_3) derived from relations R_1 and R_2. Since R_{1H} contains at most $n^{2/3}$ distinct values, we can compute the bag with a simple nested-loop join. It goes through all pairs $A_1 - (A_2, A_3)$ of distinct heavy values of A_1 in R_{1H} and tuples (A_2, A_3) in R_2. Since there are at most $n^{2/3}$ distinct heavy A_1-values and n tuples in R_2, there are $O(n^{5/3})$ such pairs. For each pair we can verify in $O(1)$ if the corresponding (A_1, A_2) combination exists in R_{1H}. The other bag computations and heavy decompositions are analogous.

For T_7, which only contains light partitions, we use a different "all-light" tree decomposition shown in Fig. 8c. It materializes each bag with a join chain: For each tuple in one "endpoint" relation, find all matches in the next relation, and so on. Consider (A_1, A_2, A_3, A_4), which is derived from R_{1L}, R_{2L}, and R_{3L}. For each tuple $t_1 \in R_{1L}$, we find all matches in R_{2L}, then join with R_{3L}. There are $O(n)$ tuples in R_{1L}, but since all relations are light, each of them joins with at most $n^{1/3}$ in the next relation. Hence total complexity for materializing bag (A_1, A_2, A_3, A_4) is $O(n \cdot n^{1/3} \cdot n^{1/3}) = O(n^{5/3})$.

In general, the tree decomposition for an ℓ-cycle query produces a union of $\ell + 1$ trees, out of which ℓ use the heavy decomposition and 1 uses the light one. By setting the heavy-light threshold to $n^{2/\ell}$, we can materialize all bags of all trees in time $O(n^{2-2/\ell})$. Note that the number of tuples in a bag is $O(n^{2-2/\ell})$. Any such union of trees can be handled by our UT-DP framework.

5.4 Putting everything together

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

Theorem 15. *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* $\text{TTF} = O(T(\mathcal{A}))$, $\text{Delay}(k) = O(\log k)$, *and* $\text{MEM}(k) = O(S(\mathcal{A}) + k)$ *in data complexity.*

Proof. In all cases we use the Take2 algorithm.

First, consider the case of path queries and recall that we had made two assumptions in Section 4.3 for simplicity: (1) that the arities of the relations are bounded, thus $|Q| = \ell$ and (2) that the operations $\oplus$ and $\otimes$ of the selective dioid take $\gamma = O(1)$. Extending the analysis of Section 4.3 to the general case of unbounded arities and $\gamma = f(|Q|)$, we get $\text{TTF} = O(f(|Q|)n)$, $\text{Delay}(k) = O(f(|Q|)(\log k + |Q|))$, and $\text{MEM}(k) = O(|Q|n + k|Q|)$. Thus, $\text{TTF} = O(n)$, $\text{Delay}(k) = O(\log k)$, and $\text{MEM}(k) = O(n + k)$ in data complexity.

For tree queries, TTF and $\text{MEM}(k)$ stay the same, however $\text{Delay}(k)$ has an additional term that is quadratic in ℓ in the absence of the inverse element as we discuss in Section 6.2. Still, the data complexity remains the same as above.

For cyclic queries, first we apply the decomposition algorithm $\mathcal{A}$ to obtain a set of acyclic queries $\boldsymbol{Q}$. The number of acyclic queries we get is $g(|Q|)$ (according to our assumptions on $\mathcal{A}$ in Section 5.3) and also note that the "bags" (i.e., the derived input relations) in each acyclic query can be at most the number of attributes m [5]. Then, we run our UT-DP framework on top of $\boldsymbol{Q}$ using Take2. The top-level priority queue `Union` takes $O(\log g(|Q|))$ to pop an element or $O(1)$ in data complexity and then $O(\log k)$ to pull the next result from the corresponding tree. To avoid duplicates, we apply our construction of Section 6.3 which imposes a lexicographic order and thus, increases the complexity of pulling results from each acyclic query by an $O(|Q|)$ factor. Also, to filter the duplicates we have to spend an additional $O(g(|Q|))$ factor in delay since the number of duplicates cannot exceed the number of acyclic queries.

Overall, TTF is $O(T(\mathcal{A})$ and $\text{Delay}(k)$ is $O(\log k)$ in data complexity. For the space consumption, note that the total size of the derived input relations of $\boldsymbol{Q}$ is bounded by $S(\mathcal{A})$ and our framework only adds an $O(k)$ term in data complexity. □

6 RANKING FUNCTION

We now look deeper into the ranking functions that our framework supports.

6.1 Attribute weights

In order to keep our formalism clean and easy to follow, we focused only on weights on tuples. It is however straightforward to also handle weights on attributes by adding unary tables with weights on single columns. We illustrate next.

Example 16 (Attribute weights). *Consider the query $Q(x, y) :- R(x, y)$ over a database $R(A, B)$ with weight function $w_R : r \in R \longrightarrow \mathbb{R}^+$ on tuples, and two weight functions $w_A : a \in \mathtt{ADom}(A) \longrightarrow \mathbb{R}^+$, and $w_b : b \in \mathtt{ADom}(B) \longrightarrow \mathbb{R}^+$ on attributes. The problem can then be translated into one with only weight functions on tuples by introducing two new relations $S(A) = \mathtt{ADom}(A)$ and $T(B) = \mathtt{ADom}(B)$ with associated weight functions $w_S : s \in S \longrightarrow \mathbb{R}^+$, and $w_T : t \in T \longrightarrow \mathbb{R}^+$ and translated query $Q'(x, y) :- R(x, y), S(A), T(B)$.*

6.2 On the existence of the inverse (groups vs. monoids)

When we presented and analysed the algorithms in Section 4, we assumed for simplicity the existence of an inverse element for the $\otimes$ operator of the selective dioid $(W, \oplus, \otimes, \bar{0}, \bar{1})$. We now discuss what happens in the absence of that inverse element. We start with some definitions.

The inverse of an operation. An *Abelian group* is a commutative monoid $(W, \otimes, \bar{1})$ for which there exists an *inverse* for each element. More formally, for each x in W, there is an inverse element x' in W such that $x \otimes x^{-1} = x^{-1} \otimes x = \bar{1}$. We also write $y \oslash x$ as short form for $y \otimes x^{-1}$ (i.e., "$\oslash y$" composes y with the inverse of x).

Example 17 (Groups vs. Monoids). *The archetypical Abelian group is $(\mathbb{R}, +, 0)$, i.e. the real numbers with addition. An example commutative monoid that is not a group (and thus has no inverse in general) is logical conjunction: $(\{0, 1\}, \wedge, 1)$.[a] Here, 1 is the identity element because $x \wedge 1 = 1 \wedge x = x$ for $x \in \{0, 1\}$. However, for 0 there is no inverse $0'$ such that $0 \wedge 0' = 0' \wedge 0 = 1$. Another operation that has no inverse is the minimum: $(\mathbb{R}, \min, \infty)$. Here ∞ is the identity element because $\min(x, \infty) = \min(\infty, x) = x$ for $x \in \mathbb{R}$. However, for no x there is an inverse x' such that $\min(x, x') = \infty$.*

In general, the inverse element allows us to perform calculations that would be otherwise impossible. Thus, it can be used to short-circuit long calculations by reusing prior results.

Example 18 (Benefit of inverse elements). *Consider a commutative monoid $(W, \otimes, \bar{1})$ and the composition $x \otimes y = z$. Assume we are given z and y and would like to calculate x. Then this is only possible in general, if each element has an inverse; in other words, if the monoid is actually a group. To illustrate this issue, consider first the real numbers with addition $(\mathbb{R}, +, 0)$ and assume $(x, y, z) = (1, 2, 3)$. Then we can calculate $x = 1$ from $z = 3$ and $y = 2$ as $x = z + y' = 3 + (-2) = 1$. Next consider logical conjunction $(\{0, 1\}, \wedge, 1)$ with $(x, y, z) = (1, 0, 0)$. Then we cannot calculate $x = 1$ from $z = 0$ and $y = 0$ (both $x = 1$ or $x = 0$ are possible). Similarly, consider minimum $(\mathbb{R}, \min, \infty)$ with $(x, y, z) = (3, 2, 2)$. Then we cannot calculate $x = 3$ from $z = 2$ and $y = 2$ (x could be any value in $[2, \infty]$).*

Ranked enumeration without an inverse. In the context of our algorithms, the inverse element is not a hard requirement. However, it can help simplify and speed up certain variants of ranked numeration. First, notice that Recursive never uses an inverse since it always constructs

[a] We write 1 for true and 0 for false.

solutions by appending one state to a suffix or a list of subtrees (see Lines 7 and 31 of Algorithm 2). Therefore the cost of the solution can easily be calculated by applying $\otimes$.

For the *ANYK-PART algorithms over T-DP*, we have to make a minor adjustment which will incur an additional $O(\ell^2)$ delay term in the complexities presented in Figure 5. To illustrate this, we use the terminology of Algorithm 1. First, notice that in the path case the inverse element is not needed. In Line 16, a new candidate is inserted into the priority queue with weight solution.prefixWeight $\otimes$ $w(\text{tail}, s) \otimes \pi_1(s)$. Intuitively this means that the "future cost" of the candidate (when optimally expanded in a solution) is the weight of its prefix composed with the weight of the new decision and with the optimal weight from there onward. Thus, we are able in $O(1)$ to calculate the weight it will have if we expand it without actually spending $O(\ell)$ to expand it. In T-DP this calculation is not possible because the weight of the optimal extension from s (which was the $\pi_1(s)$ term in the path case) involves subtrees that are not in $\lceil s \rceil$, thus it is not available at state s.

One way to circumvent this is to use the inverse and still get an $O(1)$ computation per new candidate. Let prevWeight be the weight of the prefix we popped from Cand in the current iteration. Then, the weight of the new candidate is prevWeight $\oslash$ $w(\text{tail}, \text{last}) \oslash \pi_1(last) \otimes w(\text{tail}, s) \otimes \pi_1(s)$. Intuitively, this means that to compute the weight of the new candidate we "subtract" the old decision weight and the optimal subtree weight of its target state and we "add" the new ones. If we don't have the inverse element, then the above computation is not possible; instead, we expand each of the $O(\ell)$ new candidates before inserting them into the Cand priority queue and traverse each one of them to compose the decision weights (as in Equation (6)). This costs $O(\ell^2)$ in total because we have $O(\ell)$ candidates and each one has $O(\ell)$ size.

6.3 Tie-breaking the output

We now elaborate on how to break ties between result weights consistently. This is a key element for handling cyclic queries with existing decomposition algorithms. Recall from Section 5.3 that we could use a decomposition (e.g. PANDA [5]) that generates a set of trees whose outputs are not necessarily disjoint. Thus the same result tuple could potentially be produced by multiple trees. It is easy to detect and remove those duplicates if they arrive in consecutive order (the step is then linear in number of trees, but constant in data complexity). This consecutive arrival is guaranteed if there are no ties in the weights of output tuples. If there are ties, however, the arrival between identical output tuples could be in the order of number of output tuples produced so far. To see why, imagine an extreme scenario where all the output tuples have the same weight and duplicates arrive in arbitrary order; in that case, the delay between consecutive results could be in the order of k, i.e. in the order of the number of already seen output tuples. For instance, assume 5 output tuples $\{r, s, t, u, v\}$ with the same weight, and assume 10 tree decompositions. Then a possible enumeration could be $(r, s, t, u, r, r, r, r, r, r, r, s, s, \ldots, t, t, u, \ldots)$. To prevent this, we redefine our ranking function slightly so that it breaks ties in a consistent way and thus no two output tuples will have the same weight. This guarantees again that only duplicates can have the same weight and hence all the duplicates of a tuple arrive consecutively.

Intuitively, we add a second dimension to our ranking function that captures a lexicographic order on the input tuples. Whenever two weights are equal, the tie will be broken by the value of that extra dimension, ensuring that only identical results have the same overall weight. In the end, the true weight can be recovered by looking only at the first dimension of the weight function.

Given two partially ordered sets A and B, the lexicographic order on the Cartesian product $A \times B$ is defined as

$$(a, b) \leq (a', b') \text{ iff } a < a' \text{ or } (a = a' \text{ and } b \leq b')$$

It is well known that this order is a total order if and only if the factors of the Cartesian product are totally ordered.

However, what is less known is that this order is a total order even if the first factor is just a *total preorder* (also called preference relation). Recall that a total preorder is reflective ($a \leq a$), transitive (if $a \leq b$ and $b \leq c$ then $a \leq c$), complete (for every a, b, $a \leq b$ or $b \leq a$), however not necessarily antisymmetric ($a \leq b$ and $b \leq a$ does not imply $a = b$). To illustrate this point, consider binary output tuples with domain $\{a, b, c, d, e\}$ under the attribute weight model (Section 6.1). Assume a total preorder on the domain values with $a = b < c < d < e$. Then the lexicographic order for three particular output tuples could be $(a, c) \rightarrow (b, d) \rightarrow (a, e)$. Thus the three tuples imply a total order although the domain values of the first column do not.

We now show how to use this property to force our any-k enumeration to enumerate the same output tuple with a delay that depends only on the query even if we use a decomposition method (such as PANDA) that is not disjoint. The key idea is to force that each output tuple will be enumerated *consecutively* even if there are ties, i.e. multiple output tuples with the same weights.

Assume that for an output tuple r, the original ranking function $w(r)$ was defined with operators $\oplus, \otimes$ and a total order $\leq$. Then the new ranking function is the Cartesian product $w'(r) = (w(r), \mathbf{id}(r))$, with $\mathbf{id}(r)$ capturing a lexicographic order as in Section 2.2, and the following two operators:

(1) $w'(r_1) \otimes w'(r_2) = w'(r_1)$ iff $(w(r_1) \leq w(r_2) \wedge w(r_2) \not\leq w(r_1)) \vee (w(r_1) \leq w(r_2) \wedge w(r_2) \leq w(r_1) \wedge \mathbf{id}(r_1) \preceq_{\mathrm{L}} \mathbf{id}(r_2))$, else $w'(r_1) \otimes w'(r_2) = w'(r_2)$ and

(2) $w'(r_1) \otimes w'(r_2) = (w(r_1) \otimes w(r_2), \mathbf{id}(r_1) \otimes_{\mathrm{L}} \mathbf{id}(r_2))$.

As can be easily seen, the new ranking function is also defined over a *selective dioid*, and our any-k algorithms immediately apply.

6.4 Other examples of ranking functions

Throughout the main paper we focused on the ranking function that consists of the operators min (which is selective) and addition. These correspond to the *tropical semiring* $(\mathbb{R}_{\min}, \min, +, \infty, 0)$ with $\mathbb{R}_{\min} := \mathbb{R} \cup \{\infty\}$, which is an instance of a *selective dioid* (see the definition in Section 2.2). Under this perspective, Bellman's famous *principle of optimality* discussed in Section 3 is a re-statement of the more general *distributivity of addition over minimization*: $\min(x + z, y + z) = \min(x, y) + z$.

Another example of a selective dioid that our approach works on is the Boolean semiring $(\{0, 1\}, \vee, \wedge, 0, 1)$, where the disjunction is also selective. Interestingly, our algorithms can also perform standard query evaluation by inverting the order to $1 \leq 0$. Since maintaining priority queues (or sorting) with $\{0, 1\}$ elements takes linear time, it follows that our algorithms can enumerate answers to a 4-cycle query with TTF $= O(n^{1.5})$ and TTL $= O(n^{1.5} + |\text{out}|)$. These match the best known algorithms for Boolean and full query evaluation which use the submodular width ($\text{subw}(Q_{C4}) = 1.5$) For worst-case output instances, i.e., $|\text{out}| = n^2$ we also match the AGM bound, i.e., our algorithm, like NPRR, is worst-case optimal.

Other examples of selective dioids that we can use are $(\mathbb{R}_{\max}, \max, +, -\infty, 0)$ with $\mathbb{R}_{\max} := \mathbb{R} \cup \{-\infty\}$ or $(\mathbb{R}_{\geq 0}, \max, \times, 0, 1)$ with $\mathbb{R}_{\geq 0} := [0, \infty)$. The former finds the heaviest tuples or equivalently, the "longest" paths in a graph (according to the input weights). The latter can be used to simulate bag semantics; if the weight of each input tuple reflects its multiplicity in the input relation, then by using that ranking function we first get the output tuple with the biggest multiplicity in the result and its output weight is that multiplicity.

7 EXPERIMENTS

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

Dataset	Nodes	Edges	Max/Avg Degree	Weights
Bitcoin [71, 72]	5,881	35,592	1,298 / 12.1	Provided
TwitterS [106]	8,000	87,687	6,093 / 21.9	PageRank
TwitterL [106]	80,000	2,250,298	22,072 / 56.3	PageRank

Fig. 9. Datasets used for experiments with real data.

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 [31], (4) Eager, (5) All [101] representing the anyK-part approach, and (6) Batch, which computes the full result using the Yannakakis algorithm [103] for acyclic queries and NPRR [82] for cyclic queries, both followed by sorting.

Queries. We explore *paths*, *stars*, and *simple cycles* over binary relations. The SQL queries are listed in Appendix B. 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 [82] 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 $[0, 10000]$.

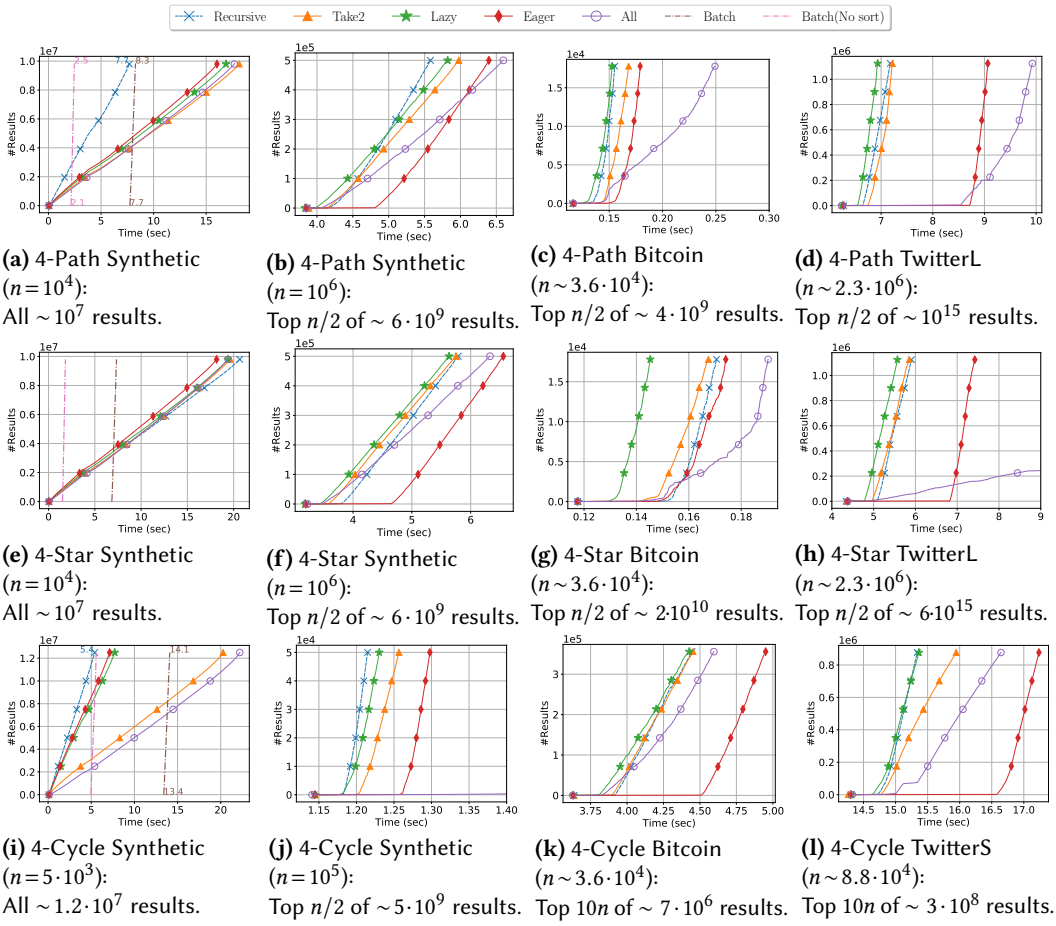
Real Data. We use two real networks. In Bitcoin OTC [71, 72], edges have weights representing the degree of trust between users. Twitter [106] edges model followership among users. Edge weight is set to the sum of the PageRanks [27] of both endpoints. To control input size, we only retain edges between 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 9 summarizes relevant statistics. Note that the size of our relations n is equal to the number of edges.

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 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 [33, 73]. To address this issue in the experiments, we use "bulk inserts" which heapify the inserted elements [31] or standard binary heaps when query size is small.

7.1 Experimental results

Figure 10 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*



(a) 4-Path Synthetic ($n = 10^4$): All $\sim 10^7$ results.
(b) 4-Path Synthetic ($n = 10^6$): Top $n/2$ of $\sim 6 \cdot 10^9$ results.
(c) 4-Path Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $n/2$ of $\sim 4 \cdot 10^9$ results.
(d) 4-Path TwitterL ($n \sim 2.3 \cdot 10^6$): Top $n/2$ of $\sim 10^{15}$ results.

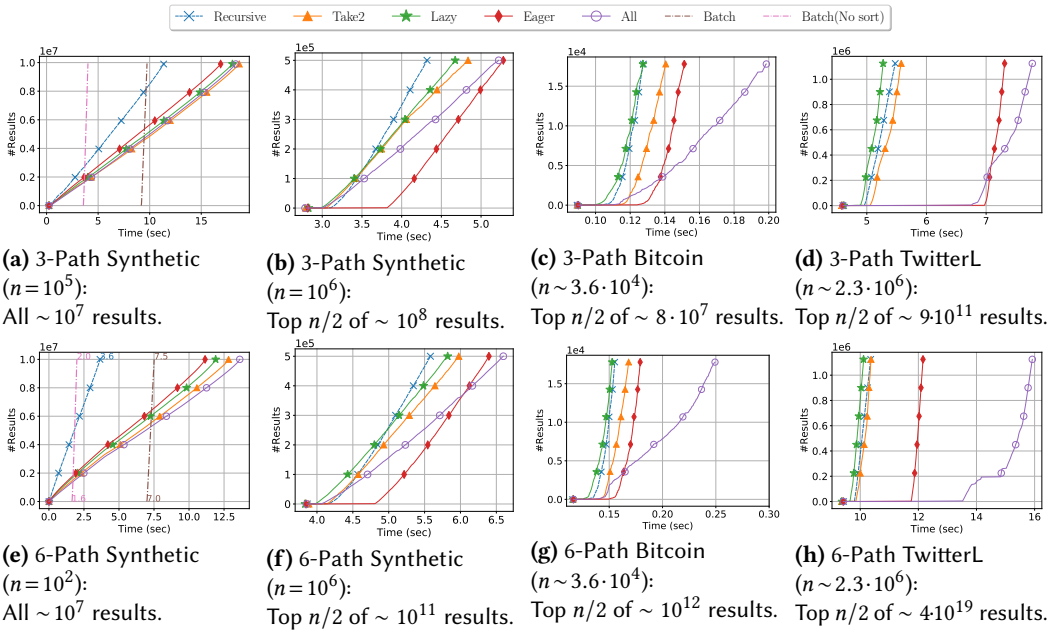
(e) 4-Star Synthetic ($n = 10^4$): All $\sim 10^7$ results.
(f) 4-Star Synthetic ($n = 10^6$): Top $n/2$ of $\sim 6 \cdot 10^9$ results.
(g) 4-Star Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $n/2$ of $\sim 2 \cdot 10^{10}$ results.
(h) 4-Star TwitterL ($n \sim 2.3 \cdot 10^6$): Top $n/2$ of $\sim 6 \cdot 10^{15}$ 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 $\sim 7 \cdot 10^6$ results.
(l) 4-Cycle TwitterS ($n \sim 8.8 \cdot 10^4$): Top $10n$ of $\sim 3 \cdot 10^8$ results.

Fig. 10. Experiments on queries of size 4 (Section 7.1).

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. 10a and 10i) 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 [82], 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. 10e). 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 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. 10c, 10g, 10h, 10k and 10l). Recursive starts off slower, but often overtakes the others for sufficiently large k (Figs. 10b and 10j). Eager is also slow in the beginning because it has to sort each time it accesses a new choice set. Take2 showed mixed



(a) 3-Path Synthetic ($n = 10^5$): All $\sim 10^7$ results.

(b) 3-Path Synthetic ($n = 10^6$): Top $n/2$ of $\sim 10^8$ results.

(c) 3-Path Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $n/2$ of $\sim 8 \cdot 10^7$ results.

(d) 3-Path TwitterL ($n \sim 2.3 \cdot 10^6$): Top $n/2$ of $\sim 9 \cdot 10^{11}$ results.

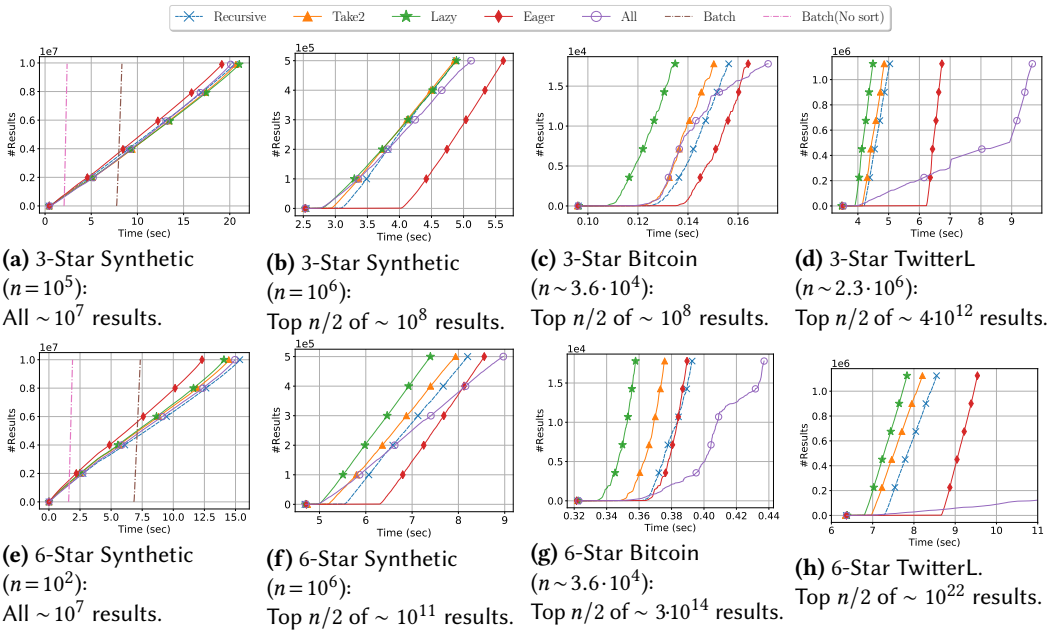
(e) 6-Path Synthetic ($n = 10^2$): All $\sim 10^7$ results.

(f) 6-Path Synthetic ($n = 10^6$): Top $n/2$ of $\sim 10^{11}$ results.

(g) 6-Path Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $n/2$ of $\sim 10^{12}$ results.

(h) 6-Path TwitterL ($n \sim 2.3 \cdot 10^6$): Top $n/2$ of $\sim 4 \cdot 10^{19}$ results.

Fig. 11. Experiments on path queries of sizes 3 and 6.



(a) 3-Star Synthetic ($n = 10^5$): All $\sim 10^7$ results.

(b) 3-Star Synthetic ($n = 10^6$): Top $n/2$ of $\sim 10^8$ results.

(c) 3-Star Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $n/2$ of $\sim 10^8$ results.

(d) 3-Star TwitterL ($n \sim 2.3 \cdot 10^6$): Top $n/2$ of $\sim 4 \cdot 10^{12}$ results.

(e) 6-Star Synthetic ($n = 10^2$): All $\sim 10^7$ results.

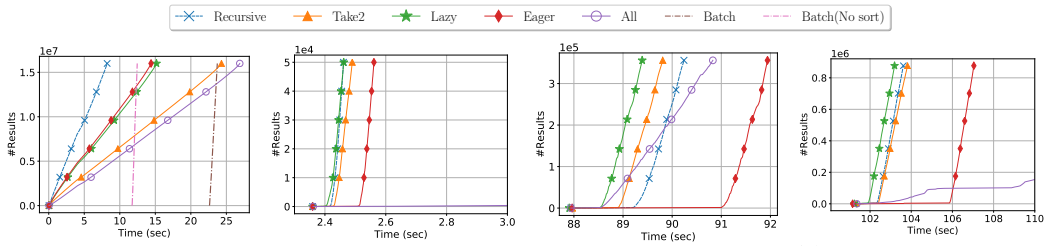
(f) 6-Star Synthetic ($n = 10^6$): Top $n/2$ of $\sim 10^{11}$ results.

(g) 6-Star Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $n/2$ of $\sim 3 \cdot 10^{14}$ results.

(h) 6-Star TwitterL. Top $n/2$ of $\sim 10^{22}$ results.

Fig. 12. Experiments on star queries of sizes 3 and 6.

results, performing near the top (Fig. 10f) or near the bottom (Fig. 10l). ALL performs poorly overall due to the large number of successors it inserts into its priority queue.



(a) 6-Cycle Synthetic ($n = 4 \cdot 10^2$): All $\sim 1.6 \cdot 10^7$ results.

(b) 6-Cycle Synthetic ($n = 10^5$): Top $n/2$ of $\sim 3 \cdot 10^{14}$ results.

(c) 6-Cycle Bitcoin ($n \sim 3.6 \cdot 10^4$): Top $50n$ of $> 10^9$ results.

(d) 6-Cycle TwitterS ($n \sim 8.8 \cdot 10^4$): Top $50n$ of $> 10^9$ results.

Fig. 13. Experiments on cycle queries of size 6.

7.2 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. We do not consider the cycle of length 3 (i.e., the triangle query) because our simple cycle decomposition does not give any bound that would be better than the Batch algorithm. Our goal with these experiments is to observe how the conclusions we made in Section 7.1 are affected when the query size changes.

Figures 11 to 13 depicts our results. The main observation is that Recursive's TTL benefits more from longer queries (Figures 11e and 13a) than shorter ones (Figure 11a). This makes sense because in a long path, there are more solutions that share the same suffixes and Recursive essentially reuses the ranking of those common suffixes to sort the entire solution set faster. In general, the situation that we saw in Fig. 10 is repeated. Lazy is again the winning algorithm for small k across the board, while All generally underperforms. Eager only makes sense in cases where a large number of results is returned if Recursive cannot reuse computation as it does in the path case, e.g. in the extreme case of star queries (Figures 12a and 12e)

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.*

7.3 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: we turn off fsync, synchronous_commit, full_page_writes, we set bgwriter_delay to the maximum (10 sec), bgwriter_lru_maxpages to 0, checkpoint_timeout to 1 hour and max_wal_size to a large value (1000 GB). We also give shared_buffers and work_mem 32 GB and set the isolation level to the lowest possible (READ UNCOMMITED). Like before we run 200 instances and report the median result. For each of those instances, we run PSQL 3 times and time only the last run to ensure that the input relations are cached.

Our results for the synthetic datasets are gathered in Figure 14. Overall, we found our Batch implementation to be 12% to 54% faster than PSQL. 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.

	3-Path $n = 10^5$ 10^7 Results	**4-Path** $n = 10^4$ 10^7 Results	**6-Path** $n = 10^2$ 10^7 Results	**3-Star** $n = 10^5$ 10^7 Results	**4-Star** $n = 10^4$ 10^7 Results	**6-Star** $n = 10^2$ 10^7 Results	**4-Cycle** $n = 5 \cdot 10^3$ $1.25 \cdot 10^7$ Results	**6-Cycle** $n = 4 \cdot 10^2$ $1.6 \cdot 10^7$ Results
Batch	9.74	8.27	7.51	8.32	7.34	7.35	14.09	23.72
PSQL	12.18	13.39	16.45	11.84	13.10	16.04	30.36	26.86
% faster	20%	38%	54%	30%	44%	54%	54%	12%

Fig. 14. Seconds to return the full result for Batch and PSQL on our synthetic data.

8 EXTENSIONS

8.1 Join queries with projections

So far, we have only considered *full* conjunctive queries, i.e. those that can be written in Datalog as $Q(\mathbf{x}) :- g_1(\mathbf{x}_1), \ldots, g_\ell(\mathbf{x}_\ell)$ where $\mathbf{x} = \bigcup_{i=1}^{\ell} \mathbf{x}_i$. A non-full conjunctive query (also called a join query with projection) $Q(\mathbf{y}) :- g_1(\mathbf{x}_1), \ldots, g_\ell(\mathbf{x}_\ell)$ has $\mathbf{y} \subset \mathbf{x}$ and asks to return only the *free* variables $\mathbf{y}$, while the remaining variables $\mathbf{x} \setminus \mathbf{y}$ (also called existentially quantified variables) are projected away. As mentioned in Section 2.1, our approach covers in principle all conjunctive queries: For non-full queries, we can perform the enumeration as if they were full and then project the output tuples on the free variables, discarding the duplicates. However, this approach might not always be ideal. In this section, we investigate the different possible semantics of ranked enumeration with projections and extend our approach to cover some of these cases more efficiently.

Two principal ways to define ranked enumeration. There are at least two reasonable semantics for ranked enumeration over joins queries with projections. Consider the 2-path query $Q(x_1) :- R_1(x_1, x_2), R_2(x_2, x_3)$ where we want to return only the first attribute x_1. Recall that we assume input weights have been placed on the relation tuples. What do we do if the same value v_1 of x_1 appears in two different results of the full query (v_1, v_2, v_3) and (v_1, v_2', v_3') with weights w and w', respectively? We identify two different semantics:

(1) *All-weight-projection* semantics: The first option is to return v_1 twice with both weights w, w' in the correct sequence. The corresponding SQL query would be:

```
SELECT   R1.A1, R1.W + R2.W as Weight
FROM     R1, R2
WHERE    R1.A2=R2.A2
ORDER BY Weight ASC
LIMIT    k
```

In general, we return the results and the weights that the full conjunctive query would return projected on the variables $\mathbf{y}$.[9] Thus, it is trivial to extend our approach to all-weight-projection semantics, as it is essentially equivalent to the ranked enumeration of full CQs: We enumerate the full CQ $Q(\mathbf{x})$ as before, yet we apply a projection $\pi_{\mathbf{y}}(r)$ to the output tuples r before returning them. The guarantees that we get in this case are the same as in Theorem 15.

(2) *Min-weight-projection* semantics: The second option is to return v_1 only once with the best (minimum) of the two weights. In this case, the SQL query is:

```
SELECT   X.A1, X.Weight
FROM
    (SELECT   R1.A1, MIN(R1.W + R2.W) as Weight
    FROM     R1, R2
    WHERE    R1.A2=R2.A2
    GROUP BY R1.A1) X
ORDER BY X.Weight
LIMIT  k
```

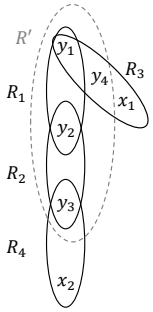
[9] In the case that two output results have the same weight, we still return both of them.

If r is a result of the query $Q(\mathbf{y})$ denoted by $r \in Q(\mathbf{y})$, we return the results of the query $Q(\mathbf{y})$ ranked by weight $w(r) = \min_{r' \in Q(\mathbf{y}) : \pi_{\mathbf{y}}(r') = r} \{w(r')\}$. In other words, each returned tuple r has the minimum weight over all tuples r' of the full query $Q(\mathbf{x})$ that map to r if projected on $\mathbf{y}$. While it is still possible to apply the projection as a post-filtering step and get a correct algorithm, there is no guarantee on the delay. If a lot of consecutive results project to the same variables y, then we might have to wait for the next result for a time that can be as high as $O(|\text{out}|)$ in the worst case. We next discuss a non-trivial extension that can handle min-weight-projection semantics with logarithmic delay in certain cases.

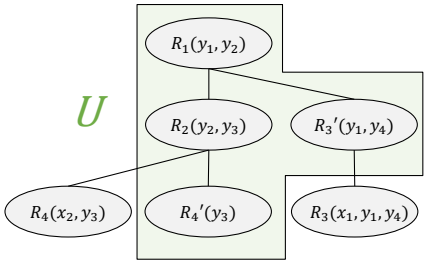
Min-weight-projection semantics. To efficiently handle min-weight-projection semantics, we resort to the techniques that have been developed for *unranked* constant-delay enumeration. Bagan et al. [11] show that the acylic queries which are *free-connex* admit constant delay enumeration after linear-time preprocessing. Multiple characterizations of these queries exist [18]. One particularly useful way to identify them is to check for the acyclicity of the corresponding hypergraph that includes an additional hyperedge connecting the head variables $\mathbf{y}$ [25]. For further reading on the topic, we refer the reader to the paper that introduced the concept [11] and recent surveys and tutorials [18, 41, 93]. In the following, we proceed to modify some of the common techniques for free-connex acyclic queries in order to accommodate efficient ranked enumeration under min-weight-projection semantics.

Intuitively, unranked constant-delay enumeration on free-connex acyclic queries [18] works by constructing an appropriate join tree that groups the free variables together. The tree is first swept bottom-up with semi-joins as in the Yannakakis algorithm [103] and then pruned so that only the free variables remain. The answers to the query can then be enumerated as if it were full (with no projections). We present a modification of this approach for ranked enumeration under min-weight-projection semantics with a logarithmic (instead of constant) delay. Essentially, it involves replacing the semi-joins with our Dynamic Programming scheme.

Example 19 (Free-connex query). *Consider the free-connex acyclic query* $Q(y_1, y_2, y_3, y_4) :- R_1(y_1, y_2), R_2(y_2, y_3), R_3(x_1, y_1, y_4), R_4(x_2, y_3)$. *We can check that it is indeed free-connex if we add an additional relation* R' *that encompasses all the free variables* $\mathbf{y}$ *(Fig. 15a) and then verify that the modified query is acyclic (e.g. by finding a join tree). Using the algorithm of Brault-Baron [25], we can construct a join tree for* Q *such that a connected subset of nodes* U *contain all the free variables and no existentially quantified ones (Fig. 15b). In order to achieve that, we have to introduce two additional atoms with two new relation symbols:* $R'_3 = \pi_{Y_1, Y_4}(R_3)$ *and* $R'_4 = \pi_{Y_3}(R_4)$. *Given this join tree and a database instance (Fig. 15c), we can construct the T-DP state space* T *shown in Fig. 15d. The non-artificial stages (depicted in gray) correspond to the nodes of the join tree and are populated by states (depicted by small white circles) that correspond to tuples from the relations. If we were to run our ranked enumeration algorithms on* T*, we would enumerate the answers to the full query* $Q(x_1, x_2, y_1, y_2, y_3, y_4)$. *Instead, we only run the bottom-up phase that computes the values* $\pi_1(s)$ *for all states* s*, shown with purple color in the figure. We proceed by removing the stages that do not belong to* U *and replacing them with artifical terminal nodes, thereby getting a modified state-space* T' *shown in Fig. 15e. Observe that ranked enumeration on* T' *will now enumerate the answers to the query* $Q(y_1, y_2, y_3, y_4)$. *To get the correct min-weight-projection semantics, we also have to modify the input weights on* T'*. Consider state* $s_3 \in S_2$ *on* T *that has two branches, one towards* S_4 *and one towards* S_5*. For the first branch, we have to choose among two decisions* (s_3, s_6) *and* (s_3, s_7)*. The minimum is achieved with* (s_3, s_6) *since* $1 = w(s_3, s_6) + \pi_1(s_6) < w(s_3, s_7) + \pi_1(s_7) = 2$*. Therefore, when we remove stage* S_4 *in* T' *and replace it with a terminal stage* $S'_5 = \{t_5\}$*, we set the weight of the decision* (s_3, t_5) *to be equal to that minimum, i.e.,* $w(s_3, t_5) = \min_{(s_3, s) \in E_{24}} \{w(s_3, s) + \pi_1(s)\} = 1$. *Notice that the minimum achievable*



(a) The hypergraph of the query and an additional atom R' that is used to check the free-connex property.



(b) The join tree with a connected subset of nodes U that contain precisely the free variables.

R_1

Y_1	Y_2	w
1	1	0
2	2	2

R_2

Y_2	Y_3	w
1	1	1
2	4	3

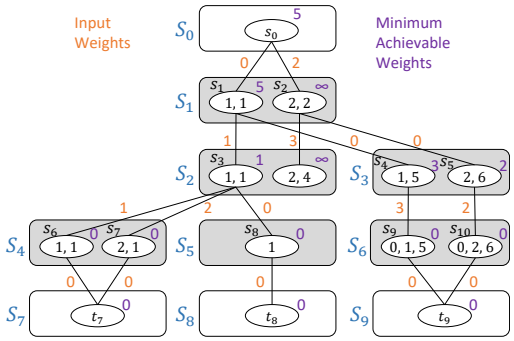
R_3

X_1	Y_1	Y_4	w
0	1	5	3
0	2	6	2

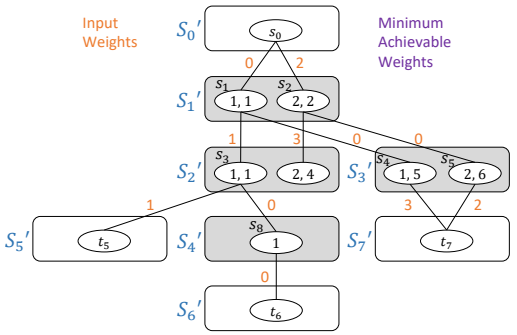
R_4

X_2	Y_3	w
1	1	1
2	1	2

(c) An example database instance.



(d) T-DP state-space T that corresponds to the full query $Q(x_1, x_2, y_1, y_2, y_3, y_4)$ using the join tree of Fig. 15b. For a state s, an identifier is depicted on its top-left and $\pi_1(s)$ on its top-right.



(e) T-DP state-space T' used for the ranked enumeration of $Q(y_1, y_2, y_3, y_4)$. Notice that stages not in U have been removed and replaced by artificial terminal ones. The input weights have been modified accordingly.

Fig. 15. Example 19: ranked enumeration under min-weight-projection semantics for the acyclic and free-connex query $Q(y_1, y_2, y_3, y_4) :- R_1(y_1, y_2), R_2(y_2, y_3), R_3(x_1, y_1, y_4), R_4(x_2, y_3)$ on an example database.

weights per branch that we need have already been computed from the bottom-up phase on T (see Eq. (7)).

Theorem 20 (Free-connex acyclic queries). *Ranked enumeration of the results of a free-connex acyclic query under min-weight-projection semantics can be performed with* TTF $= O(n)$ *and* Delay$(k) = O(\log k)$ *in data complexity.*

Proof. Let $\mathbf{y}$ be the set of free variables of a free-connex acyclic query $Q(\mathbf{y})$. Recall that in a join tree, each node represents one atom of the query – for a node t, let $\mathtt{var}(t)$ denote the set of variables of the corresponding atom. Since the query is free-connex, we can compute a join tree with a connected subset of nodes U that satisfy $\bigcup_{t \in U} \mathtt{var}(t) = \mathbf{y}$ in $O(|Q|)$ time using known techniques [18, 25]. In order to achieve this, some additional atoms might be introduced in the query. Set the input weights of all the tuples or T-DP states materialized from those atoms to 0. Also set the root of the tree to be some node $t \in U$.

Next, from the join tree, construct in a bottom-up fashion the corresponding T-DP state space graph as in Section 5.1 and denote it as T. This takes $O(n)$ time. Every T-DP solution of T is by construction an answer to the full query $Q(\mathbf{x})$ (with no projections). Given that (1) U contains all

the free variables y needed for answering $Q(\mathbf{y})$ and (2) bottom-up consistency has already been enforced, it suffices to perform ranked enumeration only to the subtree induced by U. To do that, create a copy T' of T that only retains the stages that belong to U [10]. Complete T' with an artificial starting stage as the root of the tree and terminal stages as the leaves, exactly as in Section 5.1. We argue that there is a 1 to 1 correspondence between the T-DP solutions of T' and the answers to $Q(\mathbf{y})$. To see this, first consider a T-DP solution Π of T'. It must contain states that belong to $\mathbb{S}$, (recall that these are the ones that were not removed from the bottom-up pass) hence they can reach the terminal nodes of the original T-DP state space graph T. Thus, there is a way to extend Π to a solution to the original state-space T, which corresponds to an answer to the full query $Q(\mathbf{x})$. Thus, the values assigned to the variables $\mathbf{y}$ are an answer to $Q(\mathbf{y})$. Conversely, consider an answer to $Q(\mathbf{y})$ which is an assignment of values to the the y variables. Since the subset U of the join tree contains precisely the free variables y, we can find tuples in the materialized relations of the join tree and equivalently, states in T' that form a T-DP solution using those values.

To get min-weight-projection semantics, we have to make adjustments to the input weights of T'. In particular, we introduce some additional terminal stages to T' and set the weights of the decisions that reach them according to the weights of T that have been removed from T'. Let S_r be a (non-artificial) stage in T and $S_p = pr(S_r)$ its parent such that $S_r \notin U$ and $S_p \in U$. Also let S'_p be the copy of S_p in T'. We add to T' a stage $S'_t = \{s'_t\}$ and decisions (s'_p, s'_t) for $s'_p \in S'_p$ with $\pi_1(s_p) \neq \infty$. Notice that this does not add or remove any T-DP solutions from T'. The weight of the new decisions is set to be the minimum achievable weight that s_p could reach in T from the S_r branch: $w(s'_p, s'_t) = \min_{(s_p, s_r) \in E_{pr}} = \{w(s_p, s_r) + \pi_1(s_r)\}$. This can be done in time linear in T. For a solution Π of T and a solution Π' of T', let $\Pi' \subset \Pi$ denote the fact that Π is an extension of Π', i.e., they agree on the subset U. By our construction, it is easy to see that for a solution Π' of T' we get $w(\Pi') = \min_{\Pi : \Pi' \subset \Pi} w(\Pi)$.

The total time spent so far is linear in the size of the database. After we modify the weights of the decisions as described above, we perform a bottom-up pass on T' once more and finally, apply Take2 on T' to get ranked enumeration with $O(\log k)$ delay. □

We proceed to strengthen the above result with lower bounds that also originate from the works on unranked enumeration. First, we state some complexity-theoretic assumptions that are commonly used and on which we will rely on. For more information on these conjectures, we refer the reader to the extensive discussion by Berkholz et al. [18].

- BMM is the hypothesis that two $n \times n$ Boolean matrices cannot be multiplied over the Boolean semiring in time $O(n^2)$.
- sparseBMM is the hypothesis that two Boolean matrices cannot be multiplied over the Boolean semiring in time $O(m)$, where m is the number of the non-zero entries in the input and the output.
- Triangle is the hypothesis that a triangle cannot be identified in a graph of m edges within $O(m)$ time.
- Hyperclique is the hypothesis that a $(k+1, k)$-hyperclique cannot be identified in a k-uniform hypergraph within $O(m)$, where m is the number of hyperedges and $k \geq 3$. Note that a $(k+1, k)$-hyperclique is a set of $k+1$ vertices such that every k-element subset is a hyperedge and in a k-uniform hypergraph, all hyperedges contain k vertices.

Besides the efficient algorithm for free-connex acyclic queries, Bagan et al. [11] provide a complementary negative result for queries without self-joins that rests on the BMM hypothesis. In particular, if a self-join-free acyclic query is not free-connex then we cannot enumerate its answers

[10] By slightly abusing the notation, we say that a stage belongs to U if its corresponding node in the join tree belongs to U

with constant delay after linear preprocessing. This essentially creates a dichotomy for the class of self-join-free acyclic queries: the only ones that can be handled efficiently are those that are free-connex. Later developments replace the original BMM assumption with sparseBMM, which is considered more likely to be true [18] and further extend the dichotomy to all self-join-free conjunctive queries with some additional assumptions. We restate this result below:

Theorem 21 ([11, 25]). *Assuming sparseBMM, Triangle and Hyperclique, unranked enumeration of the results of a self-join-free conjunctive query that is not acyclic free-connex cannot be done with $O(n)$ preprocessing and $O(\log n)$ delay.*

The following is immediate, since ranked enumeration is a harder problem than unranked enumeration and $\log k = O(\log n)$ in data complexity:

Corollary 22 (Ranked enumeration of conjunctive queries). *Assuming sparseBMM, Triangle and Hyperclique, ranked enumeration of the results of a self-join free conjunctive query under min-weight-projection semantics can be performed with* $\text{TTF} = O(n)$ *and* $\text{Delay}(k) = O(\log k)$ *delay in data complexity if and only if the query is acyclic and free-connex.*

8.2 Minimum-cost homomorphism

The connections between conjunctive query evaluation, constraint satisfaction, and the hypergraph homomorphism problem are well-known [30, 44, 70]. We now apply our framework to the minimum-cost homomorphism problem and generalize it in the same spirit as we generalized a standard Dynamic Programming (find the top-1 solution) to an any-k problem. In other words, we want to perform ranked enumeration, finding the min cost homomorphism, then the 2nd lowest cost homomorphism, etc. For that purpose we need to introduce a slight variation of the well-studied hypertree decompositions.

Pinned hypertree decomposition. A hypergraph $\mathcal{H}(N, E)$ is a pair $\mathcal{H} = (N, E)$ where N is a set of elements called *nodes*, and E is a set of non-empty subsets of N (i.e. $E \subseteq 2^N \setminus \emptyset$) called *edges*.

Definition 23 (TD [89]). *A tree decomposition of a hypergraph $\mathcal{H}(N, E)$ is a pair $\langle T, \chi \rangle$ where $T = (V, F)$ is a tree, and χ is a labeling function assigning to each vertex $v \in V$ a set of vertices $\chi(v) \subseteq N$, such that the following three conditions are satisfied:*

(1) (node coverage) for each node $b \in N$, there exists $v \in V$ such that $b \in \chi(v)$;
(2) (edge coverage) for each hyperedge $h \in E$, there exists $v \in V$ such that $h \subseteq \chi(v)$; and
(3) (coherence) for each node $b \in N$, the set $\chi^{-1}(b) = \{v \in V \mid b \in \chi(v)\}$ induces a connected subtree of T.

To distinguish between vertices of $\mathcal{H}$ and T, we will denote the former *nodes* N, and the latter *vertices* V. Thus, we also call the set $\chi(v)$ for $v \in V$ the nodes of v.

Definition 24 (HD [54]). *A (generalized) hypertree decomposition HD of a hypergraph $\mathcal{H}$ is a triple $HD = \langle T, \chi, \lambda \rangle$, called a hypertree for $\mathcal{H}$, where $\langle T, \chi \rangle$ is a tree decomposition of $\mathcal{H}$, and λ is a function labeling the vertices of T by sets of hyperedges of $\mathcal{H}$ such that,*

(4) for each vertex v of T, $\chi(v) \subseteq \bigcup_{h \in \lambda(v)} h$.[11]

In other words, the additional condition is that all nodes in the χ labeling of the TD are covered by hyperedges in the λ labeling.

A rooted hypertree decomposition $\langle T, \chi, \lambda, r \rangle$ of $\mathcal{H}$ is obtained by additionally choosing a root $r \in V$, which defines a child/parent relation between every pair of adjacent vertices, and

[11] Notice we use continuous numbering for the conditions as they build upon each other and we will need 6 conditions in total for our formulation.

ancestors/descendants in the usual way: In a rooted tree, the parent of a vertex is the vertex connected to it on the path to the root; every vertex except the root has a unique parent. A child of a vertex v is a vertex of which v is the parent. A descendant of any vertex v is any vertex which is either the child of v or is (recursively) the descendant of any of the children of v. We write $p(v)$ for the parent of node v, $C(v)$ for the set of children, and $D(v)$ for the set of descendents. A node without children is called a leaf.

Define as *reverse tree-order* the partial ordering on the vertices $V(T)$ with $u \leq_T v$ if and only if the unique path from u to the the root passes through v [39]. if $u <_T v$ we say that u lies below v in T. We call

$$\lceil v \rceil := \{u | u \leq_T v\}$$

the *down-closure* of y. In other words, $\lceil v \rceil = D(v) \cup \{v\}$.

Let $\langle T, \chi, r \rangle$ be a rooted tree decomposition of $\mathcal{H}$. For a node $v \in V$, we denote $\chi(\lceil v \rceil) = \bigcup_u \chi(u)$, with $u \in \lceil v \rceil$. In other words, $\chi(\lceil v \rceil)$ contains any node that is contained in either v or any of its descendants. We also define the subgraph $\mathcal{H}(\lceil v \rceil)$ as $\mathcal{H}(\lceil v \rceil) = \mathcal{H}[\chi(\lceil v \rceil)]$.

A vertex v *introduces* node b if $b \in \chi(v) \setminus \bigcup_c \chi(c)$, where the union is taken over all children $c \in C(v)$ of v [23]. In other words, a vertex introduces a node if that node is contained in the vertex but none of its children. Analogously, a vertex v *forgets* (or "projects away") node b if $b \in \bigcup_c \chi(c) \setminus \chi(v)$. Since a node can only be present in a connected set of vertices (forming a subtree), each node can be introduced multiple times, but only forgotten once.

DP algorithms rely on the following two key properties, which follow easily from Definition 23: (*i*) first, $\mathcal{H}(\lceil r \rceil) = \mathcal{H}$; (*ii*) second, for every $v \in V$, the only vertices of $\mathcal{H}(\lceil v \rceil)$ that (in $\mathcal{H}$) may be incident with edges that are not in $\mathcal{H}(\lceil v \rceil)$ are vertices in $\chi(v)$.

For our particular formulation of dynamic programming (DP) over hypertree decompositions, we need to add one more labeling function to the hypertree decomposition. We explain the intuition first: In a *HD*, an edge $h \in E$ can be mapped to multiple vertices in *HD*. However, when we add up the weights of a homomorphism, we need to make sure that weights are counted only once. We thus use a "pinning" function that maps each h to exactly one vertex in T in which h appears without projection (i.e., $\chi(v) \supseteq h$). We say that vertex v "pins" edge h.[12]

Definition 25 (Pinned HD). *A pinned hypertree decomposition of a hypergraph $\mathcal{H}$ is a quadruple $\langle T, \chi, \lambda, \rho \rangle$, where $\langle T, \chi, \lambda \rangle$ is a HD and ρ is a function labeling the vertices of T by sets of hyperedges of $\mathcal{H}$ such that,*

(5) for each $h \in E$, there exists exactly one *$v \in V$, such that $h \subseteq \rho(v)$ and $\chi(v) \supseteq h$; and*

(6) for each $v \in V : \rho(v) \subseteq \lambda(v)$.

Min Cost Homomorphism. Let $\mathcal{H}$ and $\mathcal{G}$ be hypergraphs, possibly with loops. A homomorphism from $\mathcal{H}$ to $\mathcal{G}$ is a function $\theta : V(\mathcal{H}) \to V(\mathcal{G})$ such that for all $h \in E(\mathcal{H})$, $\theta(h) \in E(\mathcal{G})$. Let $w : E(\mathcal{G}) \to \mathbb{R}$ be a cost function. The cost of a homomorphism θ from $\mathcal{H}$ to $\mathcal{G}$ is then

$$w(\theta) = \bigoplus_{h \in E(\mathcal{H})} w\big(\theta(h)\big)$$

The Minimum Cost Homomorphism problem (MCH) is then defined as

Definition 26 (Min Cost Homomorphism (MCH)). *Given hypergraphs $\mathcal{H}$ and $\mathcal{G}$, with edge weights w, decide whether a homomorphism from $\mathcal{H}$ to $\mathcal{G}$ exists, and if so, compute one of minimum*

[12] In the literature of tree decompositions of NP-hard problems, this problem is often solved by defining a "nice" tree decomposition that can be constructed from arbitrary tree decompositions with polynomial overhead, and subtracting weights at special vertices called "joins." This construction requires $\oplus$ to be a group instead of a simpler monoid (because we need an inverse element). Because in our work, we do care about polynomial overhead, we prefer to use a definition that avoids requiring an inverse operation.

weight.

$$w^* = \min_{\theta} \{w(\theta)\} \tag{10}$$

Let $\langle T, \chi, \lambda, \rho, r\rangle$ be a rooted pinned hypertree decomposition of $\mathcal{H}$. For a node $v \in V(T)$, our DP computes values $\text{val}(v, \theta)$ for every $\theta : \chi(v) \to N(\mathcal{G})$, defined as follows:

$$\text{val}(v, \theta) = \min \{w(\mu) \mid \mu : \chi(\lceil v \rceil) \to N(\mathcal{G}) \text{ s.t. } \mu|_{\chi(v)} = \theta\}.$$

So $\text{val}(v, \theta)$ is the minimum weight of a homomorphism μ from $\mathcal{H}(\lceil u \rceil)$ to $\mathcal{G}$ that coincides with θ. Then since $\mathcal{H}(\lceil r \rceil) = \mathcal{H}$, the minimum weight of a homomorphism from $\mathcal{H}$ to $\mathcal{G}$ is computed by taking the minimum value of $\text{val}(r, \theta)$ over all $\mu : \chi(r) \to N(\mathcal{G})$.

The values $\text{val}(v, \theta)$ can then be computed recursively in any sequence consistent with the reverse tree order $\leq_T$ as follows (Algorithm 3):

(1) If v is a leaf node, then initialize the weights with all grounded weights $\text{val}(v, \theta) = w(\theta)$ (Line 3).
(2) If v is not a leaf node, then first let S be the set of variables that appear in either v or its children (Line 5). Then for each homomorphism μ of the nodes in S to nodes $N(\mathcal{G})$, determine the cost for the sum of: (*i*) cost inherited all children consistent with μ, and (*ii*) additional cost incurred at that node v for all pinned edges $h \in \rho(v)$ (Line 6).

$$\text{val}(v, \mu) = \bigoplus_{c \in C(v)} \text{val}(c, \mu|_{X(c)}) \oplus \bigoplus_{h \in \rho(v)} w(\mu(h))$$

Then determine the minimum over all μ consistent with the variables in v (Line 7).
(3) Finally, determine the minimum weight of all homomorphisms consistent with the root (Line 8).

The minimum weight homomorphism θ^* can then be reassembled in one pass forward from the root to the leaves in the standard way as explained in Section 5. (To simplify the exposition, we are following the example of Bertsekas [22]: all dynamic programming algorithms construct the solution from the trace by which the optimum cost is found.)

Theorem 27. *[Correctness of* MCH-DP*] If the operation $\oplus$ is commutative, associative, and nondecreasing, then Algorithm 3 finds all optimal solutions of Eq. (10).*

Proof. We prove that MCH-DP finds the optimal value w^* of $w(\theta)$. At each vertex v of the hypertree decomposition, the computation eliminates ("forgets") a set of nodes T from its children: $T := \bigcup_c \chi(c) \setminus \chi(v)$. This elimination replaces all homomorphisms μ involving T and $\chi(v)$ with the restricted homomorphism θ where the image contains only variables in $\chi(v)$ (Line 8). So it suffices to show that the elimination of T does not change the minimum value.

The proof succeeds by induction. Consider a vertex v that forgets variables T and for which none of its descendants forgets any variable. Let $\rho(\lceil v \rceil)$ be the set of hyperedges $h \in E(\mathcal{H})$ that were pinned by any of $\lceil v \rceil$: $\rho(\lceil v \rceil) = \{h \in E(\mathcal{H}) \mid \exists u \in \lceil v \rceil : h \in \rho(u)\}$. Similarly, let $\rho(D(v))$ be the set of hyperedges $h \in E(\mathcal{H})$ that were pinned by any of the descendants of v, but not v. Further, let $\theta = \mu|_{\chi(v)}$.

Then

$$\min_{\theta} \Big\{ \bigoplus_{h \in E(\mathcal{H})} w(\theta(h)) \Big\} =$$
$$\min_{\mu} \Big\{ \text{val}(v, \mu|_{\chi(v)}) \oplus \bigoplus_{h \in E(\mathcal{H}) \setminus \rho[v]} w(\mu(h)) \Big\} \tag{11}$$

Algorithm 3: DP Formulation for Minimum Cost Homomorphism Problem over Pinned Hypertree Decomposition

Algorithm: MCH-DP
Input: Hypergraphs $\mathcal{H}$ and G. Cost function $w : V(\mathcal{G}) \to \mathbb{R}$. Rooted pinned *HD* of $\mathcal{H}$: $\langle T, \chi, \lambda, \rho, r\rangle$ with reverse tree order $\leq_T$.
Output: $w^* = \min_\theta \{w(\theta)\}$

for each vertex $v \in V(T)$ in order $\leq_T$ **do**
  if v *is a leaf node* **then**
    $\text{val}(v, \theta) = w(\theta)$
  else
    Let $S = \chi(v) \cup \bigcup_c \chi(c)$ with $c \in C(v)$
    Define a homomorphism $\mu : S \to N(\mathcal{G})$ s.t. $\text{val}(v, \mu) = \bigoplus_{c \in C(v)} \text{val}(c, \mu|_{X(c)}) \oplus \bigoplus_{h \in \rho(v)} w(\mu(h))$
    $\text{val}(v, \theta) = \min \{\text{val}(v, \mu) \mid \mu : S \to N(\mathcal{G}),\ \text{s.t. } \mu|_{\chi(v)} = \theta\}$
return $\min_\theta \{\text{val}(r, \theta)\}$

But the right-hand side Eq. (11) is now

$$\min_{\mu} \Big\{ \min_{\theta} \Big\{ \bigoplus_{h \in \rho[v]} w\big(\theta(h)\big) \Big\} \oplus \bigoplus_{h \in E(\mathcal{H}) \setminus \rho[v]} w\big(\theta(h)\big) \Big\}$$

By the commutativity, associativity, and nondecreasing monotonicity of $\oplus$, this expression is equal to the left-hand side Eq. (11). By straight-forward induction, it follows that $\min_\theta \text{val}(r, \theta)$ is the optimal value Eq. (10). □

9 RELATED WORK

Top-k. Top-k queries received significant attention in the database community [6, 7, 14, 28, 32, 64, 95, 96]. 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) [43] only accounts for the *cost of fetching* input tuples from external sources. Later work such as J* [80], Rank-Join [63], LARA-J* [75], and a-FRPA [45] 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 Section 9.1.3). We also refer the reader to a recent tutorial [98] that explores the relationship between top-k and the paradigms discussed in this paper.

Optimality in Join Processing. Acyclic Boolean queries can be evaluated optimally in $O(n)$ data complexity by the Yannakakis algorithm [103]. The AGM bound [9], a tight bound on the worst-case output size for full conjunctive queries, motivated worst-case optimal algorithms [81, 82, 83, 100] and was extended to more general scenarios, such as the presence of functional dependencies [53] 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 [89], (generalized) hypertree width (ghw) [54, 55, 56, 57, 59], fractional hypertree width (fhw) [61], and submodular width (subw) [78]. 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, 78] 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, 19, 29, 93, 94]. 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 [20] 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, 85, 86, 92] exploit the distributivity of product over union to represent query results compactly and generalize the results on bounded fhw to the non-Boolean case [87]. 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 Section 9.1.2 for the full example). Related to this line of work are different representation schemes [67] and the exploration of the continuum between representation size and enumeration delay [37].

Ranked enumeration. Both Chang et al. [31] and Yang et al. [101] provide any-k algorithms for *graph queries* instead of the more general CQs; they describe the ideas behind Lazy and All respectively. Kimelfeld and Sagiv [69] give an any-k algorithm for acyclic queries with polynomial delay. Similar algorithms have appeared for the equivalent Constraint Satisfaction Problem (CSP) [52, 58]. 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 [102], an unpublished paper [38] on arXiv proposed an algorithm for it. Without realizing it, the authors reinvented the REA algorithm [66], 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 $O(n^{1.5})$, while Deep and Koutris [38] require $O(n^2)$. 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, 40, 42, 62, 65, 66, 68, 74, 77, 76, 104]. 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 [62] introduces the concept of "deviations" as a sufficient condition for finding the k^{th} shortest path. Building on that idea, Dreyfus [40] proposes an algorithm that can be seen as a modification to the procedure of Bellman and Kalaba [17]. The *Recursive Enumeration Algorithm* (REA) [66] 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 [74] generalizes an earlier algorithm of Murty [79] and applies it to k-shortest paths. Aside from

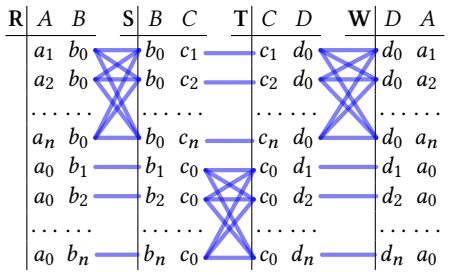


Fig. 16. Database I_1 showing sub-optimality of NPRR for TTF.

k-shortest paths, the Lawler procedure has been widely used for a variety of problems in the database community [48]. Along with the Hoffman-Pavley deviations, they are one of the main ingredients of our ANYK-PART approach. Eppstein's algorithm [42, 65] 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.

9.1 Detailed comparison to other paradigms

9.1.1 WCO join algorithms. We now show how the NPRR algorithm [82] fails to find the top ranked result in the same time bound as our approach. The key innovation of such worst-case optimal join algorithms is that they achieve the same complexity as the worst-case size of the output for every query. In the case of a 4-cycle query, NPRR produces the full join result in $O(n^2)$, a tight worst-case optimal bound. We next demonstrate with the help of the example database I_1 in Figure 16 that it requires $O(n^2)$ for the top-1 result as well, which cannot be easily fixed, whereas the techniques presented in this paper yield $O(n^{1.5})$.

NPRR execution on I_1. We follow the general treatment and formalism of [82].

(Step 1) WLOG, we use the total order of attributes $B \rightarrow C \rightarrow A \rightarrow D$, which implies choosing relation $W(A, D)$ in the first iteration: $f = \{A, D\}$, $\overline{f} = \{B, C\}$. The implied relations are: $E_1 = \{(B, C), (A, B), (C, D)\}$, and $E_2 = \{(A, D), (A, B), (C, D)\}$.

(Step 2) The algorithm will compute an intermediate set of values L for attributes in $\overline{f}$ with a recursive call on E_1. In our example, L will be a set of (b, c) pairs that satisfy $R(a_i, b), S(b, c), T(c, d_j)$ for some a_i and d_j. Its size is $|L| = 2n$.

(Step 3) For every (b, c) pair in L, there are two ways to find an (a, d) pair that forms a 4-cycle (a, b, c, d):

- *light pair:* if the number of (a, d) pairs joining with (b, c) in $R(a, b), S(b, c), T(c, d)$ is smaller than $|W(d, a)|$, iterate through those (a, d) pairs. For any such pair, if it is in $W(D, A)$ output (a, b, c, d).
- *heavy pair:* if $|W(d, a)|$ is smaller, then check for each (d, a) in $W(D, A)$, whether (a, b) is in $R(a, b)$ and (c, d) is in $T(c, d)$. Output (a, b, c, d) if both conditions are true.

All (b, c) pairs in L are light pairs in our I_2 example, since the join-generated pairs are of size n, while $|W| = 2n$.

Ranked enumeration with NPRR. A straightforward way to to turn this algorithm into a ranked enumeration algorithm is to compute all output tuples (a, b, c, d) and then sort them, which incurs $O(n^2 \log n)$ for TTF. Is it possible to do better than that? We will next show that any reasonable attempt to "retrofit" this algorithm fails to achieve $O(n^{1.5})$ TTF for our example. After Step 2 above,

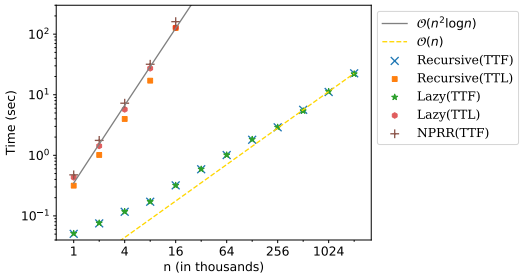


Fig. 17. TTF of NPRR vs our algorithms

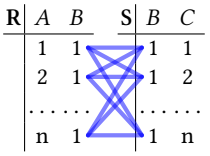


Fig. 18. Example 28: Database showing sub-optimality of factorised databases for certain lexicographic orders.

we have $2n$ pairs in L, the weight of which is known. Let us revisit Step 3 and break it further into the following parts:

(i) For every (b, c) pair in L, we know its weight w_S (the subscript refers to the relation of origin). We have already established that the number of (a, d) pairs that can be connected from any (b, c) is $1 \cdot n = n < |W(D, A)| = 2n$. Since all (b, c) in L are all light pairs, the execution plan is always to compute (a, d) pairs that satisfy $R(a, b), S(b, c), T(c, d)$. In this step, we can compute their weights as $w_{\text{light}} = w_R + w_S + w_T$. Therefore, for each (b, c) pair, we have a pool of matching pairs (a, d), each associated with a weight.

(ii) There are $2n$ (b, c) pairs and each one has n matching pairs of (a, d) in its pool. To find the result with the minimum weight, we need to go through the pool for each such (b, c) pair. For each combination, we have to verify that it is a result by checking $W(D, A)$ and also compute the total weight by adding w_W. Thus, $O(n^2)$ in total.

Experimental results. To better illustrate our point, we run NPRR against our algorithms (Recursive and Lazy) on a 4-cycle query Q_{C4} and database I_1 (Fig. 16) for various sizes n. Notice that even though our decomposition method guarantees $O(n^{1.5})$ for a 4-cycle query, it only needs $O(n)$ on I_1, since every relation has only one heavy value. We use the same experimental setup as in Section 7 and plot the time-to-first (TTF) and time-to-last (TTL). For NPRR we only plot the TTF, since TTL is very similar. We also plot two lines that show the trend of a linear and a quadratic function.

Figure 17 shows the results. We can clearly see that NPRR, as well as the TTL of our algorithms grow quadratically with n and soon become infeasible for large n. On the contrary, despite an initial overhead for small n, the TTF of our algorithms closely follows the linear line and is viable even for n in the order of millions of tuples: For example, for 16k tuples, our algorithm returns the top-1 result in 300 msec, while NPRR takes over 100 secs.

9.1.2 Comparison to Factorised Databases. Factorised databases (FDBs) [13, 85, 86, 87] support constant-delay enumeration of query results according to a desired lexicographic order on the attributes [12]. Lexicographic orders are a special case of the ranking function considered in this paper and our approach supports them (see Section 2.2), albeit with logarithmic delay. Here we look closer at the differences between the two approaches for this special case of lexicographic orders

and show that our approach can be asymptotically better in certain cases despite the logarithmic delay. Also note that the lexicographic ordering we describe in Section 2.2 is on the relations instead of the attributes but combined with the method in Section 6.1, we also support lexicographic orders on the attributes.

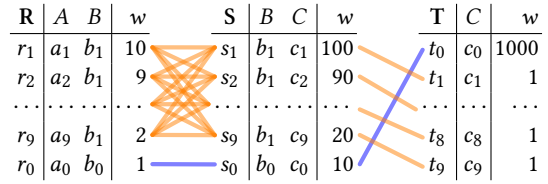
First, we provide a very short description of the main idea behind factorised databases and we refer the reader to the original papers for an in-depth presentation. To achieve a succinct representation, factorised databases repeatedly apply the distributivity law in an order described by a tree structure whose nodes are the attributes [86]. Intuitively, if X is the attribute of a node of the tree and anc(X) are its ancestor attributes, then every value $x \in X$ is represented at most once for each combination of values of anc(X). D-representations [87] provide further succinctness by making the dependencies of each attribute in the tree explicit. This means that some attributes in anc(X) might not actually determine what the possible X values are. Truly dependent ancestor attributes of a node are denoted as key(X). Each value $x \in X$ is then represented at most once for each combination of values of key(X).

These factorised representations provide unranked constant-delay enumeration out-of-the-box. Yet for specific lexicographic orders, there are two conditions that have to be met: (i) the order-by attributes have to be "at the top" of the tree and (ii) the tree order has to agree with the lexicographic order. If the tree order is not in agreement (e.g., we want A before B but A is a child of B in the tree), then the whole representation has to be restructured. The restructuring operation takes an input representation and transforms it to an output representation consistent with the lexicographic order in time linear (ignoring log factors) in the input *and output representation sizes*. However, the output representation itself could be inefficient. We next illustrate the simplest example where an ill-chosen lexicographic order results in a quadratic representation for a simple binary join.

Example 28 (Lexicographic orders). *Consider the path-2 query $Q_{P2}(A, B, C) :- R(A, B), S(B, C)$. As usual, n is the maximum number of tuples in a relation. Ideally, we would want to factorise it using a tree that has B as the root and A, C as its children. That way, every A and C value in the query result would be represented independently for each B value. However, for the lexicographic order $A \rightarrow C \rightarrow B$ this factorisation is not in agreement since B comes after A and C. The* only possible *tree that satisfies the condition (ii) above is a path from A to C to B. Note that the tree with A as the root and B, C as the children is not possible because of the path condition in factorised databases: attributes that belong to the same relation (B and C here) are in general dependent and have to lie in the same root-to-leaf path. In that tree,* key$(B) = \{A, C\}$. *According to Lemma 7.20 in [87], there exist arbitrarily large databases such that the number of B values in the representation is at least $n^{\rho^*(B \cup key(B))}$, where ρ^* is the fractional edge cover, thus $\Omega(n^2)$.*

Figure 18 presents a concrete instance where this happens. For this database, the single B-value 1 will be represented once for each combination of A, C values and there are n^2 of them. In contrast, our approach begins the enumeration after only linear time preprocessing. Thus in this case, the preprocessing step of FDBs takes $O(n^2)$ after which results can be enumerated in constant time. In contrast, our approach has TTF in $O(n)$ and TTL in $O(n^2)$ with logarithmic delay.

9.1.3 Comparison to top-k join algorithms. Consider the database I_2 from Fig. 19 with $\ell = 3$ relations and $n = 10$ tuples per relation. The top output tuple is marked in blue; it consists of the lightest tuples from the first $\ell - 1$ relations and the heaviest tuple from R_ℓ. J* [80] and Rank-Join [63] access the tuples in the input relations by decreasing weight. Their cost model takes into account only the number of database accesses, hence they try to minimize the depth up to which the sorted relations have to be accessed in order to find the top-k results. In this case, *both J* and Rank-Join will consider the $(n-1)^{\ell-1}$ combinations between R and S before getting to the the top-1 tuple (r_0, s_0, t_0).*



R	A	B	w
r_1	a_1	b_1	10
r_2	a_2	b_1	9
...	...	...	...
r_9	a_9	b_1	2
r_0	a_0	b_0	1

S	B	C	w
s_1	b_1	c_1	100
s_2	b_1	c_2	90
...	...	...	...
s_9	b_1	c_9	20
s_0	b_0	c_0	10

T	C	w
t_0	c_0	1000
t_1	c_1	1
...	...	...
t_8	c_8	1
t_9	c_9	1

Fig. 19. Database I_2 showing sub-optimality of J* and Rank-Join. (Section 9.1.3)

This happens because J* over-estimates their weight by using the large weight of t_0 to upper-bound them, while Rank-Join by default joins each newly encountered tuple with all the other ones seen so far. In contrast, our approach achieves $O(n \cdot \ell)$ for the top ranked result.

10 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 [91] of our algorithms.

Acknowledgements.

We are grateful to Hung Q. Ngo for reading drafts of this paper and providing valuable feedback. 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, F. Gerhardt, and N. Schweikardt. Constant delay enumeration for conjunctive queries: a tutorial. *ACM SIGLOG News*, 7(1):4–33, 2020. DOI: 10.1145/3385634.3385636.
[19] C. Berkholz, J. Keppeler, and N. Schweikardt. Answering conjunctive queries under updates. In *PODS*, pages 303–318, 2017. DOI: 10.1145/3034786.3034789.
[20] 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.
[21] U. Bertele and F. Brioschi. *Nonserial dynamic programming*. Academic Press, 1972. URL: https://dl.acm.org/doi/book/10.5555/578817.
[22] D. P. Bertsekas. *Dynamic Programming and Optimal Control*, volume I. Athena Scientific, 3rd edition, 2005. URL: http://www.athenasc.com/dpbook.html.
[23] H. L. Bodlaender, P. Bonsma, and D. Lokshtanov. The fine details of fast dynamic programming over tree decompositions. In *Parameterized and Exact Computation*, pages 41–53. Springer, 2013. DOI: 10.1007/978-3-319-03898-8_5.
[24] 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.
[25] J. Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013. URL: https://hal.archives-ouvertes.fr/tel-01081392.
[26] J. Brault-Baron. Hypergraph acyclicity revisited. *ACM Comput. Surv.*, 49(3), Dec. 2016. DOI: 10.1145/2983573. URL: https://doi.org/10.1145/2983573.
[27] 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.

[28] 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.

[29] 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.

[30] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90. ACM, 1977. DOI: 10.1145/800105.803397.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

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

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

[39] R. Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, 2005. DOI: 10.1007/978-3-662-53622-3.

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

[41] A. Durand. Fine-Grained Complexity Analysis of Queries: From Decision to Counting and Enumeration. In *PODS*, pages 331–346, 2020. DOI: 10.1145/3375395.3389130.

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

[43] 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.

[44] T. Feder and M. Y. Vardi. The computational structure of monotone monadic SNP and constraint satisfaction: A study through datalog and group theory. *SIAM J. Comput.*, 28(1):57–104, 1998. DOI: 10.1137/S0097539794266766.

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

[46] 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.

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

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1):1:1–1:24, 2007. DOI: 10.1145/1206035.1206036. URL: https://dblp.org/rec/journals/jacm/Grohe07.

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

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

[63] 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.

[64] 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.

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] 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.

[71] 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.

[72] 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.

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

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

[79] 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.

[80] 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.

[81] 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.

[82] 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.

[83] 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.

[84] U. o. Toronto. Computer Systems Research Group and M. Graham. On the universal relation. Technical report, 1980.

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

[86] 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.

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

[88] 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.

[89] 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.

[90] 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.

[91] 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.

[92] 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.

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

[94] 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.

[95] 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.

[96] 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.

[97] N. Tziavelis, D. Ajwani, W. Gatterbauer, M. Riedewald, and X. Yang. Optimal algorithms for ranked enumeration of answers to full conjunctive queries. *Proc. VLDB Endow.*, 13(9):1582–1597, 2020. DOI: 10.14778/3397230.3397250.

[98] N. Tziavelis, W. Gatterbauer, and M. Riedewald. Optimal join algorithms meet top-k:2659âĂŞ2665, 2020. DOI: 10.1145/3318464.3383132.

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

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

[101] 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.

[102] 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.

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

[104] 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.

[105] C. T. Yu and M. Z. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *COMPSAC 79. Proceedings. Computer Software and The IEEE Computer Society's*

Third International Applications Conference, 1979. Pages 306–312. IEEE, 1979. DOI: 10.1109/CMPSAC.1979.762509.

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

[107] S. Zilberstein. Using anytime algorithms in intelligent systems. *AI Magazine*, 17(3):73–83, 1996. DOI: 10.1609/aimag.v17i3.1232. URL: http://rbr.cs.umass.edu/shlomo/papers/Zaimag96.html.

A NOMENCLATURE

Symbol	Definition
Q	full conjunctive query $Q(\mathbf{x}) :- R_1(\mathbf{x}_1), \ldots, R_\ell(\mathbf{x}_\ell)$
Q^B	Boolean version of Q: $Q^B :- Q(\mathbf{x})$
$\boldsymbol{Q}$	set of acyclic queries $\{Q\}$
ℓ	number of atoms in Q or non-unary stages in DP
m	number of variables in Q
D	input database
n	number tuples in largest input relation in D
$w(r_i) \in W$	weight of input tuple r_i
$w(r) \in W$	weight of result tuple r with witness $(r_1, \ldots, r_\ell)$: $w(r) = w(r_1) \otimes \cdots \otimes w(r_\ell)$
out	set of output tuples for Q on D
s	state in DP problem
s_0, t	start and terminal state of DP problem
S_i	set of states in stage i
S	set of all states: $S = \bigcup_i S_i$
$\mathbb{S}_i$	set of states in stage i after bottom-up phase
$\mathbb{S}$	set of all states after bottom-up phase: $\mathbb{S} = \bigcup_i \mathbb{S}_i$
E	set of possible decisions: $E \subseteq S \times S$
$\mathbb{E}$	set of possible decisions after bottom-up phase
$\Pi(s)$	solution from state s
Π	complete solution: $\Pi = \Pi(s_0)$
$\Pi_j(s)$	j-th best solution from state s
$\pi_j(s)$	cost or weight of j-th best solution from state s
$\langle s_1 \ldots s_r \rangle$	solution prefix of length $r \in \mathbb{N}_1^\ell$
$\langle s_r \ldots t \rangle$	solution suffix starting at state s_r
$w(s, s')$	cost/weight of transitioning from state s to s'
$\pi(s_r)$	cost/weight of a solution/path suffix $\langle s_r s_{r+1} s_{r+2} \ldots s_\ell t \rangle$: $\pi(s_r) = w(s_r, s_{r+1}) \otimes w(s_{r+1}, s_{r+2}) \otimes \cdots \otimes w(s_\ell, t)$
$\texttt{Choices}_j(s)$	set of suffixes starting at s from which the j-th best suffix is selected
Cand	Set of prefixes containing the prefix of the next result to be returned
$\texttt{Succ}(s, s')$	successors of s' at state s that are considered by the Lawler-based partitioning approaches ANYK-PART
subw	submodular width of a query

B ADDITIONAL INFORMATION ON THE QUERIES USED FOR THE EXPERIMENTS

For completeness, we list the SQL code for the queries we used in our experiments. Each relation $R_i, i \geq 1$ has two attributes A_1 and A_2 and also an additional attribute W that contains the tuple weight. For the real datasets that correspond to networks, all relations R_i correspond to the same relation EDGES(fromNode, toNode, edgeWeight). In other words, A_1 corresponds to the source of a directed edge, A_2 corresponds to the destination and W corresponds to the edge weight.

The 4-Path query is:

```
SELECT   R1.A1, R2.A1, R3.A1, R4.A1, R4.A2
FROM     R1, R2, R3, R4
WHERE    R1.A2=R2.A1 AND R2.A2=R3.A1 AND
         R3.A2=R4.A1
ORDER BY R1.W + R2.W + R3.W + R4.W ASC
```

```
LIMIT    k
```

The 4-Star query is:

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

The 4-Cycle query is:

```
SELECT   R1.A1, R2.A1, R3.A1, R4.A1
FROM     R1, R2, R3, R4
WHERE    R1.A2=R2.A1 AND R2.A2=R3.A1 AND
         R3.A2=R4.A1 AND R4.A2=R1.A1
ORDER BY R1.W + R2.W + R3.W + R4.W ASC
LIMIT    k
```

The shorter or longer queries are similar and thus omitted.