Fast Matrix Multiplication for Query Processing*

XIAO HU, University of Waterloo, Canada

This paper studies how to use fast matrix multiplication to speed up query processing. As observed, computing a two-table join and then projecting away the join attribute is essentially the Boolean matrix multiplication problem, which can be significantly improved with fast matrix multiplication. Moving beyond this basic two-table query, we introduce output-sensitive algorithms for general join-project queries using fast matrix multiplication. These algorithms have achieved a polynomially large improvement over the classic Yannakakis framework. To the best of our knowledge, this is the first theoretical improvement for general acyclic join-project queries since 1981.

CCS Concepts: • **Theory of computation → Database query processing and optimization (theory)**.

Additional Key Words and Phrases: Join-project query, Fast matrix multiplication, Yannakakis framework

ACM Reference Format:
Xiao Hu. 2024. Fast Matrix Multiplication for Query Processing. *Proc. ACM Manag. Data* 2, 2 (PODS), Article 98 (May 2024), 25 pages. https://doi.org/10.1145/3651599

1 INTRODUCTION

Over the span of half of a century, the running time for multiplying two $U \times U$ matrices has progressively improved from $O(U^{2.808})$ [24] to $O(U^{2.371552})$ [12, 27]. It is still a big open question whether it can be solved in $U^{2+o(1)}$ time eventually. Consider the multiplication of two matrices (a_{ij}) and (b_{jk}). By treating each nonzero entry a_{ij} as a tuple (i, j) (and similarly for b_{jk}), Boolean matrix multiplication can be written as a join-project query over two relations

$$Q_{\text{matrix}} = \pi_{A,C} R_1(A, B) \bowtie R_2(B, C)$$

However, both the input matrices and output matrix could be sparse, i.e., the number of nonzero entries can be much smaller than U^2. We would like running times that depend on the number of non-zero entries in the input and output matrices, instead of U. The naive algorithm that materializes the join results and then projects out attribute B was the classic solution. Later, [1, 2, 11] investigated how to use fast matrix multiplication techniques to speedup sparse Boolean matrix multiplication, and [11] also showed strong evidence that these algorithms can significantly improve the classic solution in practice. A natural question arises: Can we exploit the fast matrix multiplication techniques in speeding up general join-project query processing? In this work, we answer this question in the affirmative by designing provably better output-sensitive algorithms for general join-project queries.

*This work was done while the author was visiting the Simons Institute for the Theory of Computing.

Author's address: Xiao Hu, xiaohu@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, Ontario, Canada, N2L 3G1.

ACM 2836-6573/2024/5-ART98
https://doi.org/10.1145/3651599

1.1 Problem Definition

Join-Project queries. A *join-project* query is defined as a triple $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, where $\mathcal{V} = \{A_1, A_2, \ldots, A_\ell\}$ models the *attributes*, $\mathcal{E} = \{e_1, e_2, \ldots, e_k\} \subseteq 2^{\mathcal{V}}$ models the subsets of attributes on which *relations* are defined and $\mathbf{y} \subseteq \mathcal{V}$ models the *output attributes* (a.k.a. *free attributes*). We assume that $\mathcal{V} = \bigcup_{e\in\mathcal{E}} e$. We also write Q as $Q = \pi_{\mathbf{y}} (R_1(e_1) \bowtie R_2(e_2) \bowtie \cdots \bowtie R_k(e_k))$. If $\mathbf{y} = \mathcal{V}$, Q is a *full join*; and if $\mathbf{y} = \emptyset$, Q is a *Boolean query*.

Let $\mathrm{dom}(A)$ be the *domain* of attribute $A \in \mathcal{V}$. Let $\mathrm{dom}(\mathbb{A}) = \prod_{A\in\mathbb{A}} \mathrm{dom}(A)$ be the domain of a subset of attributes $\mathbb{A} \subseteq \mathcal{V}$. A *tuple* t defined over a subset of attributes $\mathbb{A} \subseteq \mathcal{V}$ is a function that assigns a value from $\mathrm{dom}(A)$ to every attribute $A \in \mathbb{A}$. Furthermore, for a subset of attributes $X \subseteq \mathbb{A}$, we denote $\pi_X t$ as the set of values that t displays on every attribute in X.[1] An *instance* of Q is a set of relations $\mathcal{R} = \{R_e : e \in \mathcal{E}\}$, where each relation R_e consists a set of *tuples* defined over attributes e. Given an instance $\mathcal{R}$, the *query results* of Q on $\mathcal{R}$ is defined as

$$Q(\mathcal{R}) = \left\{t \in \mathrm{dom}(\mathbf{y}) : \exists t' \in \mathrm{dom}(\mathcal{V}), \pi_{\mathbf{y}} t' = t, \forall e \in \mathcal{E}, \pi_e t' \in R_e\right\}$$

i.e., the projection (without duplicates) of all join results (combinations of tuples, one from each relation, such that they share the same values on their common attributes) onto attributes $\mathbf{y}$. We denote $N = \sum_{e\in\mathcal{E}} |R_e|$ as the input size of instance $\mathcal{R}$ and $\mathrm{OUT} = |Q(\mathcal{R})|$ as the output size. We study the data complexity [25] of this problem, i.e., k and ℓ are constants, and measure the time complexity of algorithms in terms of N and OUT.

Classification of Queries. There are some important classes of join-project queries, which will be discussed throughout the paper.

- **Acyclic Query [5, 13, 17].** There are many equivalent definitions of α-acyclic joins, and we use the one based on generalized join tree [17]. A *generalized relation* R_e is defined on the projection of some input relation $R_{e'}$ for $e' \in \mathcal{E}$ onto a subset of attributes $e \subseteq e'$. A join-project query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ is acyclic if there exists a tree $\mathcal{T}$ such that (1) each node in $\mathcal{T}$ corresponds to an input relation or a generalized relation; (2) every input relation in $\mathcal{E}$ corresponds to a node in $\mathcal{T}$; (3) for every attribute $A \in \mathcal{V}$, the set of nodes containing it forms a connected subtree of $\mathcal{T}$. $\mathcal{T}$ is called a *generalized join tree* of Q. In the remaining of this paper, we use "join tree" to denote "generalized join tree" for simplicity.
- **Free-connex Query [3].** A join-project query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ is free-connex if Q is acyclic and $(\mathcal{V}, \mathcal{E} \cup \{\mathbf{y}\}, \mathbf{y})$ is also acyclic.

Model of Computation. We use the standard RAM model with uniform cost measure. For an instance of size N, every register has length $O(\log N)$. Any arithmetic operation (such as addition, subtraction, multiplication and division) on the values of two registers can be done in $O(1)$ time. Concatenating the values of two registers can be done in $O(1)$ time. Sorting the values of N registers can be done in $O(N \log N)$ time.

Notations. For a subset of attributes $\mathbb{A} \subseteq \mathcal{V}$, we define the *active domain* of $\mathbb{A}$ with respect to an instance $\mathcal{R}$ as the set of tuples in $\mathrm{dom}(\mathbb{A})$ appearing in at least one join result of $\mathcal{R}$, i.e., $\pi_{\mathbb{A}} \bowtie_{e\in\mathcal{E}} R_e$. For a pair of relations R_e and $R_{e'}$, we use $R_e \ltimes R_{e'} = \pi_e (R_e \bowtie R_{e'})$ to denote the semi-join between R_e and $R_{e'}$. For any $n \in \mathbb{Z}^+$, we use $[n]$ to denote $\{1, 2, \cdots, n\}$. For a pair of sets S_1 and S_2, we use $S_1 - S_2 = \{x \in S_1 : x \notin S_2\}$ to denote their set difference. We use $\widetilde{O}(t)$ to denote $O\left(t^{1+o(1)}\right)$ in all complexity results.

[1] If $|X| = 1$, say $X = \{A\}$, we also write $\pi_A t$ to denote $\pi_{\{A\}} t$ for simplicity.

1.2 The Yannakakis framework

In the RAM model, the Yannakakis framework [29] was proposed for acyclic join-project queries dated back to 1981. See Algorithm 4. It picks an arbitrary join tree[2] $\mathcal{T}$ for Q rooted at node r. It first removes all *dangling tuples* in the input instance $\mathcal{R}$, i.e., those won't appear in any full join result, in $O(N)$ time via a bottom-up and then a top-down phase of semi-joins. If $\text{OUT} = 0$, i.e., $Q(\mathcal{R}) = \emptyset$, all input tuples will be removed as dangling tuples. After done with semi-joins, Yannakakis framework performs joins and projections in a bottom-up way. Specifically, it takes two relation R_e and $R_{e'}$ such that e is a leaf and e' is the parent of e, projects away non-output attributes that appear in e but not in e' by replacing R_e with $\pi_{\mathbf{y}\cup(e\cap e')}R_e$, and replaces $R_{e'}$ with $R_e \bowtie R_{e'}$. Then R_e is removed and the step repeats until the root node R_r remains. It just outputs $\pi_{\mathbf{y}}R_r$ as the final results. The running time of the Yannakakis framework is proportional to the largest intermediate join size (after dangling tuples are removed), which is no larger than the full join size. Hence, Yannakakis framework is always better than the naive algorithm that computes the full join results and then projects out non-output attributes. We note that the largest intermediate join size could differ drastically on different *query plans*, i.e., each query plan of the Yannakakis framework corresponds to a rooted join tree together with the bottom-up computations of joins and projections.

If Q is *free-connex*, there is a query plan that only generate at most $O(\text{OUT})$ intermediate join results [4, 18], hence free-connex queries can be computed in $O(N + \text{OUT})$ time, which is optimal. Note that any acyclic full join query is free-connex. For non-free-connex queries, Yannakakis gave an upper bound of $O(N \cdot \text{OUT})$ on the largest intermediate join size. For Q_{matrix}, the simplest acyclic but non-free-connex query, this bound has been improved to $O\left(N \cdot \sqrt{\text{OUT}}\right)$ [2] by a better analysis. This is also tight, since there are instances with intermediate join results (which is also the full join results for Q_{matrix}) as large as $\Theta\left(N \cdot \sqrt{\text{OUT}}\right)$. This bound also extends to star queries,

$$Q_{\text{star}} = \pi_{A_1,A_2,\cdots,A_k} R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \cdots \bowtie R_k(A_k, B),$$

on which the tight bound is $O\left(N \cdot \text{OUT}^{1-\frac{1}{k}}\right)$. But, for line queries,

$$Q_{\text{line}} = \pi_{A_1,A_{k+1}} R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie \cdots \bowtie R_k(A_k, A_{k+1})$$

this bound $O(N \cdot \text{OUT})$ is already tight. See Appendix C.

People have incorporated generalized hypertree decomposition techniques [15] together with worst-case optimal join algorithms [21, 26] into the Yannakakis framework to handle cyclic join-project queries. Khamis et al. [19] proposed the PANDA algorithm, which, together with the Yannakakis framework, can compute a join-project query Q in $O\left(N^{\text{subw}} \cdot \text{OUT}\right)$ time, where $\text{subw} \geqslant 1$ is the sub-modular width of Q [20]. And $\text{subw} = 1$ if and only if Q is acyclic. If restricting the generalized hypertree decompositions to be free-connex, one can also obtain an algorithm that can compute a join-project query Q in $O\left(N^{\text{fc-subw}} + \text{OUT}\right)$ time, where fc-subw is the free-connex sub-modular width of Q [6]. Moreover, $\text{fc-subw} = 1$ if and only if Q is free-connex. It is not hard to see that $\text{fc-subw} \geqslant \text{subw}$ for arbitrary join-project query. These two algorithms are incomparable, unless we know the value of OUT.

1.3 Fast Matrix Multiplication for Q_{matrix}

As mentioned, people have also investigated how to use fast matrix multiplication techniques to speedup sparse Boolean matrix multiplication. The running time of computing two rectangular matrices of size $U^a \times U^b$ and $U^b \times U^c$ is denoted as $O\left(U^{\omega(a,b,c)}\right)$. The parameter ω, which is commonly-noted as the exponent of fast square matrix multiplication, refers to the case $\omega(1, 1, 1)$.

[2]Below, we always use "join tree" to denote "generalized join tree" for simplicity.

The currently best bounds on ω are $2 \leqslant \omega < 2.371552$ [28]. Rectangular matrix multiplication can always be tackled by partitioning rectangles into squares, for example $\omega(a,b,c) \leqslant a+b+c-\min\{a,b,c\}\cdot(3-\omega)$, but much better algorithms have been proposed. There are some important constants related to rectangular matrix multiplication, such as, $\alpha \leqslant 1$ defined as the largest constant such that $\omega(1,\alpha,1)=2$, and μ is the (unique) solution to the equation $\omega(\mu,1,1)=2\mu+1$. Note that $\alpha=1$ if and only if $\omega=2$, and the current best bounds on α are $0.321334 < \alpha \leqslant 1$ [28]. Moreover, $\mu=\frac{1}{2}$ if $\omega=2$, and the current best bounds on μ are $\frac{1}{2} \leqslant \mu < 0.527661$ [28].

Amossen and Pagh [2] first proposed an algorithm for Q_{matrix} by using fast matrix multiplication techniques, which runs in $\widetilde{O}\left(N^{\frac{2}{3}}\cdot \mathrm{OUT}^{\frac{2}{3}} + N^{\frac{(2-\alpha)\omega-2}{(1+\omega)(1-\alpha)}}\cdot \mathrm{OUT}^{\frac{2-\alpha\omega}{(1+\omega)(1-\alpha)}} + \mathrm{OUT}\right)$ time when $\mathrm{OUT} \geqslant N$, and $\widetilde{O}\left(N\cdot \mathrm{OUT}^{\frac{\omega-1}{\omega+1}}\right)$ time when $\mathrm{OUT} < N$. If $\omega=2$, this result degenerates to $\widetilde{O}\left(N^{\frac{2}{3}}\cdot \mathrm{OUT}^{\frac{2}{3}} + N\cdot \mathrm{OUT}^{\frac{1}{3}}\right)$. Very recently, Abboud et al. [1] have completely improved this to $\widetilde{O}\left(N\cdot \mathrm{OUT}^{\frac{\mu}{1+\mu}} + \mathrm{OUT} + N^{\frac{(2+\alpha)\mu}{1+\mu}}\cdot \mathrm{OUT}^{\frac{1-\alpha\mu}{1+\mu}}\right)$. If $\omega=2$, this complexity can be simplified as $\widetilde{O}\left(N\cdot \mathrm{OUT}^{\frac{1}{3}} + \mathrm{OUT}\right)$. Surprisingly, when OUT becomes larger than $N^{\frac{3}{2}}$, this result further degenerates to $\widetilde{O}(N+\mathrm{OUT})$, which is (almost) optimal up to a poly-logarithmic factor. On the other hand, improving this result for any value of OUT is rather difficult, assuming the hardness of the *all-edge-triangle* problem [1]. There are many algorithm proposed for sparse Boolean matrix multiplication, whose complexity are also measured by domain size of attributes; and we refer readers to [1] for details.

1.4 Other Results

Deep et al. [11] extended the algorithm for Q_{matrix} to Q_{star}, but without giving any theoretical analysis, but their results are completely dominated by our new results (see Appendix B). Hu et al. [16] studied *tree* (i.e., acyclic and each relation contains at most two attributes) join-aggregate queries defined over semi-rings in the massively parallel computation model, without using fast matrix multiplication. Interestingly, we revisit their algorithms in the RAM model for join-project queries, and observe some improvements over the Yannakakis framework on Q_{line} and Q_{star} by incorporating the best algorithm [1] for sparse matrix multiplication (see Appendix E). But, these improvements are also completely dominated by our new results. In addition, there has been a large body of works studying the fine-grained complexity of detecting or listing graph patterns (such as triangle, cycles and cliques), and their algorithms also use fast matrix multiplication. We refer interested readers to [7, 9, 10, 23] for details.

1.5 Our results

In this paper, we focus on acyclic but non-free-connex join-project queries where the output attributes can be arbitrary. We present new output-sensitive algorithms by exploiting the power of two choices - the classic Yannakakis framework [29] and the new algorithm for sparse matrix multiplication [1]. Our main results are summarized in Figure 1.

Limitation of Yannakakis Framework. One may wonder what we can benefit from incorporating the best algorithm for Q_{matrix} as a primitive into Yannakakis framework. Recall that in each step, it takes two relation R_e and $R_{e'}$ such that e is a leaf and e' is the parent of e, projects away non-output attributes that appear in e but not in e' by replacing R_e with $\pi_{\mathbf{y}\cup(e\cap e')}R_e$, and replaces $R_{e'}$ with $R_e \bowtie R_{e'}$. If e is the last child of e', then we can merge the last join with the subsequent projection of $R_{e'}$ when e' turns to be a leaf node, into one join-project query. More specifically, it suffices to compute $\pi_{\mathbf{y}\cup\mathrm{anc}(e')}R_e \bowtie R_{e'}$, where $\mathrm{anc}(e')$ denotes the set of attributes that appear in both e' and

Join-Project Query	Yannakakis Framework [29]	Semi-ring Algorithm	Fast Matrix Multiplication $\widetilde{O}(\cdot)$
			assume $\omega = 2$
			assume best ω, μ, α
			no assumption
Matrix	$\Theta\left(N \cdot \sqrt{\text{OUT}}\right)$		$\text{OUT} + N \cdot \text{OUT}^{\frac{1}{3}}$ [1]
			$N \cdot \text{OUT}^{0.3454} + \text{OUT} + N^{0.8} \cdot \text{OUT}^{0.5436}$ [1]
			$N \cdot \text{OUT}^{\frac{\mu}{1+\mu}} + \text{OUT} + N^{\frac{(2+\alpha)\mu}{1+\mu}} \cdot \text{OUT}^{\frac{1-\alpha\mu}{1+\mu}}$ [1]
Star ($k \geqslant 3$)	$\Theta\left(N \cdot \text{OUT}^{1-\frac{1}{k}}\right)$		$\text{OUT} + N \cdot \text{OUT}^{\frac{2}{3} - \frac{4/3}{3(k-1)+2}}$
			$N^{0.372} \cdot \text{OUT} + N \cdot \text{OUT}^{\frac{2(k-1)}{2.63(k-1)+2}}$
			$N^{\omega-2} \cdot \text{OUT} + N \cdot \text{OUT}^{\frac{2(k-1)}{(5-\omega)(k-1)+2}}$
Acyclic	$\Theta(N \cdot \text{OUT})$	$\Omega\left(N \cdot \text{OUT}^{1-\frac{1}{\text{freew}}}\right)$	$\text{OUT} + N \cdot \text{OUT}^{\frac{5}{6}}$
			$N^{0.372} \cdot \text{OUT} + N \cdot \text{OUT}^{0.871}$
			$N^{(\omega-2)} \cdot \text{OUT} + N \cdot \text{OUT}^{\frac{2}{3} \cdot \frac{3-\omega}{4-\omega} + \frac{1}{4-\omega}}$
General	$O\left(\min\left\{N^{\text{subw}} \cdot \text{OUT}, N^{\text{fn-subw}} + \text{OUT}\right\}\right)$ [15, 19, 21, 29]		$\text{OUT} + N^{\text{subw}} \cdot \text{OUT}^{\frac{5}{6}}$
			$N^{0.372 \cdot \text{subw}} \cdot \text{OUT} + N \cdot \text{OUT}^{0.871}$
			$N^{(\omega-2) \cdot \text{subw}} \cdot \text{OUT} + N^{\text{subw}} \cdot \text{OUT}^{\frac{2}{3} \cdot \frac{3-\omega}{4-\omega} + \frac{1}{4-\omega}}$

Fig. 1. Comparison of Yannakakis framework, semi-ring algorithms and fast-matrix-multiplication-based algorithms. Any complexity results based on fast matrix multiplication in a form of $\widetilde{O}(t)$ indicate $O(t^{1+o(1)})$. Results highlighted in red are achieved in this work. N is the input size and OUT is the output size. subw is the sub-modular width of the input query [20]. fc-subw is the free-connex sub-modular width [6]. freew is the free-width of input query (see Definition 3.2). k is the number of relations in the query. ω<2.371552 is the exponent of fast square matrix multiplication. $\mu <$ 0.527661 is the unique solution for $\omega(\mu, 1, 1) = 2\mu + 1$. $0.321334 < \alpha \leqslant 1$ is the largest constant such that $\omega(1, \alpha, 1) = 2$.

any ancestor of e'. This could be captured as a matrix multiplication problem if $e \cap \mathbf{y} - e' \neq \varnothing$, $e \cap e' \cap (\mathcal{V} - \mathbf{y}) - \text{anc}(e') \neq \varnothing$ and $(\mathbf{y} \cap e') \cup \text{anc}(e') - e \neq \varnothing$. However, its output size is not necessarily bounded by $O(\text{OUT})$. In the worst case, this output size can be as large as $\Theta(N \cdot \text{OUT})$. See Appendix C. Hence, materializing intermediate query results (not intermediate join size) is still the bottleneck of whole algorithm, no matter which algorithm is used for tackling the matrix multiplication problem.

Limitation of Semi-ring Algorithms. We next extend our scope to the whole class of algorithms under semi-ring [14, 18], which do not allow additive inverses. We first identify the *free-width* for an acyclic join-project query, denoted as freew, which measures the separation between output attributes. See Definition 3.2. Note that freew $= 1$ if and only if Q is free-connex. freew $= k$ for Q_{star} and freew $= 2$ for Q_{line}. In Appendix D, we prove that any semi-ring algorithm must run in $\Omega\left(N \cdot \text{OUT}^{1-\frac{1}{\text{freew}}}\right)$ time, which also captures the previous observation on Q_{matrix} [22] and

shows a critical separation between semi-ring algorithms and our new algorithms using fast matrix multiplication.

New Algorithms with Fast Matrix Multiplication. Due to the inherent limitations of semi-ring algorithms, we next resort to fast matrix multiplication techniques for faster algorithms.

(Section 2) We first revisit the algorithm for Q_{matrix} [1], and obtain two new observations when each attribute has a small *active domain*. We focus on star query Q_{star} with $k \geqslant 3$ and propose an algorithm that can reduce Q_{star} into a constant number of sub-queries, each of which either has bounded *intermediate join size* or *active domain size*, and will be tackled by Yannakakis framework and fast matrix multiplication separately. Our algorithm runs in $\widetilde{O}\left(N \cdot \text{OUT}^{\frac{2}{3}-\frac{4/3}{3(k-1)+2}} + \text{OUT}\right)$ time when $k \geqslant 3$,. This quantity $\frac{2}{3} - \frac{4/3}{3(k-1)+2}$ increases from $\frac{1}{2}, \frac{18}{33}, \frac{4}{7}, \cdots$, and approaches $\frac{2}{3}$ when k goes to infinity. When OUT becomes larger than $N^{\frac{3k-1}{k+1}}$, this result degenerates to $\widetilde{O}(N + \text{OUT})$, which is (almost) optimal up to poly-logarithmic factors.

(Section 3) For a general acyclic join-project query Q, we first show a *decompose* procedure based on the *existential connectivity* (formally defined in Section 3) of Q. Intuitively, computing Q is equivalent to computing the full join results of all sub-queries connected by non-output attributes. This way, it suffices to focus on an acyclic join-project query that is also existential-connected. Inheriting a similar high-level idea, we find a recursive way to reduce Q into a constant number of sub-queries each of which either has bounded intermediate join size or active domain size, hence can be computed efficiently by a hybrid strategy. Our algorithm runs in $\widetilde{O}\left(N \cdot \text{OUT}^{\frac{5}{6}} + \text{OUT}\right)$ time. Again, when OUT becomes larger than N^6, this result degenerates to $\widetilde{O}(N + \text{OUT})$, which is (almost) optimal up to poly-logarithmic factors.

(Section 4) One major technical difficulty we had to overcome is that the value of OUT is not readily available. It is known that OUT can be computed for free-connex queries within $O(N)$ time [18, 29], and can be estimated within a constant factor for line queries with high probability within $\widetilde{O}(N)$ time [8, 16]. But, how to efficiently compute or even obtain an $O(1)$-approximation of OUT for general non-free-connex queries still remains open. Inspired by the sparse recovery technique in [1], we present a general compress-recover framework to compute an $O(1)$-approximation of OUT on-the-fly. The only primitive we need is an algorithm for join-project queries when an $O(1)$-approximation of OUT is given, which is essentially our focus in Sections 2 and 3. From our analysis, this framework only increases the overall complexity by a poly-logarithmic factor.

(Section 5) Combining our new algorithm in Section 3 with generalized hypertree decompositions [19], we present an output-sensitive algorithm for cyclic join-project queries.

Remark 1. In Section 3, the decompose procedure could lead to better results if each sub-query has rather simple structures, such as tree join-project queries. See Appendix H.

Remark 2. In Section 4, the framework does not use fast matrix multiplication, hence it can be combined with any semi-ring algorithm. Moreover, we have an $O(1)$-approximation $\tilde{\text{OUT}}$ for OUT such that $\tilde{\text{OUT}} \leqslant \text{OUT} \leqslant 2 \cdot \tilde{\text{OUT}}$ when invoking the algorithms in Section 2 and Section 3.

2 STAR QUERY

In this section, we investigate star queries $Q_{\text{star}} = \pi_{A_1,A_2,\cdots,A_k} R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \cdots \bowtie R_k(A_k, B)$ with $k \geqslant 3$. We start with new analyses for the algorithm in [1]. All missing proofs are given in Appendix F.

2.1 Boolean Matrix Multiplication Revisited

Recall that for $Q_{\text{matrix}} = \pi_{A,C} R_1(A,B) \bowtie R_2(B,C)$, the active domains of attributes A, B, C is defined as $\pi_A(R_1 \bowtie R_2)$, $\pi_B(R_1 \bowtie R_2)$, $\pi_C(R_1 \bowtie R_2)$ separately. The Yannakakis framework can remove all dangling tuples in $O(N)$ time. The active domain of each attribute is essentially the collection of values that appears in at least one non-dangling tuple in the input instance. If the active domain sizes of A, B, C are small, we are able to get better upper bounds for Q_{matrix}, which will be used as primitives in our algorithms. After revisiting the algorithm in [1], we present the following results (by applying the "square" trick for rectangular matrix multiplication):

Lemma 2.1. *For an arbitrary instance $\mathcal{R}$ of Q_{matrix} with input size N and output size* OUT, *$Q_{\text{matrix}}(\mathcal{R})$ can be computed in*

- $\widetilde{O}\left(n_B^{\omega-2} \cdot \text{OUT} + N^{\frac{3-\omega}{4-\omega}} \cdot (n_B \cdot \text{OUT})^{\frac{1}{4-\omega}} + N\right)$ *or*
- $\widetilde{O}\left(n_B^{\omega-2} \cdot \text{OUT} + n_B \cdot \text{OUT}^{\omega-2} \cdot \max\{n_A, n_C\}^{3-\omega} + N\right)$

time, where n_A, n_B, n_C are the active domain sizes of A, B, C respectively.

The complexity results in Lemma 2.1 can be further improved via fast rectangular matrix multiplication. We won't pursue this direction further, which is left as the future work.

2.2 Algorithm

Now, we are ready to present our algorithm for Q_{star}. Suppose $\tilde{\text{OUT}}$ is known such that $\tilde{\text{OUT}} \leqslant \text{OUT} \leqslant 2 \cdot \tilde{\text{OUT}}$. All dangling tuples (that do not participate in any join result) are removed in $O(N)$ time. Our algorithm consists of five steps:

Step 1: Compute data statistics. We first compute for each value $b \in \text{dom}(B)$, its *degree* $d_i(b)$ in relation R_i for each $i \in [k]$, which is defined as the number of tuples displaying value b in attribute B, i.e., $d_i(b) = |\sigma_{B=b} R_i|$. It is not hard to see:

Lemma 2.2. *For any value $b \in \text{dom}(B)$, $\prod_{i=1}^{k} d_i(b) \leqslant \text{OUT}$.*

Proof of Lemma 2.2. For any value $b \in \text{dom}(B)$, the Cartesian product $\times_{i\in[k]} \sigma_{B=b} R_i$ must be a subset of final query results. Hence, $\prod_{i\in[k]} d_i(b) = \left|\times_{i\in[k]} \sigma_{B=b} R_i\right| \leqslant |Q_{\text{star}}(\mathcal{R})| = \text{OUT}$. □

Step 2: Reduce Q_{star}. For each value $b \in \text{dom}(B)$, we define a permutation $\phi_b : [k] \to [k]$ such that $d_{\phi_b(i)}(b) \geqslant d_{\phi_b(j)}(b)$ for any $1 \leqslant i \leqslant j \leqslant k$. Note that we can determine ϕ_b for all b's by sorting all values in $d_i(b)$, breaking ties. There are at most $k!$ number of permutations in total. Each permutation ϕ over $[k]$ defines a subset of values in $\text{dom}(B)$ as $B_\phi = \{b \in \text{dom}(B) : \phi_b = \phi\}$. Given a parameter $0 < \rho \leqslant 1$ (whose value will be determined later), we further divide each B_ϕ into two subsets:

$$B_\phi^{\text{light}} = \left\{ b \in B_\phi : \left(\prod_{i=2}^{k} d_{\phi(i)}(b) \right) \leqslant \tilde{\text{OUT}}^\rho \right\}$$

$$B_\phi^{\text{heavy}} = \left\{ b \in B_\phi : \left(\prod_{i=2}^{k} d_{\phi(i)}(b) \right) > \tilde{\text{OUT}}^\rho \right\}$$

Then, we reduce Q_{star} into the following at most $2 \cdot k!$ sub-queries:

$$Q_\phi^? = \pi_{A_1,A_2,\cdots,A_k} \bowtie_{i\in[k]} R_i(A_i, B_\phi^?)$$

where ? can be heavy or light, and $R_i(A_i, B_\phi^?)$ denote the set of tuples from R_i whose value in attribute B falls into $B_\phi^?$.

Step 3: Compute Q_ϕ^{light}. We compute the full join and then project out attribute B.

Step 4: Compute Q_ϕ^{heavy}. Let (η, β) be a partition of $[k]$, where $\eta = \{i \in [k] : i \text{ is odd}\}$ and $\beta = [k] - \eta$. We then materialize the following two intermediate joins:

$$R_{\phi,\eta}\left(\mathbb{A}, B_\phi^{\text{heavy}}\right) = \bowtie_{i\in\eta}\left(R_{\phi(i)}(A_{\phi(i)}, B_\phi^{\text{heavy}})\right)$$
$$R_{\phi,\beta}\left(\mathbb{C}, B_\phi^{\text{heavy}}\right) = \bowtie_{j\in\beta}\left(R_{\phi(j)}(A_{\phi(j)}, B_\phi^{\text{heavy}})\right)$$

where $\mathbb{A} = \bigcup_{i\in\eta} A_{\phi(i)}$ and $\mathbb{C} = \bigcup_{j\in\beta} A_{\phi(j)}$. Then, we can reduce Q_ϕ^{heavy} to Q_{matrix} as below:

$$\pi_{\mathbb{A},\mathbb{C}}\left(R_{\phi,\eta}\left(\mathbb{A}, B_\phi^{\text{heavy}}\right) \bowtie R_{\phi,\beta}\left(B_\phi^{\text{heavy}}, \mathbb{C}\right)\right) \tag{1}$$

and invoke the algorithm for sparse matrix multiplication in [1].

Step 5: Remove Duplicates. From above, each of the at most $2 \cdot k!$ subqueries could produce $O(\text{OUT})$ query results. The last step is to remove possible duplicates between them via sorting.

2.3 Analysis

We next analyze the time cost of each step. Step 1 and 2 can be done in $O(N)$ time. Step 5 can be done in $\widetilde{O}(\text{OUT})$ time. Step 3 takes $O\left(N \cdot \text{OUT}^\rho\right)$ time. As there are at most N tuples in $R_{\phi(1)}$, and each tuple $t \in R_{\phi(1)}$ can be joined with at most $\left(\prod_{i=2}^k d_{\phi(i)}\left(\pi_B t\right)\right) \leqslant \tilde{\text{OUT}}^\rho$ tuples by the definition of B_ϕ^{light}, the number of join results is $O\left(N \cdot \text{OUT}^\rho\right)$. Before analyzing step 4, we need a helper lemma:

Lemma 2.3. *Suppose $\prod_{i\in[k]} d_i = \lambda$ and $d_k \leqslant d_{k-1} \leqslant \cdots \leqslant d_1$. Let $\eta = \{i \in [k] : i \text{ is odd}\}$ and $\beta = [k] - \eta$. We obtain:*

- $\prod_{j\in\beta-\{2\}} d_j \leqslant \prod_{i\in\eta-\{1\}} d_i \leqslant \sqrt{\lambda} \leqslant \prod_{i\in\eta} d_i$;
- *if $d_1 \leqslant \lambda'$, then $\sqrt{\frac{\lambda}{\lambda'}} \leqslant \prod_{j\in\beta} d_j \leqslant \prod_{i\in\eta} d_i \leqslant \sqrt{\lambda\cdot\lambda'}$.*

Lemma 2.4. *The instance in (1) satisfies the following constraints:*

- $\max\left\{\left|R_{\phi,\eta}\right|, \left|R_{\phi,\beta}\right|\right\} \leqslant N \cdot \sqrt{\text{OUT}}$;
- *the active domain size of B_ϕ^{heavy} is $O\left(\frac{N}{\text{OUT}^{\rho/(k-1)}}\right)$;*
- *the active domain size of $\mathbb{A}$ and $\mathbb{C}$ is $O\left(\text{OUT}^{1-\frac{\rho}{2}}\right)$;*

Proof of Lemma 2.4. Note that each tuple $(a, b) \in R_{\phi(1)}$ can join with $\prod_{i\in\eta-\{1\}} d_{\phi(i)}(b) \leqslant \sqrt{\text{OUT}}$ results in $\left(\bowtie_{i\in\eta} R_{\phi(i)}\right)$, implied by Lemma 2.3. As there are at most N tuples in $R_{\phi(1)}$, $|R_{\phi,\eta}|$ can be bounded by $O(N \cdot \sqrt{\text{OUT}})$. A similar argument applies for $R_{\phi,\beta}$. We next bound the active domain size of each attribute. For every value $b \in B_\phi^{\text{heavy}}$, we have $d_{\phi(2)}(b) \geqslant \left(\prod_{i=2}^k d_{\phi(i)}(b)\right)^{\frac{1}{k-1}} > \tilde{\text{OUT}}^{\frac{\rho}{k-1}}$, where the last inequality is implied by the definition of B_ϕ^{heavy}. As there are at most N tuples in $R_{\phi(2)}$,

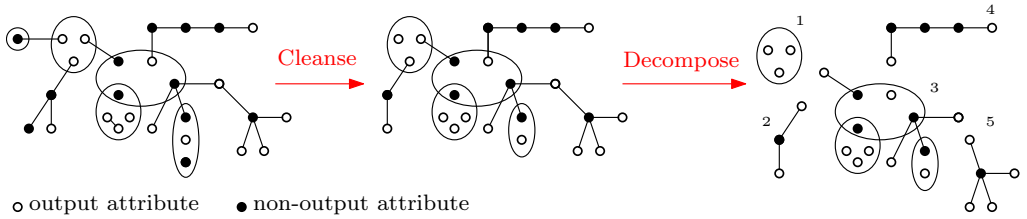

Fig. 2. The left figure shows the hypergraph of an acyclic join-project query Q, where each attribute is represented as a vertex and each relation is represented as a (hyper)-edge. The middle figure shows the residual query by applying the cleanse procedure. The right figure shows the sub-queries of Q defined by existential connectivity. Q_1 is a single relation containing all output attributes. Q_2 is Q_{matrix}. Q_3 is still a complicated join-project query, which is the focus of our algorithm in Section 3.1. Q_4 is Q_{line} with $k = 4$. Q_5 is Q_{star} with $k = 4$. The free-width of Q_1, Q_2, Q_3, Q_4, Q_5 is 1, 2, 6, 2, 4 separately. The free-width of Q is 6.

$\left|B_\phi^{\text{heavy}}\right| = O\left(\frac{N}{\text{OUT}^{\rho/(k-1)}}\right)$. After removing dangling tuples, each distinct value $c \in \text{dom}(\mathbb{C})$ participates in at least $\prod_{i\in\eta} d_{\phi(i)}(b) \geqslant \left(\prod_{j=1}^{k} d_{\phi(i)}(b)\right)^{1/2} \geqslant \left(\prod_{j=2}^{k} d_{\phi(i)}(b)\right)^{1/2} \geqslant \tilde{\text{OUT}}^{\frac{\rho}{2}}$ query results in (1) via some tuple $(b, c) \in R_{\phi,\beta}$, where the first inequality is implied by Lemma 2.3 and the third inequality is implied by the definition of B_ϕ^{heavy}. There are OUT query results in total, so the active domain size of attribute $\mathbb{C}$ is $O\left(\text{OUT}^{1-\frac{\rho}{2}}\right)$. Similarly, each value $a \in \text{dom}(\mathbb{A})$ participates in at least $\prod_{j\in\beta} d_{\phi(j)}(b) \geqslant \left(\prod_{i=2}^{k} d_{\phi(i)}(b)\right)^{1/2} > \tilde{\text{OUT}}^{\frac{\rho}{2}}$ query results in (1) via some tuple $(a, b) \in R_{\phi,\beta}$. As there are OUT query results in total, the active domain size of $\mathbb{A}$ follows. □

Combining all steps by plugging Lemma 2.4 into Lemma 2.1, we obtain (setting $\rho = \frac{2(k-1)}{(5-\omega)(k-1)+2}$):

Theorem 2.5. *For the star query Q_{star} with $k \geqslant 3$ and any instance $\mathcal{R}$ with input size N and output size* OUT*, if an $O(1)$-approximation of* OUT *is known, the query result $Q_{\text{star}}(\mathcal{R})$ can be computed in $\tilde{O}\left(N^{\omega-2} \cdot \text{OUT} + N \cdot \text{OUT}^{\frac{2(k-1)}{(5-\omega)(k-1)+2}}\right)$ time, where k is the number of relations and ω is the exponent of fast square matrix multiplication.*

3 ACYCLIC QUERIES

We finally move to general acyclic join-project queries. Consider an acyclic join-project query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ and an instance $\mathcal{R}$. All dangling tuples (that do not participate in any join result) are removed in $O(N)$ time. We start with two procedures *cleanse* and *decompose*. An attribute $A \in \mathcal{V}$ is *unique* if it only appears in one relation, i.e., $|\{e \in \mathcal{E} : A \in e\}| = 1$, and *joint* otherwise.

Cleanse$(Q, \mathcal{R})$. For a join-aggregate query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ and an instance $\mathcal{R}$, the cleanse procedure iteratively (i) removes a unique non-output attribute $A \in e$ and updates R_e with $\pi_{e-A} R_e$; or (ii) removes a relation $e \in \mathcal{E}$ if there exists another relation $e' \in \mathcal{E}$ with $e \subseteq e'$ and updates $R_{e'}$ with $R_{e'} \ltimes R_e$. We describe this procedure in Algorithm 5. This can be done in $O(N)$ time, where N is the input size of the instance. A join-project query Q is called *cleansed* if no more attribute or relation can be removed by the cleanse procedure, and *non-cleansed* otherwise. In a cleansed query, every unique attribute must be an output attribute.

Decompose$(Q, \mathcal{R})$**.** We define the *existential connectivity* of $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ by modeling it as a graph $G_Q^{\exists}$, where each $e \in \mathcal{E}$ is a vertex, and there is an edge between $e, e' \in \mathcal{E}$ if they share some non-output attribute(s), i.e., $e \cap e' - \mathbf{y} \neq \emptyset$. Let $\mathcal{E}_1, \mathcal{E}_2, \cdots, \mathcal{E}_h \subseteq \mathcal{E}$ be the connected components of $G_Q^{\exists}$. Each $\mathcal{E}_i$ defines a subquery $Q_i = \left(\bigcup_{e\in\mathcal{E}_i} e, \mathcal{E}_i, \bigcup_{e\in\mathcal{E}_i} e \cap \mathbf{y}\right)$ and a sub-instance $\mathcal{R}_i = \{R_j \in \mathcal{R} : e_j \in \mathcal{E}_i\}$. It is not hard to see that if Q is cleansed, each sub-query Q_i is also cleansed, since no more non-output attributes become unique and no more relations become a subset of another relation in the decompose procedure. A critical observation is stated below (intuitively, relations across different sub-queries can only join via output attributes):

Lemma 3.1. $Q(\mathcal{R}) = \bowtie_{i\in[h]} Q_i(\mathcal{R}_i)$.

Then, it suffices to compute the query results for each Q_i, whose output size is bounded by $O(\text{OUT})$, and then compute their full join by invoking the Yannakakis framework. The last step takes at most $O(\text{OUT})$ time. Hence, it suffices to focus on an acyclic join-project query Q that is cleansed and also existentially connected.

Now, we are also ready to introduce the notion of *free-width* (see an example in Figure 2), which plays an important role in capturing the lower bound of semi-ring algorithms (see Theorem D.1):

Definition 3.2 (Free-width). For any acyclic join-project query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, its free-width $\mathsf{freew}(Q)$ is defined as follows:

- If Q is existentially disconnected, $\mathsf{freew}(Q) = \max_{i\in[h]} \mathsf{freew}(Q_i)$, where $Q_1, Q_2, \cdots, Q_h$ are the connected sub-queries of $G_Q^{\exists}$.
- If Q is existentially connected but not cleansed, $\mathsf{freew}(Q) = \mathsf{freew}(Q')$, where Q' is the cleansed version of Q.
- If Q is existentially connected and cleansed,

$$\mathsf{freew}(Q) = \max_{S\subseteq\mathcal{E}:\forall e\in S, e\cap\mathcal{V}_\bullet\neq\emptyset} |S|,$$

where $\mathcal{V}_\bullet$ denotes the set of unique attributes in Q.

All missing proofs in Section 3 are given in Appendix G.

3.1 A Recursive Algorithm

Our recursive algorithm consists of following components:

Base Cases. We consider the following base cases for $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$:

- if $|\mathcal{E}| = 1$, say $\mathcal{E} = \{e\}$, we just return $\pi_{\mathbf{y}} R_e$;
- if $|\mathcal{E}| = 2$, we invoke the algorithm in [1];

General Case. Let $\mathcal{T}$ be a join tree of Q, where each node corresponds to a relation in the input query. A node is *incident* to another node $\mathcal{T}$ if there is an edge between them in $\mathcal{T}$. A node is a *leaf* node if it is only incident to one node, and *internal* otherwise. Let $\mathcal{T}'$ be the residual tree by removing all leaf nodes of $\mathcal{T}$. Let $\mathcal{L} \subseteq \mathcal{E}$ be the set of leaf nodes in $\mathcal{T}'$. We distinguish two cases.

General Case (1): $|\mathcal{L}| = 1$. We denote such a join-project query Q as a *flower*. We separately describe the algorithm in Section 3.2.

General Case (2): $|\mathcal{L}| \geqslant 2$. Our algorithm will iteratively choose a node $e \in \mathcal{L}$ and merge all leaf nodes in $\mathcal{T}$ incident to e. In this way, we keep partitioning the input instance into multiple pieces, while decreasing the size of Q until $|\mathcal{L}| = 1$. We show this algorithm in Section 3.3.

Post-processing. Our recursive algorithm above will reduce Q into $O(1)$ sub-queries, each of them could produce $O(\text{OUT})$ query results. The last step is to remove possible duplicates between them, which can be done via sorting.

3.2 General Case (1): $|\mathcal{L}| = 1$

Let $\mathcal{T}$ be the join tree of a flower query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$, with $\mathcal{L} = \{e_0\}$ as the *core* and $\mathcal{E} - \{e_0\} = \{e_1, e_2, \cdots, e_k\}$ as the set of *petals*. Implied by the property of $\mathcal{T}$, we observe:

- for any pair of distinct relations $e_i, e_j \in \mathcal{E} - \{e_0\}$, $e_i \cap e_j \subseteq e_0$;
- for every relation $e_i \in \mathcal{E} - \{e_0\}$, $e_i - e_0 \neq \emptyset$ and $e_i - e_0 \subseteq \mathbf{y}$;
- $e_0 - e_1 - e_2 - \cdots - e_k \subseteq \mathbf{y}$.

Intuitively, any common attributes shared by any pair of petals also appear in the core. Any attribute that appears in some petal but not the core (as well as all other petals) must be an output attribute. Any attribute that only appears in core must be an output attribute. It is not hard to see that the star query is a degenerated flower (with an arbitrary relation as the core).

A Primitive Query. We will use a slightly generalized query $Q = \pi_{A,C,D} R_1(A, B, D) \bowtie R_2(B, C, D)$ of Q_{matrix}, which can be computed within similar complexity as Lemma 2.1. The intuition is to transform any instance $\mathcal{R}$ for Q into an instance $\mathcal{R}_{\text{matrix}}$ for Q_{matrix} with the same input and output size, such that there is a one-to-one correspondence between $Q(\mathcal{R})$ and $Q_{\text{matrix}}(\mathcal{R}_{\text{matrix}})$. So, it suffices to construct $Q(\mathcal{R})$ from $Q_{\text{matrix}}(\mathcal{R}_{\text{matrix}})$ that can be computed with the algorithm in [1].

Lemma 3.3. *For any instance $\mathcal{R}$ of $Q = \pi_{A,C,D} R_1(A, B, D) \bowtie R_2(B, C, D)$ with input size N and output size* OUT*, the query result $Q(\mathcal{R})$ can be computed in $\widetilde{O}\left(n^{\omega-2} \cdot \text{OUT} + N^{\frac{3-\omega}{4-\omega}} \cdot (n \cdot \text{OUT})^{\frac{1}{4-\omega}} + N\right)$ time, where n is the active domain sizes of $\{B, D\}$.*

Proof of Lemma 3.3. Given any instance $\mathcal{R}$ for $Q = \pi_{A,C,D} R_1(A, B, D) \bowtie R_2(B, C, D)$ of input size N and output size OUT, we transform it into an instance $\mathcal{R}_{\text{matrix}}$ of input size N for $Q_{\text{matrix}} = \pi_{X,Z} R_3(X, Y) \bowtie R_4(Y, Z)$ within $O(N)$ time such that there is a one-to-one correspondence between $Q_{\text{matrix}}(\mathcal{R}_{\text{matrix}})$ and $Q(\mathcal{R})$. We set $\text{dom}(X) = \text{dom}(A) \times \text{dom}(D)$, $\text{dom}(Y) = \text{dom}(B) \times \text{dom}(D)$, and $\text{dom}(Z) = \text{dom}(C)$. For each tuple $(a, b, d) \in R_1$, we add a tuple $((a, d), (b, d))$ to R_3, and for each tuple $(b, c, d) \in R_2$, we add a tuple $((b, d), c)$ to R_4. Next, we show that there is a one-to-one correspondence between $Q_{\text{matrix}}(\mathcal{R}_{\text{matrix}})$ and $Q(\mathcal{R})$;

- For each query result $((a, d), c) \in Q_{\text{matrix}}(\mathcal{R}_{\text{matrix}})$, there must some tuple $(b, d) \in \text{dom}(B) \times \text{dom}(D)$ such that $((a, d), (b, d)) \in R_3$ and $((b, d), c) \in R_4$. From the construction above, we must have $(a, b, d) \in R_1$ and $(b, c, d) \in R_2$. So, $(a, c, d) \in Q(\mathcal{R})$.
- For each query result $(a, c, d) \in Q(\mathcal{R})$, there must some value $b \in \text{dom}(B)$ such that $(a, b, d) \in R_1$ and $(b, c, d) \in R_2$. From the construction above, we must have $((a, d), (b, d)) \in R_3$ and $((b, d), c) \in R_4$. So, $((a, d), c) \in Q_{\text{matrix}}(\mathcal{R}_{\text{matrix}})$.

In this way, it suffices to invoke any algorithm for Q_{matrix} to compute $Q_{\text{matrix}}(\mathcal{R}_{\text{matrix}})$ and then construct $Q(\mathcal{R})$ as follows. For every result $((a, d), c) \in Q_{\text{matrix}}(\mathcal{R}_{\text{matrix}})$ reported, we just report a query result (a, c, d) for $Q(\mathcal{R})$. The construction step takes $O(\text{OUT})$ time, since there are OUT query results in $Q_{\text{matrix}}(\mathcal{R}_{\text{matrix}})$. All complexity results in Lemma 2.1 apply to Q with $X = \{A, D\}$, $Y = \{B, D\}$ and $Z = \{C\}$. □

Now, we are ready to present the algorithm for a flower query. Suppose $\tilde{\text{OUT}}$ is known such that $\tilde{\text{OUT}} \leqslant \text{OUT} \leqslant 2 \cdot \tilde{\text{OUT}}$. Our algorithm consists of four steps. Let $\mathbf{x} = e_0 \cap (e_1 \cup e_2 \cup \cdots \cup e_k)$.

Step 1: Compute data statistics. We compute for each tuple $t \in \pi_{\mathbf{x}} R_0$ its *degree* $d_i(t)$ in relation R_i for each $i \in [k]$, defined as $d_i(t) = \left|\sigma_{e_i \cap e_0 = \pi_{e_i \cap e_0} t} R_i\right|$. We point out an observation below:

Lemma 3.4. *For any tuple $t \in \pi_{\mathbf{x}} R_0$, $\prod_{i=1}^{k} d_i(t) \leqslant \mathrm{OUT}$.*

Proof of Lemma 3.4. Consider an arbitrary tuple $t \in \pi_{\mathbf{x}} R_0$. Recall that $e_i - e_0 \subseteq \mathbf{y}$ and $(e_i - e_0) \cap (e_j - e_0) = \varnothing$ for every distinct pair $i, j \in [k]$. The Cartesian product of

$$\left(\pi_{e_0 - \mathbf{x}} \left(R_0 \ltimes t\right)\right) \times \left\{ \times_{i \in [k]} \left(\pi_{e_i - e_0} \left(R_i \ltimes t\right)\right)\right\}$$

must be a subset of the projection of final query results onto output attributes $\mathbf{y} - e_0$. Hence,

$$\begin{aligned} \mathrm{OUT} &\geqslant \left|\left(\pi_{e_0 - \mathbf{x}} \left(R_0 \ltimes t\right)\right) \times \left\{ \times_{i \in [k]} \left(\pi_{e_i - e_0} \left(R_i \ltimes t\right)\right)\right\}\right| \\ &= \left|\pi_{e_0 - \mathbf{x}} \left(R_0 \ltimes t\right)\right| \cdot \prod_{i=1}^{k} \left|\pi_{e_i - e_0} \left(R_i(e_i) \ltimes t\right)\right| \geqslant \prod_{i=1}^{k} d_i(t). \end{aligned}$$

□

Step 2: Reduce Q. We define a permutation $\phi_t : [k] \to [k]$ such that $d_{\phi_t(i)}(t) \geqslant d_{\phi_t(j)}(t)$ for any $1 \leqslant i \leqslant j \leqslant k$. We can determine ϕ_t for all t's by sorting all values in $d_i(t)$, breaking ties. There are at most $k!$ number of permutations in total. Each permutation ϕ over $[k]$ defines a subset of tuples in $\pi_{\mathbf{x}} R_0$ as $\mathbf{x}_\phi = \{t \in \pi_{\mathbf{x}} R_0 : \phi_t = \phi\}$. Given a parameter $\gamma \in (0, \frac{1}{2}]$, whose values will be determined later, we further divide each $\mathbf{x}_\phi$ into two subsets:

$$\begin{aligned} \mathbf{x}_\phi^{\mathsf{light}} &= \left\{ t \in \mathbf{x}_\phi : d_{\phi(1)}(t) > \tilde{\mathrm{OUT}}^\gamma \right\} \\ \mathbf{x}_\phi^{\mathsf{heavy}} &= \left\{ t \in \mathbf{x}_\phi : d_{\phi(1)}(t) \leqslant \tilde{\mathrm{OUT}}^\gamma \right\} \end{aligned}$$

Then, we can reduce Q into the following at most $2 \cdot k!$ sub-queries:

$$Q_\phi^? = \pi_{\mathbf{y}} \left(R_0 \ltimes \mathbf{x}_\phi^?\right) \bowtie \left(\bowtie_{i \in [k]} R_i\right),$$

where ? can be heavy or light.

Step 3: Compute Q_ϕ^{light}. We first compute the join between $\left(R_0 \ltimes \mathbf{x}_\phi^{\mathsf{light}}\right), R_{\phi(2)}, \cdots, R_{\phi(k)}$ and then project out non-output attributes that do not appear in $e_{\phi(1)}$:

$$R_{\mathsf{new}} = \pi_{\mathbf{y} \cup (e_0 \cap e_{\phi(1)})} \left\{ \left(R_0 \ltimes \mathbf{x}_\phi^{\mathsf{light}}\right) \bowtie \left(\bowtie_{i \in \{2,3,\cdots,k\}} R_{\phi(i)}\right)\right\}$$

This way, we can reduce Q_ϕ^{light} to the following query:

$$\pi_{\mathbb{A},\mathbb{C},\mathbb{D}} \left(R_{\mathsf{new}}(\mathbb{A}, \mathbb{B}, \mathbb{D}) \bowtie R_{\phi(1)}(\mathbb{B}, \mathbb{D}, \mathbb{C})\right) \tag{2}$$

where $\mathbb{A} = \mathbf{y} - e_{\phi(1)}, \mathbb{B} = e_0 \cap e_{\phi(1)} - \mathbf{y}, \mathbb{C} = \mathbf{y} \cap e_{\phi(1)} - e_0$ and $\mathbb{D} = e_0 \cap e_{\phi(1)} \cap \mathbf{y}$. We just apply the algorithm in Lemma 3.3.

Step 4: Compute Q_ϕ^{heavy}. Let (η, β) be a partition of $[k]$, where $\eta = \{i \in [k] : i \text{ is odd}\}$ and $\beta = [k] - \eta$. We materialize the following two intermediate joins by the Yannakakis framework:

$$\begin{aligned} R_{\phi,\eta}\left(\mathbb{A}, \mathbb{B}, \mathbb{D}\right) &= \left(R_0 \ltimes \mathbf{x}_\phi^{\mathsf{heavy}}\right) \bowtie \left(\bowtie_{i \in \eta} R_{\phi(i)}\right) \\ R_{\phi,\beta}\left(\mathbb{B}, \mathbb{C}, \mathbb{D}\right) &= \left(R_0 \ltimes \mathbf{x}_\phi^{\mathsf{heavy}}\right) \bowtie \left(\bowtie_{j \in \beta} R_{\phi(j)}\right) \end{aligned}$$

where $\mathbb{A} = \bigcup_{i \in \eta} e_{\phi(i)} - e_0, \mathbb{B} = e_0 - \mathbf{y}, \mathbb{C} = \bigcup_{j \in \beta} e_{\phi(j)} - e_0$ and $\mathbb{D} = e_0 \cap \mathbf{y}$. Then, we reduce Q_ϕ^{heavy} to the following query:

$$\pi_{\mathbb{A},\mathbb{C},\mathbb{D}} \left(R_{\phi,\eta}\left(\mathbb{A}, \mathbb{B}, \mathbb{D}\right) \bowtie R_{\phi,\beta}\left(\mathbb{B}, \mathbb{C}, \mathbb{D}\right)\right). \tag{3}$$

We just apply the algorithm in Lemma 3.3.

Step 5: Remove Duplicates. From above, each of the at most $2 \cdot k!$ subqueries could produce $O(\text{OUT})$ query results. The last step is to remove possible duplicates between them via sorting.

3.3 General Case (2): $|\mathcal{L}| \geqslant 2$

For every node $e \in \mathcal{L}$, let e_{inc} be the unique node incident to e in $\mathcal{T}'$. All other nodes incident to e in $\mathcal{T}$ must be leaf nodes of $\mathcal{T}$, which are denoted as $\mathcal{E}_e \subseteq \mathcal{E}$. Let $\mathbf{y}_e \subseteq \mathbf{y}$ be the set of output attributes that only appears in some relation in $\mathcal{E}_e$, i.e., $\mathbf{y}_e = \bigcup_{e' \in \mathcal{E}_e} e' \cap \mathbf{y} - e$. As Q is cleansed, for every leaf node $e' \in \mathcal{E}_e$, there is $e' - e \subseteq \mathbf{y}$.

Step 1: Computing data statistics. Consider an arbitrary node $e \in \mathcal{L}$ with a set of leaf nodes $\mathcal{E}_e$ incident to it in $\mathcal{T}$. For each tuple $t \in R_e$, we define $\triangle(t)$ as the number of distinct tuples in $\text{dom}(\mathbf{y}_e)$ that can be joined with t, i.e., $\triangle(t) = \prod_{e' \in \mathcal{E}_e} |R_{e'} \ltimes t|$.

Step 2: Reduce Q. For an arbitrary node $e \in \mathcal{L}$, a tuple $t \in R_e$ is *heavy* if $\triangle(t) > \sqrt{2 \cdot \tilde{\text{OUT}}}$, and *light* otherwise. Let $R_e^{\text{heavy}}, R_e^{\text{light}}$ be the heavy and light tuples in R_e separately. We can reduce the original query into $O\left(2^{|\mathcal{L}|}\right)$ subqueries:

$$Q^{\chi} = \pi_{\mathbf{y}} \left(\ltimes_{e \in \mathcal{L}} R_e^{?} \right) \bowtie \left(\ltimes_{e' \in \mathcal{E} - \mathcal{L}} R_{e'} \right)$$

where $? \in \{\text{heavy}, \text{light}\}$ can be different for different nodes $e \in \mathcal{L}$, and $\chi \in \{\text{heavy}, \text{light}\}^{\mathcal{L}}$ is simply the collection of "labels" for all nodes in $\mathcal{L}$. Below, we point out a critical observation (therefore we only need to consider subqueries with at most one heavy relation in $\mathcal{L}$):

Lemma 3.5. *If $?$ is heavy for at least two distinct nodes $e, e' \in \mathcal{L}$, then $Q^{\chi} = \emptyset$.*

Proof of Lemma 3.5. By contradiction, suppose there exists a pair of nodes $e, e' \in \mathcal{L}$ such that $R_e, R_{e'}$ are heavy, and the query results of Q^{ξ} is not emtpy. Then, it is always feasible to find a join result t of all relations involved. Let $t \in R_e^{\text{heavy}}, t \in R_{e'}^{\text{heavy}}$ be the projection of t onto e, e' separately. Note that $\Delta(t) > \sqrt{2 \cdot \tilde{\text{OUT}}}$ and $\Delta(t') > \sqrt{2 \cdot \tilde{\text{OUT}}}$. And, $\mathbf{y}_e \cap \mathbf{y}_{e'} = \emptyset$. This way, the Cartesian product $(\ltimes_{e'' \in \mathcal{E}_e} R_{e''} \ltimes t) \times \left(\ltimes_{e'' \in \mathcal{E}_{e'}} R_{e''} \ltimes t\right)$ is a subset of the projection of final query results $Q(\mathcal{R})$ onto $\mathbf{y}_e \cup \mathbf{y}_{e'}$, after all dangling tuples are removed. Hence,

$$\text{OUT} < 2 \cdot \tilde{\text{OUT}} = \sqrt{2 \cdot \tilde{\text{OUT}}} \cdot \sqrt{2 \cdot \tilde{\text{OUT}}} < \Delta(t) \cdot \Delta(t') \leqslant \text{OUT},$$

coming to a contradiction. □

Step 3: Compute Q_{ϕ}^{χ}. As $|\mathcal{L}| \geqslant 2$, together with Lemma 3.5, there is at least one light relation in $\mathcal{L}$, say e. We compute the full join results of R_e with relations in $\mathcal{E}_e$, and project out non-output attributes that do not appear in e_{inc}, i.e.,

$$R_{\text{new}} = \pi_{\mathbf{y}_e \cup (e \cap \mathbf{y}) \cup (e \cap e_{\text{inc}} - \mathbf{y})} \left(R_e^{\text{light}} \bowtie \left(\ltimes_{e' \in \mathcal{E}_e} R_{e'} \right) \right).$$

Recall that $\mathbf{y}_e$ is the set of output attributes only appearing in some relation in $\mathcal{E}_e$, $e \cap \mathbf{y}$ is the set of output attributes appearing in e, and $e \cap e_{\text{inc}} - \mathbf{y}$ is the set of non-output attributes appearing in both e, e_{inc}. We update Q_{ϕ}^{χ} by removing all relations in $\mathcal{E}_e$, replacing e with $\text{new} = \mathbf{y}_e \cup (e \cap \mathbf{y}) \cup (e \cap e_{\text{inc}} - \mathbf{y})$, and updating R_e with R_{new}. At last, we invoke the whole algorithm recursively.

3.4 Analysis

General case (1): $|\mathcal{L}| = 1$. We consider a more general scenario for an instance $\mathcal{R} = \{R_0, R_1, \cdots, R_k\}$ with *core size* $N = |R_0|$, *petal size* $N' = \max_{i \in [k]} |R_i|$ for $N' = O(N \cdot \sqrt{\text{OUT}})$, and output size OUT. We start with two important observations:

Lemma 3.6. *The instance in (2) satisfies the following constraints:*

- $|R_{\text{new}}| = O\left(N \cdot \text{OUT}^{1-\gamma}\right)$ *and* $|R_{\phi(1)}| \leqslant N' = O(N \cdot \sqrt{\text{OUT}})$;
- *the active domain size of* $\mathbb{B} \cup \mathbb{D}$ *is* $O(N)$;

Proof of Lemma 3.6. The active domain size of $\mathbb{B} \cup \mathbb{D}$ is $O(N)$, since $\mathbb{B} \cup \mathbb{D} \subseteq e_0$ and $|R_0| = N$. We observe that $\prod_{i=2}^{k} d_{\phi(i)}(\pi_{\mathbf{x}} t) \leqslant \tilde{\text{OUT}}^{1-\gamma}$ for every tuple $t \in R_0 \ltimes \mathbf{x}_\phi^{\text{light}}$, implied by Lemma 3.4 and the definition of $\mathbf{x}_\phi^{\text{light}}$. As there are at most N tuples in R_0, and each tuple $t \in R_0 \ltimes \mathbf{x}_\phi^{\text{light}}$ can be joined with $\prod_{i=2}^{k} d_{\phi(i)}(\pi_{\mathbf{x}} t) \leqslant \tilde{\text{OUT}}^{1-\gamma}$ tuples in the join $\left(\bowtie_{i \in \{2,3,\cdots,k\}} R_{\phi(i)}\right)$. So, the number of intermediate join results materialized for computing R_{new}, as well as the size of R_{new}, is $O\left(N \cdot \text{OUT}^{1-\gamma}\right)$. □

Lemma 3.7. *The instance in (3) satisfies the following constraints:*

- $\left|R_{\phi,\eta}\right| = O\left(N \cdot \text{OUT}^{\frac{1+\gamma}{2}}\right)$ *and* $\left|R_{\phi,\beta}\right| = O\left(N \cdot \sqrt{\text{OUT}}\right)$;
- *the active domain size of* $\mathbb{B} \cup \mathbb{D}$ *is* $O(N)$.

Proof of Lemma 3.7. We first observe that the active domain size of $\mathbb{B} \cup \mathbb{D}$ is $O(N)$, since $\mathbb{B} \cup \mathbb{D} \subseteq e_0$ and $|R_0| \leqslant N$. Consider any tuple $t \in R_0 \ltimes \mathbf{x}_\phi^{\text{heavy}}$. It participates in at most $\prod_{j \in \eta} d_{\phi(j)}(\pi_{\mathbf{x}} t) \leqslant \left(d_{\phi(1)}(\pi_{\mathbf{x}} t) \cdot \prod_{j=1}^{k} d_{\phi(j)}(\pi_{\mathbf{x}} t)\right)^{\frac{1}{2}} \leqslant \sqrt{\tilde{\text{OUT}}^{\gamma} \cdot \text{OUT}}$ join results of $\left(\bowtie_{j \in \eta} R_{\phi(j)}\right)$, implied by Lemma 2.3 and Lemma 3.4, hence $\left|R_{\phi,\eta}\right|$ is $O\left(N \cdot \text{OUT}^{\frac{1+\gamma}{2}}\right)$. Similarly, it participates in at most $\prod_{j \in \beta} d_{\phi(j)}(\pi_{\mathbf{x}} t) \leqslant \left(\prod_{j=1}^{k} d_{\phi(j)}(\pi_{\mathbf{x}} t)\right)^{\frac{1}{2}} \leqslant \sqrt{\text{OUT}}$ join results of $\bowtie_{i \in \beta} R_{\phi(i)}$, hence $|R_{\phi,\beta}|$ is $O\left(N \cdot \sqrt{\text{OUT}}\right)$. □

Now, we are ready to analyze the cost of each step separately. Steps 1 and 2 can be done within $O(N + N') = O(N \cdot \sqrt{\text{OUT}})$ time. Step 5 can be done within $\tilde{O}(\text{OUT})$ time. Plugging Lemma 3.6 into Lemma 3.3, Step 3 takes

$$\tilde{O}\left(N \cdot \text{OUT}^{(1-\gamma)\cdot\frac{3-\omega}{4-\omega}+\frac{1}{4-\omega}} + N^{\omega-2} \cdot \text{OUT} + N \cdot \text{OUT}^{1-\gamma}\right)$$

time. Plugging Lemma 3.7 into Lemma 3.3, Step 4 takes

$$\tilde{O}\left(N \cdot \text{OUT}^{\frac{1+\gamma}{2}\cdot\frac{3-\omega}{4-\omega}+\frac{1}{4-\omega}} + N^{\omega-2} \cdot \text{OUT} + N \cdot \text{OUT}^{\frac{1+\gamma}{2}}\right)$$

time. Combining all steps, we obtain (by setting $\gamma = \frac{1}{3}$):

Lemma 3.8. *For a flower query* Q *and an arbitrary instance* $\mathcal{R}$ *with core size* N*, output size* OUT*, and petal size* $N' = O\left(N \cdot \sqrt{\text{OUT}}\right)$*, if an* $O(1)$*-approximation of* OUT *is known, the query results* $Q(\mathcal{R})$ *can be computed in*

$$\tilde{O}\left(N \cdot \text{OUT}^{\frac{2}{3}\cdot\frac{3-\omega}{4-\omega}+\frac{1}{4-\omega}} + N^{\omega-2} \cdot \text{OUT}\right)$$

time, where ω *is the exponent of fast square matrix multiplication.*

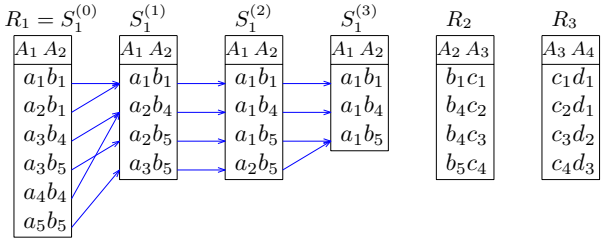

Fig. 3. Compress relation R_1 for the line query $\pi_{A_1,A_4} R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4)$. The arrow from t to t' indicates that t' is constructed from t at line 10 of Algorithm 1. After obtaining $S_1^{(3)}$, the residual query Q' is defined as $\pi_{A_4}\left(R_2(A_2, A_3) \ltimes S_1^{(3)}\right) \bowtie R_3(A_3, A_4)$. When invoking Algorithm 1 recursively, we apply the CLEANSE procedure on Q', which essentially removes A_2 and e_2 iteratively. We are left with R_3, which falls into the base case at line 2. It is easy to see Lemma 4.1 on this example. Moreover, $|\mathcal{J}^{(3)}| = 3$, $|\mathcal{J}^{(2)}| = 4$, $|\mathcal{J}^{(1)}| = 5$, and $|\mathcal{J}^{(0)}| = 8$.

General case (2): $|\mathcal{L}| \geqslant 2$. We analyze the cost of reducing a join-project query from general case (2) to general case (1). Suppose we are given an instance $\mathcal{R}$ with input size N and output size OUT.

Step 1 and 2 can be done in $O(N)$ time. In Step 3, the size of R_{new} can be bounded by $O(N \cdot \sqrt{\text{OUT}})$, since there are at most N tuples in R_e and each tuple $t \in R_e^{\text{light}}$ can be joined with at most $\triangle(t) \leqslant \sqrt{\widetilde{\text{OUT}}}$ tuples in $\bowtie_{e' \in \mathcal{E}_e} R_{e'}$. It removes all leaf nodes in $\mathcal{E}_e$ as well as e, and adds a new leaf node R_{new} of size $O(N \cdot \sqrt{\text{OUT}})$.

Moreover, we point out an invariant of this recursive algorithm that the size of internal relations is always bounded by $O(N)$ and the size of leaf relations is always bounded by $O\left(N \cdot \sqrt{\text{OUT}}\right)$.

The Recursive Algorithm. From our analysis above, an input join-project query Q and an instance $\mathcal{R}$ of input size N and output size OUT, is reduced to a constant number of instances for flower queries, each of core size $O(N)$, petal size $O\left(N \cdot \sqrt{\text{OUT}}\right)$, and output size $O(\text{OUT})$. Plugging into Lemma 3.8, we obtain:

THEOREM 3.9. *For any acyclic but non-free-connex join-project query Q and an instance $\mathcal{R}$ with input size N and output size* OUT*, if an $O(1)$-approximation of* OUT *is known, the query results $Q(\mathcal{R})$ can be computed within*

$$\widetilde{O}\left(N \cdot \text{OUT}^{\frac{2}{3} \cdot \frac{3-\omega}{4-\omega} + \frac{1}{4-\omega}} + N^{\omega-2} \cdot \text{OUT}\right)$$

time, where ω is the exponent of fast square matrix multiplication.

4 ESTIMATE OUT ON-THE-FLY

So far, we assume that an $O(1)$-approximation of OUT is known in advance. We now remove this assumption by showing our framework in Algorithm 1. We note that this is definitely not the only way to approximate OUT. As long as an $O(1)$-approximation of OUT is obtained, it can be fed into our output-aware algorithms presented in Sections 2-3.

In Algorithm 1, we first cleanse the input query $Q = (\mathcal{V}, \mathcal{E}, \mathbf{y})$ and instance $\mathcal{R}$ (line 1). If $\mathcal{E}$ only contains one relation, say $\mathcal{E} = \{e\}$, we just return the projection of R_e onto output attributes $\mathbf{y}$ (line 2). In general, we always start with a relation $e \in \mathcal{E}$, such that after removing its *unique* attributes (those only appear in e), e becomes a subset of another relation e' (line 3). As Q is already cleansed, all unique attributes in e are output attributes.

(line 4-13) We iteratively compress R_e by halving the active domain size of $e - e'$, until there is a single value left (line 13). The compression works as follows. Suppose all values in $\pi_{e-e'} R_e$

Algorithm 1: $\textsc{Acyclic}(\mathcal{Q} = (\mathcal{V}, \mathcal{E}, \mathbf{y}), \mathcal{R})$

1 $(\mathcal{Q}, \mathcal{R}) \leftarrow \textsc{Cleanse}(\mathcal{Q}, \mathcal{R})$; ▸ Algorithm 5;
2 **if** $|\mathcal{E}| = 1$, *say* $\mathcal{E} = \{e\}$ **then return** $\pi_{\mathbf{y}} R_e$;
3 Find $e \in \mathcal{E}$ such that there exists some $e' \in \mathcal{E}$ with $e - U \subseteq e'$, where U is the set of unique attributes in e, i.e., those only appear in e;
4 Relabel tuples in $\pi_{e-e'} R_e$ as $a_1, a_2, a_3, \cdots$;
5 $S_e^{(0)} \leftarrow R_e$, $i \leftarrow 1$;
6 **while true do**
7 $\quad S_e^{(i)} \leftarrow \emptyset$;
8 $\quad$ **foreach** $t \in S_e^{(i-1)}$ **do**
9 $\quad\quad$ Suppose $\pi_{e-e'} t = a_j$;
10 $\quad\quad t' \leftarrow$ a tuple with $\pi_{e-e'} t' = a_{\lfloor \frac{j+1}{2} \rfloor}$ and $\pi_A t' = \pi_A t$ for any attribute $A \in e \cap e'$;
11 $\quad\quad S_e^{(i)} \leftarrow S_e^{(i)} \cup \{t'\}$;
12 $\quad$ **if** $\left|\pi_{e-e'} S_e^{(i)}\right| = 1$ **then break**;
13 $\quad i \leftarrow i + 1$;
14 $R_{e'} \leftarrow R_{e'} \ltimes S_e^{(i)}$;
15 $\mathcal{Q}' \leftarrow (\mathcal{V} - (e - e'), \mathcal{E} - \{e\}, \mathbf{y} - e)$;
16 $\mathcal{J}^{(i)} \leftarrow \left(\pi_{e-e'} S_e^{(i)}\right) \times \textsc{Acyclic}\left(\mathcal{Q}', \mathcal{R} - \{R_e\}\right)$; ▸ Algorithm 1;
17 **while** $i > 0$ **do**
18 $\quad \mathcal{J}^{(i-1)} \leftarrow$ Compute $\mathcal{Q}$ over $\mathcal{R} - \{R_e\} \cup \left\{S_e^{(i-1)}\right\}$ using $2 \cdot |\mathcal{J}^{(i)}|$ as the approximated output size; ▸ Algorithms in Section 2 or Section 3;
19 $\quad i \leftarrow i - 1$;
20 **return** $\mathcal{J}^{(0)}$;

are labeled as $a_1, a_2, a_3, \cdots$. Initially, set $S_e^{(0)} = R_e$. Suppose $S^{(h)}$ is created recursively. We next compress $S_e^{(h)}$ into $S_e^{(h+1)}$ as follows. For every tuple $t \in S_e^{(h)}$, with $\pi_{e-e'} t = a_j$, we add a tuple t' to $S_e^{(h+1)}$ such that t' shares the same value as t on every attribute in $e \cap e'$, and $\pi_{e-e'} t' = a_{\lfloor \frac{j+1}{2} \rfloor}$. This way, for any pair of tuples $t_1, t_2 \in S_e^{(h)}$ such that $\pi_{e \cap e'} t_1 = \pi_{e \cap e'} t_2$, $\pi_{e-e'} t_1 = a_j$ and $\pi_{e-e'} t_2 = a_{j+1}$ for some odd j, they will be "compressed" into one tuple. Then, we continue applying the procedure to compress $S_e^{(h+1)}$.

(line 14-20) Let $S_e^{(i)}$ be the compressed relation such that all tuples share the same value in $e - e'$. We update $R_{e'}$ with $R_{e'} \ltimes S_e^{(i)}$ (line 14), remove e from $\mathcal{E}$ as well as all unique attributes in e from $\mathcal{V}$ (line 15), and handle the residual query by invoking the whole algorithm recursively (line 16). Define $\mathcal{R}^{(i)} = (\mathcal{R} - \{R_e\}) \cup \left\{S_e^{(i)}\right\}$. Let $\mathcal{J}^{(i)}$ be the results returned for $\mathcal{R}^{(i)}$. As we will prove in Lemma 4.1, $\left|\mathcal{J}^{(i)}\right|$ is a 2-approximation of the output size of $\mathcal{Q}\left(\mathcal{R}^{(i-1)}\right)$. Hence, we iteratively invoke our output-sensitive algorithms in Section 2 (if $\mathcal{Q}$ is $\mathcal{Q}_{\text{star}}$) or Section 3 (if $\mathcal{Q}$ is a general acyclic join-project query) for computing $\mathcal{R}^{(i-1)}, \mathcal{R}^{(i-2)}, \cdots, \mathcal{R}^{(0)}$. Note that $\mathcal{R}^{(0)} = \mathcal{R}$. At last, we return $\mathcal{Q}\left(\mathcal{R}^{(0)}\right)$, i.e., $\mathcal{J}^{(0)}$. See an example in Figure 3.

Correctness. We next show the correctness of this algorithm by establishing the following lemma:

Lemma 4.1. $|\mathcal{J}^{(i)}| \leqslant |\mathcal{J}^{(i-1)}| \leqslant 2 \cdot |\mathcal{J}^{(i)}|$.

Proof. From the construction of $S_e^{(i)}$, it is not hard to see that if there exists a query result $t' \in Q\left(\mathcal{R}^{(i-1)}\right)$ with $\pi_{e-e'}t' = a_{2j-1}$ or $\pi_{e-e'}t = a_{2j}$, there exists a query result $t \in Q\left(\mathcal{R}^{(i)}\right)$ such that $\pi_{e\cap e'}t = \pi_{e\cap e'}t'$ and $\pi_{e-e'}t = a_j$. So, $|\mathcal{J}^{(i)}| \geqslant \frac{1}{2} \cdot |\mathcal{J}^{(i-1)}|$. On the other hand, if there exists a query result $t \in Q\left(\mathcal{R}^{(i)}\right)$ with $\pi_{e-e'}t = a_j$, there must exist a query result $t' \in Q\left(\mathcal{R}^{(i-1)}\right)$ such that $\pi_{e\cap e'}t = \pi_{e-e'}t'$, and $\pi_{e-e'}t' = a_{2j-1}$ or $\pi_{e-e'}t = a_{2j}$. So, $|\mathcal{J}^{(i)}| \leqslant |\mathcal{J}^{(i-1)}|$. Together, we complete the proof. □

With Lemma 4.1, the query results $Q\left(\mathcal{R}^{(i-1)}\right)$ can be computed by our algorithms in Section 2-Section 3 at line 18. Moreover, $\mathcal{J}^{(0)} = Q(\mathcal{R})$. Hence, for any acyclic join-project query Q and instance $\mathcal{R}$, the query results $Q(\mathcal{R})$ can be computed by Algorithm 1.

Analysis. At last, we come to the time complexity of Algorithm 1. The while-loop at line 6-13 takes $O(N \log N)$ time. Note that each iteration takes $O(N)$ time to construct $S_e^{(i)}$. After $O(\log N)$ iterations, the active domain size of $e - e'$ will decrease to 1, since the active domain size of the input instance is $O(N)$ after all dangling tuples are removed. Moreover, while-loop at line 17-19 takes $O\left(f_Q(N, \mathrm{OUT}) \cdot \log N\right)$ time in total, if an instance for Q with input size N and output size OUT can be computed in $f_Q(N, \mathrm{OUT})$ time, when an $O(1)$-approximation of OUT is known.

Theorem 4.2. *For an arbitrary acyclic join-project query $Q = (\mathcal{V}, \mathcal{E}, \mathrm{y})$ and an instance $\mathcal{R}$ of input size N and output size* OUT*, if there is an algorithm that can compute $Q(\mathcal{R})$ within $f_Q(N, \mathrm{OUT})$ time when an $O(1)$-approximation of* OUT *is known, Algorithm 1 can compute $Q(\mathcal{R})$ within $O\left(N \cdot \log N + \mathrm{OUT} + f_Q(N, \mathrm{OUT}) \cdot \log N\right)$ time.*

Proof of Theorem 4.2. First, line 1-5 take $O(N)$ time and line 14-15 takes $O(1)$ time. By hypothesis, assume Acyclic $(Q', \mathcal{R} - \{R_e\})$ takes $O\left(N \log N + \mathrm{OUT} + f_{Q'}(N, \mathrm{OUT}) \cdot (|\mathcal{E}| - 1) \cdot \log N\right)$ time. The while-loop at line 6-13 takes $O(N \log N)$ time. Each iteration takes $O(N)$ time to construct $S_e^{(i)}$. There are at most N values in $\pi_{e-e'}R_e$, and each iteration would decrease this number by a factor of 2. After $O(\log N)$ iterations, line 12 will be triggered. Moreover, when the while-loop at line 6-13 is broken, we have $i = O(\log N)$. The while-loop at line 17-19 takes $O\left(f(N, \mathrm{OUT}) \cdot \log N\right)$ time. As $f_{Q'}(N, \mathrm{OUT}) \leqslant f_Q(N, \mathrm{OUT})$, Acyclic $(Q, \mathcal{R})$ incurs a cost of

$$N \log N + \mathrm{OUT} + f_Q(N, \mathrm{OUT}) \cdot |\mathcal{E}| \cdot \log N = O\left(N \log N + \mathrm{OUT} + f_Q(N, \mathrm{OUT}) \cdot \log N\right). \quad \square$$

Plugging Theorem 2.5 and Theorem 3.9 into Theorem 4.2 separately, we obtain:

Theorem 4.3. *For the star query Q_{star} with $k \geqslant 3$ and any instance $\mathcal{R}$ with input size N and output size* OUT*, the query result $Q_{\mathrm{star}}(\mathcal{R})$ can be computed within $\widetilde{O}\left(N^{\omega-2} \cdot \mathrm{OUT} + N \cdot \mathrm{OUT}^{\frac{2(k-1)}{(5-\omega)(k-1)+2}}\right)$ time, where k is the number of relations and ω is the exponent of fast square matrix multiplication.*

Theorem 4.4. *For any acyclic but non-free-connex join-project query Q and an instance $\mathcal{R}$ with input size N and output size* OUT*, the query results $Q(\mathcal{R})$ can be computed within*

$$\widetilde{O}\left(N \cdot \mathrm{OUT}^{\frac{2}{3} \cdot \frac{3-\omega}{4-\omega} + \frac{1}{4-\omega}} + N^{\omega-2} \cdot \mathrm{OUT}\right)$$

time, where ω is the exponent of fast square matrix multiplication.

5 IMPLICATIONS TO CYCLIC QUERIES

At last, we turn to cyclic join-project queries. Combining our algorithm for acyclic queries with the generalized hypertree decomposition techniques, worst-case optimal join algorithms [15, 19, 21, 26] and our new algorithm for acyclic join-project queries, we obtain:

Algorithm 2: Cyclic$(\mathcal{Q}, \mathcal{R})$

$(\mathcal{Q}_1, \mathcal{R}_1), (\mathcal{Q}_2, \mathcal{R}_2), \cdots, (\mathcal{Q}_\ell, \mathcal{R}_\ell) \leftarrow$ a set of decomposed sub-instances for $(\mathcal{Q}, \mathcal{R})$ by [19];
foreach $i \in [\ell]$ **do** $\mathcal{J}_i \leftarrow$ Acyclic$(\mathcal{Q}_i, \mathcal{R}_i)$; ▸ Algorithm 1;
return $\mathcal{J}_1 \cup \mathcal{J}_2 \cup \cdots \cup \mathcal{J}_\ell$;

Theorem 5.1. *For any non-free-connex join-project query $\mathcal{Q}$ and an instance $\mathcal{R}$ of input size N and output size* OUT*, the query results $\mathcal{Q}(\mathcal{R})$ can be computed within*

$$\widetilde{O}\left(N^{\mathsf{subw}} \cdot \mathrm{OUT}^{\frac{2}{3}\cdot\frac{3-\omega}{4-\omega}+\frac{1}{4-\omega}} + N^{(\omega-2)\cdot\mathsf{subw}} \cdot \mathrm{OUT}\right)$$

time, where subw *is the sub-modular width of $\mathcal{Q}$ and ω is the exponent of fast square matrix multiplication.*

The complete procedure is illustrated in Algorithm 2. Consider an arbitrary cyclic join-project query $\mathcal{Q}$ and an instance $\mathcal{R}$ of input size N and output size OUT. We first apply the hypertree decomposition techniques proposed in [19] to decompose the input pair $(\mathcal{Q}, \mathcal{R})$ into a set of input pairs $(\mathcal{Q}_1, \mathcal{R}_1)$,$(\mathcal{Q}_2, \mathcal{R}_2)$, $\cdots$,$(\mathcal{Q}_\ell, \mathcal{R}_\ell)$ for some constant ℓ, such that $\mathcal{Q}(\mathcal{R}) = \bigcup_{j\in[\ell]} \mathcal{Q}_j(\mathcal{R}_j)$ and for any $j \in [\ell]$, the following properties hold: (i) $\mathcal{Q}_j$ is an acyclic join-project query; (ii) the input size of $\mathcal{R}_j$ is $O\left(N^{\mathsf{subw}}\right)$; (iii) the output size of $\mathcal{Q}_j(\mathcal{R}_j)$ is $O(\mathrm{OUT})$. For each $j \in [\ell]$, we simply invoke Algorithm 1 to compute $\mathcal{Q}_j(\mathcal{R}_j)$ separately. At last, we remove duplicates over all sub-instances.

For simplicity, let $\mathrm{OUT}_j = |\mathcal{Q}_j(\mathcal{R}_j)|$ be the output size of $\mathcal{Q}_j$ over $\mathcal{R}_j$. Implied by Theorem 4.4, the $\mathcal{Q}_j(\mathcal{R}_j)$ can be computed within

$$\widetilde{O}\left(N^{\mathsf{subw}} \cdot \mathrm{OUT}_j^{\frac{2}{3}\cdot\frac{3-\omega}{4-\omega}+\frac{1}{4-\omega}} + N^{(\omega-2)\cdot\mathsf{subw}} \cdot \mathrm{OUT}_i\right)$$

time, where subw is the sub-modular width of $\mathcal{Q}$ and ω is the exponent of fast square matrix multiplication. As $\mathrm{OUT}_i \leqslant \mathrm{OUT}$, this complexity is also bounded by

$$\widetilde{O}\left(N^{\mathsf{subw}} \cdot \mathrm{OUT}^{\frac{2}{3}\cdot\frac{3-\omega}{4-\omega}+\frac{1}{4-\omega}} + N^{(\omega-2)\cdot\mathsf{subw}} \cdot \mathrm{OUT}\right).$$

As there are $O(1)$ sub-instances, and each of them generates at most $O(\mathrm{OUT})$ query results, the last step of removing duplicates can be done within $\widetilde{O}(\mathrm{OUT})$ time. Putting everything together, we obtain Theorem 5.1.

Example 5.2. For a join-project query $\pi_{A_1,A_2,A_3} R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4) \bowtie R_4(A_1, A_4)$ with $\mathsf{subw} = \frac{3}{2}$ and $\mathsf{fc\text{-}subw} = 2$, our new algorithm can compute it within $\widetilde{O}\left(N^{\frac{3}{2}} \cdot \mathrm{OUT}^{\frac{5}{6}} + \mathrm{OUT}\right)$ time, which has improved the previous result $O\left(\min\left\{N^{\frac{3}{2}} \cdot \mathrm{OUT}, N^2\right\}\right)$ by (almost) a factor of $\mathrm{OUT}^{\frac{1}{6}}$ if $\mathrm{OUT} \leqslant N^{\frac{1}{2}}$ and by a factor of $\frac{N^{1/2}}{\mathrm{OUT}^{5/6}}$ if $N^{\frac{1}{2}} < \mathrm{OUT} < N^{\frac{3}{5}}$.

REFERENCES

[1] A. Abboud, K. Bringmann, N. Fischer, and M. Künnemann. The time complexity of fully sparse matrix multiplication. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 4670–4703. SIAM, 2024.

[2] R. R. Amossen and R. Pagh. Faster join-projects and sparse matrix multiplications. In *Proceedings of the 12th International Conference on Database Theory*, pages 121–126. ACM, 2009.

[3] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *International Workshop on Computer Science Logic*, pages 208–222. Springer, 2007.

[4] N. Bakibayev, T. Kocisky, D. Olteanu, and J. Zavodny. Aggregation and ordering in factorised databases. In *Proc. International Conference on Very Large Data Bases*, 2013.

[5] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM (JACM)*, 30(3):479–513, 1983.

[6] C. Berkholz and N. Schweikardt. Constant delay enumeration with fpt-preprocessing for conjunctive queries of bounded submodular width. *arXiv preprint arXiv:2003.01075*, 2020.

[7] A. Björklund, R. Pagh, V. V. Williams, and U. Zwick. Listing triangles. In *International Colloquium on Automata, Languages, and Programming*, pages 223–234. Springer, 2014.

[8] E. Cohen. Estimating the size of the transitive closure in linear time. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 190–200. IEEE, 1994.

[9] M. Dalirrooyfard, S. Mathialagan, V. V. Williams, and Y. Xu. Listing cliques from smaller cliques. *arXiv preprint arXiv:2307.15871*, 2023.

[10] M. Dalirrooyfard and V. V. Williams. Induced cycles and paths are harder than you think. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 531–542. IEEE, 2022.

[11] S. Deep, X. Hu, and P. Koutris. Fast join project query evaluation using matrix multiplication. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1213–1223, 2020.

[12] R. Duan, H. Wu, and R. Zhou. Faster matrix multiplication via asymmetric hashing. In *2023 IEEE 64th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 2129–2138. IEEE, 2023.

[13] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *JACM*, 30(3):514–550, 1983.

[14] T. J. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, 2007.

[15] M. Grohe and D. Marx. Constraint solving via fractional edge covers. *ACM Transactions on Algorithms (TALG)*, 11(1):1–20, 2014.

[16] X. Hu and K. Yi. Parallel algorithms for sparse matrix multiplication and join-aggregate queries. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, page 411–425, 2020.

[17] M. Idris, M. Ugarte, and S. Vansummeren. The dynamic Yannakakis algorithm: Compact and efficient query processing under updates. In *Proc. ACM SIGMOD International Conference on Management of Data*, 2017.

[18] M. R. Joglekar, R. Puttagunta, and C. Ré. Ajar: Aggregations and joins over annotated relations. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 91–106, 2016.

[19] M. A. Khamis, H. Q. Ngo, and D. Suciu. What do shannon-type inequalities, submodular width, and disjunctive datalog have to do with one another? In *Proc. ACM Symposium on Principles of Database Systems*, 2017.

[20] D. Marx. Tractable hypergraph properties for constraint satisfaction and conjunctive queries. *Journal of the ACM (JACM)*, 60(6):1–51, 2013.

[21] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*, pages 111–124. ACM, 2018.

[22] R. Pagh and M. Stöckel. The input/output complexity of sparse matrix multiplication. In *European Symposium on Algorithms*, pages 750–761. Springer, 2014.

[23] M. Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proc. ACM Symposium on Theory of Computing*, pages 603–610. ACM, 2010.

[24] V. Strassen. Gaussian elimination is not optimal. *Numer. Math.*, 13:354–356, 1969.

[25] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 137–146, 1982.

[26] T. L. Veldhuizen. Leapfrog triejoin: A simple, worst-case optimal join algorithm. In *Proc. International Conference on Database Theory*, 2014.

[27] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.

[28] V. V. Williams, Y. Xu, Z. Xu, and R. Zhou. New bounds for matrix multiplication: from alpha to omega. In *Proceedings of the 2024 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 3792–3835. SIAM, 2024.

[29] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. International Conference on Very Large Data Bases*, pages 82–94, 1981.

Algorithm 3: DeDangle$(Q, \mathcal{R})$ [29]

Let $\mathcal{T}$ be a join tree of Q rooted at r;
while *visit nodes a bottom-up way* **do**
  foreach *node e visited with* $e \neq r$ **do**
    $e' \leftarrow$ the parent of e;
    $R_{e'} \leftarrow R_{e'} \ltimes R_e$;
while *visit nodes a top-down way* **do**
  foreach *node e visited with* $e \neq r$ **do**
    $e' \leftarrow$ the parent of e;
    $R_e \leftarrow R_e \ltimes R_{e'}$;
return updated $\mathcal{R}$;

Algorithm 4: Yannakakis$(Q, \mathcal{R})$ [29]

$\mathcal{R} \leftarrow$ DeDangle$(Q, \mathcal{R})$;
Let $\mathcal{T}$ be a join tree of Q rooted at r;
while *visit nodes a bottom-up way* **do**
  foreach *node e visited with* $e \neq r$ **do**
    $e' \leftarrow$ the parent of e;
    $R_e \leftarrow \pi_{\mathbf{y} \cup (e \cap e')} R_e$;
    $R_{e'} \leftarrow R_e \bowtie R_{e'}$;
return $\pi_{\mathbf{y}} R_r$;

A DETAILS OF YANNAKAKIS FRAMEWORK

In Algorithm 3, we show how to remove dangling tuples in an input instance $\mathcal{R}$ for an acyclic join-project query Q, which runs in $O(N)$ time. In Algorithm 4, we show the Yannakakis framework. In Section 1.5, we also mentioned a slightly modified version of Yannakakis framework, in which some specific operations can be captured as the Boolean matrix multiplication problem. Suppose we choose a join tree rooted at R_k. It starts with R_1. As R_1 is the only child of R_2, it updates R_2 with $\pi_{A_1,A_3} R_1(A_1, A_2) \bowtie R_2(A_2, A_3)$ directly, and then removes R_1. In the next iteration with R_2, it does not need to project away any attribute, and just further updates R_3 with $\pi_{A_1,A_4} R_2(A_1, A_3) \bowtie R_3(A_3, A_4)$. It repeats this procedure until only one relation is left.

B NEW ANALYSIS OF STAR QUERY IN [11]

Recall the a star query Q_{star} is defined as $Q_{\text{star}} = \pi_{A_1,A_2,\cdots,A_k} R_1(A_1, B) \bowtie R_2(A_2, B) \bowtie \cdots \bowtie R_k(A_k, B)$. This algorithm is also built on the heavy-light decomposition technique. Suppose two parameter $\triangle_1, \triangle_2$ are given, whose values will be determined later. For any $i \in [n]$, a value $a_i \in \text{dom}(A_i)$ is *heavy* if it appears in more than $\triangle_2$ tuples in R_i and *light* otherwise. A value $b \in \text{dom}(B)$ is *heavy* if it appears in more than $\triangle_1$ tuples in R_i for some $i \in [k]$. Then, the input query can be reduced into the following sub-queries:

$$\pi_{A_1,A_2,\cdots A_k} R_1(A_1^?, B^?) \bowtie R_2(A_2^?, B^?) \bowtie \cdots \bowtie R_k(A_k^?, B^?)$$

where ? can be either heavy or light, and can be different over different attributes.

- If $A_i^? = A_i^{\text{light}}$ for some $i \in [k]$, it invokes the Yannakakis framework. The number of intermediate join results is at most $O(\text{OUT} \cdot \triangle_2)$, hence the time cost of this step is $O(\text{OUT} \cdot \triangle_2)$.
- If $B^? = B^{\text{light}}$, it invokes the Yannakakis framework. The number of intermediate join results is at most $O(N \cdot \triangle_1^{k-1})$, hence the time cost of this step is $O(N \cdot \triangle_1^{k-1})$.
- If $A_i^? = A_i^{\text{heavy}}$ for every $i \in [k]$ and $B^? = B^{\text{heavy}}$, it is reduced to a rectangular matrix multiplication problem of sizes $\mathsf{MM}\left((\frac{N}{\triangle_2})^{\lfloor \frac{k}{2} \rfloor}, \frac{N}{\triangle_1}, (\frac{N}{\triangle_2})^{\lceil \frac{k}{2} \rceil}\right)$.

Hence, the overall cost is $N \cdot \Delta_1^{k-1} + \text{OUT} \cdot \Delta_2 + \mathsf{MM}\left((\frac{N}{\triangle_2})^{\lfloor \frac{k}{2} \rfloor}, \frac{N}{\triangle_1}, (\frac{N}{\triangle_2})^{\lceil \frac{k}{2} \rceil}\right)$. To achieve the minimum cost, we always require $N \cdot \Delta_1^{k-1} = \text{OUT} \cdot \Delta_2$. For simplicity, we assume $\omega = 2$ and k is even. We distinguish two more cases:

- $(\frac{N}{\triangle_2})^\ell \leqslant \frac{N}{\triangle_1}$: The cost of matrix multiplication is $\frac{N}{\triangle_1} \cdot (\frac{N}{\triangle_2})^\ell$. Hence, the total cost is
$$N \cdot \triangle_1^{2\ell-1} + \mathrm{OUT} \cdot \triangle_2 + \frac{N}{\triangle_1} \cdot (\frac{N}{\triangle_2})^\ell \geqslant N \cdot \mathrm{OUT}^{\frac{2\ell-1}{2\ell+1}}$$
where the equality is achieved when $\triangle_1 = \mathrm{OUT}^{\frac{1}{2\ell+1}}$ and $\triangle_2 = N \cdot \mathrm{OUT}^{-\frac{2}{2\ell+1}}$. The $(\frac{N}{\triangle_2})^\ell < \frac{N}{\triangle_1}$ holds if $\mathrm{OUT} \leqslant N$.
- $(\frac{N}{\triangle_2})^\ell > \frac{N}{\triangle_1}$: The cost of matrix multiplication is $(\frac{N}{\triangle_2})^k$. Hence, the total cost is
$$N \cdot \triangle_1^{2\ell-1} + \mathrm{OUT} \cdot \triangle_2 + (\frac{N}{\triangle_2})^{2\ell} \geqslant (N \cdot \mathrm{OUT})^{\frac{2\ell}{2\ell+1}}$$
by setting $\triangle_1 = \left(\frac{\mathrm{OUT}^{2\ell}}{N}\right)^{\frac{1}{(2\ell+1)(2\ell-1)}}$ and $\triangle_2 = \frac{N^{2\ell/(2\ell+1)}}{\mathrm{OUT}^{1/(2\ell+1)}}$. The $(\frac{N}{\triangle_2})^\ell > \frac{N}{\triangle_1}$ holds if $\mathrm{OUT} > N$.

Hence, this algorithm runs in $O\left(N \cdot \mathrm{OUT}^{\frac{k-1}{k+1}} + (N \cdot \mathrm{OUT})^{\frac{k}{k+1}}\right)$ time for even k. We next consider the case when k is odd. When $k = 3$, this algorithm runs in $O\left(N \cdot \mathrm{OUT}^{4/5} + (N \cdot \mathrm{OUT})^{3/4}\right)$. When $k = 5$, this algorithm runs in $O(N \cdot \mathrm{OUT}^{2/3} + (N \cdot \mathrm{OUT})^{5/6})$ time. For $k \geqslant 6$, this algorithms runs in $\Omega\left(N \cdot \mathrm{OUT}^{5/7}\right)$ time. Putting everything together, our new result in Figure 1 completely improves this result for every value of k.

C LIMITATIONS OF YANNAKAKIS

We consider the line query with $k = 3$: $\pi_{A_1,A_4} R_1(A_1, A_2) \bowtie R_2(A_2, A_3) \bowtie R_3(A_3, A_4)$. There are two query plans under the Yannakakis framework: plan 1 of $\pi_{A_1,A_4}(\pi_{A_1,A_3}(R_1 \bowtie R_2)) \bowtie R_3$ and plan 2 of $\pi_{A_1,A_4} R_1 \bowtie (\pi_{A_2,A_4}(R_2 \bowtie R_3))$.

We first construct an instance (see the top half of Figure 4). Assume $\mathrm{OUT} \leqslant N$. Attributes A_1, A_2, A_3, A_4 have domain sizes $\frac{\mathrm{OUT}}{2}$, $\frac{N}{\mathrm{OUT}}$, $\frac{N}{2}$, 1. Relations R_1 is a Cartesian product between A_1 and A_2, R_2 is a many-to-one mapping from A_2 to A_3, and R_3 is a one-to-many mapping from A_3 to A_4. It can be easily checked that this instance has input size $\Theta(N)$ and output size $\Theta(\mathrm{OUT})$. Note that plan 1 incurs a cost of $\Theta(N \cdot \mathrm{OUT})$ since $|R_1 \bowtie R_2| = \frac{N \cdot \mathrm{OUT}}{2}$, while plan 2 incurs a cost of $\Theta(N)$ since $|R_2 \bowtie R_3| = N$ and $|R_1 \times \mathrm{dom}(A_4)| = N$. We next construct a symmetric instance (see the bottom half of Figure 4). In this case, these two plans would have opposite behaviors. At last, we consider the instance by combining these two sub-instances together. In this instance, both plan incurs a cost of $\Theta(N \cdot \mathrm{OUT})$, since $|R_1 \bowtie R_2| = |R_2 \bowtie R_3| = \Theta(N \cdot \mathrm{OUT})$. Similarly, we can construct an instance for general line queries, such that the intermediate join results materialized by Yannakakis algorithm is $\Theta(N \cdot \mathrm{OUT})$.

Moreover, this hard instance also shows that $|\pi_{A_1,A_3}(R_1 \bowtie R_2)| = |\pi_{A_2,A_4}(R_2 \bowtie R_3)| = \Theta(N \cdot \mathrm{OUT})$. Hence, no matter which algorithm is used for computing $\pi_{A_1,A_3}(R_1 \bowtie R_2)$ or $\pi_{A_2,A_4}(R_2 \bowtie R_3)$, even with fast matrix multiplication techniques, the bottleneck is still to materialize the output of the intermediate matrix multiplication, which is as expensive as $\Theta(N \cdot \mathrm{OUT})$.

D LIMITATION OF SEMI-RING ALGORITHMS

Theorem D.1. *For an acyclic join-project query Q with free-width* fhtw*, given parameters $1 \leqslant N, \mathrm{OUT} \leqslant N^{\mathsf{fhtw}}$, there exists an instance $\mathcal{R}$ for Q of input size $\Theta(N)$ and output size $\Theta(\mathrm{OUT})$ such that any semi-ring algorithm for computing the query results $Q(\mathcal{R})$ requires $\Omega\left(N \cdot \mathrm{OUT}^{1-\frac{1}{\mathsf{fhtw}}} + \mathrm{OUT}\right)$ time.*

Hard Instance for Matrix Multiplication [2, 22]. We start with the hard instance for $Q_{\mathsf{matrix}} = \pi_{A,C} R_1(A, B) \bowtie R_2(B, C)$. Attributes A, B, C have domain size $\sqrt{\mathrm{OUT}}$, $\frac{N}{\sqrt{\mathrm{OUT}}}$, $\sqrt{\mathrm{OUT}}$. Relations R_1 is a Cartesian product between A and B, and R_2 is a Cartesian product between B and C. It has been

Algorithm 5: $\textsc{Cleanse}(Q = (\mathcal{V}, \mathcal{E}, \mathbf{y}), \mathcal{R})$

```
R ← DeDangle(Q, R);
while true do
    if ∃B ∈ V − y s.t. B only appears in one relation, say e then
        R_e ← π_{e−B} R_e;
        V ← V − {B};
    if ∃e, e' ∈ E s.t. e ⊆ e' then E ← E − {e};
    if Q does not change from last iteration then
        break;
return updated Q, R;
```

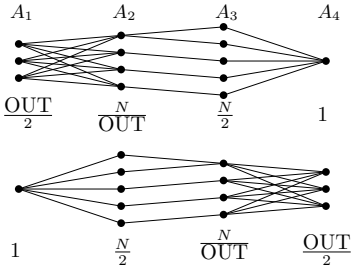

Fig. 4. A hard instance of line-3 query for Yannakakis framework. The numbers below are the domain sizes.

proved by [22] that when only semi-ring operations are allowed, any algorithm needs to incur a cost of $\Omega\left(N \cdot \sqrt{\mathrm{OUT}}\right)$.

Hard Instance for Star Queries [2]. We can generalize the hard instance for Q_{matrix} to Q_{star} as follows. Attribute A_i has domain size $\mathrm{OUT}^{1/k}$ for any $i \in [m]$, and attribute B has domain size $\frac{N}{\mathrm{OUT}^{1/k}}$. Relations R_i is a Cartesian product between A_i and B. The similar argument can be made such that when only semi-ring operations are allowed, any algorithm needs to incur a cost of $\Omega\left(N \cdot \mathrm{OUT}^{1-1/k}\right)$.

Hard Instance for General Acyclic Join-Project Queries. Consider an acyclic join-project query Q that is also cleansed. We will remove this assumption at last. Let $Q_1, Q_2, \cdots, Q_h$ be the connected sub-queries in $G_Q^{\exists}$. Wlog, assume Q_1 has the maximum number of relations that contain unique output attributes. Let $e_1, e_2, \cdots, e_\ell$ be the set of relations that contain unique output attributes. By definition, $\ell = \mathsf{freew}$. We pick an arbitrary unique output attribute from e_i denoted as A_i.

We construct a hard instance as follows. Each chosen output attribute A_i has its domain size $\mathrm{OUT}^{1/\ell}$. For any remaining output attribute $A \in \mathbf{y} - \{A_1, A_2, \cdots, A_\ell\}$, we set $\mathrm{dom}(A) = \{*\}$. Every non-output attribute $B \in \mathcal{V} - \mathbf{y}$ has domain size $\frac{N}{\mathrm{OUT}^{1/\ell}}$. Consider an arbitrary relation R_e. There is a one-to-one mapping between all non-output attributes. If $e \in \{e_1, e_2, \cdots, e_\ell\}$, R_e is a Cartesian product between the unique output attribute and non-output attributes, with $\mathrm{OUT}^{1/\ell} \cdot \frac{N}{\mathrm{OUT}^{1/\ell}} = N$ tuples. Otherwise, R_e is a one-to-one mapping between all non-output attributes, with $\frac{N}{\mathrm{OUT}^{1/\ell}}$ tuples. It can be easily shown that for any non-output attribute B, the projection of the query results $Q(\mathcal{R})$ onto attributes $A_1, A_2, \cdots, A_\ell, B$ has its size as large as

$$\left(\mathrm{OUT}^{1/\ell}\right)^{\ell} \cdot \frac{N}{\mathrm{OUT}^{1/\ell}} = N \cdot \mathrm{OUT}^{1-\frac{1}{\ell}},$$

which is equivalent to the hard instance for Q_{star} with ℓ relations. See an example in Figure 5. Any semi-ring algorithm for Q and $\mathcal{R}$ would imply a semi-ring algorithm for Q_{star} with the corresponding hard instance. Hence, when only semi-ring operations are allowed, any algorithm needs to incur a cost of $\Omega\left(N \cdot \mathrm{OUT}^{1-\frac{1}{\mathsf{freew}}}\right)$.

Now, we consider the case when Q is not cleansed. Let Q' be the cleansed version of Q. We construct the hard instance for Q' as above. For every non-output attribute that has been removed by the cleanse procedure from Q, we set its domain as $\{*\}$. For every relation e that has been removed by the cleanse procedure from Q, there must exist some relation e' in Q' such that $e \subseteq e'$. We just set $R_e = \pi_e R_{e'}$ as the projection of $R_{e'}$. All arguments apply for Q.

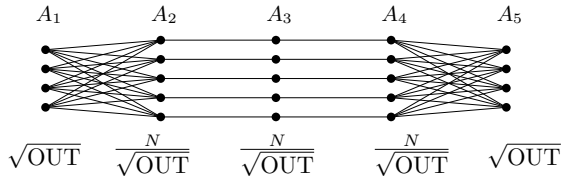

Fig. 5. A hard instance of line-4 query for semi-ring algorithms. The numbers below are the domain sizes.

E NEW ANALYSIS OF [16] IN RAM MODEL

Hu et al. [16] have investigated the join-aggregate queries defined over semi-ring in the massively parallel computational model. Note that join-project query is defined on a special semi-ring $(\{0,1\}, \cap, \cup)$. Revisiting this algorithm in the RAM model, we observe:

- given any instance $\mathcal{R}$ of Q_{line} with input size $O(N \cdot \sqrt{\text{OUT}})$ and output size OUT, there is an algorithm running in $O(N \cdot \sqrt{\text{OUT}})$ time that can reduce $\mathcal{R}$ into $O(1)$ instances of Q_{matrix} with input size $O(N \cdot \sqrt{\text{OUT}})$ and output size OUT.
- given any instance $\mathcal{R}$ of Q_{star} with input size $O(N \cdot \sqrt{\text{OUT}})$ and output size OUT, there is an algorithm runs in $O(N \cdot \sqrt{\text{OUT}})$ time that can reduce $\mathcal{R}$ into $O(1)$ instances of Q_{matrix} with input size $O(N \cdot \sqrt{\text{OUT}})$ and output size OUT, and runs in $O(N \cdot \text{OUT})$ time.
- given any instance $\mathcal{R}$ of tree query Q (an acyclic query such that each relation contains at most two attributes), with input size N and output size OUT, there is an algorithm running in $O(N \cdot \text{OUT}^{2/3})$ time that can reduce $\mathcal{R}$ into $O(1)$ instances of Q_{matrix} with input size $O\left(N \cdot \text{OUT}^{2/3}\right)$ and output size OUT.

In incorporating the best algorithm [1] for sparse Boolean matrix multiplication, we obtain the following results (assuming $\omega = 2$):

- Given an instance $\mathcal{R}$ for Q_{line} or Q_{star} with input size N and output size OUT, the $Q_{\text{line}}(\mathcal{R})$ and $Q_{\text{star}}(\mathcal{R})$ can be computed in $\widetilde{O}\left(N \cdot \text{OUT}^{\frac{5}{6}} + \text{OUT}\right)$ time.
- Given an instance $\mathcal{R}$ for a tree query Q with input size N and output size OUT, the $Q(\mathcal{R})$ can be computed in $\widetilde{O}\left(N \cdot \text{OUT} + N + \text{OUT}\right)$ time. This is no better than the Yannakakis algorithm.

F MISSING PROOFS IN SECTION 2

Proof of Lemma 2.1. We review the algorithm in [1]. Suppose we are given two input matrices $\mathbb{M}_1 \in \{0,1\}^{n_A \times n_B}$ and $\mathbb{M}_2 \in \{0,1\}^{n_B \times n_C}$. For an arbitrary Boolean matrix $\mathbb{M} \in \{0,1\}^{n_1 \times n_2}$, we denote $\text{supp}(\mathbb{M}) = \{(i,j) \in [n_1] \times [n_2] : \mathbb{M}[i,j] = 1\}$ be the *support* of $\mathbb{M}$, i.e., the set of indices for all non-zero entries in $\mathbb{A}$. Let $N = \text{supp}(\mathbb{M}_1) + \text{supp}(\mathbb{M}_2)$ and $\text{OUT} = \text{supp}(\mathbb{M}_1 \cdot \mathbb{M}_2)$. The essence of the algorithm presented in [1] is a primitive for sparse input and dense output matrix where $n_A \cdot n_C \leqslant \text{OUT}$, which is also the bottleneck of whole algorithm. We note that all other primitives together take $\widetilde{O}(N + \text{OUT})$ time. Suppose an $O(1)$-approximation of OUT is known.

The First Complexity Result. Set $\Delta = (\frac{n_B \cdot \text{OUT}}{N})^{\frac{1}{4-\omega}}$. A value $b \in \text{dom}(B)$ is *heavy* if its degree is greater than Δ in R_1 or R_2, and *light* otherwise. Let $B^{\text{heavy}}, B^{\text{light}}$ be the set of heavy, light values in B respectively. We then divide Q_{matrix} into two sub-queries: $Q^{?}_{\text{matrix}} = \pi_{A,C} R_1(A, B^{?}) \bowtie R_2(B^{?}, C)$, where ? can either be heavy or light. For $Q^{\text{light}}_{\text{matrix}}$, we compute the full join results and then project out B. The cost of this step is $O(N \cdot \Delta) = O\left(N^{\frac{3-\omega}{4-\omega}} \cdot (n_B \cdot \text{OUT})^{\frac{1}{4-\omega}}\right)$. For $Q^{\text{heavy}}_{\text{matrix}}$, we compute the

following rectangular matrix multiplication:

$$\underset{\substack{x\cdot z\leqslant \mathrm{OUT},\\ \Delta\leqslant x,\\ \Delta\leqslant z}}{\mathsf{MM}}(x,n_B,z)\leqslant \max_{\substack{x\cdot z\leqslant \mathrm{OUT},\\ \Delta\leqslant x\leqslant \frac{\mathrm{OUT}}{\Delta},\\ \Delta\leqslant z\leqslant \frac{\mathrm{OUT}}{\Delta}}} \frac{x\cdot z\cdot n_B}{\min\{x,n_B,z\}^{3-\omega}} \tag{4}$$

We further distinguish two more cases:

- Case 1: $\min\{x,z\}<n_B$. Then, we can further bound (4) $\leqslant \frac{n_B\cdot \mathrm{OUT}}{\Delta^{3-\omega}}\leqslant N^{\frac{3-\omega}{4-\omega}}\cdot(n_B\cdot\mathrm{OUT})^{\frac{1}{4-\omega}}$.
- Case 2: $\min\{x,z\}>n_B$. Then, we can further bound (4) $\leqslant \max_{\substack{x\cdot z\leqslant \mathrm{OUT},\\ \Delta\leqslant x\leqslant \frac{\mathrm{OUT}}{\Delta},\\ \Delta\leqslant z\leqslant \frac{\mathrm{OUT}}{\Delta}}} \frac{x\cdot z\cdot n_B}{n_B^{3-\omega}}\leqslant \mathrm{OUT}\cdot n_B^{\omega-2}$.

Combining these two cases, we obtain the following result as desired.

The Second Complexity Result. We skip the heavy-light strategy and compute the rectangular matrix multiplications directly:

$$\underset{\substack{x\cdot z\leqslant \mathrm{OUT},\\ x\leqslant n_A,\\ z\leqslant n_C}}{\mathsf{MM}}(x,n_B,z)\leqslant \max_{\substack{x\cdot z\leqslant \mathrm{OUT},\\ x\leqslant n_A,\\ z\leqslant n_C}} \frac{x\cdot z\cdot n_B}{\min\{x,n_B,z\}^{3-\omega}} \tag{5}$$

We further distinguish two more cases:

- Case 1: $\min\{x,z\}\leqslant n_B$. Then, we can further bound (5) as

$$\begin{aligned}(5)\leqslant \max_{\substack{x\cdot z\leqslant \mathrm{OUT},\\ x\leqslant n_A,\\ z\leqslant n_C}} \frac{x\cdot z\cdot n_B}{\min\{x,z\}^{3-\omega}} &= \max_{\substack{x\cdot z\leqslant \mathrm{OUT},\\ x\leqslant n_A,\\ z\leqslant n_C}} n_B\cdot\max\{x,z\}\cdot\min\{x,z\}^{\omega-2}\\ &= \max_{\substack{x\cdot z\leqslant \mathrm{OUT},\\ x\leqslant n_A,\\ z\leqslant n_C}} n_B\cdot(x\cdot z)^{\omega-2}\cdot\max\{x,z\}^{3-\omega}\\ &\leqslant n_B\cdot\mathrm{OUT}^{\omega-2}\cdot\max\{n_A,n_C\}^{3-\omega}\end{aligned}$$

- Case 2: $\min\{x,z\}>n_B$. Then, we can further bound (5) $\leqslant \max_{\substack{x\cdot z\leqslant \mathrm{OUT},\\ x\leqslant n_A,\\ z\leqslant n_C}} \frac{x\cdot z\cdot n_B}{n_B^{3-\omega}}\leqslant \mathrm{OUT}\cdot n_B^{\omega-2}$.

Combining these two cases, we obtain the result as desired. □

Proof of Lemma 2.3. As $d_k\leqslant d_{k-1}\leqslant\cdots\leqslant d_1$, $\prod_{i\in\eta} d_i\geqslant\prod_{j\in\beta} d_j$ and $\prod_{i\in\eta-\{1\}} d_i\geqslant\prod_{j\in\beta-\{2\}} d_j$. As $\prod_{i\in\eta} d_i\cdot\prod_{j\in\beta} d_j=\lambda$, we have $\prod_{i\in\eta} d_i=\max\left\{\prod_{i\in\eta} d_i,\prod_{j\in\beta} d_j\right\}\geqslant\lambda^{1/2}$ Wlog, assume k is even (the odd case can be proved similarly). As $d_1\geqslant d_2\geqslant\cdots\geqslant d_{k-1}$, $\prod_{i\in\eta\setminus\{1\}} d_i\leqslant\left(\prod_{j=2}^{k} d_j\right)^{1/2}=\left(\frac{\lambda}{d_1}\right)^{1/2}$. Hence, $\prod_{i\in\eta\setminus\{1\}} d_i\leqslant\lambda^{1/2}$. Moreover, if $d_1\leqslant\lambda'$, $\prod_{i\in\eta} d_i=\prod_{i\in\eta\setminus\{1\}} d_i\cdot d_1\leqslant\left(\frac{\lambda}{d_1}\right)^{1/2}\cdot d_1=(\lambda\cdot d_1)^{1/2}\leqslant\sqrt{\lambda\cdot\lambda'}$. Again, as $\prod_{i\in\eta} d_i\cdot\prod_{j\in\beta} d_j=\lambda$, we have $\prod_{j\in\beta} d_j\geqslant\sqrt{\frac{\lambda}{\lambda'}}$. □

Correctness Proof of Theorem 2.5. Given any input instance $\mathcal{R}$ for Q_{star}, we denote $\mathcal{S}$ as the results returned by the algorithm in Section 2.2. To prove the correctness, we next show $Q_{\text{star}}(\mathcal{R}) = \mathcal{S}$. *Direction $\subseteq$.* For each query result $(a_1, a_2, \cdots, a_k) \in Q_{\text{star}}(\mathcal{R})$, there must exist some value $b \in \text{dom}(B)$, such that $(a_i, b) \in R_i$ for each $i \in [k]$. Then, in step 1, let $\phi = \phi_b$ be the permutation of b for simplicity. If $b \in B_\phi^{\text{light}}$, then $(a_1, a_2, \cdots, a_k)$ is computed by Step 3. If $b \in B_\phi^{\text{heavy}}$, $(a_1, a_2, \cdots, a_k)$ is computed by Step 4. Hence, every query result must be reported by this algorithm. *Direction $\supseteq$.* For each tuple $(a_1, a_2, \cdots, a_k) \in \mathcal{S}$ reported by this algorithm. If $(a_1, a_2, \cdots, a_k)$ is reported by Step 3 or Step 4, it is easy to check that there exists some $b \in \text{dom}(B)$ such that $(a_i, b) \in R_i$ for each $i \in [k]$, hence $(a_1, a_2, \cdots, a_k) \in Q_{\text{star}}(\mathcal{R})$. □

G MISSING PROOFS IN SECTION 3

Correctness Proof of Lemma 3.8. Given any input instance $\mathcal{R}$ for a flower query Q, we denote $\mathcal{S}$ as the results returned by the algorithm in Section 3.2. To prove the correctness, we next show $Q(\mathcal{R}) = \mathcal{S}$. *Direction $\subseteq$.* For each query result $t_1 \in Q(\mathcal{R})$, there must exist a full join result $t_2 \in \bowtie_{e\in\mathcal{E}} R_e$ such that $\pi_{\mathbf{y}} t_2 = t_1$. Let $t = \pi_{\mathbf{x}} t_2$. In step 1, let $\phi = \phi_t$ be the permutation of t for simplicity. If $t \in \mathbf{x}_\phi^{\text{light}}$, then t_1 is computed by Step 3. If $t \in \mathbf{x}_\phi^{\text{heavy}}$, t_1 is computed by Step 4. Hence, every query result must be reported by this algorithm. *Direction $\supseteq$.* For each tuple $t_1 \in \mathcal{S}$ reported by this algorithm. If t_1 is reported by step 3 or step 4, it is easy to check that there exists some full join result $t_2 \in \bowtie_{e\in\mathcal{E}} R_e$ such that $\pi_{\mathbf{y}} t_2 = t_1$, hence $t_2 \in Q_{\text{star}}(\mathcal{R})$. □

Correctness Proof of Theorem 3.9. Given any input instance $\mathcal{R}$ for a general acyclic join-project query Q, we denote $\mathcal{S}$ as the results returned by the algorithm in Section 3.2. To prove the correctness, we next show $Q(\mathcal{R}) = \mathcal{S}$. This trivially holds for all base cases as well as the general case (1). We next prove it for general case (2) by induction. By hypothesis, we assume that each subquery Q^χ is correctly computed. Then, it suffices to show that the union of query results of Q^χ over all possible labels χ is exactly $Q(\mathcal{R})$. This is straightforwardly guaranteed by the decomposition in Step 2. Putting everything together, we complete the whole proof. □

H BETTER RESULTS ON TREE QUERY

Corollary H.1. *For a tree join-project query Q with its cleansed version Q', where each connected component of $G_{Q'}^{\exists}$ is Q_{matrix}, and an instance $\mathcal{R}$ of input size N and output size* OUT*, the $Q(\mathcal{R})$ can be computed in $\widetilde{O}\left(N \cdot \text{OUT}^{\frac{\mu}{1+\mu}} + \text{OUT} + N^{\frac{(2+\alpha)\mu}{1+\mu}} \cdot \text{OUT}^{\frac{1-\alpha\mu}{1+\mu}}\right)$ time. If $\omega = 2$, it degenerates to $\widetilde{O}\left(N \cdot \text{OUT}^{1/3} + \text{OUT}\right)$.*

Corollary H.2. *For a tree query Q with its cleansed version Q' where each connected component of $G_{Q'}^{\exists}$ is Q_{star}, and an instance $\mathcal{R}$ of input size N and output size* OUT*, the $Q(\mathcal{R})$ can be computed in*

$$\widetilde{O}\left(N^{\omega-2} \cdot \text{OUT} + N \cdot \text{OUT}^{\frac{2(\text{freew}-1)}{(5-\omega)(\text{freew}-1)+2}}\right)$$

time, where freew *is the free-width of Q. If $\omega = 2$, it degenerates to $\widetilde{O}\left(N \cdot \text{OUT}^{\frac{2(\text{freew}-1)}{3(\text{freew}-1)+2}} + \text{OUT}\right)$.*

Received December 2023; revised February 2024; accepted March 2024