Jaguar: A Primal Algorithm for Conjunctive Query Evaluation in Submodular-Width Time

MAHMOUD ABO KHAMIS, RelationalAI, USA
HUBIE CHEN, King's College London, UK

The *submodular width* is a complexity measure of conjunctive queries, which assigns a nonnegative real number $\mathrm{subw}(Q)$ to each conjunctive query Q. An existing algorithm, the PANDA algorithm, performs conjunctive query evaluation in polynomial time where the exponent is essentially $\mathrm{subw}(Q)$. Formally, for every Boolean conjunctive query Q, the PANDA algorithm performs conjunctive query evaluation for Q in time $O(N^{\mathrm{subw}(Q)} \cdot \mathrm{polylog}(N))$, where N denotes the input database size; moreover, there is complexity-theoretic evidence that, for a number of Boolean conjunctive queries, no exponent strictly below $\mathrm{subw}(Q)$ can be achieved by combinatorial algorithms. On the outer level, the submodular width of a conjunctive query Q can be described as the optimum of a maximization problem over certain polymatroids; polymatroids are set functions on the variables of Q that satisfy Shannon inequalities. The PANDA algorithm in a sense works in the dual space of this maximization problem; it makes use of information theory, manipulates Shannon inequalities, and transforms a conjunctive query into a set of disjunctive datalog rules which are individually solved.

In this article, we introduce a new algorithm for conjunctive query evaluation which achieves, for each Boolean conjunctive query Q and for all $\epsilon > 0$, a running time of $O(N^{\mathrm{subw}(Q)+\epsilon})$. This new algorithm's description and analysis are, in our view, significantly simpler than those of PANDA. We refer to it as a *primal* algorithm as it operates in the primal space of the described maximization problem, by throughout maintaining a feasible primal solution, namely, a polymatroid. Indeed, this algorithm deals directly with the input conjunctive query and adaptively computes a sequence of joins, in a guided fashion, so that the cost of these join computations is bounded. Additionally, this algorithm can achieve the stated runtime for the generalization of submodular width incorporating degree constraints. We dub our algorithm *Jaguar*, as it is a join-adaptive guided algorithm.

CCS Concepts: • **Theory of computation** → **Database query processing and optimization (theory)**; • **Information systems** → **Join algorithms**; • **Mathematics of computing** → *Linear programming*.

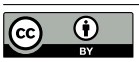
Additional Key Words and Phrases: conjunctive queries; submodular width; polymatroids; primal/dual

ACM Reference Format:
Mahmoud Abo Khamis and Hubie Chen. 2026. Jaguar: A Primal Algorithm for Conjunctive Query Evaluation in Submodular-Width Time. *Proc. ACM Manag. Data* 4, 2 (PODS), Article 108 (May 2026), 21 pages. https://doi.org/10.1145/3801904

1 Introduction

Conjunctive query evaluation, the computational problem of evaluating a conjunctive query on a relational database, is a fundamental problem that appears in many guises all throughout computer science, for example, in database management, graph algorithms, logic, constraint satisfaction, and graphical model inference [1, 4, 16, 21]. In the database context, conjunctive queries are considered

Authors' Contact Information: Mahmoud Abo Khamis, mahmoudabo@gmail.com, RelationalAI, Berkeley, CA, USA; Hubie Chen, hubie.chen@kcl.ac.uk, King's College London, London, UK.



ACM 2836-6573/2026/5-ART108
https://doi.org/10.1145/3801904

to be the most commonly occurring queries and are the most heavily studied form of database queries [1].

With the motivation of understanding which classes of Boolean conjunctive queries allow for tractable query evaluation, D. Marx introduced a measure of Boolean conjunctive queries called the *submodular width* [24]. Let us assume for now that all conjunctive queries under discussion are Boolean. In an established parameterized complexity model, Marx proved a dichotomy theorem [24]: on a class of conjunctive queries having *bounded submodular width*—that is, for which there exists a constant bounding the submodular width of all queries in the class—query evaluation is tractable; moreover, on any class of conjunctive queries not having bounded submodular width, query evaluation is intractable, under a complexity-theoretic assumption. This dichotomy theorem thus comprehensively describes every class of conjunctive queries as either tractable or intractable, with respect to query evaluation, and reveals that the submodular width serves as the critical parameter describing the location of the boundary between these two behaviors.

While Marx's dichotomy theorem describes the tractability status of every class of conjunctive queries, it left open a more fine-grained question regarding the data complexity of conjunctive queries. For every Boolean conjunctive query Q, it is known that conjunctive query evaluation on Q can be performed in polynomial time: there is a constant α such that conjunctive query evaluation on Q can be performed in $O(N^\alpha)$ time; here, N denotes the input database size. However, there is no general description of how to determine, for a conjunctive query Q, the lowest exponent α enjoying this property. Nonetheless, there are some general time complexity upper bounds known, in this vein, which are described in terms of the submodular width. Let $\mathrm{subw}(Q)$ denote the submodular width of a Boolean conjunctive query Q.

- Abo Khamis, Ngo, and Suciu [5, 6] presented an information-theoretic algorithm called *PANDA* that, for any Boolean conjunctive query Q, performs conjunctive query evaluation on Q within time $O(N^{\mathrm{subw}(Q)} \cdot \mathrm{polylog}(N))$.
- Drawing upon Marx's algorithm and analysis [24],[1] Berkholz and Schweikardt [10] presented an algorithm, which we call the Marx-Berkholz-Schweikardt algorithm. For any Boolean conjunctive query Q and any $\epsilon > 0$, this algorithm performs query evaluation on Q within time $O(N^{2\cdot\mathrm{subw}(Q)+\epsilon})$.

That is, the first algorithm essentially achieves runtime with exponent $\mathrm{subw}(Q)$, while the second algorithm essentially achieves runtime with exponent of $2 \cdot \mathrm{subw}(Q)$. Let us mention here that there are techniques for lower bounding the fine-grained time complexity of evaluating conjunctive queries. In particular, Fan, Koutris, and Zhao [14] presented such techniques and—among other results—proved that, for a number of different queries, no combinatorial algorithm can achieve a running time of the form $O(N^{\mathrm{subw}(Q)-\epsilon})$, with $\epsilon > 0$, unless an established fine-grained complexity-theoretic assumption fails.

The two algorithms named above, in addition to differing in terms of the time upper bounds that are demonstrated, differ in two aspects which we wish to highlight.

- First, the PANDA algorithm was shown to perform query evaluation not just within the stated time bound $O(N^{\mathrm{subw}(Q)} \cdot \mathrm{polylog}(N))$, but achieves a running time of similar form where the submodular width is replaced with a natural generalization of the submodular width that incorporates information about the degrees of an input database. Such information is formalized using a notion which is called *degree constraints* and also, in the context of conjunctive query evaluation, is simply called a *statistics*. A *statistics* is a pair $(\Delta, \boldsymbol{n})$ where, speaking on a high level, each element of Δ is a term consisting of a pair of variable subsets, and

[1] It is worth remarking here that the running time claimed by Marx [24] for his original algorithm was $N^{O(\mathrm{subw}(Q))}$ times a multiplicative overhead.

where $\boldsymbol{n}$ maps each term in Δ to a number. The mentioned generalization of the submodular width [6] maps each query Q and statistics $(\Delta, \boldsymbol{n})$ to a value $\mathrm{subw}(Q, \Delta, \boldsymbol{n})$, and PANDA can achieve a running time of $O(N^{\mathrm{subw}(Q,\Delta,\boldsymbol{n})} \cdot \mathrm{polylog}(N))$ when evaluating a query Q on databases satisfying the statistics $(\Delta, \boldsymbol{n})$. For any query Q and any statistics $(\Delta, \boldsymbol{n})$, it holds that $\mathrm{subw}(Q, \Delta, \boldsymbol{n}) \leq \mathrm{subw}(Q)$; it is thus transparent to see that incorporating statistics can only result in an improved running time. We are not aware of any work showing that an algorithm along the lines of the Marx-Berkholz-Schweikardt algorithm can utilize this information.

- Second, there is—in our view—a notable difference both in the objects that these algorithms and their analyses reason about, and also in the tools used by the analyses. To describe this difference, we begin by reflecting on the definition of submodular width. The submodular width is defined as the optimum of a maximization problem over the space of certain polymatroids, where the goal is to find the polymatroid maximizing a certain objective. The dual problem is a minimization problem over the space of Shannon inequalities, where the goal is to infer a Shannon inequality yielding the tightest upper bound on the submodular width. It can be shown by LP duality that these two problems have the same optimal value [6].

  The PANDA algorithm operates in the dual space, that is, the space of Shannon inequalities; it always maintains a Shannon inequality that upper bounds the submodular width. In contrast, the Marx-Berkholz-Schweikardt algorithm deals directly with the conjunctive query under evaluation; it uses recursion, and for any run of the algorithm, it is shown that at each leaf of the recursive tree, there is a polymatroid describing the data at that leaf. We view PANDA as a *dual* algorithm and the Marx-Berkholz-Schweikardt algorithm as a *primal* algorithm. Although we do not give formal definitions of *dual* and *primal* algorithms, we believe that the distinction is a meaningful one; certainly, information theory is not explicitly employed by the Marx-Berkholz-Schweikardt algorithm nor its analysis.

Contributions. We introduce a new algorithm for conjunctive query evaluation called *Jaguar*. Our algorithm has the following features.

- It is fast. It achieves, for every query Q and value $\epsilon > 0$, a running time of $O(N^{\mathrm{subw}(Q)+\epsilon})$. Its running time is thus similar to that of PANDA.
- It is general. It can make use of degree information; for every query Q, any statistics $(\Delta, \boldsymbol{n})$, and any value $\epsilon > 0$, it can achieve a running time of $O(N^{\mathrm{subw}(Q,\Delta,\boldsymbol{n})+\epsilon})$ when evaluating Q on databases satisfying the statistics $(\Delta, \boldsymbol{n})$.
- It and its analysis are relatively simple. With respect to the above distinction between dual and primal algorithms, we view Jaguar as a primal algorithm. Throughout an execution, Jaguar maintains a set function $\boldsymbol{g}$ that is defined on all variable subsets and that describes the data; this set function—in particular, the points at which it violates submodularity—is used to guide the process of partitioning the data and computing joins. Our analysis shows and crucially uses the fact that this set function induces a polymatroid that—in a certain precise sense—lower bounds the submodular width.

  We believe that Jaguar and its analysis directly shed light on the role of submodularity in conjunctive query evaluation. We view Jaguar and its analysis as simpler than that of PANDA; note that its study requires no explicit use of information theory. At the same time, we view Jaguar and its analysis as simpler and more direct than that of the Marx-Berkholz-Schweikardt algorithm, which makes use of two notions of *consistency* and, throughout its execution, generically searches for pairs of subsets of the variables that violate a condition called *ϵ-uniformity*; when a violation is found, a general partitioning procedure is invoked.

To sum up, we believe that the introduction of Jaguar and its analysis is valuable, since it (1) essentially achieves the runtime exponent $\mathsf{subw}(Q, \Delta, \boldsymbol{n})$ that can be essentially achieved by PANDA, but is not known to be achievable by the Marx-Berkholz-Schweikardt algorithm, which essentially achieves an exponent of $2 \cdot \mathsf{subw}(Q)$, and (2) accomplishes this via a simple algorithm having an analysis that is, in our view, quite direct and deals very openly and clearly with the notion of submodularity.

Let us remark that our main result on Jaguar's performance addresses not just Boolean conjunctive queries, but also addresses conjunctive queries that may have free variables. We work with a suitable and established generalization of submodular width to arbitrary conjunctive queries, and show that Jaguar achieves a runtime of $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})+\epsilon} + \mathsf{OUT})$, where OUT denotes the output size. Note that each of the PANDA and the Marx-Berkholz-Schweikardt algorithms, on general conjunctive queries, can similarly achieve the running time described for them, plus the output size.

Related work. The introduction of the submodular width followed a long and substantial body of work defining and studying complexity measures of conjunctive queries, including hypertree width [18, 19], generalized hypertree width [7, 13], and fractional hypertree width [17, 21, 23]; here, we have offered a sampling of references. For each conjunctive query Q, all of these measures provide upper bounds on the discussed exponent describing the running time of query evaluation on Q. Abo Khamis, Hu, and Suciu [3] introduced an algebraic extension of the submodular width whose corresponding algorithm can be seen as combining techniques used by PANDA with fast matrix multiplication.

As mentioned, Fan, Koutris, and Zhao [14] presented techniques for lower bounding this discussed exponent and, for a number of queries, demonstrated (essentially) that the submodular width serves as a lower bound for combinatorial algorithms; these results thus in a sense show that, on these queries, PANDA and Jaguar yield optimal running times—as combinatorial algorithms. Related lower bound technology was also developed by Bringmann and Gorbachev [11], who studied queries Q where $\mathsf{subw}(Q) < 2$ and all relations are binary, although the queries they study are *aggregate* conjunctive queries over the tropical semiring $(\min, +)$. In some sense, using the tropical semiring is another way to rule out the use of fast matrix multiplication, thus enforcing combinatorial algorithms. These two works demonstrated lower bounds under established assumptions from fine-grained complexity. Fan, Koutris, and Zhao [15] presented a different form of lower bounds, namely, circuit lower bounds, relative to a measure related to the submodular width; we discuss this work further in Section 7.

Organization. Preliminaries and notation are given in Section 2. Section 3 explains a key technical lemma that will be at the heart of the new Jaguar algorithm. In Section 4, we present a warmup example that demonstrates the main ideas and intuition behind the algorithm. In Section 5, we explain the general Jaguar algorithm in detail. We prove its correctness and analyze its runtime in Section 6. Finally, in Section 7, we conclude with a recap and some open problems.

2 Preliminaries

Given a function $f : A \to B$, a value $a \in A$, and a value $b \in B$, we define $f[a \to b]$ to be a new function $f' : A \to B$ such that

$$f'(x) \stackrel{\text{def}}{=} \begin{cases} b, & \text{if } x = a \\ f(x), & \text{otherwise} \end{cases} \tag{1}$$

Throughout the paper, we *fix* a finite set of variables $\boldsymbol{V}$ where each variable $X \in \boldsymbol{V}$ takes values from an infinite domain Dom. We use a capital letter X to denote a variable and a small letter x to

denote a value of the corresponding variable X. We use a bold capital letter $\boldsymbol{X}$ to denote a set of variables and a bold small letter $\boldsymbol{x}$ to denote a tuple of values for the variables in $\boldsymbol{X}$. We use $\mathsf{Dom}^{\boldsymbol{X}}$ to denote the set of all possible tuples of values $\boldsymbol{x}$ for the variables in $\boldsymbol{X}$. Given a tuple $\boldsymbol{x} \in \mathsf{Dom}^{\boldsymbol{X}}$ and a subset $\boldsymbol{Y} \subseteq \boldsymbol{X}$, we use $\pi_{\boldsymbol{Y}}(\boldsymbol{x})$ to denote the *projection of* $\boldsymbol{x}$ *onto* $\boldsymbol{Y}$, which is the tuple of values for the variables in $\boldsymbol{Y}$ that are consistent with $\boldsymbol{x}$.

Database Instances. A database instance $\mathcal{D}$ is a finite set of relation instances $\{R_1^{\mathcal{D}}(\boldsymbol{X}_1), \ldots, R_m^{\mathcal{D}}(\boldsymbol{X}_m)\}$, where each relation instance $R_i^{\mathcal{D}}(\boldsymbol{X}_i) \subseteq \mathsf{Dom}^{\boldsymbol{X}_i}$ is a finite set of tuples over the variables $\boldsymbol{X}_i \subseteq \boldsymbol{V}$. When the database instance $\mathcal{D}$ is clear from the context, we drop the superscript and write $R_i(\boldsymbol{X}_i)$ instead of $R_i^{\mathcal{D}}(\boldsymbol{X}_i)$. The *size* of a database instance $\mathcal{D}$, denoted $|\mathcal{D}|$, is defined as the total number of tuples in all relation instances in $\mathcal{D}$, i.e., $|\mathcal{D}| \stackrel{\text{def}}{=} \sum_{i=1}^{m} |R_i(\boldsymbol{X}_i)|$. A database instance $\mathcal{D}$ is called *signature-unique* if, for every $\boldsymbol{X} \subseteq \boldsymbol{V}$, $\mathcal{D}$ contains at most one relation instance $R(\boldsymbol{X})$. Note that according to this definition, a signature-unique database instance $\mathcal{D}$ can still contain two relation instances $R_1(\boldsymbol{X}_1)$ and $R_2(\boldsymbol{X}_2)$ where $\boldsymbol{X}_1 \subset \boldsymbol{X}_2$. Given a signature-unique database instance $\mathcal{D}$ and a relation instance $R(\boldsymbol{X})$ where $\boldsymbol{X} \subseteq \boldsymbol{V}$, we define $\mathcal{D}$ *augmented with* $R(\boldsymbol{X})$, denoted $\mathcal{D} \uplus \{R(\boldsymbol{X})\}$, to be a new database instance that results from adding $R(\boldsymbol{X})$ to $\mathcal{D}$, if no $R'(\boldsymbol{X})$ is already in $\mathcal{D}$, or, otherwise, by replacing the existing $R'(\boldsymbol{X})$ with its intersection with $R(\boldsymbol{X})$:

$$\mathcal{D} \uplus \{R(\boldsymbol{X})\} \stackrel{\text{def}}{=} \begin{cases} \mathcal{D} \cup \{R(\boldsymbol{X})\}, & \text{if there is no } R'(\boldsymbol{X}) \in \mathcal{D} \\ (\mathcal{D} \setminus \{R'(\boldsymbol{X})\}) \cup \{R(\boldsymbol{X}) \cap R'(\boldsymbol{X})\}, & \text{if there is } R'(\boldsymbol{X}) \in \mathcal{D} \end{cases} \tag{2}$$

Note that $\mathcal{D} \uplus \{R(\boldsymbol{X})\}$ is also signature-unique.

We use the standard notions of *projection* and *selection* of relations. In particular, given a relation instance $R(\boldsymbol{Z})$ and a subset $\boldsymbol{X} \subseteq \boldsymbol{Z}$, we define the *projection of* $R(\boldsymbol{Z})$ *onto* $\boldsymbol{X}$, denoted $\pi_{\boldsymbol{X}}(R)$, as the relation instance over $\boldsymbol{X}$ that contains the projections of all tuples in R onto $\boldsymbol{X}$. Moreover, given a tuple $\boldsymbol{x} \in \mathsf{Dom}^{\boldsymbol{X}}$, we use $\sigma_{\boldsymbol{X}=\boldsymbol{x}}(R)$ to denote the *selection of* R *with* $\boldsymbol{X} = \boldsymbol{x}$, which is the relation instance over $\boldsymbol{Z}$ that contains all tuples $\boldsymbol{z}$ in R that are consistent with $\boldsymbol{x}$, i.e., where $\pi_{\boldsymbol{X}}(\boldsymbol{z}) = \boldsymbol{x}$. Given a relation instance $R(\boldsymbol{Z})$, two sets of variables $\boldsymbol{X}, \boldsymbol{Y} \subseteq \boldsymbol{Z}$, and a tuple $\boldsymbol{x}$ for $\boldsymbol{X}$, we define the *degree* of $\boldsymbol{Y}$ given $\boldsymbol{X} = \boldsymbol{x}$ in R, denoted $\deg_R(\boldsymbol{Y}|\boldsymbol{X} = \boldsymbol{x})$, and the *degree* of $\boldsymbol{Y}$ given $\boldsymbol{X}$ in R, denoted $\deg_R(\boldsymbol{Y}|\boldsymbol{X})$, as follows, respectively:

$$\deg_R(\boldsymbol{Y}|\boldsymbol{X} = \boldsymbol{x}) \stackrel{\text{def}}{=} |\pi_{\boldsymbol{Y}}(\sigma_{\boldsymbol{X}=\boldsymbol{x}}(R))|$$

$$\deg_R(\boldsymbol{Y}|\boldsymbol{X}) \stackrel{\text{def}}{=} \max_{\boldsymbol{x} \in \mathsf{Dom}^{\boldsymbol{X}}} \deg_R(\boldsymbol{Y}|\boldsymbol{X} = \boldsymbol{x})$$

Conjunctive Queries. A conjunctive query (CQ) Q is specified with a set of variables $\boldsymbol{V}$, a set of *free variables* $\boldsymbol{F}$, and a set of atoms $\mathsf{atoms}(Q)$, and has the form:

$$Q(\boldsymbol{F}) \text{ :- } \bigwedge_{R(\boldsymbol{X}) \in \mathsf{atoms}(Q)} R(\boldsymbol{X}), \tag{3}$$

where $\boldsymbol{V} = \bigcup_{R(\boldsymbol{X}) \in \mathsf{atoms}(Q)} \boldsymbol{X}$. The query Q is called *Boolean* if $\boldsymbol{F} = \emptyset$ and *full* if $\boldsymbol{F} = \boldsymbol{V}$. A database instance $\mathcal{D}$ is said to *include the schema* of Q if, for every atom $R(\boldsymbol{X}) \in \mathsf{atoms}(Q)$, $\mathcal{D}$ contains a corresponding relation instance $R^{\mathcal{D}}(\boldsymbol{X})$. A database instance $\mathcal{D}$ is said to *match the schema* of Q if it includes the schema of Q *and* contains no other relation instances. We adopt the standard semantics for evaluating a CQ Q on a database instance $\mathcal{D}$ that includes the schema of Q, and we use $Q(\mathcal{D})$ to denote the evaluation result. Throughout the paper, we use *data complexity*, i.e., we assume that the query Q is fixed, hence its size is a constant, and we measure the complexity of evaluating Q only in terms of the size, N, of the input database instance $\mathcal{D}$.

Statistics and Polymatroids. Using definitions and notation from [6], we define a collection of *statistics* over a variable set $\boldsymbol{V}$ to be a pair $(\Delta, \boldsymbol{n})$, where:

- Δ is a set of *conditional terms* of the form $\boldsymbol{Y}|\boldsymbol{X}$ where $\boldsymbol{X}, \boldsymbol{Y} \subseteq \boldsymbol{V}$.
- $\boldsymbol{n} : \Delta \to \mathbb{R}_+$ maps each term $\boldsymbol{Y}|\boldsymbol{X}$ in Δ to a non-negative real number $n_{\boldsymbol{Y}|\boldsymbol{X}}$.

Let $\mathcal{D}$ be a database instance of size N. The instance $\mathcal{D}$ is said to *satisfy* the statistics $(\Delta, \boldsymbol{n})$, denoted $\mathcal{D} \models (\Delta, \boldsymbol{n})$, if for every conditional term $(\boldsymbol{Y}|\boldsymbol{X}) \in \Delta$, $\mathcal{D}$ contains a relation $R(\boldsymbol{Z})$, called the *guard* of $\boldsymbol{Y}|\boldsymbol{X}$, that has variables $\boldsymbol{Z} \supseteq \boldsymbol{X} \cup \boldsymbol{Y}$ and satisfies:

$$\log_N \deg_R(\boldsymbol{Y}|\boldsymbol{X}) \leq n_{\boldsymbol{Y}|\boldsymbol{X}} \tag{4}$$

Throughout the paper, *all* logarithms are base N where N is the size of the input database instance $\mathcal{D}$. This is in contrast to [6], where logarithms are base 2.

A set function $\boldsymbol{h} : 2^{\boldsymbol{V}} \to \mathbb{R}_+$ is called a *polymatroid* if it satisfies *Shannon inequalities*:

$$h(\emptyset) = 0 \tag{5}$$

$$h(\boldsymbol{X}) \leq h(\boldsymbol{Y}), \quad \forall \boldsymbol{X} \subseteq \boldsymbol{Y} \subseteq \boldsymbol{V}, \qquad \text{(Monotonicity)} \tag{6}$$

$$h(\boldsymbol{X}) + h(\boldsymbol{Y}) \geq h(\boldsymbol{X} \cap \boldsymbol{Y}) + h(\boldsymbol{X} \cup \boldsymbol{Y}), \quad \forall \boldsymbol{X}, \boldsymbol{Y} \subseteq \boldsymbol{V}, \qquad \text{(Submodularity)} \tag{7}$$

A set function $\boldsymbol{h} : 2^{\boldsymbol{V}} \to \mathbb{R}_+ \cup \{\infty\}$ is said to *satisfy* given statistics $(\Delta, \boldsymbol{n})$, denoted $\boldsymbol{h} \models (\Delta, \boldsymbol{n})$, if for every conditional term $\boldsymbol{Y}|\boldsymbol{X} \in \Delta$, we have $h(\boldsymbol{Y}|\boldsymbol{X}) \stackrel{\text{def}}{=} h(\boldsymbol{Y} \cup \boldsymbol{X}) - h(\boldsymbol{X}) \leq n_{\boldsymbol{Y}|\boldsymbol{X}}$. (In general, we deal with arithmetic expressions and comparisons involving infinity in the standard way. So here, we understand that this condition fails when $h(\boldsymbol{Y} \cup \boldsymbol{X}) = \infty$, and that it holds when $h(\boldsymbol{Y} \cup \boldsymbol{X}) \neq \infty = h(\boldsymbol{X})$.)

Tree decompositions. A *tree decomposition* of Q is a pair (T, χ), where T is a tree and $\chi : \text{nodes}(T) \to 2^{\boldsymbol{V}}$ is a map from the nodes of T to subsets of $\boldsymbol{V}$ that satisfies the following properties: for every atom $R(\boldsymbol{X}) \in \text{atoms}(Q)$, there is a node $t \in \text{nodes}(T)$ such that $\boldsymbol{X} \subseteq \chi(t)$; and, for every variable $X \in \boldsymbol{V}$, the set $\{t \mid X \in \chi(t)\}$ forms a connected sub-tree of T. Each set $\chi(t)$ is called a *bag* of the tree decomposition, and we will assume w.l.o.g. that the bags are distinct, i.e. $\chi(t) \neq \chi(t')$ when $t \neq t'$ are nodes in T. A *free-connex* tree decomposition of Q is a tree decomposition of Q with the additional property that there is a connected subtree T' of T that contains all the free variables and no other variables [2, 9], i.e., $\bigcup_{t \in \text{nodes}(T')} \chi(t) = \boldsymbol{F}$. Let $\mathcal{T}(Q)$ be the set of *free-connex* tree decompositions of Q. (If the query is Boolean or full, i.e., if $\boldsymbol{F} = \emptyset$ or $\boldsymbol{F} = \boldsymbol{V}$, then $\mathcal{T}(Q)$ is the set of *all* tree decompositions of Q. Readers only interested in Boolean or full conjunctive queries can thus ignore the free-connex notion.)

The Submodular Width. The submodular width was originally introduced by Daniel Marx [24] and was later generalized in [6]. We adopt the generalized definition from [6]. In particular, given a query Q and statistics $(\Delta, \boldsymbol{n})$, we define the *submodular width* of Q w.r.t. the statistics $(\Delta, \boldsymbol{n})$ as:

$$\text{subw}(Q, \Delta, \boldsymbol{n}) \stackrel{\text{def}}{=} \max_{\substack{\text{Polymatroid } \boldsymbol{h} \\ \boldsymbol{h} \models (\Delta, \boldsymbol{n})}} \min_{(T,\chi) \in \mathcal{T}(Q)} \max_{t \in \text{nodes}(T)} h(\chi(t)) \tag{8}$$

The above definition generalizes the original definition of Daniel Marx [24] in two aspects:

- It accounts for arbitrary statistics $(\Delta, \boldsymbol{n})$, as opposed to the special case of "edge-domination" constraints[2] from [24].
- It handles arbitrary conjunctive queries (that are not necessarily Boolean or full) by using the notion of *free-connex* tree decompositions.

[2]Edge-domination constraints are restricted to the form $h(\boldsymbol{X}) \leq 1$ for every atom $R(\boldsymbol{X})$ in the query.

Remark 2.1. Since in Eq. (4), we use logarithms base N where N is the size of the input database instance $\mathcal{D}$, our runtimes later will be stated in terms of $N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})}$. This is more consistent with the original definition of the submodular width [24]. However, it deviates from [6] where logarithms were base 2, hence runtimes were stated in terms of $2^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})}$.

Given a conjunctive query Q with variables $\boldsymbol{V}$ and tree decompositions $\mathcal{T}(Q)$, a *bag selector* $\mathcal{S}$ is a subset of $2^{\boldsymbol{V}}$ that consists of a bag $\chi(t)$ out of every tree decomposition $(T,\chi) \in \mathcal{T}(Q)$. Let $\mathsf{BS}(Q)$ denote the set of all bag selectors of Q. By distributing the min over the inner max in Eq. (8), we can rewrite the submodular width as follows:

$$\begin{aligned} \mathsf{subw}(Q,\Delta,\boldsymbol{n}) &= \max_{\substack{\text{Polymatroid } \boldsymbol{h} \\ \boldsymbol{h}\models(\Delta,\boldsymbol{n})}} \max_{\mathcal{S}\in\mathsf{BS}(Q)} \min_{\boldsymbol{Z}\in\mathcal{S}} h(\boldsymbol{Z}) \\ &= \max_{\mathcal{S}\in\mathsf{BS}(Q)} \max_{\substack{\text{Polymatroid } \boldsymbol{h} \\ \boldsymbol{h}\models(\Delta,\boldsymbol{n})}} \min_{\boldsymbol{Z}\in\mathcal{S}} h(\boldsymbol{Z}) \end{aligned} \tag{9}$$

Constant-delay enumeration. Given a query Q and a database instance $\mathcal{D}$ that includes the schema of Q, *constant-delay enumeration (CDE)* refers to enumerating tuples in the output $Q(\mathcal{D})$ one by one without repetitions while keeping the *delay* constant. The delay is the maximum of three times: the time to output the first tuple, the time between any two consecutive tuples, and the time after outputting the last tuple until the end of enumeration. See [25] for a survey.

3 Truncation Lemma

Before describing the Jaguar algorithm, we present a key technical lemma that is at the heart of this algorithm's analysis. In essence, this lemma converts a monotone set function $\boldsymbol{g}$ to a polymatroid $\boldsymbol{h}$. The polymatroid $\boldsymbol{h}$ is constructed by taking the lowest value c at which $\boldsymbol{g}$ violates submodularity, and truncating the function $\boldsymbol{g}$ above c, so that c becomes the maximum value after truncation. The truncated $\boldsymbol{h}$ no longer violates submodularity, hence is a polymatroid. But now, we can utilize $\boldsymbol{h}$ in the context of the submodular width as follows. By construction, the polymatroid $\boldsymbol{h}$ has a maximum value of c, and, in turn, the submodular width is defined as a maximum over polymatroids. Hence, the lemma shows that c is upper bounded by the submodular width, under a certain condition.

Given a query Q over variable set $\boldsymbol{V}$ and a tree decomposition $(T,\chi) \in \mathcal{T}(Q)$, a set $\mathcal{I} \subseteq 2^{\boldsymbol{V}}$ is said to *cover* the tree decomposition (T,χ) if every bag $\chi(t)$ of (T,χ) is in $\mathcal{I}$.

Lemma 3.1 (Truncation lemma). *Let $\boldsymbol{g} : 2^{\boldsymbol{V}} \to \mathbb{R}_+ \cup \{\infty\}$ be a monotone set function satisfying $g(\emptyset) = 0$. Define the following:*[3]

$$f(\boldsymbol{X},\boldsymbol{Y}) \stackrel{def}{=} g(\boldsymbol{X}) + g(\boldsymbol{Y}) - g(\boldsymbol{X}\cap\boldsymbol{Y}), \qquad \forall \boldsymbol{X},\boldsymbol{Y} \subseteq \boldsymbol{V} \tag{10}$$

$$c \stackrel{def}{=} \min_{\substack{\boldsymbol{X},\boldsymbol{Y}\subseteq\boldsymbol{V} \\ g(\boldsymbol{X}\cup\boldsymbol{Y})>f(\boldsymbol{X},\boldsymbol{Y})}} f(\boldsymbol{X},\boldsymbol{Y}) \tag{11}$$

$$h(\boldsymbol{X}) \stackrel{def}{=} \min(g(\boldsymbol{X}), c), \qquad \forall \boldsymbol{X} \subseteq \boldsymbol{V} \tag{12}$$

$$\mathcal{I} \stackrel{def}{=} \{\boldsymbol{X} \subseteq \boldsymbol{V} \mid g(\boldsymbol{X}) \leq c\} \tag{13}$$

Then, the following claims hold:

(1) The function $\boldsymbol{h}$ is a polymatroid that satisfies $\boldsymbol{h} \leq \boldsymbol{g}$ and $\boldsymbol{h} \leq c$.

[3]Note that f can be undefined, for example when $g(\boldsymbol{X}) = g(\boldsymbol{X}\cap\boldsymbol{Y}) = \infty$. We follow the convention that the minimum over an empty set is ∞, hence $c \in \mathbb{R}_+ \cup \{\infty\}$.

(2) Let Q be a CQ with variable set $\boldsymbol{V}$ and let $(\Delta, \boldsymbol{n})$ be a collection of statistics over $\boldsymbol{V}$ such that $\boldsymbol{g} \models (\Delta, \boldsymbol{n})$. Then, $\boldsymbol{h} \models (\Delta, \boldsymbol{n})$. In addition, if $\mathcal{I}$ does not cover any tree decomposition of Q, then $c \leq subw(Q, \Delta, \boldsymbol{n})$.

Proof. First, we prove Claim (1). It is straightforward to verify that the function $\boldsymbol{h}$ always takes on values in $\mathbb{R}_+$ and that $\boldsymbol{h}$ is upper bounded by both $\boldsymbol{g}$ and c. Since $\boldsymbol{g}$ is monotone and satisfies $g(\emptyset) = 0$, the function $\boldsymbol{h}$ is also monotone and satisfies $h(\emptyset) = 0$. It remains to show that $\boldsymbol{h}$ is submodular, that is, for every $\boldsymbol{X}, \boldsymbol{Y} \subseteq \boldsymbol{V}$, we have $h(\boldsymbol{X} \cup \boldsymbol{Y}) \leq h(\boldsymbol{X}) + h(\boldsymbol{Y}) - h(\boldsymbol{X} \cap \boldsymbol{Y})$. We recognize the following cases:

- Case 1: $g(\boldsymbol{X}), g(\boldsymbol{Y}) \leq c$. This implies that $h(\boldsymbol{X}) = g(\boldsymbol{X})$, $h(\boldsymbol{Y}) = g(\boldsymbol{Y})$, and $h(\boldsymbol{X} \cap \boldsymbol{Y}) = g(\boldsymbol{X} \cap \boldsymbol{Y})$. If $g(\boldsymbol{X} \cup \boldsymbol{Y}) \leq f(\boldsymbol{X}, \boldsymbol{Y})$, then $h(\boldsymbol{X} \cup \boldsymbol{Y}) \leq g(\boldsymbol{X} \cup \boldsymbol{Y}) \leq f(\boldsymbol{X}, \boldsymbol{Y})$ and we are done. Otherwise, by definition of c, we have $h(\boldsymbol{X} \cup \boldsymbol{Y}) = c \leq f(\boldsymbol{X}, \boldsymbol{Y})$.
- Case 2: $g(\boldsymbol{X}) \leq c$ and $g(\boldsymbol{Y}) > c$. This implies that $g(\boldsymbol{X} \cap \boldsymbol{Y}) \leq c$ and $g(\boldsymbol{X} \cup \boldsymbol{Y}) > c$. Therefore, $h(\boldsymbol{X} \cup \boldsymbol{Y}) \leq h(\boldsymbol{X}) + h(\boldsymbol{Y}) - h(\boldsymbol{X} \cap \boldsymbol{Y})$ holds because $h(\boldsymbol{X} \cup \boldsymbol{Y}) = h(\boldsymbol{Y}) = c$ and monotonicity implies that $h(\boldsymbol{X} \cap \boldsymbol{Y}) \leq h(\boldsymbol{X})$.
- Case 3: $g(\boldsymbol{Y}) \leq c$ and $g(\boldsymbol{X}) > c$. Symmetric to Case 2.
- Case 4: $g(\boldsymbol{X}), g(\boldsymbol{Y}) > c$. This implies that $g(\boldsymbol{X} \cup \boldsymbol{Y}) > c$ and $h(\boldsymbol{X} \cup \boldsymbol{Y}) = h(\boldsymbol{X}) = h(\boldsymbol{Y}) = c$. Since $h(\boldsymbol{X} \cap \boldsymbol{Y}) \leq c$ by definition of $\boldsymbol{h}$, the submodularity holds.

Now, we prove Claim (2). We start by showing that $\boldsymbol{h} \models (\Delta, \boldsymbol{n})$. For every $\boldsymbol{X}, \boldsymbol{Y} \subseteq \boldsymbol{V}$, we have

$$\begin{aligned} h(\boldsymbol{Y}|\boldsymbol{X}) &\stackrel{\text{def}}{=} h(\boldsymbol{Y} \cup \boldsymbol{X}) - h(\boldsymbol{X}) \\ &= \min(g(\boldsymbol{X} \cup \boldsymbol{Y}), c) - \min(g(\boldsymbol{X}), c) \\ &\leq g(\boldsymbol{Y} \cup \boldsymbol{X}) - g(\boldsymbol{X}) && \text{(By monotonicity of } \boldsymbol{g}) \\ &\leq n_{Y|X} && \text{(Because } \boldsymbol{g} \models (\Delta, \boldsymbol{n})) \end{aligned}$$

This proves that $\boldsymbol{h} \models (\Delta, \boldsymbol{n})$. Now we prove that $c \leq \text{subw}(Q, \Delta, \boldsymbol{n})$. If $\mathcal{I}$ does not contain all the bags of any tree decomposition in $\mathcal{T}(Q)$, then the complement of $\mathcal{I}$, denoted $\overline{\mathcal{I}} \stackrel{\text{def}}{=} 2^{\boldsymbol{V}} \setminus \mathcal{I}$, must contain at least one bag of every tree decomposition in $\mathcal{T}(Q)$. Hence, $\overline{\mathcal{I}}$ must be a superset of some *bag selector* $\mathcal{S}^* \in \text{BS}(Q)$. By Eq. (9), we have:

$$\begin{aligned} \text{subw}(Q, \Delta, \boldsymbol{n}) &= \max_{\substack{\text{Polymatroid } \boldsymbol{h} \\ \boldsymbol{h} \models (\Delta, \boldsymbol{n})}} \max_{\mathcal{S} \in \text{BS}(Q)} \min_{Z \in \mathcal{S}} h(Z) \\ &\geq \max_{\mathcal{S} \in \text{BS}(Q)} \min_{Z \in \mathcal{S}} h(Z) && \text{(By choosing } \boldsymbol{h} \text{ from Eq. (12))} \\ &\geq \min_{Z \in \mathcal{S}^*} h(Z) && \text{(By choosing } \mathcal{S}^*) \\ &= c && \text{(because } h(Z) = c \text{ for every } Z \in \overline{\mathcal{I}} \supseteq \mathcal{S}^*) \end{aligned}$$

□

4 Warmup Example

Before presenting the full Jaguar algorithm in Section 5, we start in this section with a warmup example that demonstrates the main ideas behind the algorithm. Consider the following CQ $Q_\square$:

$$Q_\square(X, Y, Z, W) \text{ :- } R(X, Y), S(Y, Z), T(Z, W), U(W, X). \tag{14}$$

The query $Q_\square$ is known to have a *fractional hypertree width* [20] of 2 and a submodular width of 3/2 [5, 8]. In addition to the trivial tree decomposition that puts all variables in one bag, $Q_\square$ has two free-connex tree decompositions:

- $\mathcal{T}_1$ with bags $\{X, Y, Z\}$ and $\{Z, W, X\}$, and

- $\mathcal{T}_2$ with bags $\{W, X, Y\}$ and $\{Y, Z, W\}$.

The query and both tree decompositions are depicted in Figure 1. Let N be an *even* number. Consider the following input database instance $\mathcal{D}_\square$, where $[N]$ is a shorthand for $\{1, 2, \ldots, N\}$:

$$R = S = T = U = ([N/2] \times \{1\}) \cup (\{1\} \times [N/2])$$

Note that no matter which tree decomposition we choose, there is always a bag whose size is $\Omega(N^2)$ over this database instance. For example, the bag $\{X, Y, Z\}$ of $\mathcal{T}_1$ requires computing the join $R(X, Y) \bowtie S(Y, Z)$, which has size $\Omega(N^2)$. This is consistent with the fact that the fractional hypertree width of $Q_\square$ is 2: Using a single tree decomposition, we cannot beat the $\Omega(N^2)$ runtime in the worst-case. We show below how the Jaguar algorithm can achieve a runtime of $O(N)$ on the specific input database instance $\mathcal{D}_\square$ by utilizing both tree decompositions $\mathcal{T}_1$ and $\mathcal{T}_2$, thus mimicking the algorithm from [8]. We focus on this database instance for the purpose of illustrating our algorithm; note that the general time bound achieved by our algorithm on the query $Q_\square$ is $O(N^{3/2+\epsilon})$, as this query's submodular width is 3/2.

At a high level, the Jaguar algorithm starts by initializing a monotone set function $\boldsymbol{g} : 2^{\boldsymbol{V}} \to \mathbb{R}_+ \cup \{\infty\}$ where $\boldsymbol{V} := \{X, Y, Z, W\}$ in this example using the sizes of the input relations and their projections on $\log_N$-scale. In particular, the function $\boldsymbol{g}$ is initialized as follows:

$$\begin{aligned} g(XY) = g(YZ) = g(ZW) = g(WX) &= \log_N N = 1, \\ g(X) = g(Y) = g(Z) = g(W) &= \log_N N = 1, \\ g(\emptyset) &= \log_N 1 = 0, \\ g(\boldsymbol{X}) &= \infty \qquad \text{otherwise.} \end{aligned}$$

Now, we find the "smallest" violation of submodularity by $\boldsymbol{g}$. Specifically, we look for a violated submodularity $g(\boldsymbol{X} \cup \boldsymbol{Y}) > f(\boldsymbol{X}, \boldsymbol{Y}) \stackrel{\text{def}}{=} g(\boldsymbol{X}) + g(\boldsymbol{Y}) - g(\boldsymbol{X} \cap \boldsymbol{Y})$ that minimizes $f(\boldsymbol{X}, \boldsymbol{Y})$, in the same spirit as in Lemma 3.1. Breaking ties arbitrarily, we could pick $g(XYZ) = \infty > g(XY) + g(YZ) - g(Y) = 1$. In order to fix this submodularity, we need to lower the value of $g(XYZ)$ from ∞ down to 1 (or less). To translate this change in the function $\boldsymbol{g}$ into the database world, we need to be able compute a relation over $\{X, Y, Z\}$ of size at most $N^1 = N$. Note that $g(Z|Y) \stackrel{\text{def}}{=} g(YZ) - g(Y) = 0$. We interpret this as saying that we can only process a Y-value in S if it has at most one matching Z-value. We call such Y-values *light*, and the rest of the Y-values *heavy*. Now, the execution of the algorithm branches into two corresponding branches:

- On the *light-Y* branch, we compute the join $R(X, Y) \bowtie S(Y, Z)$ but *only* over Y-values in S that are *light*, i.e., have at most one matching Z-value in S. The size of this join is at most N, thus allowing us to lower the value of $g(XYZ)$ down to $\log_N N = 1$. Thanks to this change, the submodularity is fixed $g(XYZ) = 1 \le g(XY) + g(YZ) - g(Y) = 1$. Now we can look for the next "smallest" violation among the remaining submodularities. Breaking ties arbitrarily again, suppose now we pick $g(ZWX) = \infty > g(ZW) + g(WX) - g(W) = 1$. Just like before, now we partition the W-values into light and heavy based on the number of matching X-values in $U(W, X)$, and we create two corresponding branches:
  - On the light-W branch, we compute the join $T(Z, W) \bowtie U(W, X)$ over only the light W-values, hence this join has size at most N. After we compute this join, we now have two relations over $\{X, Y, Z\}$ and $\{Z, W, X\}$ corresponding to the two bags of $\mathcal{T}_1$. We can enumerate the output of $Q_\square$ by running Yannakakis algorithm [26] over $\mathcal{T}_1$ in time $O(\mathsf{OUT})$, where OUT is the output size for this particular branch.
  - On the heavy-W branch, we cannot afford to compute the join $T(Z, W) \bowtie U(W, X)$ over the heavy W-values, thus we are not allowed to lower the value of $g(ZWX)$ down to 1.

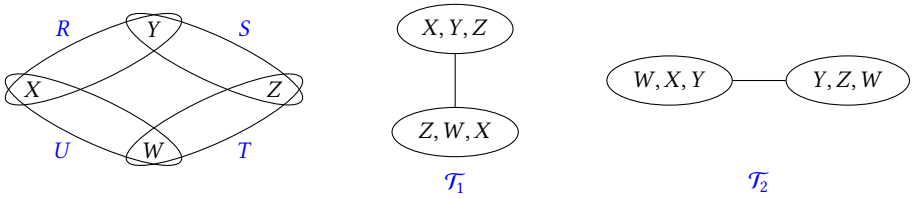


Fig. 1. Query $Q_\square$ from Eq. (14) along with the two free-connex tree decompositions.

Instead, we take advantage of the fact that the number of heavy W-values is very small, namely 1 in this particular database instance $\mathcal{D}_\square$. This allows us to lower the value of $g(W)$ down to $\log_N 1 = 0$. Note that this change does not really fix the submodularity $g(ZWX) = \infty > g(ZW) + g(WX) - g(W) = 2$. Nevertheless, we can now look for the "smallest" submodularity violation in the resulting $\boldsymbol{g}$. Let's say we pick $g(YZW) = \infty \geq g(YZ) + g(W) = 1$. To fix this submodularity, we need to lower the value of $g(YZW)$ down to 1 (or less). In the database world, we need to be able to mirror this change by computing a relation over $\{Y, Z, W\}$ of size at most $N^1 = N$. Luckily, we can compute this directly by expanding tuples in $S(Y, Z)$ with the single heavy W-value. Now we look for another submodularity violation, let's say we pick $g(WXY) = \infty > g(W) + g(XY) = 1$. Just like before, we need to lower the value of $g(WXY)$ down to 1, which corresponds to computing a relation over $\{W, X, Y\}$ of size at most N^1, which is again easily doable by expanding $R(X, Y)$ with the heavy W-value. Now, we have two relations over $\{W, X, Y\}$ and $\{Y, Z, W\}$ corresponding to the two bags of $\mathcal{T}_2$. We run Yannakakis algorithm over $\mathcal{T}_2$.

- On the *heavy* Y-branch, we *cannot* afford to compute the join $R(X, Y) \bowtie S(Y, Z)$ over the heavy Y-values, thus we are *not* allowed to lower the value of $g(XYZ)$ down to 1. On this branch, we cannot make progress by fixing the violated submodularity $g(XYZ) = \infty > g(XY) + g(YZ) - g(Y) = 1$. Instead, we will have to make progress by utilizing something else: The number of heavy Y-values is very small, namely 1 in this example. Instead of lowering the value of $g(XYZ)$ as we did in the light-Y branch, here we will lower the value of $g(Y)$ down to $\log_N 1 = 0$, which corresponds to processing the single heavy Y-value. Note that the submodularity will remain violated $g(XYZ) = \infty > g(XY) + g(YZ) - g(Y) = 2$. However, now we can pick another submodularity to fix, for example $g(YZW) > g(Y) + g(ZW) = 1$. To fix this submodularity, we need to lower the value of $g(YZW)$ down to 1, which corresponds to computing a relation over $\{Y, Z, W\}$ of size at most N^1. This in turn is easily doable by expanding $T(Z, W)$ with the single heavy Y-value. After this change, we can pick another submodularity violation, for example $g(WXY) > g(WX) + g(Y) = 1$. This is also easily doable by expanding $U(W, X)$ with the single heavy Y-value. Now we have two relations over $\{W, X, Y\}$ and $\{Y, Z, W\}$ corresponding to the two bags of $\mathcal{T}_2$, and we can run Yannakakis algorithm over $\mathcal{T}_2$.

In all branches above, the Jaguar algorithm reports the output of $Q_\square$ in time $O(N + \mathsf{OUT})$ over the input database instance $\mathcal{D}_\square$, where OUT is the output size. As mentioned before, over an arbitrary input database instance, the runtime of the Jaguar algorithm on $Q_\square$ is $O(N^{3/2+\epsilon} + \mathsf{OUT})$.

5 The Algorithm

We now present the Jaguar algorithm for evaluating a CQ Q over a database instance $\mathcal{D}$.

5.1 Parameters, Inputs, and Outputs

The Jaguar algorithm is depicted in Algorithm 1. It is parameterized by the following:

- A constant $\epsilon > 0$.
- A CQ Q (Eq. (3)) that has variables $\boldsymbol{V}$ and free variables $\boldsymbol{F} \subseteq \boldsymbol{V}$.

Jaguar takes the following inputs:

- A collection of statistics $(\Delta, \boldsymbol{n})$ over $\boldsymbol{V}$.
- A database instance $\mathcal{D}$, that *includes the schema* of Q (Section 2) and satisfies the statistics $(\Delta, \boldsymbol{n})$, that is, $\mathcal{D} \models (\Delta, \boldsymbol{n})$. Without loss of generality, we assume that $\mathcal{D}$ is *signature-unique*.[4] Moreover, we assume that $\mathcal{D}$ is over the same set $\boldsymbol{V}$ of variables of Q, i.e., for every $R(\boldsymbol{X}) \in \mathcal{D}$, we assume $\boldsymbol{X} \subseteq \boldsymbol{V}$.
- *[Optional argument]* A set function $\boldsymbol{g} : 2^{\boldsymbol{V}} \to \mathbb{R}_+ \cup \{\infty\}$ that satisfies the following invariants:
  - **Cardinality Invariant:** For every $\boldsymbol{X} \subseteq \boldsymbol{V}$, we must have:

$$\begin{aligned} &\boldsymbol{g}(\boldsymbol{X}) \geq \log_N |R(\boldsymbol{X})| && \text{if } R(\boldsymbol{X}) \in \mathcal{D}, \\ &\boldsymbol{g}(\boldsymbol{X}) = \infty && \text{otherwise.} \end{aligned}$$

  - **Monotonicity Invariant:** $\boldsymbol{g}$ is monotone and satisfies $\boldsymbol{g}(\emptyset) = 0$.
  - **Statistics Invariant:** $\boldsymbol{g} \models (\Delta, \boldsymbol{n})$.

Jaguar is a recursive algorithm, where ϵ, Q and the statistics $(\Delta, \boldsymbol{n})$ remain *the same* across all recursive calls, whereas the database instance $\mathcal{D}$ and the function $\boldsymbol{g}$ change from one recursive call to another. The top-level call to Jaguar is made with a database instance $\mathcal{D}_0$ that *matches the schema* of Q (Section 2), and this top-level call returns the answer of Q over $\mathcal{D}_0$, denoted $Q(\mathcal{D}_0)$. Every recursive call to Jaguar is made with a database instance $\mathcal{D}$, which is an *augmented* version of $\mathcal{D}_0$ (Section 2). In particular, such an instance $\mathcal{D}$ always includes the schema of Q, but does not necessarily match the schema of Q. To every recursive call to Jaguar over a database instance $\mathcal{D}$, we associate a CQ, $Q_{\mathcal{D}}$, whose body is the join of *all* relations in the current database instance $\mathcal{D}$, and whose free variables are $\boldsymbol{F}$, the free variables of the original query Q:

$$Q_{\mathcal{D}}(\boldsymbol{F}) \text{ :- } \bigwedge_{R(\boldsymbol{X}) \in \mathcal{D}} R(\boldsymbol{X}). \tag{15}$$

We will show later that every recursive call to Jaguar on a database instance $\mathcal{D}$ returns a set $\mathcal{A}$ of tuples over the variables $\boldsymbol{F}$ that satisfies the following:[5]

$$Q_{\mathcal{D}}(\mathcal{D}) \quad \subseteq \quad \mathcal{A} \quad \subseteq \quad Q(\mathcal{D}_0) \tag{16}$$

Throughout the algorithm, we use N to refer to the size of the *original* input database instance $\mathcal{D}_0$: $N \stackrel{\text{def}}{=} |\mathcal{D}_0|$. In particular, the value of N remains *the same* across all recursive calls to Jaguar.

[4] If there are multiple relations $R(\boldsymbol{X})$ over the same set of variables $\boldsymbol{X}$, we can replace them with their intersection.

[5] It might be tempting to claim that every recursive call returns $\mathcal{A} := Q_{\mathcal{D}}(\mathcal{D})$, but we will show later that this is not the case, and there is a fundamental reason for that. In particular, if this was the case, then we could have solved #CQ queries in time $O(N^{\text{subw}(Q)+\epsilon})$, but counting in submodular width time remains a hard open problem [2] and no prior algorithm seems to solve it [5, 6, 10, 24].

5.2 Main Algorithm Description

In this section, we describe the steps of the Jaguar algorithm which are depicted in Algorithm 1. As mentioned before, Jaguar allows a set function $\boldsymbol{g}$ as an optional input argument. In the case that this argument is not provided, the algorithm initializes $\boldsymbol{g}$ as follows:

$$g(\boldsymbol{X}) \stackrel{\text{def}}{=} \begin{cases} \log_N |R(\boldsymbol{X})| & \text{if } R(\boldsymbol{X}) \in \mathcal{D} \\ \infty & \text{otherwise} \end{cases} \tag{17}$$

The above $\boldsymbol{g}$ satisfies the cardinality invariant but not necessarily the other two invariants (the monotonicity and statistics invariants). In order to enforce these two invariants (while maintaining the first), Jaguar invokes a special subroutine, called Calibrate. This subroutine replaces both $\mathcal{D}$ and $\boldsymbol{g}$ with new database instance $\mathcal{D}'$ and set function $\boldsymbol{g}'$ that satisfy all three invariants.

The Jaguar algorithm then terminates if there is a tree decomposition $(T, \chi) \in \mathcal{T}(Q)$ that is *covered by* [6] the current database instance $\mathcal{D}$, which means[7] that for every bag $\chi(t)$ of (T, χ), there is a relation $R(\chi(t)) \in \mathcal{D}$. If this is the case, Jaguar runs the Yannakakis algorithm [26] over those relations $R(\chi(t))$ corresponding to the bags of (T, χ)—after semijoin reducing them with all other relations to remove spurious tuples—and returns the result.

If this termination condition is not satisfied yet, then Jaguar computes the constant c from Eq. (11) in Lemma 3.1 along with the sets $\boldsymbol{X}, \boldsymbol{Y} \subseteq \boldsymbol{V}$ that achieve the minimum in Eq. (11). Later on in the analysis, we will rely on Lemma 3.1 to show that $c \leq \text{subw}(Q, \Delta, \boldsymbol{n})$. Let $\boldsymbol{W} \stackrel{\text{def}}{=} \boldsymbol{X} \cap \boldsymbol{Y}$ and $\theta \stackrel{\text{def}}{=} N^{g(\boldsymbol{Y}) - g(\boldsymbol{W})}$. Now, the algorithm partitions the relation $R(\boldsymbol{Y})$ into two parts, $R^h(\boldsymbol{Y})$ and $R^\ell(\boldsymbol{Y})$, called the *heavy* and *light* parts, respectively, using a threshold $\theta \times N^\epsilon$ on the degree $\deg_R(\boldsymbol{Y}|\boldsymbol{W})$. In particular,

$$R^h(\boldsymbol{Y}) \stackrel{\text{def}}{=} \left\{ \boldsymbol{y} \in R(\boldsymbol{Y}) \quad | \quad \deg_{R(\boldsymbol{Y})}(\boldsymbol{Y}|\boldsymbol{W} = \pi_{\boldsymbol{W}}(\boldsymbol{y})) > \theta \times N^\epsilon \right\} \tag{18}$$

$$R^\ell(\boldsymbol{Y}) \stackrel{\text{def}}{=} \left\{ \boldsymbol{y} \in R(\boldsymbol{Y}) \quad | \quad \deg_{R(\boldsymbol{Y})}(\boldsymbol{Y}|\boldsymbol{W} = \pi_{\boldsymbol{W}}(\boldsymbol{y})) \leq \theta \times N^\epsilon \right\} \tag{19}$$

In the light part $R^\ell(\boldsymbol{Y})$, every tuple $\boldsymbol{w} \in \pi_{\boldsymbol{W}}(R^\ell(\boldsymbol{Y}))$ has degree at most $\theta \times N^\epsilon$. We partition $R^\ell(\boldsymbol{Y})$ further into $\lceil N^\epsilon \rceil$ parts $R^{\ell,1}(\boldsymbol{Y}), \ldots, R^{\ell,\lceil N^\epsilon \rceil}(\boldsymbol{Y})$ such that in each part $R^{\ell,i}(\boldsymbol{Y})$, every tuple $\boldsymbol{w} \in \pi_{\boldsymbol{W}}(R^{\ell,i}(\boldsymbol{Y}))$ has degree at most θ. This can be done by a simple partitioning procedure.

Now for each light part $R^{\ell,i}(\boldsymbol{Y})$, we create a database instance $\mathcal{D}^{\ell,i}$ by *augmenting* $\mathcal{D}$ with $R(\boldsymbol{X}) \bowtie R^{\ell,i}(\boldsymbol{Y})$ (Eq. (2)). We also define a new set function $\boldsymbol{g}^\ell$ obtained from $\boldsymbol{g}$ by updating $g(\boldsymbol{X} \cup \boldsymbol{Y})$ to c (Eq. (1)). Note that

$$|R(\boldsymbol{X}) \bowtie R^{\ell,i}(\boldsymbol{Y})| \leq |R(\boldsymbol{X})| \cdot \theta \leq N^{g(\boldsymbol{X})} \cdot N^{g(\boldsymbol{Y}) - g(\boldsymbol{W})} = N^c \quad \left(\leq N^{\text{subw}(Q, \Delta, \boldsymbol{n})} \right) \tag{20}$$

Hence, $\boldsymbol{g}^\ell$ satisfies the cardinality invariant with respect to $\mathcal{D}^{\ell,i}$, and we rely on the Calibrate subroutine to update $\mathcal{D}^{\ell,i}$ and $\boldsymbol{g}^\ell$ to satisfy all three invariants. We call Jaguar recursively with the latest $\mathcal{D}^{\ell,i}$ and $\boldsymbol{g}^\ell$ as the new database instance and set function, respectively, and collect the results in a set of tuples $\mathcal{A}$.

Finally, we create a database instance $\mathcal{D}^h$ by augmenting $\mathcal{D}$ with $\pi_{\boldsymbol{W}}(R^h(\boldsymbol{Y}))$, and we recursively call Jaguar on $\mathcal{D}^h$ *without* providing the optional argument $\boldsymbol{g}$. Instead, we rely on Eq. (17) to initialize $\boldsymbol{g}$ in this recursive call. We add the result to $\mathcal{A}$ and return $\mathcal{A}$.

[6] Recall that the current database instance $\mathcal{D}$ is always assumed to be over the same set of variables $\boldsymbol{V}$ as the query Q, i.e., for every $R(\boldsymbol{X}) \in \mathcal{D}$, we have $\boldsymbol{X} \subseteq \boldsymbol{V}$.

[7] This is the same as saying (T, χ) is covered by the set $\{\boldsymbol{X} \mid R(\boldsymbol{X}) \in \mathcal{D}\}$, as defined in Lemma 3.1.

Algorithm 1 $\text{Jaguar}_{\epsilon,Q}((\Delta, \boldsymbol{n}), \mathcal{D}, [\boldsymbol{g}])$

if $\boldsymbol{g}$ was not provided **then**
  Initialize $\boldsymbol{g}$ using Eq. (17). ▷ $\boldsymbol{g}$ satisfies the cardinality invariant
  $(\mathcal{D}, \boldsymbol{g}) \leftarrow \text{Calibrate}((\Delta, \boldsymbol{n}), \mathcal{D}, \boldsymbol{g})$ ▷ Now, $\boldsymbol{g}$ satisfies all three invariants
end if
if there exists $(T, \chi) \in \mathcal{T}(Q)$ that is *covered by* $\mathcal{D}$ **then**
  Semijoin-reduce $R(\chi(t))$ for every $t \in \text{nodes}(T)$ with all relations in $\mathcal{D}$
  Evaluate $Q'_{\mathcal{D}}(\boldsymbol{F}) \coloneq \bigwedge_{t \in \text{nodes}(T)} R(\chi(t))$ using Yannakakis algorithm and **return** the result
else
  $(\boldsymbol{X}, \boldsymbol{Y}) \leftarrow \operatorname{argmin}_{\substack{\boldsymbol{X}', \boldsymbol{Y}' \subseteq \boldsymbol{V} \\ \boldsymbol{g}(\boldsymbol{X}' \cup \boldsymbol{Y}') > f(\boldsymbol{X}', \boldsymbol{Y}')}} f(\boldsymbol{X}', \boldsymbol{Y}')$ ▷ f is given by Eq. (10)
  $c \leftarrow f(\boldsymbol{X}, \boldsymbol{Y})$ ▷ This is the same c from Eq. (11)
  $\boldsymbol{W} \leftarrow \boldsymbol{X} \cap \boldsymbol{Y}$
  $\theta \leftarrow N^{\boldsymbol{g}(\boldsymbol{Y}) - \boldsymbol{g}(\boldsymbol{W})}$
  $(R^h(\boldsymbol{Y}), R^\ell(\boldsymbol{Y})) \leftarrow \text{heavy-light-partition}(R(\boldsymbol{Y}), \boldsymbol{W}, \theta \times N^\epsilon)$ ▷ Partition $R(\boldsymbol{Y})$ into heavy and light parts based on the degree $\deg_R(\boldsymbol{Y}|\boldsymbol{W} = \boldsymbol{w})$ being above or below $\theta \times N^\epsilon$
  $(R^{\ell,1}(\boldsymbol{Y}), \ldots, R^{\ell,\lceil N^\epsilon \rceil}(\boldsymbol{Y})) \leftarrow \text{equal-degree-partition}(R^\ell(\boldsymbol{Y}), \boldsymbol{W}, \theta)$ ▷ Parition $R^\ell(\boldsymbol{Y})$ into $\lceil N^\epsilon \rceil$ parts where each part $R^{\ell,i}(\boldsymbol{Y})$ has degree $\deg_{R^{\ell,i}(\boldsymbol{Y})}(\boldsymbol{Y}|\boldsymbol{W} = \boldsymbol{w})$ at most θ
  $\mathcal{A} \leftarrow \emptyset$ ▷ A set of tuples to collect all answers
  for $i = 1$ to $\lceil N^\epsilon \rceil$ **do**
    $R^{\ell,i}(\boldsymbol{X} \cup \boldsymbol{Y}) \leftarrow R(\boldsymbol{X}) \bowtie R^{\ell,i}(\boldsymbol{Y})$ ▷ $|R^{\ell,i}(\boldsymbol{X} \cup \boldsymbol{Y})| \le N^{\boldsymbol{g}(\boldsymbol{X})} \cdot \theta = N^{f(\boldsymbol{X},\boldsymbol{Y})} = N^c$
    $\mathcal{D}^{\ell,i} \leftarrow \mathcal{D} \uplus \{R^{\ell,i}(\boldsymbol{X} \cup \boldsymbol{Y})\}$
    $\boldsymbol{g}^\ell(\boldsymbol{Z}) \leftarrow \boldsymbol{g}[(\boldsymbol{X} \cup \boldsymbol{Y}) \to c]$ ▷ $\boldsymbol{g}^\ell$ satisfies the cardinality invariant w.r.t. $\mathcal{D}^{\ell,i}$
    $(\mathcal{D}^{\ell,i}, \boldsymbol{g}^\ell) \leftarrow \text{Calibrate}((\Delta, \boldsymbol{n}), \mathcal{D}^{\ell,i}, \boldsymbol{g}^\ell)$ ▷ Now, $\boldsymbol{g}^\ell$ satisfies all 3 invariants w.r.t. $\mathcal{D}^{\ell,i}$
    $\mathcal{A} \leftarrow \mathcal{A} \cup \text{Jaguar}_{\epsilon,Q}((\Delta, \boldsymbol{n}), \mathcal{D}^{\ell,i}, \boldsymbol{g}^\ell)$
  end for
  $\mathcal{D}^h \leftarrow \mathcal{D} \uplus \{\pi_{\boldsymbol{W}}(R^h(\boldsymbol{Y}))\}$
  $\mathcal{A} \leftarrow \mathcal{A} \cup \text{Jaguar}_{\epsilon,Q}((\Delta, \boldsymbol{n}), \mathcal{D}^h, \text{NULL})$ ▷ We do *not* pass $\boldsymbol{g}$ here
  return $\mathcal{A}$
end if

5.3 The Calibrate Subroutine

We now describe the steps of the Calibrate subroutine which are depicted in Algorithm 2. The Calibrate subroutine is mainly needed to handle *general* statistics $(\Delta, \boldsymbol{n})$. In particular, if all the statistics in $(\Delta, \boldsymbol{n})$ are *cardinality upper bounds*, i.e., have the form $\log_N \deg_R(\boldsymbol{Y}|\emptyset) \le n_{\boldsymbol{Y}|\emptyset}$, then Calibrate simplifies to a much simpler procedure that only enforces the monotonicity invariant, and it does so by computing projections of every relation that is added to the current database instance.

Calibrate takes as input a collection of statistics $(\Delta, \boldsymbol{n})$, a database instance $\overline{\mathcal{D}}$ that satisfies the statistics $(\Delta, \boldsymbol{n})$, and a set function $\overline{\boldsymbol{g}}$ that satisfies the cardinality invariant w.r.t. $\overline{\mathcal{D}}$. Calibrate returns a new database instance $\mathcal{D}$ (which is an *augmented* version of $\overline{\mathcal{D}}$, as defined in Section 2) and a new set function $\boldsymbol{g} \le \overline{\boldsymbol{g}}$ that satisfies all three invariants w.r.t. $\mathcal{D}$: the cardinality, monotonicity, and statistics invariants.

Calibrate starts by initializing its outputs $\mathcal{D}$ and $\boldsymbol{g}$ to be copies of the inputs $\overline{\mathcal{D}}$ and $\overline{\boldsymbol{g}}$. Then, Calibrate updates $g(\emptyset)$ to 0 (assuming it was not 0 already) and augments $\mathcal{D}$ with a nullary relation $R()$ consisting of a single tuple $()$, to ensure the cardinality invariant $g(\emptyset) \ge \log_N |R()|$. Afterwards, Calibrate repeatedly applies the following two steps until neither applies:

- If the monotonicity invariant is violated, that is, there exist $\boldsymbol{X} \subset \boldsymbol{Y} \subseteq \boldsymbol{V}$ such that $g(\boldsymbol{X}) > g(\boldsymbol{Y})$, then Calibrate updates $g(\boldsymbol{X})$ to $g(\boldsymbol{Y})$ and augments $\mathcal{D}$ with the relation $\pi_{\boldsymbol{X}}(R(\boldsymbol{Y}))$. This fixes the violated monotonicity while maintaining the cardinality invariant.
- If the statistics invariant is violated, that is, there exists a statistics $(\boldsymbol{Y}|\boldsymbol{X}) \in \Delta$ such that $g(\boldsymbol{X} \cup \boldsymbol{Y}) - g(\boldsymbol{X}) > n_{\boldsymbol{Y}|\boldsymbol{X}}$, then Calibrate updates $g(\boldsymbol{X} \cup \boldsymbol{Y})$ to become $g(\boldsymbol{X}) + n_{\boldsymbol{Y}|\boldsymbol{X}}$. In order to maintain the cardinality invariant, we need to augment $\mathcal{D}$ with a relation $R(\boldsymbol{X} \cup \boldsymbol{Y})$ such that $|R(\boldsymbol{X} \cup \boldsymbol{Y})| \leq N^{g(\boldsymbol{X})} \cdot N^{n_{\boldsymbol{Y}|\boldsymbol{X}}}$. To that end, we take the *guard* relation $R(\boldsymbol{Z})$ of the statistics $\boldsymbol{Y}|\boldsymbol{X}$ (Section 2) and set $R(\boldsymbol{X} \cup \boldsymbol{Y}) \leftarrow R(\boldsymbol{X}) \bowtie \pi_{\boldsymbol{X} \cup \boldsymbol{Y}}(R(\boldsymbol{Z}))$. By definition of guard relation, $\deg_{R(\boldsymbol{Z})}(\boldsymbol{Y}|\boldsymbol{X}) \leq N^{n_{\boldsymbol{Y}|\boldsymbol{X}}}$. Moreover, by the cardinality invariant, $|R(\boldsymbol{X})| \leq N^{g(\boldsymbol{X})}$. Therefore, $|R(\boldsymbol{X} \cup \boldsymbol{Y})|$ meets the desired bound.

Algorithm 2 Calibrate$((\Delta, \boldsymbol{n}), \overline{\mathcal{D}}, \overline{\boldsymbol{g}})$

```
(D, g) ← (D̄, ḡ)                                  ▷ Initialize the outputs to the inputs
g(∅) ← 0                                          ▷ Fix the monotonicity invariant
R() ← {()}                                        ▷ R() is a nullary relation with a single tuple
D ← D ⊎ {R()}                                     ▷ Re-inforce the cardinality invariant
while true do
    if ∃X ⊂ Y ⊆ V where g(X) > g(Y) then          ▷ If the monotonicity invariant is violated
        g(X) ← g(Y)                               ▷ Fix the monotonicity invariant
        D ← D ⊎ {π_X(R(Y))}                       ▷ Re-inforce the cardinality invariant
    else if ∃(Y|X) ∈ Δ where g(X ∪ Y) − g(X) > n_{Y|X} then   ▷ If the statistics invariant is violated
        g(X ∪ Y) ← g(X) + n_{Y|X}                 ▷ Fix the statistics invariant
        Let R(Z) be the guard of Y|X              ▷ Z ⊇ X ∪ Y and log_N deg_{R(Z)}(Y|X) ≤ n_{Y|X}
        R(X ∪ Y) ← R(X) ⋈ π_{X∪Y}(R(Z))           ▷ |R(X ∪ Y)| ≤ N^{g(X)} · N^{n_{Y|X}}
        D ← D ⊎ {R(X ∪ Y)}                        ▷ Re-inforce the cardinality invariant
    else                                          ▷ If no invariants are violated
        break
    end if
end while
return (D, g)
```

6 Algorithm Analysis

Our main result is the following theorem, which says that Jaguar runs in submodular width time, with an extra factor of $O(N^{\epsilon})$ where $\epsilon > 0$ can be arbitrarily small. Note that the definition of the submodular width used below *generalizes* the one by Daniel Marx [24] by accepting *arbitrary degree constraints* (not just upper bounds on the cardinalities of relations) as well as *arbitrary CQs* (not just Boolean or full CQs). See Section 2.

Theorem 6.1. *Let $\epsilon > 0$ be an arbitrary constant. Given a CQ Q, a set of statistics $(\Delta, \boldsymbol{n})$, and a database instance $\mathcal{D}_0$ that matches the schema of Q and satisfies the statistics $(\Delta, \boldsymbol{n})$, Jaguar (Algorithm 1) correctly computes $Q(\mathcal{D}_0)$ in time*

$$O\left(N^{subw(Q,\Delta,\boldsymbol{n})+\epsilon} + \mathsf{OUT}\right), \tag{21}$$

in data complexity, where $N \stackrel{def}{=} |\mathcal{D}_0|$, and OUT is the output size. In particular, after a preprocessing time of $O(N^{subw(Q,\Delta,\boldsymbol{n})+\epsilon})$, Jaguar supports constant-delay enumeration of the output tuples.

In order to prove Theorem 6.1, we first prove some basic properties of the Calibrate subroutine in Section 6.1. Then, we prove the correctness claim of the theorem in Section 6.2, and finally, we prove the runtime claim in Section 6.3.

6.1 Analysis of the Calibrate subroutine

The following lemma says that, on any input database instance $\overline{\mathcal{D}}$ and set function $\overline{\boldsymbol{g}}$ that satisfy the cardinality invariant, Calibrate always terminates and outputs a database instance $\mathcal{D}$ and set function $\boldsymbol{g} \leq \overline{\boldsymbol{g}}$ that satisfy all three invariants. In addition, suppose that we take the output $\boldsymbol{g}$, reduce some entry $g(\boldsymbol{W})$ down to c, for some value c, and then run Calibrate again on this modified output. Then, the lemma says that running Calibrate again cannot reduce any other entry of $\boldsymbol{g}$ below c. All properties of Calibrate become obvious once we realize that is a *shortest-path algorithm* in disguise, as we explain in the proof. In particular, the last claim in the lemma below is analogous to saying: if we compute shortest paths in a weighted graph (with non-negative weights), and then add a new edge with weight c, then this newly added edge cannot create any *new* shortest paths with weight less than c.

Lemma 6.1 (Properties of Calibrate). *Let $(\Delta, \boldsymbol{n})$ be a set of statistics, $\overline{\mathcal{D}}$ be database instance that satisfies $(\Delta, \boldsymbol{n})$, and $\overline{\boldsymbol{g}}$ be a set function that satisfies the cardinality invariant w.r.t. $\overline{\mathcal{D}}$. Then, the Calibrate subroutine (Algorithm 2) terminates and returns a database instance $\mathcal{D}$ and a set function $\boldsymbol{g}$ that satisfy the following properties:*

(1) $\boldsymbol{g}$ satisfies the cardinality, monotonicity and statistics invariants w.r.t. $\mathcal{D}$.
(2) $\boldsymbol{g} \leq \overline{\boldsymbol{g}}$.
(3) Suppose that there exists a set function $\tilde{\boldsymbol{g}}$ that satisfies all three invariants w.r.t. $\overline{\mathcal{D}}$ and the input $\overline{\boldsymbol{g}}$ satisfies: $\overline{\boldsymbol{g}} = \tilde{\boldsymbol{g}}[\boldsymbol{W} \to c]$ for some $\boldsymbol{W} \subseteq \boldsymbol{V}$, $c \in \mathbb{R}_+$ where $c < \tilde{g}(\boldsymbol{W})$. Then, the output $\boldsymbol{g}$ satisfies: $g(\boldsymbol{Z}) \geq c$ for every $\boldsymbol{Z} \subseteq \boldsymbol{V}$ where $g(\boldsymbol{Z}) < \overline{g}(\boldsymbol{Z})$.

Proof. First, we prove that Calibrate always terminates. To that end, we show that Calibrate is a shortest-path algorithm in disguise on a certain weighted directed graph. In particular, consider the following weighted directed graph $\overline{G}$ whose vertices are all subsets of $\boldsymbol{V}$ and has four kinds of weighted edges:

- For every $\boldsymbol{X} \subseteq \boldsymbol{V}$, we have an edge $\emptyset \to \boldsymbol{X}$ with weight $\overline{g}(\boldsymbol{X})$.
- We have an edge $\emptyset \to \emptyset$ with weight 0.
- For every $\boldsymbol{X} \subset \boldsymbol{Y} \subseteq \boldsymbol{V}$, we have an edge $\boldsymbol{Y} \to \boldsymbol{X}$ with weight 0.
- For every statistics $(\boldsymbol{Y}|\boldsymbol{X})$ in Δ, we have an edge $\boldsymbol{X} \to (\boldsymbol{X} \cup \boldsymbol{Y})$ with weight $n_{\boldsymbol{Y}|\boldsymbol{X}}$.

It is straightforward to verify that Calibrate is computing shortest paths from $\emptyset$ to every other vertex in $\overline{G}$. In particular, each step of Calibrate is considering some edge $\boldsymbol{X} \to \boldsymbol{Y}$ and trying to use the current shortest path to $\boldsymbol{X}$ to update the shortest path to $\boldsymbol{Y}$. Because all weights in the graph are non-negative, for a fixed $\boldsymbol{Y} \subseteq \boldsymbol{V}$, the total number of times the algorithm can lower the value of $g(\boldsymbol{Y})$ is upper bounded by the number of simple paths from $\emptyset$ to $\boldsymbol{Y}$ in $\overline{G}$. Hence, the algorithm terminates.

As long as the algorithm terminates, the output function $\boldsymbol{g}$ must satisfy the all three invariants, since the algorithm keeps going as long as any invariant is violated. Hence, Claim (1) holds. Claim (2) is obvious.

Finally, we prove Claim (3). Let $\tilde{G}$ be the weighted directed graph defined similarly to $\overline{G}$ but using $\tilde{\boldsymbol{g}}$ instead of $\overline{\boldsymbol{g}}$. In the language of shortest paths, Claim (3) says that $\tilde{G}$ is *closed* under shortest paths, meaning that for every vertex $\boldsymbol{Y}$ in $\tilde{G}$, there is a shortest path from $\emptyset$ to $\boldsymbol{Y}$ in $\tilde{G}$ consisting of a *single edge*. The claim also says that $\overline{G}$ is obtained from $\tilde{G}$ by introducing a new edge $\emptyset \to \boldsymbol{W}$ with

weight c. Because all weights are non-negative, all *new* shortest paths in $\overline{G}$ that were not present in $\tilde{G}$ (i.e., the shortest paths that use the newly added edge) must have weight at least c. □

Proposition 6.2 (Runtime of Calibrate). *The Calibrate algorithm (Algorithm 2) runs in linear time in the size of the input database instance $\overline{\mathcal{D}}$, in data complexity.*

6.2 Correctness Proof of Jaguar

As mentioned before, Jaguar is recursive where the top-level call is made with a database instance $\mathcal{D}_0$ (of size N), and each recursive call is made with an *augmented* database instance $\mathcal{D}$ (of size potentially larger than N). It is not immediately clear that the algorithm terminates, i.e., that the recursion depth is finite. However, assuming that it does, we will prove in this section that it is correct, i.e., the top-level call to Jaguar returns $Q(\mathcal{D}_0)$. In the next section, we will prove a finite bound on the recursion depth, hence proving termination.

In order to prove that Jaguar is correct, we will inductively prove the following claim:

Claim 1. Every recursive call to Jaguar on a database instance $\mathcal{D}$ returns a set of tuples $\mathcal{A} \subseteq \mathsf{Dom}^{\boldsymbol{F}}$ that satisfies Eq. (16).

Proof of Claim 1. As a base case, consider a recursive call that terminates in line 7 of Algorithm 1. The current database instance $\mathcal{D}$ is always an augmented version of the original input instance $\mathcal{D}_0$. In addition, for every relation $R(\boldsymbol{X})$ in $\mathcal{D}_0$, there is a bag $R(\chi(t))$ for some $t \in \mathsf{nodes}(T)$ where $\boldsymbol{X} \subseteq \chi(t)$ and $R(\chi(t))$ was semijoin-reduced with $R(\boldsymbol{X})$ in line 6. This proves that $\mathcal{A} \stackrel{\text{def}}{=} Q'_{\mathcal{D}}(\mathcal{D}) \subseteq Q(\mathcal{D}_0)$. Moreover, $Q_{\mathcal{D}}(\mathcal{D}) \subseteq Q'_{\mathcal{D}}(\mathcal{D})$ by definition.[8]

For the inductive step, note that we are partitioning the relation $R(\boldsymbol{Y})$ into (disjoint) parts in both lines 13 and 14. Afterwards, for each part, we augment the database instance $\mathcal{D}$ with that part and recursively call Jaguar. By induction, it is straightforward to see that $\mathcal{A} \subseteq Q(\mathcal{D}_0)$ and at the same time $\mathcal{A} \supseteq Q_{\mathcal{D}}(\mathcal{D})$. □

6.3 Runtime Analysis of Jaguar

In this section, we prove that Jaguar runs in time $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})+\epsilon} + \mathsf{OUT})$ in data complexity, and also supports constant-delay enumeration after a pre-processing time of $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})+\epsilon})$. Along with the previous section, this will complete the proof of Theorem 6.1.

The execution of Jaguar can be represented as a recursion tree, where each node is a call to Jaguar. Each internal node has $\lceil N^{\epsilon} \rceil + 1$ children, namely $\lceil N^{\epsilon} \rceil$ children corresponding to light parts in line 21 and one child corresponding to the heavy part in line 24. An edge from a parent to a child is called *light* if it corresponds to a recursive call on a light partition in line 21, and *heavy* otherwise. A path from an ancestor node to a descendant node is called a *light path* if it consists of only light edges. Our first target is to prove the following claim:

Claim 2. A light path can have length at most $2^{|\boldsymbol{V}|} = O(1)$.

In order to prove Claim 2, we will prove a series of claims. To that end, we introduce some notation. Let $f^{\ell}, c^{\ell}, h^{\ell}, \mathcal{I}^{\ell}$ be defined as in Eq. (10)-(13) but, instead of $\boldsymbol{g}$, they are with respect to $\boldsymbol{g}^{\ell}$ *after* calibration is complete in line 20.

[8] Note that in this step, we can only compute $Q'_{\mathcal{D}}(\mathcal{D})$ but *not* necessarily $Q_{\mathcal{D}}(\mathcal{D})$ because there is no guarantee that for every relation $R(\boldsymbol{X})$ in the current database instance $\mathcal{D}$, there will be a bag $R(\chi(t))$ for some $t \in \mathsf{nodes}(T)$ where $\boldsymbol{X} \subseteq \chi(t)$. This is because (T, χ) is a tree decomposition of the *original query* Q but not necessarily of $Q_{\mathcal{D}}$. This seems to be a fundamental limitation of similar algorithms, where they cannot be extended to counting (i.e., solving #CQ) in submodular width time $O(N^{\mathsf{subw}(Q)})$. The same limitation applies to the original algorithm proposed by Daniel Marx [24] as well as all subsequent algorithms [5, 6, 10].

Claim 3. For every $Z \subseteq V$, $g^\ell(Z) \le g(Z)$. Moreover, if $g^\ell(Z) < g(Z)$, then $g^\ell(Z) \ge c$.

Proof of Claim 3. Claim 3 follows immediately from Lemma 6.1. In particular, the lemma says that $\boldsymbol{g}^\ell \le \boldsymbol{g}$. Moreover, $\boldsymbol{g}$ satisfies all three invariants w.r.t. $\mathcal{D}$ and $g^\ell(Z) \stackrel{\text{def}}{=} \boldsymbol{g}[(X \cup Y) \to c]$, per line 19 of Algorithm 1. Thanks to the last claim of Lemma 6.1, the Calibrate call in line 20 cannot reduce any $g^\ell(Z)$ value below c. □

We now use Claim 3 to prove the following claim.

Claim 4. $c^\ell \ge c$ and $\mathcal{I}^\ell \supset \mathcal{I}$.

Proof of Claim 4. First, we prove that $c^\ell \ge c$. To that end, we show that for every pair $X', Y' \subseteq V$ where $\boldsymbol{g}^\ell$ violates submodularity, i.e., where $g^\ell(X' \cup Y') > f^\ell(X', Y')$, we must have $f^\ell(X', Y') \ge c$. We consider two cases:

- Suppose that the pair X', Y' above used to satisfy submodularity in $\boldsymbol{g}$, i.e., $g(X' \cup Y') \le f(X', Y')$. Since the pair violates submodularity in $\boldsymbol{g}^\ell$, it must be the case that $g^\ell(X') < g(X')$ (or symmetrically $g^\ell(Y') < g(Y')$). In this case, $g^\ell(X') \ge c$ by Claim 3, hence $f^\ell(X', Y') \stackrel{\text{def}}{=} g^\ell(X') + g^\ell(Y') - g^\ell(X' \cap Y') \ge g^\ell(X') \ge c$ where the first inequality holds by monotonicity of $\boldsymbol{g}^\ell$.
- Now suppose that the pair X', Y' already violated submodularity in $\boldsymbol{g}$, i.e., $g(X' \cup Y') > f(X', Y') \ge c$ (where the last inequality follows by definition of c). We recognize the following cases:
  - If $g^\ell(X') < g(X')$, then $g^\ell(X') \ge c$ by Claim 3 and $f^\ell(X', Y') \stackrel{\text{def}}{=} g^\ell(X') + g^\ell(Y') - g^\ell(X' \cap Y') \ge g^\ell(X') \ge c$ by monotonicity of $\boldsymbol{g}^\ell$.
  - If $g^\ell(Y') < g(Y')$, then this is symmetric to the above case.
  - If $g^\ell(X') = g(X')$ and $g^\ell(Y') = g(Y')$, then $f^\ell(X', Y') = g(X') + g(Y') - g^\ell(X' \cap Y') \ge g(X') + g(Y') - g(X' \cap Y') = f(X', Y') \ge c$.

In all above cases, we have $f^\ell(X', Y') \ge c$, hence $c^\ell \ge c$.

The fact that $\mathcal{I}^\ell \supseteq \mathcal{I}$ follows from $c^\ell \ge c$ and $g^\ell(Z) \le g(Z)$ for all Z. Moreover, $\mathcal{I}^\ell$ contains $X \cup Y$ from line 9, while $\mathcal{I}$ doesn't, thus $\mathcal{I}^\ell \supset \mathcal{I}$. □

Now, we are ready to prove Claim 2.

Proof of Claim 2. By Claim 4, the size of $\mathcal{I}$ increases by at least one with every recursive call to the light branch. Moreover, this size cannot exceed $2^{|V|}$. □

Claim 2 proves an upper bound on the number of consecutive light edges on any path from the root to a leaf in the recursion tree. In contrast, the following claim proves an upper bound on the number of heavy edges on any path from the root to a leaf. The two claims together imply an upper bound on the depth of any leaf.

Claim 5. Any path from the root to a leaf can have at most $O(1)$ heavy edges.

Proof of Claim 5. Let Γ be a constant that is larger than the full join of all relation instances in $\mathcal{D}_0$, e.g., $\Gamma \stackrel{\text{def}}{=} N^{|V|+1}$. Define the following potential function:

$$\varphi(\mathcal{D}) \stackrel{\text{def}}{=} \sum_{R(X) \in \mathcal{D}} \log_N |R(X)| + \sum_{\substack{X \subseteq V \\ R(X) \notin \mathcal{D}}} \log_N \Gamma$$

Clearly, $\varphi(\mathcal{D}) = O(1)$ in data complexity. Consider a path from the root to a leaf. We argue that between every two consecutive heavy edges (with an arbitrary number of light edges in between),

the potential decreases by at least ϵ. In particular, let $\overline{\mathcal{D}}^h$ and $\mathcal{D}^h$ be the input database instances to two consecutive heavy calls to Jaguar in line 24 (with potentially many light calls in line 21 in between), and let $\overline{\boldsymbol{g}}$ and $\boldsymbol{g}$ be the two corresponding set functions. We argue that $\varphi(\mathcal{D}^h) \le \varphi(\overline{\mathcal{D}}^h) - \epsilon$. The first heavy call will initialize $\overline{\boldsymbol{g}}$ in line 2 to satisfy the cardinality invariant. From that point until the second heavy call, the cardinality invariant will always be maintained and $g(\boldsymbol{X})$ can never increase for any $\boldsymbol{X} \subseteq \boldsymbol{V}$. At the time we reach the second heavy call in line 24, we have

$$\begin{aligned}
\log_N |R^h(\boldsymbol{X} \cap \boldsymbol{Y})| &\le \log_N \frac{|R(\boldsymbol{Y})|}{\theta \times N^\epsilon} \\
&= \log_N |R(\boldsymbol{Y})| - \log_N \theta - \epsilon \\
&\le g(\boldsymbol{Y}) - \log_N \theta - \epsilon && \text{(by the cardinality invariant)} \\
&= g(\boldsymbol{X} \cap \boldsymbol{Y}) - \epsilon && \text{(by definition of } \theta \text{ in line 12)} \\
&\le \overline{g}(\boldsymbol{X} \cap \boldsymbol{Y}) - \epsilon && (\boldsymbol{g} \text{ never increases between heavy calls}) \\
&= \log_N |\overline{R}^h(\boldsymbol{X} \cap \boldsymbol{Y})| - \epsilon && \text{(by initialization of } \overline{\boldsymbol{g}} \text{ in line 2)}
\end{aligned}$$

This proves $\varphi(\mathcal{D}^h) \le \varphi(\overline{\mathcal{D}}^h) - \epsilon$. Moreover, since $\varphi(\mathcal{D}) = O(1)$, the number of heavy edges in any path from the root to a leaf is $O(1)$. □

Claims 2 and 5 together imply the following claim:

Claim 6. The depth of the recursion tree of Jaguar is $O(1)$ in data complexity.

Claim 6 along with the fact that each internal (i.e., non-leaf) node has $\lceil N^\epsilon \rceil + 1$ children imply that the size of the recursion tree is $O(N^{\epsilon'})$ for some constant $\epsilon' > 0$. Next, we argue that the time spent at each internal node is $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})+\epsilon})$. To that end, we need the following claim:

Claim 7. In line 10, $c \le \mathsf{subw}(Q, \Delta, \boldsymbol{n})$.

Proof of Claim 7. First, we show that c in line 10 cannot be ∞. In particular, if $c = \infty$, then $\boldsymbol{g}$ is already a polymatroid, hence $\boldsymbol{g} < \infty$. By the cardinality invariant, $\mathcal{D}$ must contain a finite relation $R(\boldsymbol{X})$ for every $\boldsymbol{X} \subseteq \boldsymbol{V}$, but then $\mathcal{D}$ must cover every tree decomposition in $\mathcal{T}(Q)$, contradicting the failure of the condition in line 5.

Assuming $c < \infty$, Claim 7 follows from Lemma 3.1. In particular, the condition in line 5 ensures that there is no tree decomposition in $\mathcal{T}(Q)$ that is covered by

$$\mathcal{I}' \stackrel{\text{def}}{=} \{\boldsymbol{X} \subseteq \boldsymbol{V} \mid R(\boldsymbol{X}) \in \mathcal{D}\} \supseteq \{\boldsymbol{X} \subseteq \boldsymbol{V} \mid g(\boldsymbol{X}) < \infty\} \supseteq \{\boldsymbol{X} \subseteq \boldsymbol{V} \mid g(\boldsymbol{X}) \le c\} = \mathcal{I}$$

The first $\supseteq$ above follows from the cardinality invariant, while the second follows from $c < \infty$. The statistics invariant ensures that $\boldsymbol{g} \models (\Delta, \boldsymbol{n})$, hence, Lemma 3.1 applies. □

Thanks to Claim 7, the time needed to compute each join $R^{\ell,i}(\boldsymbol{X} \cup \boldsymbol{Y})$ in line 17 is within

$$|R^{\ell,i}(\boldsymbol{X} \cup \boldsymbol{Y})| \le |R(\boldsymbol{X})| \cdot \deg_{R^{\ell,i}}(\boldsymbol{Y}|\boldsymbol{X} \cap \boldsymbol{Y}) \le N^{g(\boldsymbol{X})} \times \theta = N^{f(\boldsymbol{X},\boldsymbol{Y})} = N^c \le N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})} \tag{22}$$

Therefore, the total time to compute the joins for i from 1 to $\lceil N^\epsilon \rceil$ is $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})+\epsilon})$. Moreover, by Proposition 6.2, the time spent in Calibrate is linear in the size of the current database $\mathcal{D}$, which cannot exceed $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})})$, by Eq. (22). This proves that the time spent at each internal node is $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})+\epsilon})$, hence the time spent at *all* internal nodes combined is $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})+\epsilon+\epsilon'})$.

Finally, we analyze the time needed to report the output at leaf nodes. The recursion tree can have $O(N^{\epsilon'})$ leaves, and each output tuple can occur in multiple leaves. However, the total number of tree decompositions is constant in data complexity, thus we can group together leaves that use the same tree decomposition in line 5 (by taking the union of their bags). Thanks to this grouping,

each output tuple can still occur in multiple groups but the total number of groups is constant, thus allowing us to remove duplicates with extra $O(1)$ overhead and report the output in $O(\mathsf{OUT})$ time. This proves that the overall runtime of Jaguar is $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})+\epsilon+\epsilon'} + \mathsf{OUT})$, as desired.

Constant-delay enumeration (CDE). In order to support CDE after a pre-processing time of $O(N^{\mathsf{subw}(Q,\Delta,\boldsymbol{n})+\epsilon+\epsilon'})$, we replace running Yannakakis algorithm in line 7 with the enumeration algorithm from [9]. (Recall from Section 2 that all tree decompositions in $\mathcal{T}(Q)$ are *free-connex*.) As mentioned above, the same output tuple can still occur in $O(1)$ leaves, hence can have $O(1)$ duplicates. However, we can use the *Cheater's Lemma* [12] as a standard technique to support CDE of the union of a constant number of sets, each of which supports CDE. This completes the proof of all claims made in Theorem 6.1.

7 Conclusion and Open Problems

We introduced an algorithm, called Jaguar, for evaluating a CQ Q in submodular width time, with an extra overhead of $O(N^{\epsilon})$ where $\epsilon > 0$ can be arbitrarily small. The algorithm handles arbitrary *statistics*, which are degree constraints, on the input database instance. These statistics can significantly lower the value of the submodular width compared to the original definition [24]. The submodular width is defined via a maximization problem which naturally has equivalent primal and dual formulations. Correspondingly, we classify query evaluation algorithms that target the submodular width into two classes: (a) *primal algorithms* which operate in the primal space (the space of polymatroids) of this maximization problem and include the algorithms from [10, 24] as well as our new algorithm; and (b) *dual algorithms* which operate in the dual space (the space of Shannon inequalities) and include PANDA [5, 6]. Because it is a dual one, PANDA is quite involved and requires transforming a CQ into disjunctive datalog rules (DDRs) as well as using Shannon entropy. In contrast, our new primal algorithm is simpler, more direct, and bypasses the need for DDRs and Shannon entropy.

We conclude with a couple of major open problems in this area.

Solving #CQs in submodular width time. It might be tempting at first to try to extend our algorithm to solve counting conjunctive queries (#CQs) in time $O(N^{\mathsf{subw}(Q)+\epsilon})$. This is because the algorithm recursively partitions the database instance into *disjoint* parts (lines 13 and 14 of Algorithm 1). However, as explained earlier, there is no guarantee that the tree decomposition found at the end of the recursion (line 5) will actually encompass all intermediate relations that the algorithm created. This seems to be a common weakness of *all* prior algorithms [5, 6, 10, 24], which also fall short of solving #CQs in submodular width time. To circumvent this issue, a weaker version of the submodular width, called the *sharp-submodular width*, that allows for counting, was introduced in [2]. However, it was shown recently that this new width is not the best possible: There are #CQs where counting can be done faster than the sharp-submodular width time [11]. This leaves open the question of whether we can actually solve #CQs in submodular width time, and if not, what is the best possible complexity measure for #CQs.

Solving CQs in *entropic width* time. The current submodular width definition from Eq. (8) takes a maximum over all polymatroids. It is possible to derive a tighter width notion by maximizing over only the *entropic functions*. This tighter notion has been referred to as the *entropic width* in [5], and we denote it here by $\mathsf{entw}(Q)$. Another way to put it is that the submodular width only considers Shannon inequalities, whereas the entropic width additionally includes *non-Shannon inequalities* [27, 28]. Recent work has shown a lower bound[9] of $\Omega(N^{(1-\epsilon)\cdot\mathsf{entw}(Q)})$ for every $\epsilon > 0$ on the size of the *semiring circuit* needed to represent the output of a CQ [15]. However, no algorithm

[9]For consistency, we assume here that $\mathsf{entw}(Q)$ is defined with log base N, similar to Remark 2.1.

is known to answer a CQ in time written in terms of $O(N^{\mathrm{entw}(Q)})$. Achieving this bound remains an open problem [22].

Acknowledgments

H. Chen acknowledges the support of EPSRC grants EP/V039318/1 and EP/X03190X/1.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. 1995. *Foundations of Databases*. Addison-Wesley. http://webdam.inria.fr/Alice/

[2] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, Xuanlong Nguyen, Dan Olteanu, and Maximilian Schleich. 2020. Functional Aggregate Queries with Additive Inequalities. *ACM Trans. Database Syst.* 45, 4, Article 17 (dec 2020), 41 pages. doi:10.1145/3426865

[3] Mahmoud Abo Khamis, Xiao Hu, and Dan Suciu. 2025. Fast Matrix Multiplication meets the Submodular Width. *Proc. ACM Manag. Data* 3, 2 (2025), 98:1–98:26.

[4] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. 2016. FAQ: Questions Asked Frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 13–28. doi:10.1145/2902251.2902280

[5] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2017. What Do Shannon-type Inequalities, Submodular Width, and Disjunctive Datalog Have to Do with One Another?. In *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, Emanuel Sallinger, Jan Van den Bussche, and Floris Geerts (Eds.). ACM, 429–444. doi:10.1145/3034786.3056105

[6] Mahmoud Abo Khamis, Hung Q. Ngo, and Dan Suciu. 2025. PANDA: Query Evaluation in Submodular Width. *TheoretiCS* Volume 4, Article 12 (Apr 2025). doi:10.46298/theoretics.25.12

[7] Isolde Adler, Georg Gottlob, and Martin Grohe. 2007. Hypertree width and related hypergraph invariants. *Eur. J. Comb.* 28, 8 (2007), 2167–2181.

[8] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and Counting Given Length Cycles. *Algorithmica* 17, 3 (1997), 209–223. doi:10.1007/BF02523189

[9] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. 2007. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings (Lecture Notes in Computer Science, Vol. 4646)*, Jacques Duparc and Thomas A. Henzinger (Eds.). Springer, 208–222. doi:10.1007/978-3-540-74915-8_18

[10] Christoph Berkholz and Nicole Schweikardt. 2019. Constant Delay Enumeration with FPT-Preprocessing for Conjunctive Queries of Bounded Submodular Width. In *44th International Symposium on Mathematical Foundations of Computer Science (MFCS 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 138)*, Peter Rossmanith, Pinar Heggernes, and Joost-Pieter Katoen (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 58:1–58:15. doi:10.4230/LIPIcs.MFCS.2019.58

[11] Karl Bringmann and Egor Gorbachev. 2025. A Fine-Grained Classification of Subquadratic Patterns for Subgraph Listing and Friends. In *Proceedings of the 57th Annual ACM Symposium on Theory of Computing* (Prague, Czechia) *(STOC '25)*. Association for Computing Machinery, New York, NY, USA, 2145–2156. doi:10.1145/3717823.3718141

[12] Nofar Carmeli and Markus Kröll. 2021. On the Enumeration Complexity of Unions of Conjunctive Queries. 46, 2, Article 5 (May 2021), 41 pages. doi:10.1145/3450263

[13] Hubie Chen and Víctor Dalmau. 2005. Beyond Hypertree Width: Decomposition Methods Without Decompositions. In *Eleventh International Conference on Principles and Practice of Constraint Programming*. 167–181.

[14] Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. 2023. The Fine-Grained Complexity of Boolean Conjunctive Queries and Sum-Product Problems. In *50th International Colloquium on Automata, Languages, and Programming (ICALP 2023) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 261)*, Kousha Etessami, Uriel Feige, and Gabriele Puppis (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 127:1–127:20. doi:10.4230/LIPIcs.ICALP.2023.127

[15] Austen Z. Fan, Paraschos Koutris, and Hangdong Zhao. 2024. Tight Bounds of Circuits for Sum-Product Queries. *Proc. ACM Manag. Data* 2, 2, Article 87 (May 2024), 20 pages. doi:10.1145/3651588

[16] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. 2016. Hypertree Decompositions: Questions and Answers. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Tova Milo and Wang-Chiew Tan (Eds.). ACM, 57–74. doi:10.1145/2902251.2902309

[17] Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. 2021. Complexity Analysis of Generalized and Fractional Hypertree Decompositions. *J. ACM* 68, 5 (2021), 38:1–38:50.

[18] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2002. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.* 64, 3 (2002), 579–627.
[19] Georg Gottlob, Nicola Leone, and Francesco Scarcello. 2003. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *J. Comput. Syst. Sci.* 66, 4 (2003), 775–808.
[20] Martin Grohe and Dániel Marx. 2006. Constraint solving via fractional edge covers. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*. ACM Press, 289–298. http://dl.acm.org/citation.cfm?id=1109557.1109590
[21] Martin Grohe and Dániel Marx. 2014. Constraint Solving via Fractional Edge Covers. *ACM Trans. Algorithms* 11, 1 (2014), 4:1–4:20.
[22] Phokion G. Kolaitis, Andrej E. Romashchenko, Milan Studený, Dan Suciu, and Tobias A. Boege. 2023. Algorithmic Aspects of Information Theory (Dagstuhl Seminar 22301). *Dagstuhl Reports* 12, 7 (2023), 180–204. doi:10.4230/DagRep.12.7.180
[23] Dániel Marx. 2010. Approximating fractional hypertree width. *ACM Trans. Algorithms* 6, 2 (2010), 29:1–29:17.
[24] Dániel Marx. 2013. Tractable Hypergraph Properties for Constraint Satisfaction and Conjunctive Queries. *J. ACM* 60, 6 (2013), 42:1–42:51. doi:10.1145/2535926
[25] Luc Segoufin. 2015. Constant Delay Enumeration for Conjunctive Queries. *SIGMOD Rec.* 44, 1 (May 2015), 10–17. doi:10.1145/2783888.2783894
[26] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*. IEEE Computer Society, 82–94.
[27] Zhen Zhang and Raymond W. Yeung. 1997. A non-Shannon-type conditional inequality of information quantities. *IEEE Trans. Information Theory* 43, 6 (1997), 1982–1986.
[28] Zhen Zhang and Raymond W Yeung. 1998. On characterization of entropy function via information inequalities. *IEEE Transactions on Information Theory* 44, 4 (1998), 1440–1452.

Received December 2025; accepted February 2026