The Space-Time Complexity of Sum-Product Queries

KYLE DEEDS, University of Washington, United States
TIMO CAMILLO MERKL, TU Wien, Austria
REINHARD PICHLER, TU Wien, Austria
DAN SUCIU, University of Washington, United States

While extensive research on query evaluation has achieved consistent improvements in the time complexity of algorithms, the space complexity of query evaluation has been largely ignored. This is a particular challenge in settings with strict pre-defined space constraints. In this paper, we examine the combined space-time complexity of conjunctive queries (CQs) and, more generally, of sum-product queries (SPQs). We propose several classes of space-efficient algorithms for evaluating SPQs, and we show that the optimal time complexity is almost always achievable with asymptotically lower space complexity than traditional approaches.

CCS Concepts: • **Theory of computation → Database theory**; *Design and analysis of algorithms.*

Additional Key Words and Phrases: Query evaluation.

ACM Reference Format:
Kyle Deeds, Timo Camillo Merkl, Reinhard Pichler, and Dan Suciu. 2025. The Space-Time Complexity of Sum-Product Queries. *Proc. ACM Manag. Data* 3, 5 (PODS), Article 283 (November 2025), 21 pages. https://doi.org/10.1145/3767719

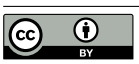
1 Introduction

Algorithms for answering conjunctive queries (CQs), often generalized to sum-product queries (SPQs), have been extensively studied. Prior work has identified tight bounds on their *time complexity* relative to a variety of structural parameters of the query, e.g. treewidth, (generalized or fractional) hypertree width, or submodular width [19–21, 30]. However, no attention has been paid to the *space complexity* of these algorithms which can often equal the time complexity.

This is a major challenge for end-users who typically run these algorithms in settings with strict pre-defined space constraints, e.g. GPU memory, main memory, or SSD size. If the algorithm has a large space complexity, the user has two unsatisfactory options; 1) reserve a moderate amount of space and risk an out-of-memory error when inputs produce large intermediates or 2) reserve a larger, more expensive server to guarantee robustness. Developers typically place a high value on stability which pushes them towards the latter, and this conservative impulse is further exacerbated by the challenges of estimating space utilization ahead-of-time [5]. In the cloud setting, this has resulted in the well-known problem of over-provisioning memory with over 90% of jobs in the Google Cluster Dataset using less than 20% of the provisioned memory [16, 29].

In this paper, we examine the combined space-time complexity of SPQs to address these space-constrained settings – illustrated here in the introduction for CQs. We begin by formally defining these notions of complexity. A *query plan* Π for a query Q is a structure that is associated with a specific algorithm for evaluating Q, e.g. a tree decomposition, or join-plan, or a variable order for

Authors' Contact Information: Kyle Deeds, University of Washington, United States; Timo Camillo Merkl, kdeeds@cs.washington.edu, TU Wien, Austria, timo.merkl@tuwien.ac.at; Reinhard Pichler, TU Wien, Austria, reinhard.pichler@tuwien.ac.at; Dan Suciu, University of Washington, United States, suciu@cs.washington.edu.



ACM 2836-6573/2025/11-ART283
https://doi.org/10.1145/3767719

generic-join. We denote a class of plans by C, e.g. all tree decompositions, and the plans of C for a query Q as $C(Q)$. Each plan Π is associated with a *space exponent* $s(\Pi)$ and a *time exponent* $t(\Pi)$. The latter bounds the associated algorithm's runtime by $\tilde{O}(|D|^{t(\Pi)})$. The former bounds the space used by the algorithm by $O(|D|^{s(\Pi)})$, excluding the space required to store the input relations. We will often refer to these jointly as the space-time exponents $e(\Pi) = (s(\Pi), t(\Pi))$.

Definition 1.1. (Plan Domination). Let Π_1, Π_2 be two plans for the same query Q. We say Π_1 *improves over* Π_2, or that Π_1 *dominates* Π_2, denoted $\Pi_1 \preceq \Pi_2$, if $t(\Pi_1) \leq t(\Pi_2)$ and $s(\Pi_1) \leq s(\Pi_2)$. We say that Π_1 *strictly dominates* Π_2, denoted $\Pi_1 \prec \Pi_2$, if $\Pi_1 \preceq \Pi_2$ but not vice versa.

A class of plans C_1 *improves over* (*dominates*) another class C_2, denoted $C_1 \preceq C_2$, if for every query Q, it holds that $\forall\, \Pi_2 \in C_2(Q), \exists\, \Pi_1 \in C_1(Q)$ such that $\Pi_1 \preceq \Pi_2$. Notice that $\preceq$ over classes of plans forms a preorder. We say that C_1 *strictly dominates* C_2, denoted $C_1 \prec C_2$, if $C_1 \preceq C_2$ holds but not vice versa.

This paper studies, compares, and improves the space-time exponents of various query evaluation methods proposed in the literature, both for database queries and for probabilistic inference in graphical models. As explained, we distinguish between the *algorithm* used to evaluate the query, and the *plan*, which is a syntactic structure (e.g. a tree, or a variable order), on which we can define simple measures (e.g. depth of a tree). Every plan is canonically associated with an algorithm, but the set of plans for a given query is finite, while that of algorithms is infinite. Our choice to distinguish these two notions may be less common in the theory community, but it is standard in database systems, where a "plan" refers to a relational algebra expression. With this distinction in mind, let's examine what types of plans have been considered in the literature.

Prior work on conjunctive query answering has focused only on the time complexity. Standard *relational algebra plans*, described in all textbooks on database systems, are known to be suboptimal, hence we do not discuss them in this paper. For a full CQ (query without projection), a *worst-case optimal join* (WCOJ) algorithm runs in time $O(|D|^{\rho^*})$, where ρ^* is the fractional edge cover number of Q; this time is proportional to the worst-case output size of the query [4]. While the first WCOJ algorithms were first introduced in [33, 37], the best known variant is Generic Join (GJ) [34]. A *GJ plan* consists of a total order on the query variables, and the associated algorithm consists of nested `for`-loops, one for each variable. Somewhat surprisingly, GJ can be proven to run in time $O(|D|^{\rho^*})$ independently of the plan[1]. GJ can easily be adapted to compute conjunctive queries with projections (i.e. non-full queries), however it is no longer guaranteed to be worst-case optimal. The space required by a GJ plan consists of the space needed to store the iteration variables of the `for`-loops, which is $O(1)$ since each variable stores a single domain value, plus the space required to store the query's output. In the case of a Boolean query, the output also has size $O(1)$, therefore the space-time exponents of a GJ plan are $(0, \rho^*)$. GJ is always space-optimal, but its time complexity is in general suboptimal for queries with projections.

Handling projections efficiently, and in particular handling Boolean queries, requires new techniques. All solutions proposed in the theoretical database community are based on tree decompositions [2, 3, 17, 18, 25, 27, 36]. As the name implies, a tree decomposition is a formalism for splitting the query into small, manageable sub-queries, called *bags*, and composing these bags into a tree. Execution proceeds by computing the result of each bag, then semi-joining the results bottom-up trough the tree decomposition. The overall time complexity is given by the time required to solve every bag, which is generally referred to as the *width* of the decomposition. In the literature one finds different approaches on how to evaluate the bags, leading to various notions of width,

[1] In practice, the choice of the plan (i.e. of the variable order) makes a huge difference for the instance-specific runtime, see [38], but we do not discuss instance-optimal algorithms in this paper.

such as *tree width*, *generalized hypertree width*, and *fractional hypertree width* (*fhw*) [13]. A *tree decomposition plan* consists of both a tree decomposition, and a choice of a plan for every bag of the tree decomposition. When we choose generic join to compute the bags, then the space-time exponents of the tree decomposition plan are (*fhw*, *fhw*).

Many more inference algorithms have been described in the field of probabilistic graphical models (PGMs); we direct the reader to [9] for a comprehensive overview. While PGM inference can be expressed as a scalar sum-product query studied in this paper, the runtime analysis of PGM inference algorithms differs from that done for query evaluation because the input data in PGMs is assumed to be dense, and the runtime is always expressed as an integer power of the domain size. For example, in the case of a tree decomposition, the time exponent is the *tree width*, instead of the (much smaller) fractional hypertree width used in the analysis of query evaluation. One of the contributions of our paper consists in adapting some of the PGM inference algorithms to query evaluation, and providing their runtime analysis. The algorithms of interest to us here are the pseudo-tree based algorithm, and its refinement to caches and resets (called *purges* in [9]).

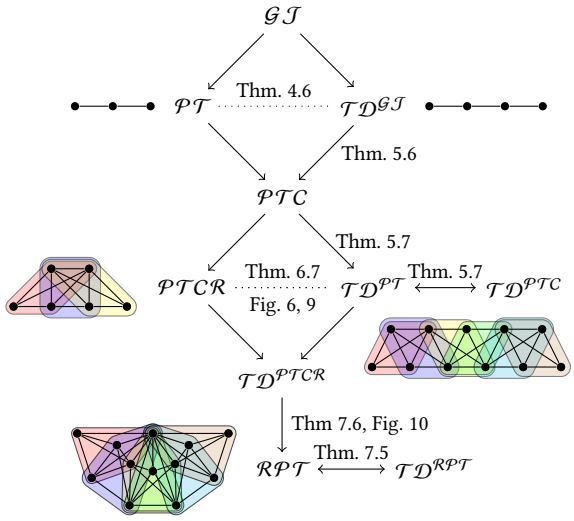
A *Pseudo-Tree* (PT) for a query is a tree whose nodes are the query variables, such that the variables of every query atom are contained in some path from the root to a leaf (formal definition in Sec. 3). A PT is a plan for the query, and its natural algorithm consists of `for`-loops, whose nesting structure is given by the PT. A Generic Join plan is a special case of a PT, where the tree consists of a single path, but, in general, a PT can have an improved time exponent because `for`-loops for independent variables can be executed sequentially, instead of nested. The space required remains optimal and is $O(1)$ (plus the space required to store the output, as we discuss in this paper). The term *pseudo-tree* was coined by Freuder and Quinn in the context of constraint optimization [15].

The runtime of a search algorithm can often be improved by the addition of a *cache*. Dechter [9] adds a cache to each node of a pseudo-tree, leading to an improved time complexity, at the cost of using more space. The cache associated to a query variable is a hash table, whose key consists of certain ancestor variables in the pseudo-tree (formal definition in Sec. 5). To allow some tradeoff between the space and time complexity, Dechter describes a refinement, by which the size of a cache can be reduced by simply removing some of these ancestors from the hash table, and resetting the cache when their value changes (details Sec. 6). But no complexity analysis is provided for these techniques, even in the simplified complexity model of the probabilistic graphical models.

While no prior work has examined the end-to-end space-time complexity of CQ evaluation, two related lines of research should be acknowledged. For one, research on *factorized databases* aims to create a space-efficient data structure from which the answers can be enumerated efficiently [35]. Second, under the name *conjunctive queries with access patterns*, prior work has explored how to materialize a space-efficient set of views to speed up subsequent query execution [39].

Our Contributions. In this paper we study the space-time tradeoff of several classes of query plans, of increased sophistication: Generic Join and Pseudo-Trees (Sec. 3), Tree Decomposition (Sec. 4), Pseudo-Trees with Caching (Sec. 5), Pseudo-Trees with Caching and Reset (Sec. 6), and finally Recursive Pseudo-Trees (Sec. 7). We fully characterize their domination relationships (Def. 1.1) and represent the resulting hierarchy in Fig. 1: lower classes have smaller exponents, and thus are better. We describe now our results in more detail, referring to this figure.

At the top of the figure is Generic Join ($\mathcal{GJ}$), which is dominated by all other classes. From there, we generalize $\mathcal{GJ}$ along two main axes. First, we consider pseudo-tree plans $\mathcal{PT}$. These were originally introduced for constraint satisfaction problems, where they correspond to Boolean queries, or, more generally, to scalar sum-product queries. We extend $\mathcal{PT}$'s to handle arbitrary outputs, and characterize their space-time complexity: unsurprisingly, they strictly dominate $\mathcal{GJ}$. We then revisit the plans based on tree decompositions, noticing that such a plan must consist



Name	Class of Query Plans Based On:
$\mathcal{GJ}$	Generic-Join
$\mathcal{PT}$	Pseudo-Tree
$\mathcal{PTC}$	Pseudo-Tree with Caching
$\mathcal{PTCR}$	Pseudo-Tree with Caching and Reset
$\mathcal{RPT}$	Recursive Pseudo-Tree
$\mathcal{TD}^{\mathcal{C}}$	Tree-Decomposition, with a query plan from $\mathcal{C}$ applied to each bag

Fig. 1. Classes of query plans, ordered by their space-time exponents: lower classes have smaller exponents and are better. An arrow $\mathcal{C}_2 \longrightarrow \mathcal{C}_1$ means that every plan in $\mathcal{C}_2$ can be mapped to a plan in $\mathcal{C}_1$ with space-time exponents at least as good; in particular, $\mathcal{C}_1 \preceq \mathcal{C}_2$ (see Def. 1.1). A missing arrow, i.e. $\mathcal{C}_2 \nrightarrow \mathcal{C}_1$, means $\mathcal{C}_1 \not\preceq \mathcal{C}_2$. In particular, all downward arrows indicate strict domination (Cor. 6.8, Thm. 7.6). The depicted graphs are examples of scalar queries with binary predicates that separate the classes; the colors represent maximal cliques.

of both the tree decomposition *and* the plans used to compute each bag. Thus, $\mathcal{TD}^{\mathcal{GJ}}$ are tree decompositions plus generic join, while $\mathcal{TD}^{\mathcal{PT}}$ are tree decompositions plus pseudo-trees. The figure shows that $\mathcal{TD}^{\mathcal{GJ}}$ dominates $\mathcal{GJ}$ (as expected); we discuss $\mathcal{TD}^{\mathcal{PT}}$ shortly.

Next we study the extension of $\mathcal{PT}$ with caches, $\mathcal{PTC}$; we slightly extend the original definition in [9] by allowing caches to be added to any subset of the nodes of the pseudo-tree, instead of all nodes. As the figure shows, $\mathcal{PTC}$ strictly dominate $\mathcal{TD}^{\mathcal{GJ}}$. This is somewhat surprising, because a tree decomposition allows a query to be computed "in small pieces", by computing one bag at a time, while a pseudo-tree consists of nested `for`-loops, requiring a global approach. Yet, by using caches, a pseudo-tree can simulate what a tree decomposition does, and, for some queries, strictly improve the space-time exponents. However, if we use pseudo-trees instead of generic join to compute the bags of a tree decomposition, then the order reverses: $\mathcal{TD}^{\mathcal{PT}}$ strictly dominates $\mathcal{PTC}$. Interestingly, if we try to improve tree decompositions by computing the bags using pseudo-trees *with caches*, $\mathcal{TD}^{\mathcal{PTC}}$, we don't gain any improvements over not using caches, $\mathcal{TD}^{\mathcal{PT}}$.

Next, we further refine pseudo-trees by allowing caches to be reset (and thus reduce their memory usage), and denote the resulting plans by $\mathcal{PTCR}$. As explained, cache reset was already discussed in [9], but no complexity analysis was provided. It turns out that computing the time complexity is more difficult in this case, because of the interaction between the various caches in the pseudo-tree. Instead, we modified the algorithm in [9], thereby both improving its time complexity, and making the analysis possible. As expected, adding resets improves the space-time complexity, and combining it with a tree decomposition, $\mathcal{TD}^{\mathcal{PTCR}}$, further improves this complexity.

Lastly, we describe a new type of query plans to dominate them all, called Recursive Pseudo-Trees. These appear to represent the best space-time tradeoff, because even by extending them with tree decompositions, the space-time exponents do not improve.

The reader may have noticed that all our results concern only upper bounds, and no lower bounds. It turns out that very few space lower bounds are known in the literature, and none of them applies to the sum-product queries studied in this paper. We discuss lower bounds in Sec. 8, were we also conjecture the space-time hardness of a specific query.

Further, all discussed methods aim at being more space efficient than existing tree decomposition based methods. As such, naturally, the time exponent is always at least as large as the fractional hypertree width (fhw). To go beyond this barrier, fundamentally different techniques are needed and are left as future work. Some intricacies of this are hinted at in Sec. 8.

Due to space limitations, proof details can be found in the full version of this paper [10]. For some of the theorems, we need to prove the non-existence of certain structures. This is done by computer-assisted exhaustive enumeration. To that end, we developed software for computing the optimal time exponent of the $\mathcal{PT}$ and $\mathcal{PTCR}$ classes when given a space exponent (https://github.com/kylebd99/submodular-width).

2 Preliminaries

Throughout this paper we fix an infinite domain $\mathbf{dom}$. We denote (sets of) variables by capital letters $A, B, C, \ldots$ ($\boldsymbol{X}, \boldsymbol{Y}, \boldsymbol{Z}, \ldots$) and (tuples of) domain values by lowercase letters $a, b, c, \ldots$ ($\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}, \ldots$). We will also refer to variables as attributes when this is more appropriate. If $\boldsymbol{X}, \boldsymbol{Y}$ are two sets of variables and $\boldsymbol{x} \in \mathbf{dom}^{\boldsymbol{X}}$, then we denote by $\boldsymbol{x}[\boldsymbol{Y}]$ the projection of $\boldsymbol{x}$ on the variables $\boldsymbol{X} \cap \boldsymbol{Y}$.

Fix a commutative semi-ring $(\mathbb{K}, \oplus, \otimes, \mathbf{0}, \mathbf{1})$. A $\mathbb{K}$-relation is a function $R\colon \mathbf{dom}^{\boldsymbol{X}} \to \mathbb{K}$ with finite support, meaning that $supp(R) := \{\boldsymbol{x} \mid R(\boldsymbol{x}) \neq \mathbf{0}\}$ is finite. When $\boldsymbol{X} = \emptyset$ then we identify R with the scalar value $s := R() \in \mathbb{K}$. By the cardinality of R, in notation $|R|$, we mean the cardinality of its support, and we write $\boldsymbol{x} \mapsto s \in R$ when $\boldsymbol{x} \in supp(R)$ and $R(\boldsymbol{x}) = s$. A $\mathbb{K}$-database D, or simply a database when $\mathbb{K}$ is clear from the context, is a tuple of $\mathbb{K}$-relations $D = (R_1^D, \ldots, R_m^D)$. Its size is $|D| := \sum_{i=1}^m |R_i^D|$.

A *sum-product query* (SPQ or query for short) is an expression of the form:

$$Q(\boldsymbol{X}) \leftarrow \bigotimes_{i=1,\ldots,m} R_i(\boldsymbol{X}_i) \tag{1}$$

where R_i are unique relation symbols and $\boldsymbol{X}, \boldsymbol{X}_1 \ldots$ are sets of variables such that $\boldsymbol{X} \subseteq var(Q) := \bigcup_i \boldsymbol{X}_i$. We refer to this query simply as Q, and call $\boldsymbol{X}$ the *output variables* or *head variables* of Q. When $\boldsymbol{X} = var(Q)$ then we say that Q is a *full query*, and when $\boldsymbol{X} = \emptyset$ then we call it a *scalar query*. If the semi-ring $\mathbb{K}$ is the set of Booleans $\mathbb{B}$, then a scalar query is called a *Boolean query*.

The semantics of Q on a database D is the $\mathbb{K}$-relation $[\![Q]\!]^D\colon \mathbf{dom}^{\boldsymbol{X}} \to \mathbb{K}$ defined as follows. Let $\boldsymbol{V} = var(Q)$ be the set of variables of the query Q in Eq. (1). Then:

$$[\![Q]\!]^D(\boldsymbol{x}) := \bigoplus_{\boldsymbol{v} \in \mathbf{dom}^{\boldsymbol{V}}:\, \boldsymbol{v}[\boldsymbol{X}] = \boldsymbol{x}} \left(\bigotimes_{i=1,\ldots,m} R_i^D(\boldsymbol{v}[\boldsymbol{X}_i]) \right)$$

We may omit the superscript D when it is clear from the context, and write R_i, $[\![Q]\!]$ for R_i^D, $[\![Q]\!]^D$.

Example 2.1. Most of our examples (in particular those used for separating families of query plans) will feature queries where all relations are binary, which allows for an intuitive representation as graphs. For instance, the 3-path on the top left of Fig. 1 represents the scalar query $Q() \leftarrow R_1(A, B) \otimes R_2(B, C)$, whose semantics is $\sum_{a,b,c \in \mathbf{dom}} R_1^D(a, b) \cdot R_2^D(b, c)$ over the natural numbers $\mathbb{N}$.

Let $\boldsymbol{Y} \subseteq var(Q)$ be a set of variables. A *fractional edge cover* of $\boldsymbol{Y}$ (with respect to Q) is a sequence of non-negative weights w_i, one for each relation R_i, such that, for every variable $A \in \boldsymbol{Y}$,

$\sum_{i:A\in\boldsymbol{X}_i} w_i \geq 1$. The *fractional edge cover number of* $\boldsymbol{Y}$, $\rho^*(\boldsymbol{Y})$, is the minimum value of $\sum_i w_i$, when the weights w_i range over fractional edge covers of $\boldsymbol{Y}$.

Complexity analysis of algorithms. We assume the RAM model, where each cell can hold a single element from the domain or from the semi-ring. As far as (normal and semi-ring) arithmetic is concerned, we assume that each operation requires constant time. For query evaluation, we only study the data complexity, i.e., the (size of the) query is considered as constant. Several of our algorithms assume a particular order of the tuples of the relations. We assume that sorting a relation does not use additional space. We will also ignore the additional log-factor due to sorting and/or lookups, and write our upper bounds using O instead of $\tilde{O}$.

3 Constant Space Query Evaluation

In this section, we briefly review Generic Join and its space-time exponents. Then, we discuss *pseudo-trees* [9, 15] and extend them in two significant ways: we generalize them to non-scalar queries, and analyze their space-time exponents using techniques similar to those used for Generic Join.

Generic Join. Consider a query $Q(\boldsymbol{X})$ in Eq. (1), with variables $var(Q) = \{A_1, \ldots, A_k\}$. The Generic Join (GJ) algorithm [32] computes $Q(D)$ in worst-case optimal time given by the AGM bound $O(|D|^{\rho^*(var(Q))})$ [4]. GJ fixes an arbitrary order on the variables, $A_1, \ldots A_k$, and computes iteratively partial assignments $\boldsymbol{y}_j = (a_1, \ldots, a_j)$ on $\boldsymbol{Y}_j = (A_1, \ldots A_j)$, for all $0 \leq j \leq k$. It starts with the empty assignment $\boldsymbol{y}_0 := ()$ and, in k nested loops, it extends it to one variable after the other, as follows. Assuming a partial tuple $\boldsymbol{y}_{j-1} = (a_1, \ldots, a_{j-1})$, the j'th nested loop is:

$$\textbf{for } a_j \textbf{ in } \bigcap_{i:A_j\in\boldsymbol{X}_i} supp(R_i[A_j|\boldsymbol{y}_{j-1}]) \textbf{ do} \ldots \tag{2}$$

where $supp(R_i[A_j|\boldsymbol{y}_{j-1}])$ represents the A_j-values in the relation R_i, restricted to tuples that agree with the values[2] $\boldsymbol{y}_{j-1}$. GJ computes the intersection above in time proportional to the smallest set, for example by iterating over the smallest set and probing (using hash tables) in all the other sets. If Q is a full SPQ, then in the inner-most loop, GJ simply outputs the assignment $\boldsymbol{y}_k \mapsto s$, where $s := \bigotimes_i R_i(\boldsymbol{y}_k[\boldsymbol{X}_i]) \in \mathbb{K}$. If Q is a scalar query, then the innermost loop computes the sum of these values $s \in \mathbb{K}$. In the general case, $\emptyset \subseteq \boldsymbol{X} \subseteq var(Q)$, GJ maintains a hash-table $OUT\colon \mathbf{dom}^{\boldsymbol{X}} \to \mathbb{K}$ to store the current output, and the inner-most loop updates $OUT(\boldsymbol{y}_k[\boldsymbol{X}]) := OUT(\boldsymbol{y}_k[\boldsymbol{X}]) \oplus s$. The time complexity of GJ is $O(|D|^{\rho^*(var(Q))})$. Its space complexity, i.e. the memory size required in addition to the input database, consists of the k variables $a_1, \ldots, a_k$, with a total size of $O(1)$, plus the space required to store the output OUT, whose size is $O(|D|^{\rho^*(\boldsymbol{X})})$.

Definition 3.1. A *Generic Join Plan* of a query Q is a total order on its variables, $\Pi = (A_1, \ldots, A_k)$. We denote by $\mathcal{GJ}$ (resp. $\mathcal{GJ}(Q)$) the set of all Generic Join plans (of Q). The space-time exponents of any $\Pi \in \mathcal{GJ}(Q)$ are $e(\Pi) = (\rho^*(\boldsymbol{X}), \rho^*(var(Q)))$; in particular, if Q is a scalar query, then the space-time exponents are $(0, \rho^*(var(Q)))$.

Theorem 3.2 ([34]). *If $\Pi \in \mathcal{GJ}(Q)$, then Π computes $[\![Q]\!]$ in space and time given by $e(\Pi)$. Concretely, the space used is $O(|D|^{\rho^*(\boldsymbol{X})})$, and the time spent is $O(|D|^{\rho^*(var(Q))})$.*

When the cardinalities of individual relations are known, $N_1 = |R_1|, N_2 = |R_2|, \ldots$, then tighter space-time bounds are given by $O(\prod_i N_i^{w_i^*})$, where $w_1^*, w_2^*, \ldots$ is the fractional edge cover of $\boldsymbol{X}$ (or $var(Q)$ respectively) that minimizes $\prod_i N_i^{w_i^*}$. In the special case when $N_1 = N_2 = \cdots = N$ this

[2] Formally, $supp(R_i[A_j|\boldsymbol{y}_{j-1}]) = \{\boldsymbol{z}[A_j] \mid \boldsymbol{z} \in supp(R_i), \boldsymbol{z}[\boldsymbol{Y}_{j-1}] = \boldsymbol{y}_{j-1}[\boldsymbol{X}_i]\}$.

is equal to $O(|D|^{\rho^*})$. In this paper we prefer to use the simpler formula, and will note where the tighter formula is needed.

Pseudo-trees. For a *full* query, any GJ plan is worst-case optimal, because the output size can be as large as $O(|D|^{\rho^*(var(Q))})$ [4]. But when Q is not full, GJ is no longer optimal. A pseudo-tree, defined below, improves the time exponent of GJ, without increasing its space exponent.

Example 3.3. For a simple intuition, consider the 3-path scalar query in Example 2.1. GJ computes it using 3 nested loops, corresponding to the variables A, B, C; intuitively it computes the expression $\sum_a \sum_b \sum_c R_1(a,b)R_2(b,c)$, with runtime $O(|R_1| \cdot |R_2|)$. A pseudo-tree based algorithm, in contrast, iterates over B first, then performs two independent loops that iterate over A and C respectively; this corresponds to the expression $\sum_b (\sum_a R_1(a,b)) \cdot (\sum_c R_2(b,c))$, and the runtime is $O(|R_1| + |R_2|)$.

If $T = (\boldsymbol{V}, E)$ is a directed tree and $A \in \boldsymbol{V}$, then we denote by $anc(A)$ the set of ancestors of A excluding A, and write $\overline{anc}(A) = anc(A) \cup \{A\}$. Similarly, we write $desc(A), \overline{desc}(A)$ for the set of descendants of A, without and with A respectively.

Fix an SPQ $Q(\boldsymbol{X}) \leftarrow \bigotimes_i R_i(\boldsymbol{X}_i)$ (see Eq. (1)), and let $\boldsymbol{V} = var(Q)$.

Definition 3.4 ([9, 15]). A *pseudo-tree* (PT) of Q is a directed tree $P = (\boldsymbol{V}, E)$, satisfying:

- Every atom $R_i(\boldsymbol{X}_i)$ is contained in a branch: formally, $\exists A \in \boldsymbol{V}$ such that $\boldsymbol{X}_i \subseteq \overline{anc}(A)$.

The term *pseudo-tree* was introduce by Freuder and Quinn in the context of constraint optimization [15], and studied extensively by Dechter [9]. Pseudo-trees are a generalization of *normal trees*, also called *Trémaux trees* used in graph theory [11, Ch.1], which are pseudo-trees where every tree edge is also an edge in the graph.

For any variable $A \in \boldsymbol{V}$, we denote by $out(A) := desc(A) \cap \boldsymbol{X}$ and by $\overline{out}(A) := \overline{desc}(A) \cap \boldsymbol{X}$.

Definition 3.5. The class of query plans $\mathcal{PT}(Q)$ consists of pseudo-trees P of Q and their space and time exponents are defined as:

$$s(P) := \max_{A \in var(Q)} \rho^*(\overline{out}(A)), \quad t(P) := \max_{A \in var(Q)} \rho^*(\overline{anc}(A) \cup out(A)).$$

When Q is a scalar query, then the space exponent is $s(P) = 0$. Pseudo-trees strictly dominate GJ, i.e. $\mathcal{PT} \prec \mathcal{GJ}$ as per Def. 1.1 because, any variable order of a GJ can be converted into a linear PT $A_1 - A_2 - \cdots - A_k$, and the two plans have the same space-time complexity thus $\mathcal{PT} \preceq \mathcal{GJ}$. On the other hand, Example 3.3 shows that $\mathcal{GJ} \not\preceq \mathcal{PT}$. This establishes the first arrow in Fig. 1.

It remains to describe an algorithm that, given a pseudo-tree P for Q, computes $[\![Q]\!]$ in space-time given by the exponents in Def. 3.5. First we need to introduce some notations. If $R(\boldsymbol{X}) : \mathbf{dom}^{\boldsymbol{X}} \to \mathbb{K}$ is a $\mathbb{K}$-relation and $A \in \boldsymbol{X}$, then we write $supp(R[A]) := \{\boldsymbol{x}[A] \mid \boldsymbol{x} \in supp(R)\}$ for the projection of $supp(R)$ on the attribute A; in other words, this is the A-column of R. We generalize this as follows. Let $\boldsymbol{Y}$ be a set of variables s.t. $A \notin \boldsymbol{Y}$, and $\boldsymbol{y} \in \mathbf{dom}^{\boldsymbol{Y}}$. We write $supp(R[A|\boldsymbol{y}])$ for the projection on A of the tuples in $supp(R)$ that agree with $\boldsymbol{y}$: $supp(R[A|\boldsymbol{y}]) := \{\boldsymbol{x}[A] \mid \boldsymbol{x} \in supp(R), \boldsymbol{x}[\boldsymbol{Y}] = \boldsymbol{y}[\boldsymbol{X}]\}$.

Algorithm 1 (ignore the gray lines for now) evaluates $Q(\boldsymbol{X})$ recursively, by following the structure of the pseudo-tree P. We start from the root A_1, and proceed recursively in the tree. Assume we have followed a path $\boldsymbol{Y} = (A_1, A_2, \ldots, A_{j-1})$, and have bound these variables to the tuple $\boldsymbol{y} \in \mathbf{dom}^{\boldsymbol{Y}}$. For a child A of A_{j-1}, SOLVE$(A, \boldsymbol{y})$ first computes the following $\mathbb{K}$-relation $[\![Q[A|\boldsymbol{y}]]\!] : \mathbf{dom}^A \to \mathbb{K}$:

$$[\![Q[A|\boldsymbol{y}]]\!] := \left\{ a \mapsto \bigotimes_{i : \boldsymbol{X}_i \setminus \boldsymbol{Y} = \{A\}} R_i(\boldsymbol{y}'[\boldsymbol{X}_i]) \;\middle|\; a \in \bigcap_{i : A \in \boldsymbol{X}_i} supp(R_i[A|\boldsymbol{y}]), \boldsymbol{y}' = (\boldsymbol{y}, a) \right\} \tag{3}$$

Algorithm 1 PT Algorithm (excl. gray parts)
Algorithm 3 PTC Algorithm (incl. gray parts)

Input: Query $Q(\boldsymbol{X})$, pseudo-tree P, caches $\boldsymbol{C} \subseteq var(Q)$
Output: $[\![Q]\!] : \mathbf{dom}^{X} \to \mathbb{K}$

return SOLVE($root(P)$, ())
for $A \in \boldsymbol{C}$
  $M_A \leftarrow \emptyset$
function SOLVE($A, \boldsymbol{y}$): $\mathbf{dom}^{\overline{out}(A)} \to \mathbb{K}$
if $A \in \boldsymbol{C} \wedge \boldsymbol{y}[con(A)] \in keys(M_A)$
  return $M_A(\boldsymbol{y}[con(A)])$
OUT $\leftarrow \{\boldsymbol{z} \mapsto \mathbf{0} \mid \boldsymbol{z} \in \mathbf{dom}^{\overline{out}(A)}\}$
for $a \mapsto s \in [\![Q[A|\boldsymbol{y}]]\!]$ ▷ see Eq. (3)
  $\boldsymbol{y}' \leftarrow (\boldsymbol{y}, a)$
  TMP $\leftarrow \{a[\boldsymbol{X} \cap \{A\}] \mapsto s\}$
  for $B \in child(A)$
    TMP $\leftarrow$ TMP $\otimes$ SOLVE($B, \boldsymbol{y}'$)
  OUT $\leftarrow$ OUT $\oplus$ TMP
if $A \in \boldsymbol{C}$
  $M_A \leftarrow M_A \cup \{\boldsymbol{y}[con(A)] \mapsto$ OUT $\}$
return OUT

Algorithm 2 Alg. 1 for the PT in Fig. 3

$\text{OUT}^B \leftarrow 0$
for $b \mapsto s^b \in [\![Q[B]]\!]$ ▷ $s^b = 1$
  ▷ $b \in R_1[B] \cap \cdots \cap R_5[B]$
  $\text{OUT}^A \leftarrow 0$
  for $a \mapsto s^a \in [\![Q[A|b]]\!]$ ▷ $s^a = R_1(a, b)$
    $\text{OUT}^A \leftarrow \text{OUT}^A + s^a$
  $s^b \leftarrow s^b \cdot \text{OUT}^A, \text{OUT}^E \leftarrow 0$
  for $e \mapsto s^e \in [\![Q[E|b]]\!]$ ▷ $s^e = R_4(b, e)$
    ▷ $e \in R_4[E|b] \cap R_6[E] \cap R_7[E]$
    $\text{OUT}^D \leftarrow 0$
    for $d \mapsto s^d \in [\![Q[D|be]]\!]$
      ▷ $s^d = R_3(b, d) \cdot R_6(e, d)$
      $\text{OUT}^D \leftarrow \text{OUT}^D + s^d$
    $s^e \leftarrow s^e \cdot \text{OUT}^D, \text{OUT}^F \leftarrow 0$
    for $f \mapsto s^f \in [\![Q[F|be]]\!]$
      $\text{OUT}^F \leftarrow \text{OUT}^F + s^f$
    $s^e \leftarrow s^e \cdot \text{OUT}^F, \text{OUT}^E \leftarrow \text{OUT}^E + s^e$
  $s^b \leftarrow s^b \cdot \text{OUT}^E, \text{OUT}^C \leftarrow 0$
  for $c \mapsto s^c \in [\![Q[C|b]]\!]$
    $\text{OUT}^C \leftarrow \text{OUT}^C + s^c$
  $s^b \leftarrow s^b \cdot \text{OUT}^C, \text{OUT}^B \leftarrow \text{OUT}^B + s^b$
return OUT^B

Intuitively, $[\![Q[A|\boldsymbol{y}]]\!]$ contains the possible values of A that extend $\boldsymbol{y}$ in a manner consistent with Q. To compute (3), the algorithm iterates over all values $a \in \bigcap_i supp(R_i[A|\boldsymbol{y}])$, by intersecting the A-attributes of all relations that contain the attribute A (similarly to Generic Join in Eq. (2)), then maps each such value a to $s \in \mathbb{K}$, where s is the product of all $\mathbb{K}$-values of the relations whose last attribute (in the order of the pseudo-tree) is A: the condition $\boldsymbol{X}_i \setminus Y = \{A\}$ checks that A is the last attribute of $R_i(\boldsymbol{X}_i)$, while $R_i(\boldsymbol{y}'[X_i]) \in \mathbb{K}$ is its value associated to $\boldsymbol{y}' := (\boldsymbol{y}, a)$. Like GJ, the algorithm computes the intersection of $supp(R_i[A|\boldsymbol{y}])$ on the fly in time proportional to the smallest set. That is, the algorithm iterates over the values $a \mapsto s$ in $[\![Q[A|\boldsymbol{y}]]\!]$, and performs a recursive call on each child B of A. When Q is a scalar query, then both TMP and OUT are scalars (because $\overline{out}(A) = \emptyset$), and the algorithm simply multiplies the values of all children B, then adds up these values over all a's. When Q has output variables $\boldsymbol{X}$, then both TMP and OUT are $\mathbb{K}$-relations. OUT has type $\mathbf{dom}^{\overline{out}(A)} \to \mathbb{K}$. Initially, TMP has attributes $\boldsymbol{X} \cap \{A\}$ (i.e. either $\emptyset$ or $\{A\}$), while the natural join[3] $TMP \leftarrow TMP \otimes$ SOLVE($B, \boldsymbol{y}'$) extends TMP with $\overline{out}(B)$, so that, after processing all children B of A, TMP has the same schema as OUT, and the algorithm adds up these values over all a's.

THEOREM 3.6. *If $P \in \mathcal{PT}(Q)$, then Algorithm 1 computes $[\![Q]\!]$ in time $O(|D|^{t(P)})$ and uses $O(|D|^{s(P)})$ space (where s, t are given by Def. 3.5).*

[3]The natural join $R_1 \otimes R_2$ of relations $R_i : \mathbf{dom}^{\boldsymbol{U}_i} \to \mathbb{K}$ is defined as $(R_1 \otimes R_2)(\boldsymbol{u}) := R_1(\boldsymbol{u}[\boldsymbol{U}_1]) \otimes R_2(\boldsymbol{u}[\boldsymbol{U}_2])$ where $\boldsymbol{u} \in \mathbf{dom}^{\boldsymbol{U}_1 \cup \boldsymbol{U}_2}$. Note the schemas of TMP and SOLVE($B, \boldsymbol{y}'$) are disjoint and, hence, it degenerates to a Cartesian product.

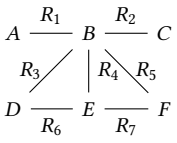


Fig. 2. Query $Q_{\triangle}$

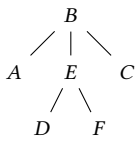


Fig. 3. Pseudo-tree for $Q_{\triangle}$

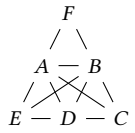


Fig. 4. Query $Q_{\triangleleft}$

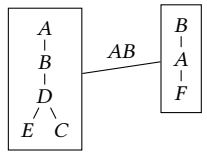


Fig. 5. $\mathcal{TD}^{\mathcal{PT}}$ plan for $Q_{\triangleleft}$

Example 3.7. Consider the scalar query $Q_{\triangle}()$ depicted in Fig. 2, over 7 $\mathbb{N}$-relations. Its fractional edge cover number is $\rho^* := \rho^*(var(Q)) = 4$. Generic Join computes the query using 6 nested loops, one for each variable $A, B, \ldots, F$, and its runtime is $O(|D|^4)$. Consider now the PT P in Fig. 3. We check that it satisfies the condition in Def. 3.4: indeed, each relation is included in a branch, e.g., relation $R_5(B, F)$ is included in the branch $B - E - F$. Its space-time exponents are $e(P) = (s(P), t(P)) = (0, 3/2)$; indeed, $s(P) = \rho^*(\emptyset) = 0$ because $Q_{\triangle}$ is a scalar query, and $t(P) = \max(\rho^*(\overline{anc}(A)), \rho^*(\overline{anc}(B)), \rho^*(\overline{anc}(C)), \rho^*(\overline{anc}(D)), \ldots) = \max(1, 1, 1, 3/2, 1, 3/2) = 3/2$. Algorithm 2 (the expansion of Algorithm 1) computes $Q_{\triangle}$ following the PT. It starts with a loop for the variable B, but, unlike Generic Join, it continues with three independent loops, one for each variable A, E, C respectively. The for-loop for E contains another two, independent for-loops, for D and F. Notice that the first loop, for B, is over the intersection of the attributes B of all relations that contain B (to reduce clutter we omitted *supp* in the comment), while the associated value s^b is 1, because there are no relations that "end" at B. On the other hand, the loop for A associates to s^a the value $R_1(a, b)$, because A is the last attribute of the relation $R_1(A, B)$. Similarly, the loop for e is over the intersection of the E-columns R_4, R_6, R_7 (where R_4 is restricted to the value b), and the value we associate to e is $s^e = R_4(b, e)$, because $R_4(B, E)$ is the only relation that ends at E. The runtime of Algorithm 2 is $O(|D|^{3/2})$, because it is dominated by the nested loops $B - E - D$ and $B - E - F$, each requiring only $O(|D|^{3/2})$ steps. Since $Q_{\triangle}$ is scalar, both TMP and OUT are scalars (to reduce clutter we did not include the variables TMP but used s instead). Consider what happens if we modify the query to have output variables $\boldsymbol{X} = \{D, F\}$. Then, $\text{OUT}^A, \text{OUT}^C$ are still scalars while OUT^D is a $\mathbb{K}$-relation with attribute D, OUT^F has the attribute F, and $\text{OUT}^B, \text{OUT}^E$ have attributes DF. Further, the space-time exponents become $(\rho^*(DF), \rho^*(BEDF)) = (2, 2)$ computed, e.g., at E.

4 Revisiting Tree Decompositions

Tree decompositions (TDs) have been extensively studied in the literature [19–21, 30]. Like a PT, a TD may decrease the time exponent of generic join, but it may increase the space exponent. We briefly review the definition of TDs , then show that they are incomparable to PTs.

In this section we fix a query $Q(\boldsymbol{X}) \leftarrow \bigotimes_i R_i(\boldsymbol{X}_i)$, as in Eq. (1).

Definition 4.1. A tree decomposition of $Q(\boldsymbol{X})$ is a tuple $TD = (T, \chi)$ where $T = (V, E)$ is a directed tree and $\chi : V \to 2^{var(Q)}$ is a function from the nodes to sets of variables, satisfying:

(1) $\boldsymbol{X} \subseteq \chi(root(T))$,
(2) $\forall A \in var(Q)$, the nodes v with $A \in \chi(v)$ must form a connected subset of V,
(3) $\forall \boldsymbol{X}_i \exists v \in V$ s.t. $\boldsymbol{X}_i \subseteq \chi(v)$. We pick an arbitrary such v and say $R_i(\boldsymbol{X}_i)$ is covered by v.

The sets $\chi(v)$ are called *bags*. Readers familiar with free-connex tree decomposition may notice that condition (1) is more restrictive, but that's OK for our purpose, because we only consider worst-case optimal algorithms, and do not consider constant-delay algorithms [6]. Any free-connex tree decomposition can be converted into a tree that satisfies Def. 4.1, by adding a bag with all output variables $\boldsymbol{X}$, without increasing its worst-case total runtime.

To each tree vertex $v \in V$ we associate a query, as follows. Let $\boldsymbol{Y}^v := \chi(v)$ and $\boldsymbol{Z}^v := \chi(v) \cap \chi(\mathit{parent}(v))$ (when v is the root node, then $\boldsymbol{Z}^v := \boldsymbol{X}$) and define the $\mathbb{K}$-relation $R_i^v : \mathbf{dom}^{\boldsymbol{X}_i \cap \boldsymbol{Y}^v} \to \mathbb{K}$ as $R_i^v := R_i$ when R_i is covered by v, and $R_i^v(\boldsymbol{z}) := (\{\boldsymbol{z} \mapsto \mathbf{1} \mid \exists \boldsymbol{x} \in \mathit{supp}(R_i) \text{ s.t. } \boldsymbol{z} = \boldsymbol{x}[\boldsymbol{Y}^v]\})$ otherwise. In other words, for all $v \in V$ but one, the relation R_i^v is a $\{\mathbf{0}, \mathbf{1}\}$-relation consisting of the projection of $\mathit{supp}(R_i)$ on the variables $\boldsymbol{X}_i \cap \boldsymbol{Y}^v$. Define the *sub-query* Q^v at node v as:

$$Q^v(\boldsymbol{Z}^v) \leftarrow \bigotimes_i R_i^v(\boldsymbol{X}_i \cap \boldsymbol{Y}^v) \otimes \bigotimes_{w \in \mathit{child}(v)} Q^w(\boldsymbol{Z}^w) \tag{4}$$

Notice that $\mathit{var}(Q^v) = \chi(v) = \bigcup_i \mathit{var}(R_i^v) = \boldsymbol{Y}^v$, since $\boldsymbol{Z}^w \subseteq \chi(v)$ for all $w \in \mathit{child}(v)$.

We use the TD to compute $Q(\boldsymbol{X})$ by computing all subqueries Q^v, bottom-up. For each $v \in V$, first compute recursively the subqueries $Q^w(\boldsymbol{Z}^w)$ of its children w, materialize these results, then compute Q^v as in Eq. (4). The materialized results $Q^w(\boldsymbol{Z}^w)$ are called *messages* in the literature.

It remains to decide what plan we use to compute the subqueries Q^v. This justifies the following:

Definition 4.2. Let $\mathcal{C}$ be a class of query plans for evaluating SPQs. The class of plans $\mathcal{TD}^{\mathcal{C}}(Q)$ consists of pairs (TD, π), where TD is a tree decomposition of Q, and π is a function that maps each vertex v in TD to a plan $\pi^v \in \mathcal{C}(Q^v)$. The space and time exponents are

$$s(TD, \pi) = \max_v s(\pi^v), \quad t(TD, \pi) = \max_v t(\pi^v). \tag{5}$$

Theorem 4.3. *Let s, t be the space and time exponents of the plans $\mathcal{C}$, such that for every query Q' and plan $\pi \in \mathcal{C}(Q')$, $[\![Q']\!]$ can be computed in space $O(|D|^{s(\pi)})$ and time $O(|D|^{t(\pi)})$. Then, for every plan $(TD, \pi) \in \mathcal{TD}^{\mathcal{C}}(Q)$, $[\![Q]\!]$ can be computed in space $O(|D|^{s(TD,\pi)})$ and time $O(|D|^{t(TD,\pi)})$, where $s(TD, \pi)$ and $t(TD, \pi)$ are given by Eq.* (5).

Two remarks are in order. First, we notice that the space needed to store the message Q^v that is sent to the parent is already accounted for by the space exponent $s(\pi^v)$. Second, when computing the space-time exponents of the query Q^v, we need to account for both the sizes of the input relations R_i, and for the sizes of the incoming messages Q^w. The latter can be asymptotically larger: if the bags are computed using Generic Join, or Pseudo-Trees, then we need to use the tighter upper bound expression $O(\prod_i N_i^{w_i^*})$ rather than $O(|D|^{\rho^*})$, see Sec. 3. We illustrate with an example.

Example 4.4. Consider the 4-cycle query $Q_\square() \leftarrow E_1(A_1, A_2) \otimes E_2(A_2, A_3) \otimes E_3(A_3, A_4) \otimes E_4(A_1, A_4)$, and the tree decomposition with two bags $\chi(v) = \{A_1A_2A_3\}$, $\chi(w) = \{A_3A_4A_1\}$, where v is the root. Assume $|E_1| = |E_2| = |E_3| = |E_4| = N$. The sub-query at w is $Q^w(A_1A_3) = E_3(A_3A_4) \otimes E_4(A_1A_4)$. Its space-time exponents are $(2, 2)$, because the optimal fractional edge cover (for both $\mathit{var}(Q)$ and $\{A_1, A_3\}$) is $w_3 = w_4 = 1$. This means that GJ can compute it in time $O(N^2)$, and its output takes space $O(N^2)$. Consider next the sub-query $Q^v() = E_1(A_1A_2) \otimes E_2(A_2A_3) \otimes Q^w(A_1A_3)$. Although Q^v has the shape of a triangle query, GJ does not compute it in time $O(N^{1.5})$ but rather in time $O(N^2)$, because the message Q^w can be as large as $O(N^2)$. The optimal fractional edge cover, which minimizes $|E_1|^{w_1} \cdot |E_2|^{w_2} \cdot |Q^w|^{w_0}$, is $w_1 = w_2 = 1$ and $w_0 = 0$, and GJ will compute this query in time $O(N^2)$ and space $O(1)$. Therefore, the space-time exponents of this tree decomposition are $(2, 2)$.

In this simple 4-cycle example, using pseudo-trees instead of GJ to compute the bags does not improve either the space or time exponent. However, we will see in Example 4.5 that replacing GJ with PT can lead to asymptotic improvements.

When all cardinalities are equal $|R_1| = |R_2| = \cdots$ then the simplified formula $O(|D|^{\rho^*})$ still gives an upper bound on the time and space complexity, assuming that we ignore the messages Q^w when

computing ρ^*; equivalently, we assign each of them the weight 0. Hence, for a $\mathcal{TD}^{\mathcal{GJ}}$ plan with tree decomposition (T, χ), we get the following space-time exponents (known as folklore):

$$s(T,\chi) = \max\left(\rho^*(\boldsymbol{X}), \max_{(u,v)\in E(T)} \rho^*(\chi(u)\cap\chi(v))\right), \quad t(T,\chi) = \max_{v\in V(T)} \rho^*(\chi(v)) = fhw(T,\chi). \quad (6)$$

In words, the space exponent is the maximal fractional edge cover number of the intersection of adjacent bags (or the maximal output size) while the time exponent is simply the fraction hypertree width of the tree decomposition.

We end this section by comparing $\mathcal{TD}^C$ for different C and proving some of the domination relations in Fig. 1.

Example 4.5. $\mathcal{TD}^{\mathcal{PT}}$ plans can strictly improve both the time and the space exponents of $\mathcal{TD}^{\mathcal{GJ}}$ plans. For example, consider the query $Q_{\triangle}()$ in Fig. 4, and the $\mathcal{TD}^{\mathcal{PT}}$ plan in Fig. 5. Using GJ to process the bags results in space-time exponents of $(1, 5/2)$ while using the PTs results in $(1, 2)$. We can improve the exponents even further, by using a TD with a single bag $ABCDEF$. To compute it, add the node F as a child of B to the left PT in Fig. 5. The space-time exponents decreased to $(0, 2)$.

Theorem 4.6. *The following hold:* $\mathcal{PT} \not\preceq \mathcal{TD}^{\mathcal{GJ}}, \mathcal{TD}^{\mathcal{GJ}} \not\preceq \mathcal{PT}$, *and* $\forall C\colon \mathcal{TD}^C \preceq C$.

Proof Sketch. $\mathcal{PT} \not\preceq \mathcal{TD}^{\mathcal{GJ}}$ follows from the fact that the 4-path query $R(AB)\otimes S(BC)\otimes T(CD)$ admits a $\mathcal{TD}^{\mathcal{GJ}}$ query plan Π with $e(\Pi) = (1, 1)$, while time exponent 1 is not feasible in $\mathcal{PT}$. $\mathcal{TD}^{\mathcal{GJ}} \not\preceq \mathcal{PT}$ is proved by showing that the 3-path query in Example 2.1 has a plan $\Pi \in \mathcal{PT}$ with $e(\Pi) = (0, 1)$, while $\mathcal{TD}^{\mathcal{GJ}}$ allows for plans Π' with $e(\Pi') = (1, 1)$ or $e(\Pi') = (0, 2)$ but not $(0, 1)$; none dominate $e(\Pi) = (0, 1)$. $\mathcal{TD}^C \preceq C$ follows as $Q^v = Q$ for the single-bag TD. □

5 Caching

In this section, we describe the addition of caching to pseudo-trees. While this method was introduced in [9], we extend it here to handle output variables, and perform its (non-obvious!) space-time analysis. A *cache* is a data structure that maps from a set of *keys* to a set of *values*.

Example 5.1. To motivate caching, consider the 4-path query $Q() \leftarrow R(AB)\otimes S(BC)\otimes T(CD)$, over the semiring $\mathbb{N}$, and consider the linear PT $A - B - C - D$, where A is the root. The runtime of this plan is given by the AGM bound, $O(N^2)$. Intuitively, this query plan corresponds to the summation $\sum_a (\sum_b R(ab)\cdot(\sum_c S(bc)\cdot(\sum_d T(cd))))$. We note that the subexpression $M_C(b) := \sum_c S(bc)\cdot\sum_d T(cd)$ is independent of a, and, by caching the values $M_C(b)$, we can avoid recomputing this expression. Similarly, we can cache $M_D(c) := \sum_d T(cd)$. By adding caches to Algorithm 1 we can trade off space for time. In our example, the two caches decrease the runtime of the PT above from $O(N^2)$ to $O(N)$, while the space increases from $O(1)$ to $O(N)$.

Throughout this section we fix a query $Q(\boldsymbol{X}) \leftarrow \bigotimes_i R_i(\boldsymbol{X}_i)$ and a pseudo-tree $P = (\boldsymbol{V}, E)$, where $\boldsymbol{V} = var(Q)$. Assume we decide to cache the values returned by $\textsc{solve}(A, \boldsymbol{y})$ of the recursive Algorithm 1, for some $A \in \boldsymbol{V}$. The *key* of the cache M_A at A is called the *context* of A.

Definition 5.2 ([9]). The *context* of a variable $A \in \boldsymbol{V}$ is defined as

$$con(A) = \{B \in anc(A) \mid \exists C \in \overline{desc}(A), \text{ s.t. } B, C \in \boldsymbol{X}_i \text{ for some atom } R_i(\boldsymbol{X}_i) \text{ of } Q\},$$

and the *closed context* of A is $\overline{con}(A) = con(A)\cup\{A\}$.

The main property of $con(A)$ is that the value returned by $\textsc{solve}(A, \boldsymbol{y})$ depends only on $\boldsymbol{y}[con(A)]$ and not on the entire tuple $\boldsymbol{y}$. Therefore, we can cache these values in a cache M_A with key $con(A)$, whose values are $\mathbb{K}$-relations of type $\mathbf{dom}^{\overline{out}(A)} \to \mathbb{K}$ (the type returned by $\textsc{solve}(A, \boldsymbol{y})$). The

type of this cache is $M_A \colon \mathbf{dom}^{con(A)} \to (\mathbf{dom}^{\overline{out}(A)} \to \mathbb{K})$, which is equivalent, through curry-uncurry, to $M_A \colon \mathbf{dom}^{con(A) \cup \overline{out}(A)} \to \mathbb{K}$. Therefore, the space usage of the cache M_A is given by $\rho^*(con(A) \cup \overline{out}(A))$. The time spent by the algorithm at node A will be reduced, because it only needs to call $\textsc{solve}(A, \boldsymbol{y})$ once for each distinct value $\boldsymbol{y}[con(A)]$. We are now ready to define a pseudo-tree with caching:

Definition 5.3. A *pseudo-tree with caching* (PTC) of Q is a pair $(P, \boldsymbol{C})$, where $P = (\boldsymbol{V}, E)$ is a PT of Q, and $\boldsymbol{C} \subseteq \boldsymbol{V}$ is a subset of the variables for which we add a cache. We require $root(P) \in \boldsymbol{C}$.

The gray lines in Algorithm 1 represent its extension to caching, which we call Algorithm 3. We now compute its space-time complexity. We have already seen the space requirement for a cache M_A and, hence, know the space complexity. So, let us focus on the time complexity. If we do not use any caches, then the time complexity of the `for`-loop in line 8 of the Algorithm is $\rho^*(\overline{anc}(A) \cup out(A))$: this is what we used in Def. 3.5. But if some $B \in \overline{anc}(A)$ uses a cache, then it suffices to consider only the set[4] $con(B) \cup [A, B]$ instead of $\overline{anc}(A)$. Indeed, consider two calls to $\textsc{solve}(A, -)$: first, $\textsc{solve}(A, \boldsymbol{x})$, followed at some later time by a second call $\textsc{solve}(A, \boldsymbol{y})$. If $\boldsymbol{x}[con(B) \cup (A, B]] = \boldsymbol{y}[con(B) \cup (A, B]]$ then the second call will not happen, because of the cache at B. The only variables in $\overline{anc}(A)$ relevant to the time consumption in line 8 are $con(B)$, and $[A, B]$. If A has multiple ancestors B with a cache, then we will only consider the lowest one (closest to A). This justifies the following generalization of Def. 3.5:

Definition 5.4. The class of query plans $\mathcal{PTC}(Q)$ consists of pseudo-trees with caching $(P, \boldsymbol{C})$ of Q and their space and time exponents are defined as:

$$s(P, \boldsymbol{C}) := \max_{A \in \boldsymbol{C}} \rho^*(con(A) \cup \overline{out}(A)), \quad t(P, \boldsymbol{C}) := \max_{A \in V(P)} \rho^*(con(B_A) \cup [A, B_A] \cup out(A)),$$

where[5] $B_A := \min(\boldsymbol{C} \cap \overline{anc}(A))$.

The reader may check that, when there are no caches (i.e., $\boldsymbol{C} = \{root(P)\}$), then the space and time exponents of the PTCs coincide with those of PTs in Def. 3.5. In general, we prove:

Theorem 5.5. *If $(P, \boldsymbol{C}) \in \mathcal{PTC}(Q)$, then Algorithm 3 computes $[\![Q]\!]$ in time $O(|D|^{t(P,\boldsymbol{C})})$ and uses space $O(|D|^{s(P,\boldsymbol{C})})$.*

We end this section by establishing the domination relationships involving $\mathcal{PTC}$ in Fig. 1: $\mathcal{PTC}$ improves upon $\mathcal{TD}^{\mathcal{GJ}}$, and augmenting TDs with $\mathcal{PT}$ or with $\mathcal{PTC}$ give equivalent classes. The separation shown in Theorem 4.6 implies that $\mathcal{PTC}$ strictly improves upon all classes above it. However, we state strict dominations collectively later in Corollary 6.8

Theorem 5.6. *The class $\mathcal{PTC}$ dominates $\mathcal{TD}^{\mathcal{GJ}}$, i.e., $\mathcal{PTC} \preceq \mathcal{TD}^{\mathcal{GJ}}$.*

Proof Sketch. Given a plan $\Pi := ((T, \chi), \pi) \in \mathcal{TD}^{\mathcal{GJ}}$, we construct a plan $\Pi_0 := (P, \boldsymbol{C}) \in \mathcal{PTC}$ such that $\Pi_0 \preceq \Pi$. The construction is based on the variable elimination procedure for a tree decomposition [24], and proceeds by induction on the number of bags in T. If T has a single bag, then Π is essentially a $\mathcal{GJ}$ plan, and the claim follows from $\mathcal{PT} \preceq \mathcal{GJ}$. Otherwise, let v be a leaf of T, and $p := parent(v)$. We eliminate all variables $\{A_1, \ldots, A_k\} := \chi(v) \setminus \chi(p)$. Let $\boldsymbol{Z}$ be their neighbors, $\boldsymbol{Z} := \{B \mid \exists \text{ atom } R_i(\boldsymbol{X}_i), \exists j, \text{ s.t. } A_j, B \in \boldsymbol{X}_i\}$. Let Q' to be the query obtained from Q by removing all variables $A_1, \ldots, A_k$, and adding a new atom $R(\boldsymbol{Z})$. Let $\Pi' = ((T', \chi), \pi)$ be the plan obtained from Π by removing the leaf v. By induction hypothesis, Π' can be converted to a $\mathcal{PTC}$

[4] $[A, B]$ denotes the set of nodes between A and B; $(A, B]$ denotes $[A, B] \setminus \{A\}$.

[5] Given a nonempty set $\boldsymbol{S}$ on some branch of the tree, i.e. $\boldsymbol{S} \subseteq \overline{anc}(A)$ for some A, we denote by $\min(\boldsymbol{S})$ its smallest element, i.e. $\min(\boldsymbol{S}) \in \boldsymbol{S}$ and $\boldsymbol{S} \subseteq \overline{anc}(\min(\boldsymbol{S}))$.

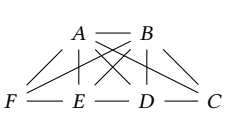


Fig. 6. Query $Q_{\triangle}$

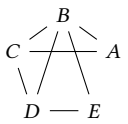


Fig. 7. Query $Q_{\pentagon}$

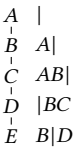


Fig. 8. PTCR for $Q_{\pentagon}$

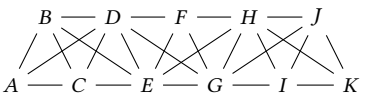


Fig. 9. Query $Q_{\triangle\triangle}$

plan $\Pi'_0 = (P', \boldsymbol{C}')$ such that $\Pi'_0 \leq \Pi'$. All variables $\boldsymbol{Z}$ belong to a branch of Π'_0 (because of the atom $R(\boldsymbol{Z})$). Construct the pseudo-tree P from P' by adding a branch A_1-A_2-$\cdots$-A_k as a child of the last variable in $\boldsymbol{Z}$. Finally, define $\Pi_0 := (P, \boldsymbol{C} \cup \{A_1\})$ (only A_1 receives a cache). It can be checked that $\Pi_0 \preceq \Pi$, which proves $\mathcal{PTC} \preceq \mathcal{TD}^{\mathcal{GJ}}$. □

Theorem 5.7. *$\mathcal{TD}^{\mathcal{PT}} \preceq \mathcal{TD}^{\mathcal{PTC}}$. Therefore, $\mathcal{TD}^{\mathcal{PT}} \equiv \mathcal{TD}^{\mathcal{PTC}}$ and $\mathcal{TD}^{\mathcal{PT}} \preceq \mathcal{PTC}$.*

6 Resetting the Cache

We saw how the addition of caches reduces the time exponent while increasing the space exponent of a pseudo-tree. We describe here *pseudo-tree with caching and resets*, $\mathcal{PTCR}$, which allows for a finer tradeoff between space and time. The basic principle was introduced in [9], but with no analysis of its complexity. We provide an analysis, and describe a non-trivial improvement, using a notion called *relevant ancestors*, which reduces the asymptotic time complexity of the algorithm.

Example 6.1. To motivate resetting caches, consider the scalar query $Q_{\triangle}()$ in Fig. 6 and the PT consisting of a single path (*A-B-C-D-E-F*), and with root *A*. Adding caches to E, F leads to space-time exponents of $(1.5, 2)$. For example, the cache M_E for variable E has key $con(E) = ABD$, and its space usage is given by $\rho^*(ABD) = 1.5$. When $\textsc{solve}(E, abcd)$ is called, Algorithm 3 stores the result in $M_E(abd)$; in later calls, if c changed while abd are the same, the algorithm immediately returns the cached value. Note that once the values of a or b change we can safely discard (or reset) all entries $M_E(ab-)$, because the values abc arrive at $\textsc{solve}(E, -)$ in lexicographic order. We can reduce the space of the cache by only keeping cached entries whose keys agree on AB – essentially only storing the results for different values of D. This decreases the space usage to $\rho^*(D) = 1$, eq. for the cache M_F. With this "reset" improvement, $Q_{\triangle}$ can be computed with space-time complexity $(1, 2)$.

We describe now this technique in general. Fix an SPQ $Q(\boldsymbol{X}) \leftarrow \bigotimes_i R_i(\boldsymbol{X}_i)$.

Definition 6.2. A pseudo-tree with caching and resets (PTCR) of Q is a pair (P, C), where $P = (\boldsymbol{V}, E)$ is a pseudo-tree and C is a function $C : \boldsymbol{V} \to \mathbb{N}$.

Fix a variable $A \in \boldsymbol{V}$, and let its context be $con(A) = \{A_1, \ldots, A_n\}$; recall that $con(A) \subseteq anc(A)$. We order $con(A)$ such that A_1 is closest to the root and A_n is closest to A. The function C in Def. 6.2 indicates how many variables from $con(A)$ will be stored simultaneously. If $k := \min(C(A), |con(A)|)$, we partition $con(A)$ into $conInst(A) \cup conSto(A)$, where $conInst(A) := \{A_1, \ldots, A_{n-k}\}$ is the *instantiated context* and $conSto(A) := \{A_{n-k+1}, \ldots, A_n\}$ is the *stored context*. The keys of the cache M_A will always agree on $conInst(A)$ but may differ on $conSto(A)$. Any change of a variable in $conInst(A)$ invalidates (resets) M_A. We use a bar to indicate the partition of $con(A)$: in Example 6.1, if $C(E) = 1$ then we write $con(E) = AB|D$. If $C(A) = 0$ for a variable A then it is equivalent to having no cache[6].

[6] Strictly speaking, if $C(A) = 0$ then we cache at A a single returned value $M_A(\boldsymbol{x}[con(A)]) := \textsc{solve}(A, \boldsymbol{x})$, and can reuse it as long as $\boldsymbol{x}[con(A)]$ doesn't change. However, if $con(A)$ contains $parent(A)$, then we cannot expect to ever reuse

A basic algorithm for handling resets described in [9] is as follows. Each variable A has a cache M_A with key $con(A)$, and an *instantiated* tuple $conInst_A \in \mathbf{dom}^{conInst(A)}$ storing the last value of $conInst(A)$. When $\textsc{solve}(A, \boldsymbol{y})$ is called, it first checks whether $conInst_A = \boldsymbol{y}[conInst(A)]$. If yes, then it uses the cache like Algorithm 3. If not, then it resets the cache M_A. However, this algorithm is not optimal, as we explain in the next example.

Example 6.3. Consider the scalar query $Q_{\bigcirc}()$ in Fig. 7 and the PTCR (P, C) in Fig. 8. The figure also shows the partitions of each context. For example, $con(E) = B|D$, means that $conInst(E) = B$ and $conSto(E) = D$, and therefore the keys of its cache $M_E(BD)$ always agree on B; similarly, D has a full cache $M_D(BC)$. We want to compute the time complexity of E, and for that we need to reason about how often the cache M_E is reset due to B changing its value. We can do this in two ways. Either we notice that the only ancestor of B is A and, thus, the number of calls to $\textsc{solve}(E, -)$ with a new B is bounded by $\rho^*(AB)$. Or, we notice that D has a fully stored cache $M_D(BC)$, and calls $\textsc{solve}(E, -)$ only with unique BC pairs, thus the complexity is also bounded by $\rho^*(BC)$.

Our algorithm reduces this to $\rho^*(B)$, by computing the cache $M_D(BC)$ eagerly. When $\textsc{solve}(D, abc)$ is called for the first time, D will ignore the values bc, and instead it fills its cache $M_D(BC)$ entirely with *all* values of BC: it iterates over the distinct values[7] $bc \in supp(\llbracket Q[BC] \rrbracket)$, calls $\textsc{solve}(E, abcd)$ recursively, and stores the result in $M_D(bc)$. D traverse the $supp(\llbracket Q[BC] \rrbracket)$ by grouping on B (e.g., by sorting it lexicographically), so that the same B-values occur consecutively, e.g. $b_1c_1, b_1c_2, b_1c_3, b_2c_1, \ldots$ When B changes from b_1 to b_2, E resets its cache, but there is no loss of work, because b_1 will never be seen again; this is similar to the argument in Example 6.1. The number of cache resets is reduced[8] to $\rho^*(B)$. This example was simple, because D had no instantiated variables, $conInst(D) = \emptyset$; the general case requires the technical Definition 6.4 below.

Filling the cache eagerly is a significant extension of GJ and all its implementations in practice [14, 37], where the values of $AB\ldots$ are examined in strict lexicographic order, e.g. $a_1b_1, a_1b_2, a_2b_1, a_2b_2 \ldots$ Instead, the values of B arrive at the function $\textsc{solve}(E, -)$ in sorted order $b_1, b_1, b_2, b_2, b_3, \ldots$

Algorithm 4 extends Algorithm 3 from $\mathcal{PTC}$ to $\mathcal{PTCR}$, and uses the following:

Definition 6.4. For a PTCR (P, C) and variable $A \in \boldsymbol{V}(P)$, we define the *(closed) relevant (instantiated) ancestors* $\overline{ra}(A)$, $ra(A)$, $ria(A)$ — where $\overline{con}(A) \subseteq \overline{ra}(A) \subseteq \overline{anc}(A)$, $con(A) \subseteq ra(A) \subseteq anc(A)$, and (for $conInst(A) \neq \emptyset$) $conInst(A) \subseteq ria(A) \subseteq \overline{anc}(\min(conInst(A)))$ — recursively as follows:

$$ria(A) = \begin{cases} \overline{ra}(parent(A)) \cap \overline{anc}(\min(conInst(A))) & conInst(A) \neq \emptyset, \\ \emptyset & conInst(A) = \emptyset, \end{cases}$$

$$ra(A) = ria(A) \cup conSto(A), \quad \overline{ra}(A) = ra(A) \cup \{A\}.$$

The key difference between Algorithms 4 and 3 is that the function $\textsc{solve}(A, \boldsymbol{x})$ fills its cache M_A eagerly. This is done by FILLCACHE, which iterates over all these values in line 11. Notice that $supp(\llbracket Q[conSto(A)|ria_A] \rrbracket)$ is a *set*: the algorithm processes these values lexicographically. However, the function FILLCACHE is called by $\textsc{solve}(A, -)$ only once for each value of the variables $ria(A)$. Referring to Example 6.3, when the function $\textsc{solve}(D, -)$ is called, we have $ria(D) = \emptyset$, hence FILLCACHE is called only once, and in line 11 it iterates over $supp(\llbracket Q[BC] \rrbracket)$.

---

the value $M_A(\boldsymbol{x}[con(A)])$, which is equivalent to not having a cache. If $parent(A) \notin con(A)$, then the PT is suboptimal: we can simply connect A to $parent(parent(A))$, that is, A and $parent(A)$ become siblings.

[7] $supp(\llbracket Q[BC] \rrbracket)$ means the projection of the query on BC. It is a natural extension of Eq. (3).

[8] The careful reader may have noticed that $\rho^*(AB) = \rho^*(BC) = \rho^*(B) = 1$, so for this simple example our improved algorithm does not reduce the runtime but the dependency. The runtime reduction does happen for more complex examples.

Algorithm 4 PTCR Algorithm

Input: Query $Q(\boldsymbol{X})$, PTCR (P, C)
Output: $[\![Q]\!]$

1: **for** $A \in V(P)$
2: $\quad M_A \leftarrow \emptyset, ria_A \leftarrow \perp$
3: **return** SOLVE($root(P)$, ())
4: **function** SOLVE($A, \boldsymbol{x}$)
5: $\quad$ **if** $\boldsymbol{x}[ria(A)] = ria_A$ ▷ Cache hit
6: $\quad\quad$ **return** $M_A(\boldsymbol{x}[con(A)])$
7: $\quad ria_A \leftarrow \boldsymbol{x}[ria(A)]$ ▷ Cache miss
8: $\quad$ FILLCACHE($A, \boldsymbol{x}[anc(A) \setminus conSto(A)]$)
9: $\quad$ **return** $M_A(\boldsymbol{x}[con(A)])$
10: **function** FILLCACHE($A, \boldsymbol{y}_{\text{anc}}$)
11: $\quad$ **for** $\boldsymbol{y}_{\text{sto}} \in supp([\![Q[conSto(A)|ria_A]]\!])$
12: $\quad\quad \boldsymbol{y} \leftarrow (\boldsymbol{y}_{\text{anc}}, \boldsymbol{y}_{\text{sto}})$
13: $\quad\quad$ OUT $\leftarrow \{\boldsymbol{z} \mapsto \boldsymbol{0} \mid \boldsymbol{z} \in \mathbf{dom}^{\overline{out}(A)}\}$
14: $\quad\quad$ **for** $a \mapsto s \in [\![Q[A|ria_A, \boldsymbol{y}_{\text{sto}}]]\!]$
15: $\quad\quad\quad \boldsymbol{y}' \leftarrow (\boldsymbol{y}, a)$ ▷ L. 15–19 as in Alg. 1
16: $\quad\quad\quad$ TMP $\leftarrow \{a[\boldsymbol{X} \cap \{A\}] \mapsto s\}$
17: $\quad\quad\quad$ **for** $B \in child(A)$
18: $\quad\quad\quad\quad$ TMP $\leftarrow$ TMP $\otimes$ SOLVE($B, \boldsymbol{y}'$)
19: $\quad\quad\quad$ OUT $\leftarrow$ OUT $\oplus$ TMP
20: $\quad\quad M_A \leftarrow M_A \cup \{\boldsymbol{y}[con(A)] \mapsto \text{OUT}\}$

Definition 6.5. The class of query plans $\mathcal{PTCR}(Q)$ consists of PTCR (P, C) of Q and their space and time exponents are defined as:

$$s(P,C) = \max_{A \in V(P)} \rho^*(conSto(A) \cup \overline{out}(A)), \quad t(P,C) = \max_{A \in V(P)} \rho^*(\overline{ra}(A) \cup out(A)).$$

THEOREM 6.6. *If $(P, C) \in \mathcal{PTCR}(Q)$, then Algorithm 4 computes $[\![Q]\!]$ in $O(|D|^{t(P,C)})$ time and uses $O(|D|^{s(P,C)})$ space.*

Lastly, we prove the remaining relationships in Fig. 1 up to $\mathcal{TD}^{\mathcal{PTCR}}$, in particular demonstrating the difference in strength of $\mathcal{TD}^{\mathcal{PT}}$ and $\mathcal{PTCR}$. This separation together with the separation given in Theorem 4.6 imply that all dominations corresponding to downward arrows up to $\mathcal{TD}^{\mathcal{PTCR}}$ are strict.

THEOREM 6.7. *$\mathcal{PTCR} \not\preceq \mathcal{TD}^{\mathcal{PT}}$ and $\mathcal{TD}^{\mathcal{PT}} \not\preceq \mathcal{PTCR}$.*

PROOF SKETCH. For $\mathcal{PTCR} \not\preceq \mathcal{TD}^{\mathcal{PT}}$, we use the query $Q_{\triangle}()$ depicted in Fig. 6 and the $\mathcal{PTCR}$ in given Example 6.1, with space-time exponents $(1, 2)$. Because no edge separates the query, any TD with 2+ bags uses superlinear space, and a single bag is equivalent to a PT, which takes more than quadratic time. For $\mathcal{TD}^{\mathcal{PT}} \not\preceq \mathcal{PTCR}$, we use the query $Q_{\triangle\triangle}()$ depicted in Fig. 9 and a TD with three bags, separated by DE and GH; its space-time exponents are $(1, 2)$. The best $\mathcal{PTCR}$ has space-time exponents of $(1.5, 2)$ or $(1, 2.5)$. □

COROLLARY 6.8. *The dominations relations represented by downward arrows up to the ones ending at $\mathcal{TD}^{\mathcal{PTCR}}$ in Fig. 1 are all strict.*

7 Using Recursion to Reorient Sub-Trees

We now present a final class of plans termed recursive pseudo-trees, $\mathcal{RPT}$, that unify the strengths of PTs and TDs into a single approach. While $\mathcal{PTCR}$ (and also $\mathcal{PTC}$) captured some of the advantages of TDs, as seen by $\mathcal{PTCR} \prec \mathcal{TD}^{\mathcal{GJ}}$, TDs can bring further benefits when computed with PTs, as seen by $\mathcal{TD}^{\mathcal{PTCR}} \prec \mathcal{PTCR}$. One benefit of TD-based plans is that different bags can use different variable orders, for example in Fig. 5 we could use A-B in one bag, and B-A in the other, which is not possible in a PT. $\mathcal{RPT}$ loosens this restriction, and fully captures the benefit of TDs: we will show the equivalence of $\mathcal{RPT}$ and $\mathcal{TD}^{\mathcal{RPT}}$ in Thm. 7.3.

To begin, we revisit the function FILLCACHE($A, \boldsymbol{y}_{\text{anc}}$) of Algorithm 4. This function evaluates for every tuple $\boldsymbol{y}_{\text{sto}} \in supp([\![Q[conSto(A)|ria_A]]\!])$ (see line 11) the subquery of Q restricted to variables

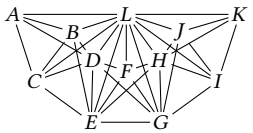


Fig. 10. Query $Q_{\smile}$

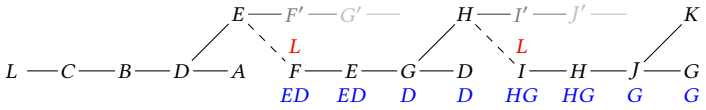


Fig. 11. An RPT for $Q_{\smile}$

$\overline{desc}(A)$ and caches the result in M_A. To do this, the algorithm based on a PTCR plan first iterates through the variables in $conSto(A)$, then proceeds to the variables in $\overline{desc}(A)$. While intuitive, this is an arbitrary decision. One could instead simply use a different PTCR (P', C') to compute this subquery where $conSto(A)$ is included as output variables. Effectively, we fill the cache using a different PTCR and see this as replacing the subtree of (P, C) rooted at A with the new sub-plan (P', C'). We call the resulting structure a *recursive pseudo-tree*.

Example 7.1. To motivate using different PTCR to fill the caches, let us consider the scalar query $Q_{\smile}()$ given in Fig. 10. As there are many 5-cliques, our aim will be to arrive at a time complexity of 2.5 and minimize the space complexity. Note the structure of the query: L is connected to everything, $\mathcal{S}_1 = LDE$ and $\mathcal{S}_2 = LHG$ are the minimal non-trivial separators, and $\mathcal{K}_1 = LBDC, \mathcal{K}_2 = LFEG, \mathcal{K}_3 = LHJI$ are three 4-cliques which each extend in two ways to a 5-clique – $\mathcal{K}_1$ together with A and E is a 5-clique, respectively, as well as $\mathcal{K}_2$ together with D and H, and $\mathcal{K}_3$ together with G and K. Thus, a natural way of evaluating $Q_{\smile}$ would be to start like a $\mathcal{PTCR}$ plan. That is, we start with a path $\mathcal{K}_2 \colon L - F - E - G$ of nested loops and then branch to the separators $\mathcal{S}_1 \colon D$ and $\mathcal{S}_2 \colon H$ separately. Let us focus on the left branch where we continue with, say, C. At this point, we need a cache to not increase the time complexity beyond 2.5. We set the size of the cache to 2 (the space complexity will be 1 but the cache contains 2 variables), thus get the context partition $con(C) = L|ED$, and now would have to execute FillCache(C, l). That is, we have to find and cache the possible values of $conSto(C) = ED$ that fit to l and solve the remaining query on A, B, C. This can be done by a loop structure that first extends l to values bdc and then loops through A and E independently. We have seen in previous sections that such a loop structure will take time $O(N^{2.5})$ and space $O(N)$ including the loop over the values l. Naturally, the right branch is symmetric. Thus, we have arrived at an algorithm with the desired space and time consumption. However, note that this algorithm is *not* the result of a $\mathcal{PTCR}$ plan as this is not the way FillCache(C, l) would have filled the cache of C (no matter how we complete the PTCR). Crucially, we inverted the order of E and D to fill the cache; we will return to this example a couple more times in this section.

To define recursive pseudo-trees formally, we introduce SPQs with input variables $\boldsymbol{I}$ (these will be $ria(A)$). That is, let $Q(\boldsymbol{X})$ be an SPQ and let $\boldsymbol{I} \subseteq var(Q) \setminus \boldsymbol{X}$ denote a subset of the variables in Q, which we refer to as the *input variables* of $Q(\boldsymbol{X})$. Then a PTCR of $Q(\boldsymbol{X})$ is also a PTCR of $(Q(\boldsymbol{X}), \boldsymbol{I})$ when $\boldsymbol{I}$ are variables that occur above all other variables. That is, these variables form a chain and we call the first variable $root \in V(P) \setminus \boldsymbol{I}$ with $anc(root) = \boldsymbol{I}, \overline{desc}(root) = var(Q) \setminus \boldsymbol{I}$ the (real) root of $(P, C, \boldsymbol{I})$. Furthermore, we require $ria(root) = \boldsymbol{I}$.

Definition 7.2. A *recursive pseudo-tree* (RPT) of an SPQ with input variables $(Q(\boldsymbol{X}), \boldsymbol{I})$ is recursively defined as follows:

- *Base case.* A PTCR $(P, C, \boldsymbol{I})$ of $(Q(\boldsymbol{X}), \boldsymbol{I})$ is an RPT of $(Q(\boldsymbol{X}), \boldsymbol{I})$.
- *Recursion.* Let $(P, C, \boldsymbol{I})$ be a PTCR of $(Q(\boldsymbol{X}), \boldsymbol{I})$, and let $A_1, \ldots, A_l$ be vertices in $\boldsymbol{V}(P) \setminus \boldsymbol{I}$, such that $A_i \notin \overline{anc}(A_j) \forall\, i \neq j$. Let Q_i be the subquery of $Q(\boldsymbol{X})$ restricted to the variables

$desc(A_i) \cup \overline{ra}(A_i)$ and with head variables $conSto(A_i) \cup \overline{out}(A_i)$. Let $(\mathcal{P}_i, C_i, \mathcal{I}_i)$ be RPTs of $(Q_i, ria(A_i))$. Then, $(\mathcal{P}, C, \mathcal{I}) := ((P, (\mathcal{P}_i)_i), (C, (C_i)_i), (\boldsymbol{I}, (\mathcal{I}_i)_i))$ is an RPT of $(Q(\boldsymbol{X}), \boldsymbol{I})$.

The class of query plan $\mathcal{RPT}(Q)$ consist of RPTs $(\mathcal{P}, C, \mathcal{I})$ of $(Q, \emptyset)$ and their space and time exponents are defined recursively via:

$$s(\mathcal{P}, C, \mathcal{I}) = \max(\max_{A \in \bigcup_i anc(A_i)} \rho^*(conSto(A) \cup \overline{out}(A)); \max_i s(\mathcal{P}_i, C_i, \mathcal{I}_i)),$$
$$t(\mathcal{P}, C, \mathcal{I}) = \max(\max_{A \in \bigcup_i anc(A_i)} \rho^*(\overline{ra}(A) \cup \overline{out}(A)); \max_i t(\mathcal{P}_i, C_i, \mathcal{I}_i)).$$

Using the RPTs $(\mathcal{P}_i, C_i, \mathcal{I}_i)$ to perform the same task as FillCache$(A_i, -)$ we get the following:

Theorem 7.3. *If $(\mathcal{P}, C, \mathcal{I}) \in \mathcal{RPT}(Q)$, then $[\![Q]\!]$ can be computed in space $O(|D|^{s(\mathcal{P}, C, \mathcal{I})})$ and time $O(|D|^{t(\mathcal{P}, C, \mathcal{I})})$.*

Example 7.4. Fig. 11 depicts an RPT $\Pi = (\mathcal{P}, C, \mathcal{I})$ of the query $Q_{\smile}()$ depicted in Fig. 10. Solid edges represent "normal" edges of a PTCR while the dashed edges represent a recursive replacement. The original PTCR had the path PT $(F'\text{-}G'\text{-}H'\text{-}I'\text{-}J'\text{-}K')$ (depicted gray) as a child of the first E while F' was the only variable with a cache. Then, $A_1 = F'$ got replaced by the sub-RPT $(\mathcal{P}_1, C_1, \mathcal{I}_1)$ rooted in F as depicted in Fig. 11 (this is the first dotted edge). Note that $L|DE = con(F')$, $L = ria(F') = ria(F)$ and $DE = conSto(F') = \overline{out}(F)$ as required (output and input variables are drawn in blue and red, respectively, in Fig. 11). $(\mathcal{P}_1, C_1, \mathcal{I}_1)$ was constructed similarly. The original PTCR had the path PT $(I'\text{-}J'\text{-}K')$ as a child of the first H while I' was the only variable with a cache. Then, $A_2 = I'$ (note that A_1 and A_2 come from different PTs) got replaced by the sub-PTCR $(P_2, C_2, \{L\})$ rooted in I as depicted in Fig. 11. Note that $L|GH = con(I')$, $L = ria(I') = ria(I)$ and $GH = conSto(I') = \overline{out}(I)$ as required. The space-time exponents are $(1, 5/2)$.

The evaluation algorithm proceeds analogously to the algorithm described in Example 7.1 with the slight change that the cliques $\mathcal{K}_1, \mathcal{K}_2, \mathcal{K}_3$ are processed one after the other in exactly that order. Thus, the RPT depicted in Fig. 11 has two recursion on one single branch while the algorithm described in Example 7.1 essentially had two branches with one recursion on each branch.

Finally, we relate $\mathcal{RPT}$ to the other plan classes. First, we note that $\mathcal{TD}^{\mathcal{RPT}}$ provides no benefit over $\mathcal{RPT}$. The proof is inductive, similar to that of Theorem 5.6.

Theorem 7.5. *The class $\mathcal{RPT}$ dominates $\mathcal{TD}^{\mathcal{RPT}}$. Thus, in particular $\mathcal{RPT} \equiv \mathcal{TD}^{\mathcal{RPT}}$.*

Considering the query $Q_{\smile}()$ in Fig. 10 and using computer-assisted exhaustive search, we verified that there is no plan $(P, C) \in \mathcal{PTCR}$ with $s(P, C) \leq 1$ and $t(P, C) \leq 5/2$. There is no linear separator in this query, so the same holds for $\mathcal{TD}^{\mathcal{PTCR}}$. Combined with Theorem 7.5, we can prove:

Theorem 7.6. *The class $\mathcal{RPT}$ strictly dominates $\mathcal{TD}^{\mathcal{PTCR}}$, i.e., $\mathcal{RPT} \prec \mathcal{TD}^{\mathcal{PTCR}}$.*

Naturally, the time exponents of RPTs are always at least as large as the fractional hypertree width (fhw). As $\mathcal{RPT}$ dominates all other classes of query plans discussed in this paper, the same is the case for them.

Theorem 7.7. *If $(\mathcal{P}, C, \mathcal{I}) \in \mathcal{RPT}(Q)$, then $t(\mathcal{P}, C, \mathcal{I}) \geq fhw(Q)$, where $fhw(Q)$ is the fractional hypertree width of Q.*

Proof Sketch. The RPT $(\mathcal{P}, C, \mathcal{I})$ describes a tree structure (e.g., see black parts of Fig. 11) and we can use this tree structure together with bags $\overline{ra}(A) \cup \overline{out}(A)$ at nodes A to construct a tree decomposition of Q. The width of this tree decomposition is the same as the time exponent of the RPT. □

8 Conclusion and a Glimpse Beyond

We have presented several novel algorithms for CQ and, more generally, SPQ evaluation by combining and significantly extending existing approaches based on pseudo-trees and tree decompositions. In most cases, we have matched the optimal time complexity of previous algorithms with asymptotically lower space complexity. We end here by discussing two important lines of future work.

Conjectures on Lower Bounds. Although some conditional lower bounds have been established for the time complexity of query evaluation [8, 12], much less is known about space-time lower bounds. In the complexity community, the space-time tradeoff is studied by proving lower bounds on the product $S \times T$ for specific problems. For example, the set difference and distinct element problems have been proven to have a lower bound $ST = \Omega(n^2)$ [7, 31]. These results are too weak to constrain query evaluation, since even acyclic CQs require at least a quadratic space-time product.

Since there are no widely accepted assumptions to build on, we take a first step toward understanding lower bounds for space-constrained query answering. We propose a problem called the *Triple k-Clique Problem*, and conjecture a lower bound on its space-time complexity. To motivate our conjecture, we briefly recall the *min-weight k-clique hypothesis*, which is a standard complexity assumption for combinatorial problems [1, 12, 28]. The problem concerns the k-clique scalar query:

$$Q() \leftarrow \bigotimes_{i<j\in[1,k]} E(A_i, A_j) \tag{7}$$

and the task is to evaluate Q in the tropical semi-ring, where $x \oplus y := \min(x, y)$, $x \otimes y := x + y$; in other words, we are asked to identify a clique with the smallest total edge weight in a weighted graph. The most commonly used variant of the min-weight k-clique hypothesis is stated relative to the number of vertices in the graph. However, [12] showed that this is equivalent to saying that, for any $\varepsilon > 0$, no algorithm can solve this problem in time $O(|E|^{\frac{k}{2}-\varepsilon})$. This problem is not a good candidate for a hard space-time problem, because GJ already computes Q in optimal time $O(|E|^{\frac{k}{2}})$ and optimal space $O(1)$. Instead, we propose the following extension:

Conjecture 8.1 (The Triple k-Clique Conjecture). *For $k := 2\ell, \ell \geq 2$, consider the query:*

$$Q() \leftarrow \bigotimes_{i<j\in[1,k]} E(A_i, A_j) \otimes \bigotimes_{i<j\in[k-\ell+1,k+\ell]} E(A_i, A_j) \otimes \bigotimes_{i<j\in[k+1,2k]} E(A_i, A_j) \tag{8}$$

Then, for any $\varepsilon > 0$, no algorithm can solve this problem over the tropical semi-ring in space $S = O(|E|^{\frac{k}{4}-\varepsilon})$ and time $T = O(|E|^{\frac{k}{2}})$.

The tree decomposition with three bags corresponding to the three $\bigotimes$-expressions above uses space[9] $S = O(|E|^{k/4})$ and time $T = O(|E|^{k/2})$; the conjecture claims that it is optimal. On the other hand, GJ computes Q in $S = O(1)$ and $T = O(|E|^k)$. This justifies a stronger version of the conjecture: for any $\varepsilon > 0$, no algorithm can compute Q such that $S^2T = O(|E|^{k-\varepsilon})$.

Aiming Toward Submodular Width. The best known time bound for Boolean conjunctive queries is based on the *submodular width* [23, 26, 30]. This measure partitions the input data based on degrees, and uses different query plans for each partition. We have not considered the submodular width in this paper, instead extended the (weaker) fractional hypertree width (see Thm. 7.7). However,

[9] For a quick computation of ρ^* observe that, for any graph G with vertices V and no isolated vertices, $\rho^*(V) \geq |V|/2$, because this is the value of the fractional vertex packing where each vertex has weight $1/2$. On the other hand, if we can partition $V = V_1 \cup V_2 \cup \cdots$ such that each graph induced by V_i is a clique, then $\rho^*(V) \leq \sum_i \rho^*(V_i) = |V|/2$. For example, for a single k-clique V we can conclude $\rho^*(V) = k/2$ and for Q in Eq. (8) we can take $V_1 = [1, k]$ and $V_2 = [k + 1, 2k]$ and conclude that $\rho^*(A_1 \cdots A_{2k}) = k$.

extending submodular width is much more intricate as it does not generalize to arbitrary semi-rings, e.g., not to the natural numbers $\mathbb{N}$ [22].

Nevertheless, aiming at adapting our approach to achieve submodular width is promising future work. However, it at least requires the introduction of degree constraints in the analysis, which complicates the picture significantly. Generic Join is no longer optimal in the presence of degree constraints, and this affects pseudo-trees too. Intuitively, a pseudo-tree with structure $A_1 - A_2 - A_3$, rooted at A_1, can benefit from constraints on the degree from A_1 to A_3 but not constraints on the degree from A_3 to A_1 when analyzing its time and space. However, in some cases, it is possible to meet the submodular-width's time complexity while minimizing the space complexity, as shown here:

Theorem 8.2. *The query $Q_\square$ below can be computed in space $O(|D|^{\frac{1}{2}})$ and time $O(|D|^{\frac{3}{2}})$:*

$$Q_\square() \leftarrow E_1(A_1, A_2) \otimes E_2(A_2, A_3) \otimes E_3(A_3, A_4) \otimes E_4(A_1, A_4)$$

Proof Sketch. To achieve this, we first perform a heavy-light partitioning of the input relations where a *heavy* join value a_j is one which appears in at least $\sqrt{|E_i|}$ tuples of E_i. As there cannot be more than $\sqrt{|E_i|}$ heavy values, we can iterate over them, instantiate them, and solve the reaming linear query in linear time and constant space. Thus, in space $O(1)$ and time $O(|D|^{\frac{3}{2}})$ we can handle all heavy values.

For the case where all values are light, a different technique is needed. There, the aim is to perform a merge join on the fly, essentially using the decomposition

$$\bigoplus_{a_1,a_3} \left(\bigoplus_{a_2} E_1(a_1, a_2) \otimes E_2(a_2, a_3) \right) \otimes \left(\bigoplus_{a_4} E_4(a_1, a_4) \otimes E_3(a_4, a_3) \right).$$

We explain how to iterate through $E_1(A_1, A_2) \otimes E_2(A_2, A_3)$ projected to A_1, A_3 in lexicographic order (the other side is symmetric). To that end, we iterate through a_1 at the top level and compute the $\leq \sqrt{|E_1|}$ values $a_2 \in supp(E_1[A_2|a_1])$ that extend a_1. For every $a_2 \in supp(E_1[A_2|a_1])$ we spawns a separate process that iterates (in an ordered manner) through $a_3 \in supp(E_2[A_3|a_2])$ – hence we use $O(|D|^{\frac{1}{2}})$ processes that all require $O(1)$ space. Merging the loops of the different a_2 results in an ordered stream of a_3 values. Thus, we go over pairs a_1, a_3 in lexicographic order. Doing the same for $E_4(a_1, a_4) \otimes E_3(a_4, a_3)$ allows us to perform a merge join. In total, this is done in space $O(|D|^{\frac{1}{2}})$ and time $O(|D|^{\frac{3}{2}})$. □

Acknowledgment

The work of Merkl and Pichler was supported by the Vienna Science and Technology Fund (WWTF) [10.47379/ICT2201, 10.47379/VRG18013, 10.47379/NXT22018]. Deeds and Suciu were partially supported by NSF IIS 2314527, NSF SHF 2312195, and NSF IIS 2507117.

References

[1] A. Abboud, V. Vassilevska Williams, and O. Weimann. Consequences of faster alignment of sequences. In J. Esparza, P. Fraigniaud, T. Husfeldt, and E. Koutsoupias, editors, *Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I*, volume 8572 of *Lecture Notes in Computer Science*, pages 39–51. Springer, 2014.

[2] C. R. Aberger, A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[3] M. Arenas, L. A. Croquevielle, R. Jayaram, and C. Riveros. When is approximate counting for conjunctive queries tractable? In S. Khuller and V. V. Williams, editors, *STOC '21: 53rd Annual ACM SIGACT Symposium on Theory of Computing, Virtual Event, Italy, June 21-25, 2021*, pages 1015–1027. ACM, 2021.

[4] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748. IEEE Computer Society, 2008.

[5] J. Bader, F. Skalski, F. Lehmann, D. Scheinert, J. Will, L. Thamsen, and O. Kao. Sizey: Memory-efficient execution of scientific workflow tasks. In *IEEE International Conference on Cluster Computing, CLUSTER 2024, Kobe, Japan, September 24-27, 2024*, pages 370–381. IEEE, 2024.

[6] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In J. Duparc and T. A. Henzinger, editors, *Computer Science Logic, 21st International Workshop, CSL 2007, 16th Annual Conference of the EACSL, Lausanne, Switzerland, September 11-15, 2007, Proceedings*, volume 4646 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 2007.

[7] P. Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991.

[8] K. Bringmann and E. Gorbachev. A fine-grained classification of subquadratic patterns for subgraph listing and friends. In M. Koucký and N. Bansal, editors, *Proceedings of the 57th Annual ACM Symposium on Theory of Computing, STOC 2025, Prague, Czechia, June 23-27, 2025*, pages 2145–2156. ACM, 2025.

[9] R. Dechter. *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms, Second Edition*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019.

[10] K. Deeds, T. C. Merkl, R. Pichler, and D. Suciu. The space-time complexity of sum-product queries. *CoRR*, abs/2509.11920, 2025.

[11] R. Diestel. *Graph Theory, 4th Edition*, volume 173 of *Graduate texts in mathematics*. Springer, 2012.

[12] A. Z. Fan, P. Koutris, and H. Zhao. The fine-grained complexity of boolean conjunctive queries and sum-product problems. In K. Etessami, U. Feige, and G. Puppis, editors, *50th International Colloquium on Automata, Languages, and Programming, ICALP 2023, July 10-14, 2023, Paderborn, Germany*, volume 261 of *LIPIcs*, pages 127:1–127:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023.

[13] W. Fischl, G. Gottlob, and R. Pichler. General and fractional hypertree decompositions: Hard and easy cases. In J. V. den Bussche and M. Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 17–32. ACM, 2018.

[14] M. J. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

[15] E. C. Freuder and M. J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In A. K. Joshi, editor, *Proceedings of the 9th International Joint Conference on Artificial Intelligence. Los Angeles, CA, USA, August 1985*, pages 1076–1078. Morgan Kaufmann, 1985.

[16] Google. Google cluster data v3. https://github.com/google/cluster-data/blob/master/clusterdata2019.md, 2019.

[17] G. Gottlob, G. Greco, N. Leone, and F. Scarcello. Hypertree decompositions: Questions and answers. In T. Milo and W. Tan, editors, *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*, pages 57–74. ACM, 2016.

[18] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. In V. Vianu and C. H. Papadimitriou, editors, *Proceedings of the Eighteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 31 - June 2, 1999, Philadelphia, Pennsylvania, USA*, pages 21–32. ACM Press, 1999.

[19] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[20] M. Grohe. The complexity of homomorphism and constraint satisfaction problems seen from the other side. *J. ACM*, 54(1):1:1–1:24, 2007.

[21] M. Grohe and D. Marx. Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, 11(1):4:1–4:20, 2014.

[22] M. A. Khamis, R. R. Curtin, B. Moseley, H. Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. On functional aggregate queries with additive inequalities. In D. Suciu, S. Skritek, and C. Koch, editors, *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 414–431. ACM, 2019.

[23] M. A. Khamis, X. Hu, and D. Suciu. Fast matrix multiplication meets the submodular width. *Proc. ACM Manag. Data*, 3(2):98:1–98:26, 2025.

[24] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. *CoRR*, abs/1504.04044, 2015.

[25] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In T. Milo and W. Tan, editors, *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*, pages 13–28. ACM, 2016.

[26] 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 E. Sallinger, J. V. den Bussche, and F. Geerts, editors, *Proceedings of the 36th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2017, Chicago, IL, USA, May 14-19, 2017*, pages 429–444. ACM, 2017.

[27] K. Kim, J. Ha, G. Fletcher, and W. Han. Guaranteeing the õ(agm/out) runtime for uniform sampling and size estimation over joins. In F. Geerts, H. Q. Ngo, and S. Sintos, editors, *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 113–125. ACM, 2023.

[28] A. Lincoln, V. Vassilevska Williams, and R. R. Williams. Tight hardness for shortest cycles and paths in sparse graphs. In A. Czumaj, editor, *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018, New Orleans, LA, USA, January 7-10, 2018*, pages 1236–1252. SIAM, 2018.

[29] G. Liu, W. Lin, H. Zhang, J. Lin, S. Peng, and K. Li. Public datasets for cloud computing: A comprehensive survey. *ACM Computing Surveys*, 57(8):1–38, 2025.

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

[31] D. M. McKay and R. R. Williams. Quadratic time-space lower bounds for computing natural functions with a random oracle. In A. Blum, editor, *10th Innovations in Theoretical Computer Science Conference, ITCS 2019, January 10-12, 2019, San Diego, California, USA*, volume 124 of *LIPIcs*, pages 56:1–56:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[32] H. Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In J. V. den Bussche and M. Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 111–124. ACM, 2018.

[33] H. Q. Ngo, E. Porat, C. Ré, and A. Rudra. Worst-case optimal join algorithms: [extended abstract]. In M. Benedikt, M. Krötzsch, and M. Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48. ACM, 2012.

[34] H. Q. Ngo, C. Ré, and A. Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.

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

[36] T. van Bremen and K. S. Meel. Probabilistic query evaluation: The combined FPRAS landscape. In F. Geerts, H. Q. Ngo, and S. Sintos, editors, *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 339–347. ACM, 2023.

[37] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In N. Schweikardt, V. Christophides, and V. Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106. OpenProceedings.org, 2014.

[38] J. Wang, I. Trummer, A. Kara, and D. Olteanu. ADOPT: adaptively optimizing attribute orders for worst-case optimal join algorithms via reinforcement learning. *Proc. VLDB Endow.*, 16(11):2805–2817, 2023.

[39] H. Zhao, S. Deep, and P. Koutris. Space-time tradeoffs for conjunctive queries with access patterns. In F. Geerts, H. Q. Ngo, and S. Sintos, editors, *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS 2023, Seattle, WA, USA, June 18-23, 2023*, pages 59–68. ACM, 2023.

Received June 2025; accepted August 2025