Query Decompositions and All That

Kyle Deeds
University of Washington, Seattle, WA, USA

Timo Camillo Merkl
TU Wien, Austria

Reinhard Pichler
TU Wien, Austria

Dan Suciu
University of Washington, Seattle, WA, USA

Abstract

The close relationship between Conjunctive Queries (CQs) and Constraint Satisfaction Problems (CSPs) has long been known. Nevertheless, apart from decomposition methods, research on efficient query evaluation or constraint solving algorithms has developed rather independently. In this article, we illustrate how search algorithms originating from the CSP community can be fruitfully applied to query evaluation – either by further developing the original search algorithms or by combining them with query decomposition methods. It turns out that the resulting approaches may indeed lead to lower time and/or space complexity than previous query evaluation methods.

2012 ACM Subject Classification Theory of computation → Database theory; Information systems → Query languages

Keywords and phrases Query evaluation, Query decompositions, Complexity

Digital Object Identifier 10.4230/LIPIcs.ICDT.2026.1

Category Invited Talk

Funding 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.

1 Introduction

Conjunctive Queries (CQs, for short) are arguably the most fundamental class of (relational) queries. Consequently, the optimization of CQs has been a major topic in DB theory and systems research. Finding an optimal query evaluation plan is a highly non-trivial (in fact, intractable) problem. A particular challenge consists in avoiding the explosion of intermediate results. For acyclic CQs (ACQs, for short), Yannakakis' algorithm [36] guarantees that intermediate results never exceed the combined size of input and output. In recent time, implementations of Yannakakis' algorithm have found their way into mature research prototypes and industrial-strength database management systems [7, 35, 6, 26].

In general, we cannot expect all CQs to be acyclic. However, extensive analyses [8, 27, 13] of queries in benchmarks and query logs have shown that the majority of those queries are "almost acyclic". Various types of query decomposition methods (with associated width-measures) have been proposed to transform such CQs into ACQs. Among the most thoroughly studied decomposition methods are tree decompositions (TDs) [31], (generalized) hypertree decompositions (GHDs and HDs, respectively) [17, 2], and fractional hypertree decompositions (FHDs) [20]. The associated width measures, that allow us to formalize the

© Kyle Deeds, Timo Camillo Merkl, Reinhard Pichler, and Dan Suciu;
licensed under Creative Commons License CC-BY 4.0

notion of "almost acyclic" CQs are tree-width (*tw*), (generalized) hypertree-width (*ghw* and *hw*, respectively), and fractional hypertree-width (*fhw*). Note that decomposition methods for query evaluation have been successfully implemented in several research prototypes [1, 32, 25].

The main "freedom" of query evaluation algorithms based on one of these decompositions is how to compute and materialize the subquery corresponding to each bag of the decomposition. Using some worst-case optional join (WCOJ) method is often considered as optimal. A basic and very intuitive form of WCOJ algorithm is the so-called Generic Join [29]. Assume the upper bound N on the size of the database. Then the time and space complexity of using an (optimal) FHD is essentially $O(N^{fhw})$ (data complexity). While this bound on the time complexity may be satisfactory, such a space complexity may actually be not acceptable.

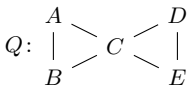
Space is a critical bottleneck in query evaluation across diverse hardware settings. Modern in-memory databases suffer significant performance degradation when forced to spill to disk [33]. Similarly, tensor algebra programs, often modeled as Sum-Product Queries (SPQs, for short), are frequently executed on GPUs where high-bandwidth memory is scarce; the cost of data movement between GPU and CPU can easily dwarf computation time [30, 19]. Furthermore, in resource-constrained environments like embedded devices, memory limits are strict and must be respected [21, 22]. In all these contexts, the existing space-complexity bounds of current query evaluation techniques are insufficiently tight for practical optimization.

Now a natural question arises: can we do better? That is, can we improve the time and space requirements of query evaluation or, at least, improve one without deteriorating the other. Here, we are particularly interested in the question whether we can improve the space complexity of previous query evaluation approaches without (significantly) compromising the time complexity.

CQs have long been known to be closely related to Constraint Satisfaction Problems (CSPs) [24]. In fact, on the logical level, both correspond to first-order formulas restricted to the connectives $\exists, \wedge$ and excluding $\forall, \vee, \neg$. The above mentioned decomposition methods have actually been equally studied and applied in both areas. However, other than that, both areas have developed their own techniques. Here, we follow mostly Rina Dechter's book [11], where "constraint networks" [10] (corresponding to Boolean CQs) as well as "probabilistic networks" and "cost networks" (which can be captured by scalar SPQs) have been studied. In [11], decomposition-based methods are referred to as "inference-based". They are contrasted with the other important paradigm in the CSP-community, namely "search-based" methods.

In their basic form (so-called "OR-search"), search algorithms work by branching into possible instantiations of one variable after the other. This search is realized by nested loops which, for each variable, iterate through the possible values. By exploring one instantiation of each variable at a time, the space consumption is proportional to the number of variables and, therefore, constant when considering data complexity. A more sophisticated approach is an "AND/OR-search", which exploits the fact that, after instantiating some variable(s), possible instantiations for other variables may be investigated independently. While OR-search is guided by an order on the variables (i.e., in which order to inspect possible instantiations), AND/OR search is guided by a tree structure, reflecting the independence of variables conditioned on their common ancestors. Here, we concentrate on *pseudo-trees*, which were introduced in [15] and studied in detail and further developed in [11]. As was observed in [12], Generic Join, which realizes the "attribute"- (= "variable-") at-a-time" paradigm, corresponds to the less sophisticated OR-search.

The following example will help to illustrate some of the characteristics of the classes of query plans mentioned so far, namely tree decompositions, Generic Join, and pseudo-trees.



Tree decomposition:	Generic Join:	Pseudo-tree:
	for $a \dots$	for $c \dots$
Materialize $Q[ABC]$	for $b \dots$	for $a \dots$
Materialize $Q[CDE]$	for $c \dots$	for $b \dots$
Intersect on C	for $d \dots$	for $d \dots$
	for $e \dots$	for $e \dots$
$S = (N^{3/2}), T = (N^{3/2})$	$S = (1), T = (N^{5/2})$	$S = (1), T = (N^{3/2})$

Figure 1 Query with 3 different plans (Example 1).

▶ **Example 1.** Consider the query

$$Q() \leftarrow E_1(A,B) \wedge E_2(B,C) \wedge E_3(C,A) \wedge E_4(D,E) \wedge E_5(E,C) \wedge E_6(C,D)$$

over some database D of size N. In Figure 1, we graphically display this query and show three different query plans:

- The first one evaluates Q by using a *tree decomposition* with bags ABC and CDE in a straightforward way. That is, we first evaluate and materialize the subqueries restricted to the bags (in time and space $O(N^{3/2})$, where the exponent corresponds to the fractional edge cover number of the triangle query, and then intersect the two results on C.
- The next plan applies the *Generic Join* algorithm. Here, we have to fix a variable order – say A, B, C, D, E – and then proceed in nested loops. Since we inspect for each variable only one value at at time, the space consumption is constant (data complexity). The time complexity is $O(N^{5/2})$, where the exponent corresponds to the fractional edge cover number $\rho^*(Q)$ of the query Q.
- The last plan uses a *pseudo-tree* with root node C and then branches into A, B and D, E, respectively. Here we make use of the fact that, for every value of C, we may evaluate the subqueries restricted to A, B, C and to C, D, E independently. We thus achieve $O(N^{3/2})$ time complexity, while the space complexity is still constant.

Our goal is to study both decompositions and search-based methods as well as the relationship between them. To this end, we first recall properties of these methods individually and then study their combination. In particular, we look at what is gained in terms of time and/or space complexity, when using pseudo-tree-based algorithms for evaluating the subquery at each bag of a (tree) decomposition. We thus analyze various classes of query plans in terms of data complexity. As "yardstick", we consider their space and time exponents. More specifically, we will say that a query plan Π in a given class of query plans has space-time exponent $e(\Pi) = (s(\Pi), t(\Pi))$, if the "canonical" evaluation algorithm executing the query plan Π has space complexity $O(N^{s(\Pi)})$ and time complexity $O(N^{t(\Pi)})$, where N is a bound on the size of the database. For instance, as fundamentally different approaches we have Generic Join plans Π with $e(\Pi) = (0, \rho^*(Q))$ and plans Π using a (tree) decomposition and WCOJ to evaluate and materialize the subqueries at each bag with $e(\Pi) = (\mathit{fhw}(Q), \mathit{fhw}(Q))$. That is, the space complexity of Generic Join plans is clearly optimal, while plans based on a decomposition plus WCOJ are, in general, superior in terms of time consumption.

The focus of this article will be on illustrating the key ideas of various classes of query plans by means of examples. Formal proofs of all results mentioned here can be found in [12]. To keep things simple and in the interest of clarity, we will concentrate on CQs and, in particular, Boolean CQs. We note that, in [12], the more general case of SPQs over some semi-ring $\mathbb{K}$ and with output variables is studied.

The remainder of this article is structured as follows: After introducing some basic definitions and results in Section 2, we revisit query plans based on decompositions in Section 3. In Section 4, we will study several variants of pseudo-trees. The combination of decompositions and pseudo-trees will then be discussed in Section 5. We conclude with a summary and an outlook on future work in Section 6.

2 Preliminaries

In this paper, we focus on *Conjunctive Queries* (CQs), which correspond to select-project-join queries in the Relational Algebra. W.l.o.g., we restrict ourselves to CQs of the form $Q = \pi_{\boldsymbol{X}}(R_1 \bowtie \ldots \bowtie R_m)$. Here we assume that equi-joins are replaced by natural joins via appropriate renaming of attributes. Moreover, we assume that selections applying to a single relation have been pushed immediately in front of this relation and the R_i's are the result of these selections. Alternatively, we denote CQs in Datalog-style notation and write them as

$$Q(\boldsymbol{X}) \leftarrow \bigwedge_{i=1}^{m} R_i(\boldsymbol{X}_i). \tag{1}$$

Q is a Boolean CQ, if $\boldsymbol{X}$ is the empty set. Recall that, from a logical point of view, CQs and Constraint Satisfaction Problems (CSPs) are, the same. Hence, our discussion equally applies to CSPs.

By slight abuse of notation, we write R_i both, to denote a relation (in a Relational Algebra expression) and a relation symbol (in Datalog notation). We denote (sets of) variables (also called attributes) 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. We write $var(Q)$ and $atoms(Q)$ to denote the set of variables in Q and the set of atoms in Q, respectively, i.e., $var(Q) = \bigcup_{i=1}^{m} \boldsymbol{X}_i$ and $atoms(Q) = \{R_i(\boldsymbol{X}_i) \mid 1 \leq i \leq m\}$. 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 $\boldsymbol{X} \cap \boldsymbol{Y}$. Throughout this paper we fix an infinite domain $\mathbf{dom}$ and we write $\mathbf{adom}_i$ to denote the (finite) set of domain values actually occurring in attribute A_i in the given database D, i.e., the active domain of A_i.

Unless mentioned otherwise, we consider *set semantics*. Then the semantics of Q in Eq. (1) on a database D is the relation $[\![Q]\!]^D \subseteq \mathbf{dom}^{\boldsymbol{X}}$ defined as

$$[\![Q]\!]^D := \{\boldsymbol{x} \in \mathbf{dom}^{\boldsymbol{X}} \mid \text{ there exists } \boldsymbol{y} \in \mathbf{dom}^{var(Q)\setminus\boldsymbol{X}}, \text{ s.t. for every } i, (\boldsymbol{x}, \boldsymbol{y})[\boldsymbol{X}_i] \in R_i^D\}.$$

Sometimes, we will switch to *bag semantics*. Then the semantics of Q in Eq. (1) on a database D is a mapping $[\![Q]\!]^D \colon \mathbf{dom}^{\boldsymbol{X}} \to \mathbb{N}$, which assigns to every $\boldsymbol{x} \in \mathbf{dom}^{\boldsymbol{X}}$ the cardinality $|\{\boldsymbol{y} \in \mathbf{dom}^{var(Q)\setminus\boldsymbol{X}} \mid \text{ for every } i, (\boldsymbol{x}, \boldsymbol{y})[\boldsymbol{X}_i] \in R_i^D\}|$. We will mostly consider Boolean queries and, in this case, under bag semantics we have to count the number of tuples $\boldsymbol{x} \in \mathbf{dom}^{var(Q)}$ that satisfy the right hand side of Eq. (1). We may omit the superscript D when it is clear from the context, and simply write R_i, $[\![Q]\!]$.

Let $\boldsymbol{Y} \subseteq var(Q)$ be a set of variables. A *fractional edge cover* of $\boldsymbol{Y}$ (with respect to Q) is a weight function $w \colon atoms(Q) \to [0, 1]$, such that, for every variable $A \in \boldsymbol{Y}$, $\sum_{i:A\in\boldsymbol{X}_i} w(R_i(\boldsymbol{X}_i)) \geq 1$. The *fractional edge cover number of* $\boldsymbol{Y}$, denoted $\rho^*(\boldsymbol{Y})$, is the minimum value of $\sum_i w(R_i(\boldsymbol{X}_i))$ attainable by fractional edge covers w of $\boldsymbol{Y}$. Due to [4], we know that $|D|^{\rho^*(\boldsymbol{Y})}$ is an upper bound on the number of tuples $\boldsymbol{y} \in \mathbf{dom}^{\boldsymbol{Y}}$ that satisfy the query Q restricted to the variables $\boldsymbol{Y}$[1].

[1] Formally, we mean that for every R_i there has to exist an $\boldsymbol{x}_i \in R_i$ such that $\boldsymbol{x}_i[\boldsymbol{Y}] = \boldsymbol{y}[\boldsymbol{X}_i]$.

A CQ is called *acyclic* (an ACQ, for short), if it has a *join tree*, i.e., a labeled tree $\langle T, \lambda \rangle$ with node-labeling function λ such that (1) for every relation R_i there exists exactly one node u of T with $\lambda(u) = R_i$ and (2) λ satisfies the so-called connectedness condition, i.e., if some attribute A occurs in both relations $\lambda(u)$ and $\lambda(v)$ for two nodes u, v of T, then A occurs in the relation $\lambda(w)$ for every node w on the path between u and v.

Yannakakis [36] has shown that ACQs can be efficiently evaluated (that is, essentially, linear w.r.t. the input+output data and linear w.r.t. the size of the query) via 3 traversals of the join tree: (1) a bottom-up traversal of semi-joins, (2) a top-down traversal of semi-joins, and (3) a bottom-up traversal of joins. Formally, let u be a node in T with child nodes $u_1, \ldots, u_k$ of u and let relations R, $R_{i_1}, \ldots, R_{i_k}$ be associated with the nodes u, $u_1, \ldots, u_k$ at some stage of the computation. Then we set

(1) $R = (((R \ltimes R_{i_1}) \ltimes R_{i_2}) \cdots) \ltimes R_{i_k}$,
(2) $R_{i_j} = R_{i_j} \ltimes R$ for every $j \in \{1, \ldots, k\}$, and
(3) $R = (((R \bowtie R_{i_1}) \bowtie R_{i_2}) \cdots) \bowtie R_{i_k}$

in the 3 traversals (1), (2), and (3). The final result of the query can be read off from the resulting relation associated with the root r of T. Importantly, in case of a Boolean CQ, only the first bottom-up traversal of semi-joins is needed: a Boolean CQ evaluates to true, iff this bottom-up traversal yields a non-empty relation at the root r of T.

In this paper, we examine the *data complexity* of evaluating CQs with various types of query plans, i.e., we measure the space and time complexity as exponentials w.r.t. an upper bound N on the number of tuples in the given database D. We consider a *query plan* Π for a query Q as a structure that is associated with a specific algorithm for evaluating Q, e.g., in case of an ACQ Q, we may consider a join tree as a query plan, which is associated with Yannakakis' algorithm. We denote a class of plans by $\mathcal{C}$, e.g. all tree decompositions, and the plans of $\mathcal{C}$ for a query Q as $\mathcal{C}(Q)$. 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 refer to these jointly as space-time exponents $e(\Pi) = (s(\Pi), t(\Pi))$.

▶ **Definition 2** (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. We may write $\Pi_1 \equiv \Pi_2$, if both $\Pi_1 \preceq, \Pi_2$ and $\Pi_1 \succeq \Pi_2$ hold.*

A class of plans $\mathcal{C}_1$ improves over *(*dominates*) another class $\mathcal{C}_2$, denoted $\mathcal{C}_1 \preceq \mathcal{C}_2$, if for every query Q, it holds that $\forall \Pi_2 \in \mathcal{C}_2(Q)$, $\exists \Pi_1 \in \mathcal{C}_1(Q)$ such that $\Pi_1 \preceq \Pi_2$. We say that $\mathcal{C}_1$* strictly dominates *$\mathcal{C}_2$, denoted $\mathcal{C}_1 \prec \mathcal{C}_2$, if $\mathcal{C}_1 \preceq \mathcal{C}_2$ holds but not vice versa. We may write $\mathcal{C}_1 \equiv \mathcal{C}_2$ if both $\mathcal{C}_1 \preceq \mathcal{C}_2$ and $\mathcal{C}_1 \succeq \mathcal{C}_2$ hold.*

3 Decompositions Revisited

Several forms of decompositions have been proposed to make use of structural properties of the query in the quest for efficient query evaluation algorithms. The most fundamental form are tree decompositions (TDs, for short). They were originally introduced for *graphs* [31]. In case of queries (notably CQs), the underlying structure is best captured by *hypergraphs*. In principle, we could therefore associate with every hypergraph a graph and define TDs of a hypergraph as TDs of the corresponding graph. We prefer to define TDs (and, based on TDs, the other forms of decompositions studied here) directly for CQs without taking the detour

via the underlying hypergraph and an associated graph. Our notion of TDs corresponds to the definition of the TD via the "primal graph" of the hypergraph, i.e., the graph whose vertices are identical with the vertices of the hypergraph and two vertices are adjacent in the graph if they jointly occur in some hyperedge. This is arguably the most common way of defining the TD of a hypergraph and, ultimately, of a CQ, but not the only possible way, e.g.: in [9], TDs defined via the so-called "incidence graph" were considered.

▶ **Definition 3.** *For a given CQ Q, a tree decomposition (TD, for short) of Q is a tuple (T, χ) where $T = (V, E)$ is a directed tree and $\chi\colon V \to 2^{var(Q)}$ is a node-labeling function that assigns a set of variables to every node, satisfying the following conditions:*

1. *for every $R_i(\boldsymbol{X}_i) \in atoms(Q)$, there exists a node $v \in V$ with $\boldsymbol{X}_i \subseteq \chi(v)$;*
2. *for every $A \in var(Q)$, the nodes v with $A \in \chi(v)$ form a connected subset of V.*

The sets $\chi(v)$ are called *bags*. The width of a TD is the maximum size of its bags minus 1, i.e., $width(T, \chi) = \max_v(|\chi(v)|) - 1$. The tree-width $tw(Q)$ of a CQ Q is defined as the minimum width attainable by a TD of Q, i.e., $tw(Q) = \min_{(T,\chi)}(width(T, \chi))$. Note that, usually, TDs are considered as undirected trees. However, since we want to use TDs as query plans, it is more convenient to consider TDs as directed trees, since we will ultimately evaluate a (Boolean) CQ by a bottom-up traversal of the tree.

A straightforward way of evaluating a CQ Q by making use of a TD is to compute and materialize, for every node v, the query restricted to the variables in the bag $\chi(v)$ to turn the CQ into an ACQ and then to apply Yannakakis' algorithm. Hence, the principal source of complexity when using a TD for query evaluation are the subqueries Q^v corresponding to each bag. Clearly, there is a choice as to what plan we use to compute the subqueries Q^v. The most naive evaluation consists in associating with each bag with variables $X_{i_1}, \ldots, X_{i_\alpha}$ the relation obtained by computing the query restricted to those atoms which are covered by $\chi(v)$ and extending the result to the variables not contained in the subquery by allowing these variables to take any value from their active domain. That is, for every node $v \in V$, we compute the subquery $Q^v = \big(\bowtie_{i \in I^v} R_i\big) \times \big(\times_{j \in J^v} \mathbf{adom}_j\big)$, where the index sets I^v and J^v are defined as $I^v = \{i \mid \boldsymbol{X}_i \subseteq \chi(v)\}$ and $J^v = \{A_j \mid A_j \in \chi(v) \text{ and } \nexists i \in I^v \text{ with } X_j \in \boldsymbol{X}_i\}$, respectively. We can then transform Q into an equivalent ACQ $\bowtie_{v \in V} Q^v$ and evaluate this ACQ by applying Yannakakis' algorithm to it. The space and time exponents of this approach are essentially $(tw(Q) + 1, tw(Q) + 1)$ (cf. [10]).

A better evaluation strategy was proposed in [17] by extending TDs to hypertree decompositions (HDs, for short):

▶ **Definition 4.** *For a given CQ Q, a hypertree decomposition (HD, for short) of Q is a tuple (T, λ, χ) where (T, χ) is a tree decomposition and $\lambda\colon V \to 2^{atoms(Q)}$ is another node-labeling function that assigns a set of atoms to every node, satisfying (in addition to the properties of TDs according to Definition 3) the following conditions:*

3. *for every $u \in V$, the atoms in $\lambda(u)$ are an integral edge cover of $\chi(u)$, i.e., $\chi(u) \subseteq var(\lambda(u))$;*
4. *for every $u \in V$, the so-called "special condition" holds, i.e., let T_u denote the subtree rooted at u, then $\big(\bigcup_{v \in T_u} \chi(v)\big) \cap var(\lambda(u)) \subseteq \chi(u)$.*

The width of an HD is the maximum size of its λ-labels, i.e., $width(T, \lambda, \chi) = \max_v(|\lambda|)$. The hypertree-width $hw(Q)$ of a CQ Q is defined as the minimum width attainable by an HD of Q, i.e., $hw(Q) = \min_{(T,\lambda,\chi)}(width(T, \lambda, \chi))$. Note that the fourth condition in Definition 4 was introduced in [17] only to make the problem of deciding, if $hw(Q) \leq k$ holds, tractable for fixed $k \geq 1$. When using an HD for query evaluation, this condition is irrelevant. If we drop the fourth condition, we get *Generalized Hypertree Decompositions (GHDs, for short)* with

generalized hypertree-width, denoted $ghw(Q)$ as the associated width measure [2]. Clearly, GHDs generalize HDs and $ghw(Q) \leq hw(Q)$ holds for every CQ Q. The downside of GHDs is that even deciding if $ghw(Q) \leq 2$ for given query Q holds, is NP-complete [18, 16].

Bounded hw and bounded ghw generalize acyclicity in that ACQs are precisely the CQs with $hw = ghw = 1$. In order to turn a CQ with given HD or GHD into an ACQ, we just need to join the relations in the λ-label and project the result to the χ-label for every node $v \in V$. That is, for every $v \in V$, we compute the subquery $Q^v = \pi_{\chi(v)}\big(\bowtie_{R_i(\boldsymbol{X}_i)\in\lambda(v)} R_i\big)$. The resulting ACQ $\bowtie_{v\in V} Q^v$ can then be evaluated by applying Yannakakis' algorithm. We thus get the space-time exponents $(hw(Q), hw(Q))$ and $(ghw(Q), ghw(Q))$, respectively [17].

A further refinement of this technique was proposed in [20] by introducing fractional hypertree decompositions (FHDs, for short):

▶ **Definition 5.** *For a given CQ Q, a fractional hypertree decomposition (FHD, for short) of Q is a tuple (T, γ, χ), where (T, χ) is a tree decomposition and $\gamma\colon V \to [0,1]^{atoms(Q)}$, satisfying (in addition to the properties of TDs according to Definition 3) the following condition:*

3. *for every $u \in V$, $\gamma(u)$ is a fractional edge cover of $\chi(u)$, i.e., $\gamma(u)\colon atoms(Q) \to [0,1]$, such that, for every $A \in \chi(v)$, $\sum_{i:A\in R_i(\boldsymbol{X}_i)} \gamma(v)(R_i(\boldsymbol{X}_i)) \geq 1$.*

The width of an FHD is defined as the maximum weight of its γ-labels, i.e., $width(T, \gamma, \chi) = \max_v(\sum_i \gamma(v)(R_i(\boldsymbol{X}_i)))$. The fractional hypertree-width $fhw(Q)$ of a CQ Q is defined as the minimum width attainable by an FHD of Q, i.e., $fhw(Q) = \min_{(T,\gamma,\chi)}(width(T, \gamma, \chi))$. In order to turn a CQ with given FHD into an ACQ, we need to evaluate the query Q restricted to the variables in $\chi(v)$ for every $v \in V$, i.e., $Q^v(\chi(v)) = \bigwedge_{i=1}^m R_i(\boldsymbol{X}_i \cap \chi(v))$. By using a WCOJ method such as Generic Join [29], the maximal output size and the time for evaluating such a query Q^v over a database D is $O(|D|^{\rho^*(\chi(v))})$. The resulting ACQ $\bowtie_{v\in V} Q^v$ can then be evaluated by Yannakakis' algorithm, leading to the space-time exponents $(fhw(Q), fhw(Q))$ [20].

For every CQ Q, we clearly have $fhw(Q) \leq ghw(Q) \leq hw(Q) \leq tw(Q)+1$. Hence, treating a TD as an FHD and evaluating the subqueries at each bag by some WCOJ techniques in order to turn Q into an ACQ, is optimal among the decomposition methods discussed in this section. As mentioned, e.g., in [11], one can even do better space-wise. More specifically, it suffices to materialize only the subqueries restricted to the variables in the intersection between the bags at a node and its parent. To each vertex $v \in V$ of a TD (T, χ), we thus associate a query as follows. Let $\boldsymbol{Y}^v := \chi(v)$ and $\boldsymbol{Z}^v := \chi(v) \cap \chi(parent(v))$ or, if v is the root node, then $\boldsymbol{Z}^v := \emptyset$. Then we compute, in a bottom-up traversal of the TD, the queries

$$Q^v(\boldsymbol{Z}^v) \leftarrow \bigwedge_{i=1}^{m} R_i(\boldsymbol{X}_i \cap \boldsymbol{Y}^v) \wedge \bigwedge_{w\in child(v)} Q^w(\boldsymbol{Z}^w).$$

Using a WCOJ method such as Generic Join for evaluating each of these subqueries, we end up with the space-time exponents $s(T, \chi) = \max_{(u,v)\in E(T)} \rho^*(\chi(u) \cap \chi(v))$ and $t(T, \chi) = \max_{v\in V(T)} \rho^*(\chi(v))$. The latter is equal to $fhw(T, \chi)$, if (T, χ) is the TD underlying an FHD of minimum width.

4 Pseudo-Trees

In this section, we discuss *pseudo-trees* [15, 11] together with several extensions. It will turn out that WCOJ algorithms such as Generic Join [29] can actually be seen as a special case of pseudo-tree-based algorithms. We therefore revisit Generic Join (GJ)) query plans first.

4.1 Generic Join

The *Generic Join* (GJ) algorithm [29] fixes an arbitrary order on the variables $A_1, \ldots, A_k$ of a given CQ Q 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 assignment $\boldsymbol{y}_{j-1} = (a_1, \ldots, a_{j-1})$, the j'th nested loop iterates through all values a_j in $\bigcap_{i:A_j \in \boldsymbol{X}_i} R_i[A_j|\boldsymbol{y}_{j-1}]$ to extend $\boldsymbol{y}_{j-1}$ to a partial assignment $\boldsymbol{y}_j = (a_1, \ldots, a_{j-1}, a_j)$. Here, we write $R_i[A_j|\boldsymbol{y}_{j-1}]$ to denote the set of values of attribute A_j in relation R_i restricted to those tuples that coincide with $\boldsymbol{y}_{j-1}$ on all attributes in $\boldsymbol{X}_i \cap \{A_1, \ldots, A_{j-1}\}$. 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.

▶ **Definition 6.** *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). In case of a Boolean query Q, the space-time exponents of any $\Pi \in \mathcal{GJ}(Q)$ are $e(\Pi) = (0, \rho^*(var(Q)))$.*

▶ **Theorem 7** ([29])**.** *Let Q be a Boolean query. If $\Pi \in \mathcal{GJ}(Q)$, then we can compute $[\![Q]\!]$ based on Π in space and time given by $e(\Pi)$. Concretely, the space used is $O(|D|^0)$ (i.e., constant), and the time spent is $O(|D|^{\rho^*(var(Q))})$.*

4.2 Basic Form of Pseudo-Trees

Pseudo-tree-based algorithms improve upon $\mathcal{GJ}$-algorithms by making use of the fact that the instantiation of some variables may split the query into disconnected parts that can be evaluated separately. This means that some of the inner loops of the $\mathcal{GJ}$ algorithm can be executed in parallel rather than in a nested fashion. This idea is illustrated by the following simple example:

▶ **Example 8.** Consider the 3-path query $Q() \leftarrow R_1(A, B) \wedge R_2(B, C)$. The bag semantics of $Q()$ is obtained as $\sum_{a,b,c \in \mathbf{dom}} R_1(a, b) \cdot R_2(b, c)$. By slight abuse of notation, we are writing $R_i(x, y)$ for the indicator expression $1_{R_i}(x, y)$, which takes the value 1 if R_i contains the tuple (x, y) and which is 0 otherwise. GJ proceeds in 3 nested loops, e.g., for the variable ordering (A, B, C), GJ computes the expression $\sum_a \sum_b \sum_c R_1(a, b)R_2(b, c)$, with runtime $O(|R_1| \cdot |R_2|)$, i.e., the time exponent is $\rho^*(Q) = 2$. 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 $O(|R_1| + |R_2|)$, i.e., the time exponent is 1. Structurally speaking, the loops of GJ can be described as a path $A \text{—} B \text{—} C$ while the loops of the pseudo-tree based algorithm are tree-structured where B is the root and A, C are its children, i.e., $\begin{smallmatrix} & B & \\ A & & C \end{smallmatrix}$.

To formally define and analyze pseudo-tree algorithms, we introduce the following notation: For a directed tree $T = (\boldsymbol{V}, E)$ and $A \in \boldsymbol{V}$, 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 $desc(A)$ and $\overline{desc}(A)$ for the set of descendants of A, without and with A respectively.

▶ **Definition 9** ([15, 11])**.** *A* pseudo-tree *(PT) of a query Q is a directed tree $P = (\boldsymbol{V}, E)$, such that every atom $R_i(\boldsymbol{X}_i)$ is contained in a branch: formally, $\exists A \in \boldsymbol{V}$ such that $\boldsymbol{X}_i \subseteq \overline{anc}(A)$.*

▶ **Definition 10.** *The class of query plans $\mathcal{PT}(Q)$ of a query Q consists of pseudo-trees P of Q. In case of a Boolean query Q, the space-time exponents of any $P \in \mathcal{PT}(Q)$ are $e(P) := (0, \max_{A \in var(Q)} \rho^*(\overline{anc}(A))$.*

Pseudo-tree plans strictly dominate GJ plans, i.e. $\mathcal{PT} \prec \mathcal{GJ}$, 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 exponents, thus $\mathcal{PT} \preceq \mathcal{GJ}$. On the other hand, the 3-path query in Example 8 shows that $\mathcal{GJ} \not\preceq \mathcal{PT}$. Indeed, while $\mathcal{PT}$-plans still require constant space, their time exponent is no longer given by the fractional edge cover number of *all* variables but by the maximum fractional edge cover number over all branches in the given pseudo-tree.

▶ **Theorem 11.** *Let Q be a Boolean query. If $P \in \mathcal{PT}(Q)$, then we can compute $[\![Q]\!]$ based on P in space and time given by $e(P)$. Concretely, the space used is $O(|D|^0)$ (i.e., constant), and the time spent is $O(|D|^{t(P)})$ with $t(P) = \max_{A \in var(Q)} \rho^*(\overline{anc}(A))$.*

Theorem 11 is based on an algorithm that simply uses a loop structure that mimics the pseudo-tree P. That is, the top-most loop iterates through the possible values a_1 for the variable at the root $R = root(P)$. Then, there is an independent nested loop for each child of R and so on .... Reconsidering the query used in Example 1, the pseudo-tree-based loops arising from the pseudo-tree $A—B—C—D—E$ are the ones depicted in Figure 1. Importantly, the loops proceed as GJ does. That is, at the loop for some $A \in var(Q)$, the loops above it fix a partial assignment $\boldsymbol{y} \in \mathbf{dom}^{anc(A)}$ which is then extended by a value $a \in \bigcap_{i:\ A \in \boldsymbol{X}_i} R_i[A|\boldsymbol{y}]$ to the partial assignment $(\boldsymbol{y}, a)$. Iterating over the smallest of these sets and probing in the other, the time at A is determined by $\rho^*(\overline{anc}(A))$ (over all $\boldsymbol{y}$).

4.3 Pseudo-Trees with Caches

We now consider the extension of pseudo-trees by caches to store intermediate results and avoid their recomputation. A *cache* is a data structure that maps from a set of *keys* to a set of *values*. As is illustrated in the next example, caches make a time-space trade-off possible.

▶ **Example 12.** To motivate caching, consider the 4-path query $Q() \leftarrow R(A,B) \wedge S(B,C) \wedge T(C,D)$ under bag semantic, 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 [3], $O(N^2)$. Intuitively, this query plan corresponds to the summation $\sum_a (\sum_b R(a,b) \cdot (\sum_c S(b,c) \cdot (\sum_d T(c,d))))$[2]. We note that the subexpression $M_C(b) := \sum_c S(b,c) \cdot \sum_d T(c,d)$ is independent of a, and, by caching the values $M_C(b)$, we can avoid recomputing this expression for different values a. Similarly, we can cache $M_D(c) := \sum_d T(c,d)$. By adding caches, we can trade space for time. In our example, filling the cache $M_D(\cdot)$ for all possible values of c requires at most linear time. Likewise, filling the cache $M_C(\cdot)$ for all possible values of b can be done in linear time, i.e.: we have to iterate through all tuples $S(b,c)$ but, due to caching, we only need to iterate once through all tuples $T(c,d)$. Hence, the two caches M_C and M_D decrease the runtime of the PT given above from $O(N^2)$ to $O(N)$, while the space increases from $O(1)$ to $O(N)$.

Suppose that we want to store the results for possible extensions of partial solutions to the attribute A in a cache M_A. The *key* of M_A is called the *context* of A, defined as follows.

▶ **Definition 13** ([11]). *The* context *of a variable $A \in \boldsymbol{V}$ is defined as*

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

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

[2] As in Example 8, we are writing $R_i(x,y)$ for the indicator expression $1_{R_i}(x,y)$, which takes the value 1 if R_i contains the tuple (x,y) and which is 0 otherwise.

The main property of $con(A)$ is that the subquery corresponding to the subtree (of the pseudo-tree) rooted at A does not depend on the instantiation of *all* ancestors of A but only on the instantiation of the variables in $con(A)$. Therefore, we can cache these values in a cache M_A with key $con(A)$, whose values are subsets of $\mathbf{dom}^A$ (equivalently, mappings $\mathbf{dom}^A \to \mathbb{B}$) or mappings $\mathbf{dom}^A \to \mathbb{N}$ for set and bag semantics, respectively. Hence, the type of this cache is $M_A \colon \mathbf{dom}^{con(A)} \to (\mathbf{dom}^A \to \mathbb{B})$ or $M_A \colon \mathbf{dom}^{con(A)} \to (\mathbf{dom}^A \to \mathbb{N})$, respectively, which is equivalent to $M_A \colon \mathbf{dom}^{\overline{con}(A)} \to \mathbb{B}$ or $M_A \colon \mathbf{dom}^{\overline{con}(A)} \to \mathbb{N}$, respectively. Therefore, the space usage of the cache M_A is given by $\rho^*(\overline{con}(A))$ as each cached tuple satisfies the query Q restricted to variables $\overline{con}(A)$.

To determine the time spent by the algorithm at node A, suppose that some $B \in \overline{anc}(A)$ uses a cache, and let B be closest to A with this property. Moreover, suppose that we want to extend a partial solution to the attribute A for some instantiation $\boldsymbol{x}$ of $anc(A)$ and later for another instantiation $\boldsymbol{y}$ of $anc(A)$. Further, let $\boldsymbol{x}[con(B)] = \boldsymbol{y}[con(B)]$. Then, $\boldsymbol{x}[anc(B)]$ and $\boldsymbol{y}[anc(B)]$ must have both resulted in a cache miss at B. However, this only happens the first time the partial instantiation $\boldsymbol{x}[anc(B)]$ is considered. Thus, $\boldsymbol{x}[anc(B)]$ and $\boldsymbol{y}[anc(B)]$ must be the same partial instantiation and we can essentially ignore $anc(B) \setminus con(B)$ in the computation of the time spent at A. Let $[A, B] \subseteq \overline{anc}(A)$ be the variables along the path connecting A to B. Put differently, at A the values of the variables $con(B) \cup [A, B]$ are unique and they are the only variables of $\overline{anc}(A)$ relevant to the time consumption by the loop at A.

▶ **Example 14.** Let us compute the space-time usage of plans for the 5-path $Q() \leftarrow R_1(A, B) \wedge \cdots \wedge R_4(D, E)$ for three different cache setups. We always use the linear PT $A{-}B{-}C{-}D{-}E$. Without a cache, we simply compute $\sum_{a,b} R_1(a, b) \sum_c \cdots \sum_e R_4(d, e)$ with $O(1)$ space and $O(N^3)$ time usage. The worst time usage is at E and given by $\rho^*(\overline{anc}(E)) = \rho^*(A \dots E) = 3$. With a cache at C, we compute $\sum_{a,b} R_1(a, b) M_C(b)$ and $M_C(b) = \sum_c R_2(b, c) \sum_d R_3(c, d) \sum_e R_4(d, e)$ with $O(N)$ space and $O(N^2)$ time usage. The worst time usage is again given at E but computed as $\rho^*(con(C) \cup [E, B]) = \rho^*(B \dots E) = 2$. Lastly, when we add caches at D and E as well, we compute $\sum_{a,b} R_1(a, b) M_C(b)$, $M_C(b) = \sum_c R_2(b, c) M_D(c)$, $M_D(c) = \sum_d R_3(c, d) M_E(d)$, and $M_E(d) = \sum_e R_4(d, e)$, resulting in $O(N)$ space and $O(N)$ time usage.

We are now ready to define a pseudo-tree with caching and its space-time exponents:

▶ **Definition 15.** *A* pseudo-tree with caching *(PTC) of a query 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 $root(P) \in \boldsymbol{C}$.*

▶ **Definition 16.** *The class of query plans $\mathcal{PTC}(Q)$ of a query Q consists of pseudo-trees with caching $(P, \boldsymbol{C})$ of Q. If Q is a Boolean query, then the space and time exponents of a $\mathcal{PTC}(Q)$ plan $(P, \boldsymbol{C})$ are defined as:*

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

where $B_A := \min(\boldsymbol{C} \cap \overline{anc}(A))$, i.e., the node in $\overline{anc}(A)$ closest to A with a cache.

Pseudo-trees with caches strictly dominate pseudo-trees, i.e. $\mathcal{PTC} \prec \mathcal{PT}$, because a $\mathcal{PTC}(Q)$-plan with no caches (i.e., $\boldsymbol{C} = \{root(P)\}$), is simply a $\mathcal{PT}(Q)$-plan. On the other hand, we have $\mathcal{PT} \not\preceq \mathcal{PTC}$, since the space-time exponents of the $\mathcal{PTC}$-plan for the 4-path query shown in Example 12 are not attainable with a $\mathcal{PT}$-plan.

▶ **Theorem 17.** *Let Q be a Boolean query. If $(P, \boldsymbol{C}) \in \mathcal{PTC}(Q)$, then we can compute $[\![Q]\!]$ based on $(P, \boldsymbol{C})$ in space $O(|D|^{s(P,\boldsymbol{C})})$ and time $O(|D|^{t(P,\boldsymbol{C})})$.*

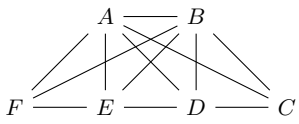


Figure 2 Query $Q_{\triangle}$.

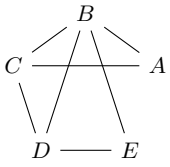


Figure 3 Query $Q_{\hexagon}$.

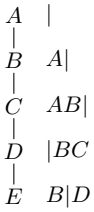


Figure 4 PTCR for $Q_{\hexagon}$.

4.4 Pseudo-Trees with Caching and Resets

We have seen how the addition of caches reduces the time exponent while increasing the space exponent of a pseudo-tree. Now we describe *pseudo-trees with caching and resets*, $\mathcal{PTCR}$, which allow for a finer trade-off between space and time. The basic principle was introduced in [11], while a complexity analysis and further improvements were presented in [12].

▶ **Example 18.** To motivate resetting caches, consider the query $Q_{\triangle}()$ in Figure 2 with bag semantics and the PT $A{-}B{-}C{-}D{-}E{-}F$. 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 extending a partial solution $abcd$ to attribute E, the result is stored in $M_E(abd)$. Later, if c has changed while abd are the same, the result can be retrieved from the cache. Note that, once the value of a or b changes, we can safely discard (or reset) all entries $M_E(ab{-})$, because the values abc are processed in lexicographic order. We can, therefore, 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 of M_E to $\rho^*(D) = 1$, and similarly for the cache M_F. With this "reset" improvement, $Q_{\triangle}$ can be computed with space-time exponents $(1, 2)$.

In Example 18, resetting the cache did not increase the time complexity. However, this is not true in general. For example, if we drop the edge AE from the query $Q_{\triangle}()$, then the context of attribute E becomes $con(E) = BD$. If we now drop B from the cache, then we will increase the time required when extending partial solutions to attribute E, as can be seen as follows: B may take a value b, then we reset the cache when B takes a new value b'; but then it can take the value b again, when A has changed. Thus, dropping B from the cache key may require us to redo some work. This increases the time complexity and it also complicates the time analysis of the algorithm. We describe now this technique in general.

▶ **Definition 19.** *A pseudo-tree with caching and resets (PTCR) of a query Q is a pair (P, C), where $P = (\boldsymbol{V}, E)$ is a pseudo-tree and C is a function $C\colon \boldsymbol{V} \to \mathbb{N}$.*

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 Definition 19 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 18, if $C(E) = 1$, then we write $con(E) = AB|D$. If $C(A) = 0$ then it is equivalent to A having no cache.

A basic algorithm for handling resets described in [11] 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 trying to extend a partial solution $\boldsymbol{y}$ to A, we first check whether $conInst_A = \boldsymbol{y}[conInst(A)]$. If yes, the cache can be used like in Section 4.4. If not, then the cache M_A has to be reset. However, this algorithm is not optimal, as we explain next.

▶ **Example 20.** Consider the Boolean query $Q_{\hexagon}()$ in Figure 3 with bag semantics and the PTCR (P, C) in Figure 4. 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; on the other hand, D has a full cache $M_D(BC)$. To determine the time complexity at variable E, 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 attempts to extend partial solutions to E with a new B is bounded by $\rho^*(AB)$. Or, we notice that D has a fully stored cache $M_D(BC)$, and extensions to E are only tried with unique BC pairs, yielding the bound $\rho^*(BC)$.

These bounds can be reduced to $\rho^*(B)$ by computing the cache $M_D(BC)$ eagerly. When trying to extend a partial solution abc to D 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 all distinct values bc of BC for the subquery $Q[BC]$ restricted to BC and stores the result for every value d of D (which includes the iteration over the values of E) in $M_D(bc)$. D traverses the subquery $Q[BC]$ 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 18. The number of cache resets is reduced[3] to $\rho^*(B)$. This example was simple, because D had no instantiated variables, $conInst(D) = \emptyset$; the general case requires the technical Definition 21 below. Notice that filling the cache eagerly is a significant extension of GJ and all its implementations in practice [34, 14], 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, we expect the values of B at E in sorted order $b_1, b_1, b_2, b_2, b_3, \ldots$

In general, to extend query evaluation based on $\mathcal{PTC}$ to $\mathcal{PTCR}$, we use the following:

▶ **Definition 21.** *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:*

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

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

Intuitively, the instantiated relevant context $ria(A)$ determines the number of resets of the cache M_A: on one hand, it contains the instantiated context $conInst(A)$. Clearly, for every different value combination of the attributes in $conInst(A)$, we have to fill the cache M_A again from scratch. On the other hand, when the ancestors of $conInst(A)$ take different

[3] In this simple example, we have $\rho^*(AB) = \rho^*(BC) = \rho^*(B) = 1$. Hence, the improved algorithm does not reduce the runtime but the dependency. However, when also considering the time required to iterate through the E-loop, we get $\rho^*(BE) = 1 < \rho^*(ABE) = 2$ and $< \rho^*(BCE) = 1.5$.

values, then $\mathit{conInst}(A)$ may later take the same value combination again. Thus, the total number of cache resets of M_A depends on the possible value combinations of *conInst and* the possible number of repetitions of the same value combinations of *conInst*. To bound this, $\mathit{ria}(A)$ lies between $\mathit{conInst}(A)$ and their ancestors $(\overline{\mathit{anc}}(\min(\mathit{conInst}(A))))$, and crucially satisfies the property that the partial instantiations $\boldsymbol{y} \in \mathbf{dom}^{\overline{\mathit{anc}}(A)}$ that "arrive" at A come in lexicographic order even when projected onto $\mathit{ria}(A)$. Hence, we can use $\rho^*(\mathit{ria}(A))$ to determine the number of cache resets.

As we have alluded to before, maintaining this lexicographic order does not come for free but requires the caches to be computed eagerly. That is, if we inductively assume $\mathit{ria}(A)$ to come lexicographically, then go through $\mathit{conSto}(A)$ lexicographically, and then also go though A lexicographically, then $\overline{\mathit{ra}}(A)$ is given to each child lexicographically. The problem with the naive approach (i.e., the lazy computation) is that an ancestor of A between $\mathit{conInst}(A)$ and A may potentially destroy the lexicographic order in which $\mathit{conSto}(A)$ should be processed and is passed to the children. Referring to Example 20, when trying to extend partial solutions to D, we have $\mathit{ria}(D) = \emptyset$. We, therefore, iterate only once over all values bc of $[\![Q[BC]]\!]$. Hence, the $-bcd$ partial assignments $(BCD = \overline{\mathit{ra}}(D))$ arrive at E lexicographically. The cache M_E at E is only reset when b changes, i.e., the relevant variables are $\mathit{ria}(E) = BCD \cap AB = B$. Thus, to bound the time spent at E we can use $\rho^*(\overline{\mathit{ra}}(E)) = \rho^*(BDE) = 1.5$.

We next define the class of query plans $\mathcal{PTCR}$ together with their space-time exponents:

▶ **Definition 22.** *The class of query plans $\mathcal{PTCR}(Q)$ for a query Q consists of PTCR (P, C) of Q. If Q is a Boolean query, then the space and time exponents of a $\mathcal{PTCR}(Q)$ (P, C) are defined as:*

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

▶ **Theorem 23.** *Let Q be a Boolean query. If $(P, C) \in \mathcal{PTCR}(Q)$, then we can compute $[\![Q]\!]$ based on (P, C) in $O(|D|^{s(P,C)})$ space and in $O(|D|^{t(P,C)})$ time.*

4.5 Pseudo-Trees Using Recursion to Reorient Sub-Trees

We now present a final extension of pseudo-trees termed *recursive pseudo-trees* $\mathcal{RPT}$ in [12]. They are motivated by the following observation: In a $\mathcal{PTCR}$-algorithm based on the PTCR (P, C), for an attribute A with $C(A) \neq 0$ and where a cache miss occurs, we evaluate, for every instantiation $\boldsymbol{y}_{\text{sto}}$ of the stored context $\mathit{conSto}(A)$, the subquery of Q restricted to the variables $\overline{\mathit{desc}}(A)$ and cache the result in M_A. The results (over all $\boldsymbol{y}_{\text{sto}}$) then make up the cache $M_A(\mathit{con}(A))$. Hence, actually, we can see $M_A(\mathit{con}(A))$ as a query itself where we have simply already fixed some variables (also some of $\mathit{con}(A)$ are fixed). Intuitively, we aim to solve the subquery

$$M_A(\mathit{con}(A)) \leftarrow Q[\overline{\mathit{ra}}(A) \cup \overline{\mathit{desc}}(A)]$$

but where the values of $\mathit{ria}(A)$ are fixed to some partial assignment $\boldsymbol{y}_{\text{ria}}$. Hence, it only remains to go though the variables $\mathit{conSto}(A) \cup \overline{\mathit{desc}}(A)$. The $\mathcal{PTCR}$-based algorithm does exactly this and proceeds as follows: First, it iterates through the variables in $\mathit{conSto}(A)$ and, then, it proceeds to the variables in $\overline{\mathit{desc}}(A)$ according to (P, C). The variables in $\mathit{conSto}(A)$ are thus processed in the order they are given as ancestors of A in P. While intuitive, this is an arbitrary decision. One could instead simply use a different PTCR (P', C') – imposing a different order on the remaining variables $\mathit{conSto}(A) \cup \overline{\mathit{desc}}(A)$ – to compute this subquery. 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').

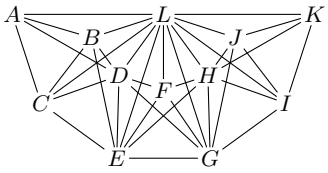


Figure 5 Query $Q_{\Box}$.

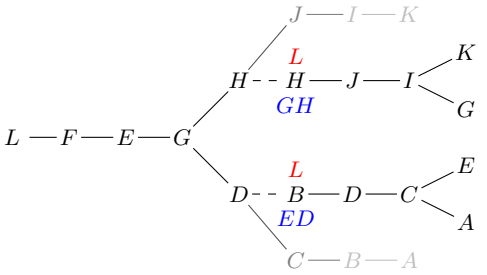


Figure 6 An RPT for $Q_{\Box}$.

▶ **Example 24.** To motivate the use of different PTCRs to fill the caches, consider the Boolean query $Q_{\Box}()$ given in Figure 5 with bag semantics. As there are many 5-cliques, our aim is to achieve time complexity 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 or E, as well as $\mathcal{K}_2$ together with D or H, and $\mathcal{K}_3$ together with G or K. Thus, a natural way of evaluating $Q_{\Box}$ 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 $con(C) = L|ED$. Now we would have to fill the cache for the possible values of $conSto(C) = ED$ that fit to the current value l of L, and then solve the remaining query on A, B, C. Thus, the query to solve is

$$M_C(l, E, D) \leftarrow Q[l, A, B, C, D, E]$$

This can be done by a loop structure that first extends l to values bdc and then loops through A and E independently. 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 can be taken care of symmetrically. 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 how a $\mathcal{PTCR}$ algorithm would have filled the cache of C (no matter how we complete the PTCR). Crucially, we have inverted the order of E and D to fill the cache.

Figure 6 depicts an RPT that follows the same idea of the sketch of an algorithm above. That is, the RPT starts out as a PTCR (solid edges) but where a different PTCR is used to fill the caches at C and J. To that end, these sub-parts (the gray parts) are replaced by the PTCRs attached with dashed edges. The red L indicates the fixed inputs that the new PTCRs have to take into consideration and the blue ED (resp. GH) are the output variables of the subquery.

The definition of recursive pseudo-trees and the analysis of their time complexity are quite involved. We, therefore, omit them here and refer the interested reader to [12]. Suffice it to mention that $\mathcal{RPT}$ plans strictly dominate $\mathcal{PTCR}$ plans, which is seen as follows: We clearly have $\mathcal{RPT} \preceq \mathcal{PTCR}$, since $\mathcal{RPT}$-plans without recursion are simply $\mathcal{PTCR}$ plans. On the other hand, we have $\mathcal{PTCR} \not\preceq \mathcal{RPT}$, as illustrated in Example 24.

5 Combining the Two Worlds

Having discussed query plans based on decompositions in Section 3 and based on various variants of pseudo-trees in Section 4, we now consider their combination. That is, we study classes of query plans which, on one hand, use tree decompositions and, on the other hand, use query plans from some class $\mathcal{C}$ for evaluating the subqueries in each bag. We denote the resulting class of query plans as $\mathcal{TD}^{\mathcal{C}}$. More specifically, we will have a closer look at the classes $\mathcal{TD}^{\mathcal{GJ}}$, $\mathcal{TD}^{\mathcal{PT}}$, $\mathcal{TD}^{\mathcal{PTC}}$, and $\mathcal{TD}^{\mathcal{PTCR}}$, which combine tree decompositions with plans in the classes $\mathcal{GJ}$, $\mathcal{PT}$, $\mathcal{PTC}$, and $\mathcal{PTCR}$, respectively, from Section 4.

5.1 Tree Decompositions and Generic Join

We have already observed in Section 3 that the space-time exponents of $\mathcal{TD}^{\mathcal{GJ}}$ plans are $s(T,\chi) = \max_{(u,v)\in E(T)} \rho^*(\chi(u)\cap\chi(v))$ and $t(T,\chi) = \max_{v\in V(T)} \rho^*(\chi(v))$; the time exponent is equal to the fractional hypertree-width of the query, if (T,χ) is the TD underlying an FHD of minimum width. We, therefore, concentrate here on a comparison with other classes of query plans. Of course, the relationship $\mathcal{TD}^{\mathcal{C}} \preceq \mathcal{C}$ holds for any class $\mathcal{C}$ of query plans, since we can simply consider the trivial tree decomposition consisting of a single bag. Below we show that $\mathcal{PT}$ and $\mathcal{TD}^{\mathcal{GJ}}$ are incomparable.

▶ **Theorem 25.** *The classes $\mathcal{PT}$ and $\mathcal{TD}^{\mathcal{GJ}}$ of query plans are incomparable, i.e., $\mathcal{PT} \not\preceq \mathcal{TD}^{\mathcal{GJ}}$ and $\mathcal{TD}^{\mathcal{GJ}} \not\preceq \mathcal{PT}$.*

Proof Sketch. To show $\mathcal{PT} \not\preceq \mathcal{TD}^{\mathcal{GJ}}$, we revisit the 4-path query $Q() \leftarrow R_1(A,B) \wedge R_2(B,C) \wedge R_3(C,D)$ from Example 12 under bag semantics. We have seen that a $\mathcal{PT}$-plan can evaluate this query with space-time exponents $(0,2)$. Now consider a TD with three bags $\chi(u) = \{AB\}$, $\chi(v) = \{BC\}$, $\chi(w) = \{CD\}$, where u is the root. Then a $\mathcal{TD}^{\mathcal{GJ}}$-plan achieves space-time exponents $(1,1)$. Clearly, time exponent 1 is not feasible with a $\mathcal{PT}$-plan.

On the other hand, to show $\mathcal{TD}^{\mathcal{GJ}} \not\preceq \mathcal{PT}$, consider again the 3-path query from Example 8. We have seen in Example 8 that it can be evaluated by a $\mathcal{PT}$-plan with space-time exponents $(0,1)$. A $\mathcal{TD}^{\mathcal{GJ}}$-plan cannot achieve this, since it either has time exponent $= 2$ (for a TD with a single bag) or space exponent 1 (for a TD with 2 bags). ◀

5.2 Tree Decompositions and Pseudo-Trees with or without Caching

In Theorem 25, we have established the relationships $\mathcal{PT} \not\preceq \mathcal{TD}^{\mathcal{GJ}}$ and $\mathcal{TD}^{\mathcal{GJ}} \not\preceq \mathcal{PT}$. Together with the trivial relationship $\mathcal{TD}^{\mathcal{PT}} \preceq \mathcal{TD}^{\mathcal{GJ}}$, we thus also have $\mathcal{TD}^{\mathcal{PT}} \prec \mathcal{TD}^{\mathcal{GJ}}$. Actually, in [12], even a slightly stronger result was shown: recall from Example 8 that not only $\mathcal{GJ} \not\preceq \mathcal{PT}$ holds, but, for the 3-path query, a $\mathcal{PT}$-plan can even achieve a strictly smaller time exponent without increasing the space exponent. Similarly, there exists a (more involved) query, for which the optimal time exponent with a $\mathcal{TD}^{\mathcal{GJ}}$ plan can be improved by a $\mathcal{TD}^{\mathcal{PT}}$ plan without increasing the space exponent. For details, see [12].

Below, we present two results on $\mathcal{PTC}$-plans: on one hand, even with decompositions, $\mathcal{PTC}$ dominates $\mathcal{TD}^{\mathcal{GJ}}$. On the other hand, if we combine tree decompositions with $\mathcal{PTC}$-plans for evaluating the subqueries at the bags (i.e., we have plans in the class $\mathcal{TD}^{\mathcal{PTC}}$), we do not get an improvement over $\mathcal{TD}^{\mathcal{PT}}$-plans.

▶ **Theorem 26.** *The class $\mathcal{PTC}$ dominates $\mathcal{TD}^{\mathcal{GJ}}$, i.e., $\mathcal{PTC} \preceq \mathcal{TD}^{\mathcal{GJ}}$.*

Proof Sketch. Given a plan $\Pi := (T,\chi) \in \mathcal{TD}^{\mathcal{GJ}}$, we construct a plan $\Pi_0 := (P, \boldsymbol{C}) \in \mathcal{PTC}$ such that $\Pi_0 \preceq \Pi$. Note that, by slight abuse of notation, we denote query plans in $\mathcal{TD}^{\mathcal{GJ}}$ by only writing the TD and not explicitly specifying the query plan for every bag $\chi(v)$. In case

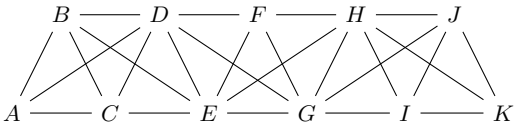


Figure 7 Query $Q_{\square\!\square}$.

of $\mathcal{GJ}$ this is justified since any variable order leads to the same time exponent, namely the fractional edge cover number $\rho^*(\chi(v))$. The construction is based on the variable elimination procedure for a tree decomposition [23], 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, let $p := \mathit{parent}(v)$, and let $\{A_1, \ldots, A_k\} := \chi(v) \setminus \chi(p)$, i.e., the set of variables that occur in v but nowhere else. Let $\boldsymbol{Z}$ be their neighbors, i.e., $\boldsymbol{Z} := \{B \mid \exists \text{ atom } R_i(\boldsymbol{X}_i), \exists j, \text{ s.t. } A_j, B \in \boldsymbol{X}_i\}$. Then, $\boldsymbol{Z} \subseteq \chi(v) \cap \chi(p)$ must hold. Let Q' be the query obtained from Q by removing all variables $A_1, \ldots, A_k$, and adding a new atom $R(\boldsymbol{Z})$, i.e., $Q' \leftarrow Q[var(Q) \setminus \{A_1, \ldots, A_k\}](\boldsymbol{X}) \wedge R(\boldsymbol{Z})$. Clearly, the plan $\Pi' = (T', \chi)$ obtained from Π by removing the leaf v is a $\mathcal{TD}^{\mathcal{GJ}}$-plan for Q'. Hence, by the induction hypothesis, Π' can be converted to a $\mathcal{PTC}$ 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 $R(\boldsymbol{Z})$, that we have "artificially" included into Q'). 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 $\Pi_0 \preceq \Pi$: for the space exponent, note that $\mathit{con}(A_1) = \boldsymbol{Z}$ is the only additional cache in Π_0 compared with Π. By $\boldsymbol{Z} \subseteq \chi(v) \cap \chi(p)$, its space exponent is not bigger than $\rho^*(\chi(v) \cap \chi(p))$. For the time exponent, we note that filling the cache at A_1 (by considering query Q restricted to $\boldsymbol{Z} \cup \{A_1, \ldots, A_k\}$) does not require more time than evaluating the subquery of Q restricted to the bag $\chi(v) \supseteq \boldsymbol{Z} \cup \{A_1, \ldots, A_k\}$. ◀

Recall from Theorem 25 that $\mathcal{PT}$ and $\mathcal{TD}^{\mathcal{GJ}}$ are incomparable. Together with the trivial relationship $\mathcal{PTC} \preceq \mathcal{PT}$, we may conclude from Theorem 26 the strict domination relationships $\mathcal{PTC} \prec \mathcal{PT}$ and $\mathcal{PTC} \prec \mathcal{TD}^{\mathcal{GJ}}$.

The crucial step in the proof of Theorem 26 was to eliminate one node v from the tree decomposition and to cover the interface between this node and its parent by a newly introduced cache. In the next theorem, we essentially proceed in the opposite direction: given a $\mathcal{TD}^{\mathcal{PTC}}$, we want to eliminate a cache at some variable A in the $\mathcal{PTC}$-plan for evaluating the subquery corresponding to a bag at a node v in the tree decomposition. Now, the crucial step is to split the bag at node v into two bags (one with the variables at the subtree rooted at A plus its cache and one with the remaining variables of the bag but again adding the cache at A). The details are worked out in [12]. We thus get:

▶ **Theorem 27.** *The class $\mathcal{TD}^{\mathcal{PT}}$ dominates $\mathcal{TD}^{\mathcal{PTC}}$, i.e., $\mathcal{TD}^{\mathcal{PT}} \preceq \mathcal{TD}^{\mathcal{PTC}}$. Therefore, $\mathcal{TD}^{\mathcal{PT}} \equiv \mathcal{TD}^{\mathcal{PTC}}$ and $\mathcal{TD}^{\mathcal{PT}} \preceq \mathcal{PTC}$.*

5.3 Tree Decompositions and Pseudo-Trees with Caching and Resets

In the previous section, we have seen that combining TDs with $\mathcal{PT}$-plans for the subqueries at the bags dominates the combination of TDs with $\mathcal{GJ}$-plans. On the other hand, replacing $\mathcal{PT}$-plans by $\mathcal{PTC}$-plans for the subqueries at the bags of a TD does not allow for further improvements. The natural next question is as to whether the use of the yet more powerful $\mathcal{PTCR}$-plans in combination with TDs has the potential for further improvements. In this

section, we give a positive answer to this question indirectly. More specifically, we illustrate that $\mathcal{PTCR}$-plans and $\mathcal{TD}^{\mathcal{PT}}$-plans are incomparable. Together with the trivial domination relationship $\mathcal{TD}^{\mathcal{PTCR}} \preceq \mathcal{PTCR}$, we may then conclude that $\mathcal{TD}^{\mathcal{PTCR}} \prec \mathcal{TD}^{\mathcal{PT}}$ holds.

▶ **Theorem 28.** *The classes $\mathcal{PTCR}$ and $\mathcal{TD}^{\mathcal{PT}}$ of query plans are incomparable, i.e., $\mathcal{PTCR} \not\preceq \mathcal{TD}^{\mathcal{PT}}$ 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 Figure 2 and the $\mathcal{PTCR}$ given in Example 18, with space-time exponents $(1, 2)$. Because no edge separates the query, any TD with 2 or more bags uses super-linear 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 Figure 7 and a TD with three bags $ABCDE$, $DEFGH$, and $GHIJK$. Its space-time exponents are $(1, 2)$: the space exponent is clear by considering the intersections by DE and GH between neighboring bags; for the time exponent of the first bag, we could, for instance, use a PT starting with path A-B-C and then branching into D and E, respectively. Then both branches have $\rho^* = 2$. The subqueries at the other two bags are evaluated analogously. On the other hand, it can be shown that the best $\mathcal{PTCR}$ for the query $Q_{\triangle\triangle}()$ has space-time exponents of $(1.5, 2)$ or $(1, 2.5)$. This result, presented in [12], was obtained by a computer-assisted exhaustive search for $\mathcal{PTCR}$-plans. ◀

5.4 Tree Decompositions and Pseudo-Trees with Recursion

We now also briefly look at the combination of TDs with $\mathcal{RPT}$-plans for the subqueries at the bags. It turns out that this may lead to a further improvement compared with $\mathcal{TD}^{\mathcal{PTCR}}$-plans. Actually, even $\mathcal{RPT}$-plans without combining them with TDs strictly dominate $\mathcal{TD}^{\mathcal{PTCR}}$-plans. We first show that $\mathcal{TD}^{\mathcal{PTCR}} \not\preceq \mathcal{RPT}$ holds.

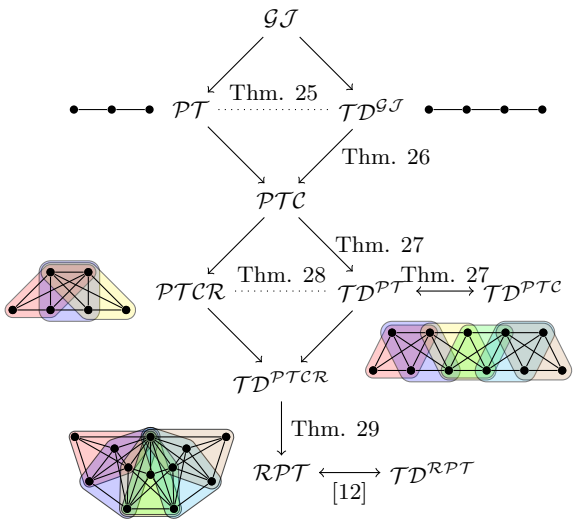
▶ **Theorem 29.** *The class $\mathcal{TD}^{\mathcal{PTCR}}$ does not dominate $\mathcal{RPT}$, i.e., $\mathcal{TD}^{\mathcal{PTCR}} \not\preceq \mathcal{RPT}$.*

Proof Sketch. To prove $\mathcal{TD}^{\mathcal{PTCR}} \not\preceq \mathcal{RPT}$, we revisit the Boolean query Q depicted in Figure 5. As was discussed in Example 24, this query can be evaluated by an $\mathcal{RPT}$-plan with space-time exponent $(1, 2.5)$. However, using a computer-assisted exhaustive search of the plan space $\mathcal{PTCR}$, it was verified in [12] that there is no plan PTCR (P, C) with a space exponent of $s(P, C) \leq 1$ and time exponent of $t(P, C) \leq 5/2$. Since there is also no linear separator in this query, this implies that the same holds for $\mathcal{TD}^{\mathcal{PTCR}}$. ◀

Finally, we mention that the combination of $\mathcal{RPT}$-plans with TDs does not provide further improvements compared with pure $\mathcal{RPT}$-plans, i.e., $\mathcal{RPT} \preceq \mathcal{TD}^{\mathcal{RPT}}$ holds. For details, see [12]. Together with the trivial domination relationship $\mathcal{TD}^{\mathcal{RPT}} \preceq \mathcal{RPT}$, we thus get $\mathcal{RPT} \equiv \mathcal{TD}^{\mathcal{RPT}}$. Moreover, by $\mathcal{TD}^{\mathcal{RPT}} \preceq \mathcal{TD}^{\mathcal{PTCR}}$ and Theorem 29, we get the strict domination relationship $\mathcal{RPT} \prec \mathcal{TD}^{\mathcal{PTCR}}$.

6 Conclusion

We have recalled query plans based on decomposition techniques, which have received a lot of attention in the database community, and query plans based on pseudo-trees and their extensions, which have been primarily studied in the constraint satisfaction community. Moreover, we have shown that techniques from both areas can be fruitfully combined by considering tree decompositions and applying pseudo-tree techniques to the subqueries at the bags of the decompositions. The comparison of the various query plans in terms of space-time exponents yields the domination relationships depicted in Figure 8.



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

Figure 8 [12] Classes of query plans, ordered by their space-time exponents: lower classes have smaller exponents and are better. An arrow $\mathcal{C}_2 \to \mathcal{C}_1$ 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 Definition 2). A missing arrow, i.e. $\mathcal{C}_2 \not\to \mathcal{C}_1$, means $\mathcal{C}_1 \not\preceq \mathcal{C}_2$. In particular, all downward arrows indicate strict domination. The depicted graphs are examples of Boolean queries with binary predicates that separate the classes; the colors represent maximal cliques.

As far future work is concerned, the widest open field is the lack of lower bounds – both in terms of space and time exponents: We have recalled several classes of query plans together with their "canonical algorithms", which lead to upper bounds on the space-time exponents. There are no formal proofs that these space-time exponents are optimal for the respective classes of query plans. More generally, we are lacking concrete space-time tradeoffs for (conjunctive) query evaluation, and there is little to built on in the literature. For example, the set difference and distinct element problems have been proven to have a lower bound $ST = \Omega(n^2)$ [5, 28]. However, these results are too weak to constrain query evaluation, since even acyclic CQs require at least a quadratic space-time product.

References

1 Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017. doi:10.1145/3129246.

2 Isolde Adler, Georg Gottlob, and Martin Grohe. Hypertree width and related hypergraph invariants. *Eur. J. Comb.*, 28(8):2167–2181, 2007. doi:10.1016/j.ejc.2007.04.013.

3 Albert Atserias, Martin Grohe, and Dániel 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. doi:10.1109/FOCS.2008.43.

4 Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013. doi:10.1137/110859440.

5 Paul Beame. A general sequential time-space tradeoff for finding unique elements. *SIAM J. Comput.*, 20(2):270–277, 1991. doi:10.1137/0220017.

6 Liese Bekkers, Frank Neven, Stijn Vansummeren, and Yisu Remy Wang. Instance-optimal acyclic join processing without regret: Engineering the yannakakis algorithm in column stores. *Proc. VLDB Endow.*, 18(8):2413–2426, 2025. doi:10.14778/3742728.3742737.

7 Altan Birler, Alfons Kemper, and Thomas Neumann. Robust join processing with diamond hardened joins. *Proc. VLDB Endow.*, 17(11):3215–3228, 2024. doi:10.14778/3681954.3681995.

8 Angela Bonifati, Wim Martens, and Thomas Timm. An analytical study of large SPARQL query logs. *VLDB J.*, 29(2-3):655–679, 2020. doi:10.1007/s00778-019-00558-9.

9 Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 239(2):211–229, 2000. doi:10.1016/S0304-3975(99)00220-0.

10 Rina Dechter. *Constraint Processing*. Morgan Kaufmann Publishers, 2003.

11 Rina Dechter. *Reasoning with Probabilistic and Deterministic Graphical Models: Exact Algorithms, Second Edition*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019. doi:10.2200/S00893ED2V01Y201901AIM041.

12 Kyle Deeds, Timo Camillo Merkl, Reinhard Pichler, and Dan Suciu. The space-time complexity of sum-product queries. *CoRR*, abs/2509.11920, 2025. Full version of a paper accepted by PODS 2026. doi:10.48550/arXiv.2509.11920.

13 Wolfgang Fischl, Georg Gottlob, Davide Mario Longo, and Reinhard Pichler. Hyperbench: A benchmark and tool for hypergraphs and empirical findings. *ACM J. Exp. Algorithmics*, 26:1.6:1–1.6:40, 2021. doi:10.1145/3440015.

14 Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020. URL: http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf.

15 Eugene C. Freuder and Michael J. Quinn. Taking advantage of stable sets of variables in constraint satisfaction problems. In Aravind 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. URL: http://ijcai.org/Proceedings/85-2/Papers/082.pdf.

16 Georg Gottlob, Matthias Lanzinger, Reinhard Pichler, and Igor Razgon. Complexity analysis of generalized and fractional hypertree decompositions. *J. ACM*, 68(5):38:1–38:50, 2021. doi:10.1145/3457374.

17 Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002. doi:10.1006/jcss.2001.1809.

18 Georg Gottlob, Zoltán Miklós, and Thomas Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *J. ACM*, 56(6):30:1–30:32, 2009. doi:10.1145/1568318.1568320.

19 Johnnie Gray and Stefanos Kourtis. Hyper-optimized tensor network contraction. *Quantum*, 5:410, 2021. doi:10.22331/Q-2021-03-15-410.

20 Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, 11(1):4:1–4:20, 2014. doi:10.1145/2636918.

21 Sunwoong Joo, Attila Dusnoki, Martyn Bliss, Ben Duckworth, Nicolas Scotto Di Perto, Markó Fabó, Gábor Lóki, Dániel Vince, Ákos Kiss, and Cheul-hee Hahm. A memory-aware performance optimization of tensor programs for embedded devices. In *2020 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia)*, pages 1–4. IEEE, 2020.

22 Manolis Katsaragakis, Lazaros Papadopoulos, Mario Konijnenburg, Francky Catthoor, and Dimitrios Soudris. Memory footprint optimization techniques for machine learning applications in embedded systems. In *IEEE International Symposium on Circuits and Systems, ISCAS 2020, Sevilla, Spain, October 10-21, 2020*, pages 1–4. IEEE, 2020. doi:10.1109/ISCAS45731.2020.9181038.

23 Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: questions asked frequently. *CoRR*, abs/1504.04044, 2015. arXiv:1504.04044.

24 Phokion G. Kolaitis and Moshe Y. Vardi. Conjunctive-query containment and constraint satisfaction. *J. Comput. Syst. Sci.*, 61(2):302–332, 2000. doi:10.1006/JCSS.2000.1713.

25 Matthias Lanzinger, Cem Okulmus, Reinhard Pichler, Alexander Selzer, and Georg Gottlob. Soft and constrained hypertree width. *Proc. ACM Manag. Data*, 3(2):114:1–114:25, 2025. doi:10.1145/3725251.

26 Matthias Lanzinger, Reinhard Pichler, and Alexander Selzer. Avoiding materialisation for guarded aggregate queries. *Proc. VLDB Endow.*, 18(5):1398–1411, 2025. URL: https://www.vldb.org/pvldb/vol18/p1398-selzer.pdf.

27 Stanislav Malyshev, Markus Krötzsch, Larry González, Julius Gonsior, and Adrian Bielefeldt. Getting the most out of wikidata: Semantic technology usage in wikipedia's knowledge graph. In Denny Vrandecic, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl, editors, *The Semantic Web - ISWC 2018 - 17th International Semantic Web Conference, Monterey, CA, USA, October 8-12, 2018, Proceedings, Part II*, volume 11137 of *Lecture Notes in Computer Science*, pages 376–394. Springer, 2018. doi:10.1007/978-3-030-00668-6_23.

28 Dylan M. McKay and Richard Ryan Williams. Quadratic time-space lower bounds for computing natural functions with a random oracle. In Avrim 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. doi:10.4230/LIPIcs.ITCS.2019.56.

29 Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew strikes back: new developments in the theory of join algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013. doi:10.1145/2590989.2590991.

30 Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: breaking the GPU memory wall for extreme scale deep learning. In Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin, editors, *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, page 59. ACM, 2021. doi:10.1145/3458817.3476205.

31 Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986. doi:10.1016/0196-6774(86)90023-4.

32 Francesco Scarcello, Gianluigi Greco, and Nicola Leone. Weighted hypertree decompositions and optimal query plans. *J. Comput. Syst. Sci.*, 73(3):475–506, 2007. doi:10.1016/J.JCSS.2006.10.010.

33 Kian-Lee Tan, Qingchao Cai, Beng Chin Ooi, Weng-Fai Wong, Chang Yao, and Hao Zhang. In-memory databases: Challenges and opportunities from software and hardware perspectives. *SIGMOD Rec.*, 44(2):35–40, 2015. doi:10.1145/2814710.2814717.

34 Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106. OpenProceedings.org, 2014. doi:10.5441/002/ICDT.2014.13.

35 Qichen Wang, Bingnan Chen, Binyang Dai, Ke Yi, Feifei Li, and Liang Lin. Yannakakis+: Practical acyclic query evaluation with theoretical guarantees. *Proc. ACM Manag. Data*, 3(3):235:1–235:28, 2025. doi:10.1145/3725423.

36 Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Proceedings VLDB*, pages 82–94. VLDB, 1981.