Hypertree Decompositions: Questions and Answers

Georg Gottlob
Computer Science Dep.
University of Oxford
gottlob@cs.ox.ac.uk

Gianluigi Greco
DEMACS
University of Calabria
ggreco@mat.unical.it

Nicola Leone
DEMACS
University of Calabria
leone@mat.unical.it

Francesco Scarcello
DIMES
University of Calabria
scarcello@dimes.unical.it

ABSTRACT

In the database context, the hypertree decomposition method is used for query optimization, whereby conjunctive queries having a low degree of cyclicity can be recognized and decomposed automatically, and efficiently evaluated. Hypertree decompositions were introduced at ACM PODS 1999. The present paper reviews –in form of questions and answers– the main relevant concepts and algorithms and surveys selected related work including applications and test results.

Keywords

Conjunctive queries; hypergraphs; query evaluation; tractability; acyclic queries; hypertree width; decompositions

1. INTRODUCTION

1.1 What Is This All About?

This paper is about *hypertree decompositions* [74], a hypergraph-based method for decomposing conjunctive queries (or, equivalently, their associated hypergraphs) into tree-structures called *hypertrees*, in order to process them more efficiently. Each hypertree decomposition of a query is associated with a width. The smallest width over all such decompositions is the query's *hypertree width*. Queries of bounded hypertree width can be answered efficiently [74] in combined complexity. In particular, while Boolean conjunctive queries are NP-complete to answer in general [30], in the case of bounded hypertree width, they can be processed in polynomial time. Moreover, conjunctive queries with output, in case of bounded hypertree width, can be processed in time polynomial in the size of the input plus the size of the output. Queries having bounded hypertree width are also highly parallelizable. This was already noted in [72, 68] and has gained renewed interest in the context of MapReduce [43] and more advanced models of parallel query processing [7].

PODS'16, June 26-July 01, 2016, San Francisco, CA, USA

DOI: http://dx.doi.org/10.1145/2902251.2902309

The notion of bounded hypertree width generalizes the well-known concept of *query acyclicity*. The hypertree width of a query is a good measure of its degree of cyclicity. In particular, the class of all conjunctive queries (CQs) of hypertree width 1 coincides with the class of acyclic CQs. Hypertree decompositions, in a precise sense, lift the great benefits of acyclic queries to much larger and more general classes of queries which, in addition to acyclic queries, contain queries that are, so to say, *mildly cyclic*. A large number of realistic queries are of bounded hypertree width, as we shall discuss in Sections 3 and 9. Such queries can be answered efficiently, by using specific algorithms that exploit hypertree decompositions. However, the same queries would often require time exponential in their length if evaluated by classical methods. Hypertree decompositions are thus a powerful query optimization method with a huge potential for practical applications. Moreover, the problem of conjunctive query answering is equivalent to the query containment problem and, in AI, to the *constraint satisfaction problem (CSPs)* [44]. Thus, the algorithms and favorable complexity results for hypertree decompositions presented in this paper also apply to these problems.

1.2 What Is so Smart about Acyclic Queries?

Query acyclicity, which will be discussed in detail in Section 2, is one of the most fundamental and most fascinating concepts in database theory. It was introduced in [18] in the context of *acyclic database schemes* and, independently, as the *tree-query property* in [59]. In [19, 58] it was shown that these notions coincide. A query Q is acyclic if its associated hypergraph $\mathcal{H}_Q$ has a join tree. This is an appropriate arrangement of the (hyperedges associated with the) query atoms in form of a tree—see Section 2 for details. The hypergraph $\mathcal{H}_Q$ associated with a Boolean conjunctive query Q has as nodes its variables and has, for each query atom, a hyperedge consisting of the set of all variables appearing in that atom. Note that, unlike for graphs, for hypergraphs there are various different definitions of acyclicity. The join-tree based definition used in the present paper coincides with α-acyclicity as defined in [50]. This type of hypergraph acyclicity is the most general acyclicity notion and is far more sophisticated than graph acyclicity. To appreciate this, consider Figure 1, which shows two acyclic hypergraphs and their associated join trees. The hypergraph on the left side is intuitively acyclic, less so the one on the right side. In fact, the latter contains an obvious cycle over the nodes A, B, C, and D, which is not covered by any single hyperedge. Still, because this cycle is appropriately covered by two consecutive edges of the join tree, the hypergraph is acyclic. Note that, unlike for graphs, an acyclic hypergraph may contain a cyclic one as a sub-hypergraph.

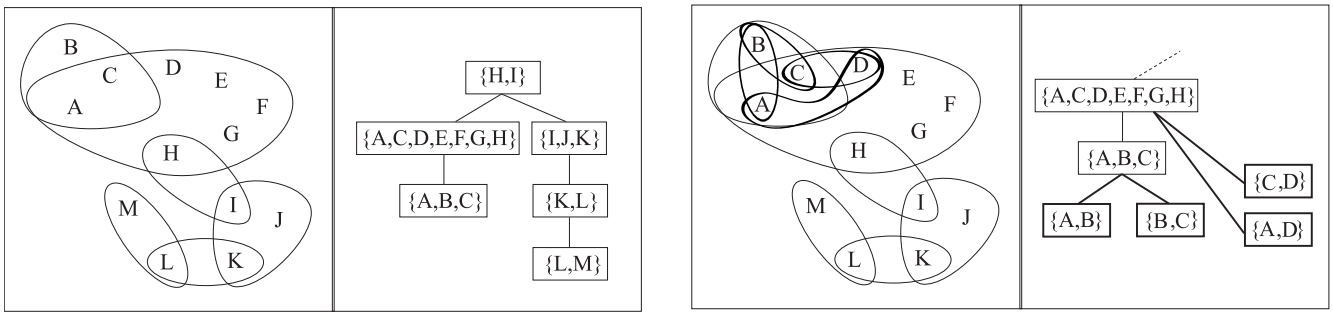

Figure 1: A hypergraph $\mathcal{H}_{Q_1}$ and a join tree for it (left), plus a hypergraph $\mathcal{H}_{Q_{0\wedge 1}}$ and a join tree for it (right).

This is illustrated yet more drastically by the following example, which also explains how recognizing acyclicity helps processing a query in polynomial time.

EXAMPLE 1.1. Let CLIQUE_r be the query expressing that a graph with edge relation E has an r-clique, i.e.,

$$\text{CLIQUE}_r \equiv \bigwedge_{1\leq i<j\leq r} E(x_i, x_j) \wedge E(x_j, x_i).$$

Let, moreover, Q_r be the query

$$\text{CLIQUE}_r \wedge R(x_1, \ldots, x_r),$$

and define the infinite class $\mathcal{C}$ of queries by $\mathcal{C} = \{Q_r | r \geq 1\}$. Each query Q_r in C is acyclic, and can thus be answered very efficiently. In fact, to answer Q_r over some database DB, we just need to start by matching the $R(x_1, \ldots, x_r)$ atom to the polynomially many tuples of the R relation of DB one by one. Each such match binds all variables, and thus for each such match it suffices to check in linear time whether all other instantiated query atoms are in the database.

However, if we used a more naïve evaluation strategy that processes the query by performing joins from left to right, we would first solve the NP-hard CLIQUE_r query and only then evaluate the $R(x_1, \ldots, x_r)$ atom. Unless P=NP, this would require an exponential effort (in combined complexity) in the worst case. ◁

Boolean acyclic conjunctive queries (ACQs) with m atoms, where r is the size of the largest database relation relevant to the query, can be answered in time $O(m \cdot r \cdot \log r)$, while non-Boolean ACQs in time $O(m \cdot N \cdot \log N))$, where N is the size of the output plus r. This is achieved by first automatically recognizing whether a query is acyclic, which is feasible in linear time [133], and if so, by processing the query efficiently using a smart algorithm, such as Yannakakis' algorithm [137], which reduces the relevant database relations via semi-joins along the edges of the query join tree in such a way that no remaining tuple is superfluous (see Section 2).

1.3 How to Generalize Query-Acyclicity?

A significant proportion of the conjunctive queries (and the constraint satisfaction problems) arising in practice are not properly acyclic, but are in some sense nearly acyclic. Efforts have therefore been made since the 1990s to weaken the notion of acyclicity by defining appropriate concepts of near acyclicity and by finding query evaluation algorithms that are efficient for nearly acyclic queries. In this context, the degree of acyclicity of a hypergraph (or of a corresponding query) is mostly referred to as its *width*.

In defining a generalization of acyclicity, one seeks to ensure the following fundamental three conditions to hold for each fixed constant $k \geq 1$:

1. **Generalization of Acyclicity:** Queries of width k include the acyclic ones.
2. **Tractable Recognizability:** Queries of width k can be recognized in polynomial time.
3. **Tractable Query-Answering:** Queries of width k can be answered in polynomial time (in combined complexity).

In Condition 3, in case of non-Boolean queries, the polynomial-time bound is considered with respect to the combined size of the input and the output (indeed, in general, we may get exponentially many tuples in the output).

In graph theory, there is an appropriate and well-known generalization of graph acyclicity: *bounded treewidth* [124, 123]. This notion is based on tree decompositions as explained in detail in Section 3. Tree decompositions can straightforwardly be defined for hypergraphs, and thus for queries. They have been considered for conjunctive query answering (or, equivalently, CSP-solving) in [46, 31, 108]. Briefly, a tree decomposition of a hypergraph $\mathcal{H}$ consists of a tree T, whose vertices are each associated with a so called *bag*, i.e., a set of nodes of $\mathcal{H}$, such that each hyperedge is covered by some bag, and such that for each node v of $\mathcal{H}$ all vertices of the decomposition tree whose bag contains v induce a connected subtree of T. The *width* of a decomposition is $s - 1$, where s is the cardinality of the largest bag. The *treewidth* $tw(\mathcal{H})$ of $\mathcal{H}$ is the minimum treewidth among all tree decompositions of $\mathcal{H}$. The treewidth $tw(Q)$ of a query Q is defined to be the treewidth $tw(\mathcal{H}_Q)$ of its associated hypergraph $\mathcal{H}_Q$. A similar convention will be adopted for other notions of width introduced below.

Conjunctive queries of treewidth k can be answered on a database DB in time $O(m' \cdot D^{k+1} \cdot \log D)$, where m' is the number of vertices of the decomposition tree T, and D is the number of distinct values occurring in DB. So, the tree decomposition method enjoys our criterion of Tractable Query-Answering (Condition 3 above), because we can always find a tree decomposition of width k, whose number of vertices m' is at most the number of variables occurring in the query. It is also well-known that, for fixed constant k, determining whether a hypergraph has treewidth k is feasible in linear time [23], therefore tree decompositions also enjoy Tractable Recognizability (Condition 2). There is, however, a significant drawback of the tree decomposition method: it does not generalize hypergraph acyclicity, and therefore does not fulfill Condition 1. For instance, for each integer r, the acyclic query Q_r of Example 1.1 has treewidth $r - 1$. Therefore, the class $\mathcal{C}$ of all these queries has unbounded treewidth.

Another method introduced in an attempt to capture the degree of acyclicity of a query is the method of *biconnected components* [52]. In this method, the degree of acyclicity of a query Q is defined as the number of nodes of the largest biconnected component of $\mathcal{H}_Q$.

This method, too, satisfies conditions 2 and 3, but not Condition 1, as it does not generalize acyclicity. According to this method, for each integer r, the acyclic query Q_r of Example 1.1 has degree of acyclicity r. Further methods that do not suitably generalize acyclicity are reviewed and compared in [70].

A significantly more appropriate, but still not fully satisfactory approach to the generalization of query acyclicity is the notion of *query width* [31], which is based on the concept of *query decompositions*. Essentially, a query decomposition of width k of a query Q is a tree decomposition of $\mathcal{H}_Q$ each of whose bags coincides with the union of k or fewer hyperedges from $\mathcal{H}_Q$, and where at least one bag is the union of precisely k such hyperedges—as shown in [72], this definition is equivalent to the original definition in [31]. The *query width (qw)* of Q is the minimum width among all possible query decompositions of Q. Bounded query width generalizes query acyclicity, as each join-tree is a query decomposition of query width 1. Thus, Condition 1 is satisfied. Queries of bounded query width can be answered in polynomial time by the same algorithms that are also used for hypertree decompositions (which we will discuss in Section 4), thus also Condition 3 is satisfied. Unfortunately, however, queries of bounded query width are hard to recognize. In fact, as shown in [74], deciding for a query Q whether $qw(Q) = k$ for fixed constants $k > 3$ is NP-complete. Thus, unless P=NP, Condition 2 is not satisfied. In addition, the fact that the bags need to coincide *exactly* to a union of $\leq k$ hyperedges is restrictive and gives rise to a higher than necessary width [74].

To redress the latter issue, it is sufficient to merely require that each bag of the tree decomposition of $\mathcal{H}_Q$ be *covered* by a union of k hyperedges (rather than requiring it coincides with their union). This generalization gives rise to the concept of *generalized hypertree decomposition (GHD)* and to the associated notion of *generalized hypertree width (ghw)*. Thus, a GHD of a hypergraph (or query) is a tree decomposition together with a specified covering of its bags by hyperedges. The ghw of a particular GHD is the maximum number of hyperedges used to cover a bag, and the generalized hypertree width of the original hypergraph (or query) is the minimum generalized hypertree width over all GHDs for this hypergraph. GHDs properly generalize query decompositions (query decompositions are GHDs but not vice-versa), and thus also acyclic queries. Hence, Condition 1 is satisfied. Queries of bounded ghw can be answered in polynomial time [74], thus Condition 3 is satisfied. However, as shown in [76], for fixed constants $k \geq 3$, it is NP-hard to decide whether for a query Q, $ghw(\mathcal{H}_Q) = k$. Thus, unless P=NP, Condition 2 is still not satisfied.

To achieve tractability, an additional restriction, called the *Descendant Condition* or also the *Special Condition*, was added to the definition of GHD. This condition, which will be defined and explained in Section 3.2, while attempting to construct a width-k GHD, reduces in a rather natural way the choice of possible child-bags of an already computed bag to a polynomial number of alternatives (each of which can be represented in logarithmic space). Based on this, a width-k decomposition of a hypergraph $\mathcal{H}$ can be computed via an alternating logspace procedure and thus in deterministic polynomial time (see Section 3.3). GHDs that satisfy the Descendant Condition are called *hypertree decompositions (HDs)*, and the associated width is the *hypertree width*, denoted by $hw(\mathcal{H})$. Hypertree decompositions were originally defined in [74], where also their fundamental properties were studied. In [70] they were carefully compared to other types of decompositions. There has been since then much further research related to (G)HDs, as well as several implementations and applications. Much of this work is reviewed in the sequel of the present paper.

1.4 What Are the Main Advantages of HDs?

We give here only a very short itemized summary of the advantages of HDs here. For more details, see Sections 3-6.

- For a fixed constant k, it can be checked in polynomial time whether a conjunctive query Q is of hypertree width $\leq k$, and if so, a HD of width $hw(\mathcal{H}_Q)$ can be computed in polynomial time. The best upper bound currently known for this is $O(m^{2k}v^2)$ time, where m and v are the number of atoms and the number of variables in Q, respectively [69]. The decision and computation problems are, moreover, at a very low level of structural complexity and are highly parallelizable problems in LOGCFL, a low class within NC^2. These results are discussed in Section 3.

- A Boolean conjunctive query Q having (generalized) hypertree width k can be answered in time $O(v \cdot r^k \cdot \log r)$ where r is the size of the largest database relation mentioned by the query. For queries with output (whose results may consist of exponentially many tuples) an output-polynomial upper bound is $O(v \cdot (r^k + s) \cdot \log(r + s))$, where s is the number of output tuples. In particular, for each query of bounded hw, we can generate in polynomial time an efficient query plan for a relational machine. This and further improved bounds are discussed in Section 4.

- Generalized hypertree decompositions (and thus, in particular, HDs) can be used for generating efficient query plans not only for standard query processing engines, but also for systems using multiway joins or for systems using parallel processing at various levels of granularity (see Section 4.1). Moreover, just as for acyclic queries, evaluating queries of bounded hw is complete for the parallel class LOGCFL [72].

- Generalized hypertree decompositions (and thus also HDs) cannot just be used for plain conjunctive query answering, but, as shown in Section 5, they are also usable for *top-k* queries, and queries with aggregation operators, such as COUNT.

- Hypertree width is not very far apart from the tighter acyclicity-measure of ghw. In fact, in [6] it was proven that for each hypergraph $\mathcal{H}$, $ghw(\mathcal{H}) \leq hw(\mathcal{H}) \leq 3 \cdot ghw(\mathcal{H}) + 1$, which implies that a class of queries has bounded ghw iff it has bounded hw.

1.5 Do HDs Have Any Disadvantage?

There is a small price we have to pay for the significant improvement HDs offer over other decompositions. While HDs of bounded width k, if they exist, can be computed in polynomial time, the exponent of the polynomial in the upper bound involves k as a factor. It is deemed very unlikely that we can get rid of the factor k in the exponent, because the problem of deciding whether a query has hypertree width k was shown to be fixed-parameter intractable [66].

In the database context this is not really a problem. First, as we will see below, in all benchmarks analyzed so far, the hypertree width of queries is extremely low. For an effective practical query optimization, it actually suffices to detect queries of hw 2 or 3. Second, while conjunctive queries can become quite large in practice, they hardly reach sizes of over, say, 50 atoms. With these figures, even exact HD algorithms can be used very effectively. Finally, note that in database applications it is very often the case that the same query is repeated extremely frequently over constantly changing data. It is then worthwhile first spending a little computational effort so as to get a really good decomposition and being rewarded a myriad of times afterwards by running a highly optimized query.

The situation is more problematic in the context of large constraint satisfaction problems. Such problems may have hundreds or

even thousands of constraint atoms and we may want to look for decompositions of hypertree width around 20. For such problems, the current exact HD algorithms often cause memory overflow or take very long time to complete. However, more efficient algorithms have been developed that use heuristics that do not necessarily aim at an optimal HD (see Section 3.4). Moreover, for such problems, if the constraint *relations* are relatively small, then methods of computing the answer of a query by implicitly exploiting an assumed small hw, but without computing the decomposition explicitly, may be appropriate. Such methods are discussed in Section 4.2.

1.6 Do HDs Matter in Practice?

Yes, definitely. As discussed in Section 6, and as noted by many authors, hypertree decompositions are a significant and very beneficial query optimization technique. The low hypertree width of all queries in the database benchmarks we and others have analyzed, and the fact that a substantial part of the queries are not plainly acyclic supports this. HDs have been used profitably in experimental systems [1, 7, 10, 8], and we believe the time is ripe for using them in for query optimization in commercial DBMSs too.

1.7 Which Applications Are HDs Suited For?

As already discussed, hypertree decompositions are, in the first place, useful for database query optimization, not only in the classical DBMS setting, but also in the context of more modern query processing using multiway joins, and for parallel and distributed query processing. In each of this contexts, HDs can be used for generating smart and efficient query plans.

Another large application area is constraint satisfaction (see Section 7.1). The constraint satisfaction problem (CSP) has been recognized to be equivalent to conjunctive query evaluation—see [92, 108, 25]. In particular, Kolaitis and Vardi [108] pointed out that both problems are, in turn, equivalent to the *homomorphism* problem between finite relational structures. However, as already noted, practical CSPs may differ from conjunctive query answering in that they have many more constraint atoms but often (though not always) quite small relations. Therefore, with CSPs, approximate algorithms using heuristics may work better than exact decomposition algorithms, for some applications.

Other applications, which we discuss in Section 7, relate to computing Nash equilibria of games, and to the problem of winner determination in combinatorial auctions. More in general, any application that is any easier on restrictions characterized by acyclic hypergraph structures may find useful to exploit (G)HDs.

1.8 Can the Concept of HD Be Improved?

It is shown in [76] that if, in addition to the original hyperedges of a hypergraph $\mathcal{H}$, we also use some proper subsets of these hyperedges for covering the bags of a tree decomposition of $\mathcal{H}$, then we still obtain a correct decomposition, whose associated width may be smaller than $hw(\mathcal{H})$. In particular, if *all* subsets of the hyperedges of H are allowed, then the obtained width coincides with $ghw(H)$. However, this causes an exponential blow-up, as there is an exponential number of hyperedge-subsets. If, on the other hand, we use a polynomial-time function that associates a *polynomial number* of such subsets with each hypergraph, then we may obtain an improvement of HD-decompositions yielding in some cases a smaller width, while still fulfilling Conditions 1-3. This is precisely the idea of subset-based decompositions as defined in [76]. Specific subset-based decompositions are the *component decompositions* defined in [76] and the *spread-cut* decompositions defined in [37], while an interesting way of computing a subset-based decomposition *dynamically* provides the new notion of *greedy hypertree decomposition* [82, 88]. In essence, however, subset-based decompositions are generalizations of HDs that are mainly based on the same principles.

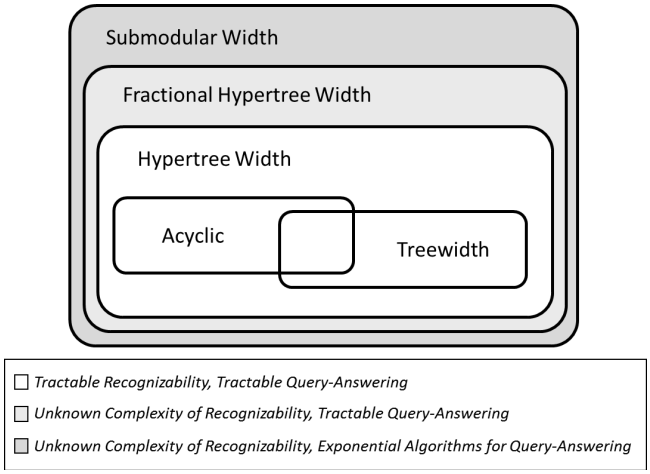

Figure 2: Relationships among hypertree decompositions and more general decomposition methods.

A more radical approach to improve HDs and even GHDs, was taken by Grohe and Marx [90], who defined *fractional hypertree decompositions (FHDs)*, giving rise to the *fractional hypertree width* $fhw(\mathcal{H})$ of a hypergraph $\mathcal{H}$ (or of a query). While hypertree decompositions exploit low cyclicity, which in a sense means a loose coherence of the variables in a query, FHDs manage to exploit, *in addition*, certain types of strong coherence between sets of variables. Moreover, even for hypergraphs $\mathcal{H}$ of low cyclicity, FHDs may yield width $fhw(\mathcal{H}) < ghw(\mathcal{H})$. The current problem with FHDs is that it is unknown whether $fhw(\mathcal{H}) \leq k$, for a fixed k, is recognizable in polynomial time. Thus, Condition 2 is not known to be satisfied. An approximate decomposition whose width is bounded by a cubic function of the fractional hypertree width can be computed in polynomial time [113]. A yet more general width-concept is *submodular width* [115]. Unfortunately, recognizing queries of bounded submodular width is not known to be tractable. Moreover, having bounded submodular width (smw) does (unfortunately) not guarantee a polynomial-time combined complexity as for GHDs, but just fixed-parameter tractability where the query hypergraph is used as a parameter. This means that we have a polynomial-time data complexity of the form $O(f_1(\mathcal{H}_Q) \cdot |\mathrm{DB}|^{f_2(\mathrm{smw})})$, where f_1 is an exponential function of the query size. Unlike the previous methods, the decompositions used to answer the query depend not only on the hypergraph $\mathcal{H}_Q$, but on the actual database DB, too. The relationships among these methods are illustrated in Figure 2—see also Section 8.

In summary, to the best of our knowledge, except for improvements such as subset-based decompositions that are essentially HD-based, there is no other improvement of hypertree decompositions that fulfills the three fundamental conditions of (1) Generalization of Acyclicity, (2) Tractable Recognizability, and (3) Tractable Query-Answering.

2. WHAT IS HYPERGRAPH ACYCLICITY?

We will adopt the standard convention of identifying a relational database instance with a logical theory consisting of ground facts [3, 134]. Accordingly, a tuple $\langle a_1, \ldots, a_k \rangle$ of constants belonging to relation r of a database DB is simply denoted by $r(a_1, \ldots, a_k) \in \mathrm{DB}$. The set of all constants in DB is denoted by $dom(\mathrm{DB})$.

2.1 Conjunctive Queries and Hypergraphs

A *conjunctive query* Q is a rule of the form

$$ans(\mathbf{u}) \leftarrow r_1(\mathbf{u_1}) \wedge ... \wedge r_m(\mathbf{u_m}),$$

where $r_1, ..., r_m$ are relation symbols, and $\mathbf{u}, \mathbf{u_1}, ..., \mathbf{u_m}$ are lists of terms (i.e., variables or constants). We denote by $atoms(Q)$ the set of all atoms in the right-hand side, i.e., $atoms(Q) = \{r_1(\mathbf{u_1}), ..., r_m(\mathbf{u_m})\}$. For a set A of atoms, $vars(A)$ is the set of all variables in A, and $vars(Q)$ is used for short in place of $vars(atoms(Q))$. If $\mathbf{u}$ is empty, then the query Q is said *Boolean*.

The *answer* of Q on a database DB, denoted by Q^{DB}, consists of the set of all ground facts $ans(\theta(\mathbf{u}))$ where $\theta : vars(Q) \mapsto dom(\text{DB})$ is a substitution such that:

$$\theta'(r_i(\mathbf{u_i})) \in \text{DB holds,}^{[1]} \text{ for each } i \in \{1, ..., m\},$$

where $\theta'(t) = \theta(t)$ if $t \in vars(Q)$ and $\theta'(t) = t$ otherwise (i.e., if the term t is a constant).

There is a very natural way to associate a hypergraph $\mathcal{H}_Q = (N, H)$ with a conjunctive query Q: The set N of nodes consists of all variables in Q; for each atom in Q, the set H of hyperedges contains a hyperedge including all its variables; and no other hyperedge is in H. Note that the cardinality of H can be smaller than the cardinality of $atoms(Q)$, because two query atoms having exactly the same set of variables in their arguments give rise to only one edge in H. In the following the set of the nodes and the set of the hyperedges of any hypergraph $\mathcal{H}$ will be shortly denoted by $nodes(\mathcal{H})$ and $edges(\mathcal{H})$, respectively.

EXAMPLE 2.1. Consider the following conjunctive query Q_0 over the set $\{A, B, C, D\}$ of variables:

$$ans() \leftarrow c_1(A, B) \wedge c_2(B, C) \wedge c_3(C, D) \wedge c_4(D, A).$$

This is a Boolean query whose associated hypergraph $\mathcal{H}_{Q_0}$ is actually a graph, because each query atom is defined over precisely two variables. Indeed, $nodes(\mathcal{H}_{Q_0}) = \{A, B, C, D\}$ and $edges(\mathcal{H}_{Q_0}) = \{\{A, B\}, \{B, C\}, \{C, D\}, \{D, A\}\}$. ◁

A hypergraph $\mathcal{H}$ is *acyclic* iff it has a join tree [20]. A *join tree* $JT(\mathcal{H})$ for a hypergraph $\mathcal{H}$ is a tree whose vertices are the hyperedges of $\mathcal{H}$ such that, whenever the same node $X \in V$ occurs in two hyperedges h_1 and h_2 of $\mathcal{H}$, then X occurs in each vertex on the unique path linking h_1 and h_2 in $JT(\mathcal{H})$. This is equivalent to requiring that the decomposition vertices where X appears induce a connected subtree of T (connectedness condition). Note that this notion of hypergraph acyclicity, known as α-acyclicity, is the most general one [50]. Moreover, it is efficiently recognizable: Deciding whether a hypergraph is acyclic is feasible in linear time [133], and also in deterministic logspace. This latter property follows from the fact that hypergraph acyclicity belongs to symmetric logspace [72], and that this is equal to deterministic logspace [122].

EXAMPLE 2.2. The (hyper)graph $\mathcal{H}_{Q_0}$ associated with the query in Example 2.1 contains a cycle over A,B, C, and D. In fact, it is not possible to build a join tree for it.

Consider, then, the following conjunctive query Q_1 over the set $\{A, ..., M\}$ of variables:

$$ans() \leftarrow r_1(A, B, C) \wedge r_2(A, C, D, E, F, G, H) \wedge r_3(H, I) \wedge r_4(I, J, K) \wedge r_5(K, L) \wedge r_6(L, M).$$

The hypergraph $\mathcal{H}_{Q_1}$ associated with the query Q_1 is the one discussed in the introduction (and reported in the left part of Figure 1). The hypergraph is acyclic as witnessed by the join tree depicted for it. More interestingly, if we consider the conjunctive query $Q_{0\wedge1}$ such that $atoms(Q_{0\wedge1}) = atoms(Q_0) \cup atoms(Q_1)$, i.e., the query obtained by joining all the atoms occurring in Q_0 and Q_1, then we obtain a hypergraph $\mathcal{H}_{Q_{0\wedge1}}$ that is acyclic. Indeed, the cycle in Q_0, due to the atoms $c_1(A, B)$, $c_2(B, C)$, $c_3(C, D)$, and $c_4(D, A)$ is now "absorbed" by two hyperedges of Q_1. In fact, a join tree for $\mathcal{H}_{Q_{0\wedge1}}$ can be easily obtained from the one shown on the left in Figure 1, by just appending the fresh vertices $\{A, B\}$, $\{B, C\}$, $\{C, D\}$, and $\{D, A\}$ as children of the vertex $\{A, C, D, E, F, G, H\}$ and $\{A, B, C\}$, as illustrated on the right of the same figure.

Instead, for the query Q_2 obtained from Q_1 by adding the atom $r_7(B, M)$ to Q_1, the associated hypergraph $\mathcal{H}_{Q_2}$ is not acyclic. This hypergraph is shown on the left of Figure 4. Note that there is no way to build a join tree for it. In particular, every attempt to add a vertex for the hyperedge $\{B, M\}$ to the join tree in Figure 1 does not satisfy the connectedness condition for B or M. ◁

[1] As usual, if $\mathbf{u_i}$ is the list of terms $t_1, ..., t_m$, then we denote $r_i(\theta'(t_1), ..., \theta'(t_m))$ by $\theta'(r_i(t_1, ..., t_m))$.

2.2 Basic Properties

Acyclic queries, i.e., queries whose associated hypergraphs are acyclic, can be *efficiently answered*. More precisely, using Yannakakis' algorithm [137], Boolean acyclic queries can be evaluated by processing any of their join trees bottom-up, by performing upward semijoins between the relations associated with the query atoms, thus keeping the size of the intermediate relations small. At the end, if the relation associated with the root of the join tree is not empty, then the query evaluates to true.

For non-Boolean queries, after the bottom-up step we have described above, one can perform the opposite top-down step by filtering each child vertex from those tuples that do not match with its parent tuples. The filtered database enjoys the *global consistency* property: every tuple in every relation participates in some solution. By exploiting this property, all solutions can be computed with a backtrack-free procedure (i.e., with backtracks used to look for further solutions, and never caused by wrong choices).

Combining the fact that acyclic queries can be efficiently answered with the fact that acyclicity is efficiently recognizable, such queries identify a so-called (accessible) "island of tractability" for the query answering problem [107]—and for equivalent problems, such as conjunctive query containment, constraint satisfaction problems, and so on [72].

3. HOW TO GENERALIZE ACYCLICITY?

While acyclic queries would be very desirable due to the properties discussed in the previous section, queries arising from real applications are often precisely of this kind. As an example, consider the following query Q_3 extracted from the well-known TPC-H benchmark (*www.tpc.org/tpch*)—the original specification was given in the SQL language, and for further details we refer to [56]:

$$\begin{aligned} &ans(Name, ExtendedPrice, Discount) \leftarrow \\ &\quad customer(CustKey, NationKey) \wedge orders(OrdKey, NationKey) \wedge \\ &\quad lineitem(SuppKey, OrdKey, ExtendedPrice, Discount) \wedge \\ &\quad supplier(SuppKey, NationKey) \wedge region(RegionKey) \wedge \\ &\quad nation(Name, NationKey, RegionKey). \end{aligned}$$

Its associated hypergraph $\mathcal{H}_{Q_3}$, shown in Figure 3, is not acyclic. However, the hypergraph is not very intricate and its "degree of cyclicity" appears to be rather limited. Accordingly, one might naturally hope that, for queries of this kind, the nice properties of acyclic queries can be preserved. Motivated by such observations, significant efforts have been spent to investigate hypergraph properties that are best suited for identifying and efficiently processing nearly-acyclic queries. This led to the definition of a number of so-called *(purely) structural decomposition methods*.

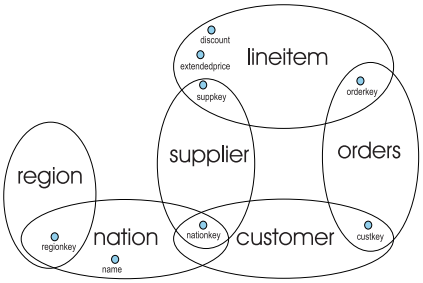

Figure 3: Hypergraph $\mathcal{H}_{Q_3}$.

3.1 Tree Decompositions

The notion of tree decomposition [123] represents a significant success story in Computer Science (see, e.g., [65]). The associated notion of *treewidth* was meant to provide a measure of the degree of cyclicity in graphs and hypergraphs.[2]

Formally, a *tree decomposition* [123] of a hypergraph $\mathcal{H}$ is a pair $\langle T, \chi \rangle$, where $T = (V, F)$ is a tree, and χ is a labeling function assigning to each vertex $p \in V$ a set of vertices $\chi(p) \subseteq nodes(\mathcal{H})$, such that the following three conditions are satisfied: (1) for each node b of $\mathcal{H}$, there exists $p \in V$ such that $b \in \chi(p)$; (2) for each hyperedge $h \in edges(\mathcal{H})$, there exists $p \in V$ such that $h \subseteq \chi(p)$; and (3) for each node b in $nodes(\mathcal{H})$, the set $\{p \in V \mid b \in \chi(p)\}$ induces a connected subtree of T. The *width* of $\langle T, \chi \rangle$ is the number $\max_{p \in V}(|\chi(p)| - 1)$. The *treewidth* of $\mathcal{H}$, denoted by $tw(\mathcal{H})$, is the minimum width over all its tree decompositions.

Recall from the introduction that the notion of treewidth does not generalize hypergraph acyclicity. For instance, consider the query Q_1 and its associated hypergraph shown in Figure 1. We have already observed that the hypergraph is acyclic (see Example 2.2), but the treewidth of this acyclic hypergraph is 6—note that there is a hyperedge defined over the variables $\{A, C, D, E, F, G, H\}$.

On the other hand, treewidth is a true generalization of graph acyclicity. Indeed, a graph is acyclic if, and only if, it has treewidth 1. In particular, note that the treewidth of a hypergraph $\mathcal{H}$ coincides with the treewidth of its *primal graph*, which is defined over the same set $nodes(\mathcal{H})$ of nodes of $\mathcal{H}$ and contains an edge for each pair of nodes included in some hyperedge of $edges(\mathcal{H})$. In fact, the notion can be applied to other graph-based representations of hypergraphs, including *dual* and *incidence* graphs representations. These approaches are contrasted with each other in [81].

Determining the treewidth of a (hyper)graph is NP-hard. However, for each fixed natural number k, checking whether its treewidth is bounded by k, and if so, computing a tree decomposition of optimal width, is achievable in linear time [23], and was shown to be achievable in logarithmic space [48]. Note that the multiplicative constant factor of Bodlaender's linear algorithm [23] is a fast growing exponential in k. However, there are algorithms that find in reasonable time tree decompositions whose width is a good upper approximation of the treewidth, in many cases of practical relevance [24].

3.2 Hypertree Decompositions

A crucial limitation for the practical use of the tree decomposition method in databases is that the method obscures, in many cases, the actual degree of cyclicity of the query hypergraph. Intuitively, many problems are easy if cycles are kept under control.

[2]Some notions strongly related to the treewidth appeared even before the 80's in the literature. It is the case of the *dimension*, defined in 1972 by Bertelé and Brioschi in the context of dynamic programming [21]. For a more detailed story, we refer to [44].

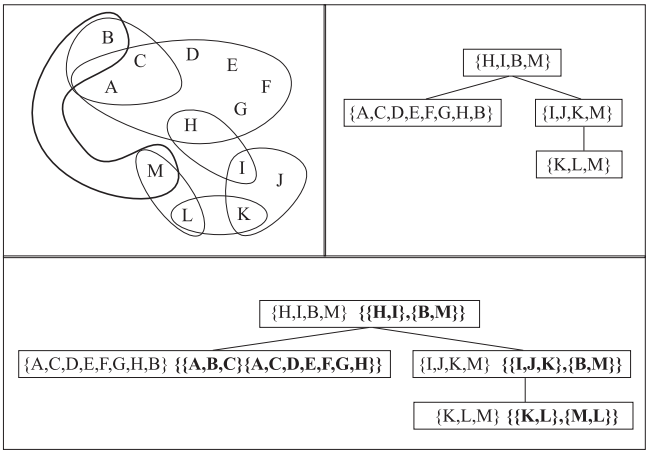

Figure 4: A hypergraph $\mathcal{H}_{Q_2}$, a tree decomposition, and a width-2 hypertree decomposition for $\mathcal{H}_{Q_2}$.

In tree decompositions, this is obtained by using suitable bags of variables in the vertices of the decomposition trees. However, in hypergraph-based problems, any hyperedge may contain a large number of variables at once, so that in order to keep cycles under control, one can exploit the power of hyperedges. Therefore, we expect that a useful measure of degree of cyclicity for hypergraphs depends on bags of hyperedges, rather than just bags of nodes. With this respect, the natural counter-part of the tree decomposition method over hypergraphs is the notion of (generalized) hypertree decomposition [74, 75].

DEFINITION 3.1 ([74]). A *generalized hypertree decomposition* of a hypergraph $\mathcal{H}$ is a triple $HD = \langle T, \chi, \lambda \rangle$, called a *hypertree* for $\mathcal{H}$, where $\langle T, \chi \rangle$ is a tree decomposition of $\mathcal{H}$, and λ is a function labeling the vertices of T by sets of hyperedges of $\mathcal{H}$ such that, for each vertex p of T, $\chi(p) \subseteq \bigcup_{h \in \lambda(v)} h$. That is, all nodes in the χ labeling are covered by hyperedges in the λ labeling.

A *hypertree decomposition* is a generalized hypertree decomposition that satisfies the following additional condition, called *Descendant Condition* or also *special condition*: for each vertex p of T and for each hyperedge $h \in \lambda(p)$, it holds that $h \cap \chi(T_p) \subseteq \chi(p)$, where T_p denotes the subtree of T rooted at p, and $\chi(T_p)$ the set of all variables occurring in the χ labeling of this subtree. Note that we are really interested in the variables occurring in the χ labels, with hyperedges in the λ labels allowing us to effectively deal with such variables. In particular, according to the Descendant Condition, if we have to deal with some variable X in the subtree T_p, then we must immediately care about it in p if it is possible (i.e., if X occurs in some hyperedge in $\lambda(p)$). This condition was introduced to simplify the computation of decompositions, as we shall see in the next section. Moreover, note that all nodes that appear in the hyperedges of $\lambda(p)$ but that are not included in $\chi(p)$ are "ineffective" for v and do not count w.r.t. the connectedness condition.

The *width* of $\langle T, \chi, \lambda \rangle$ is the number $\max_{p \in V}(|\lambda(p)|)$. The (generalized) *hypertree width* of $\mathcal{H}$, denoted by $(g)hw(\mathcal{H})$, is the minimum width over all its (generalized) hypertree decompositions. A class of hypergraphs has bounded (generalized) hypertree width if every hypergraph in the class has (generalized) hypertree width at most k, for some finite natural number $k > 0$.

The notions of hypertree width and generalized hypertree width are true generalizations of acyclicity, as the acyclic hypergraphs are precisely those hypergraphs having (generalized) hypertree width

one [75]. Moreover, recall from the introduction that for each hypergraph $\mathcal{H}$, $ghw(\mathcal{H}) \leq hw(\mathcal{H}) \leq 3 \cdot ghw(\mathcal{H}) + 1$. Hence, this entails that a class of hypergraphs has bounded generalized hypertree width if, and only if, it has bounded hypertree width.

EXAMPLE 3.2. Recall the hypergraph $\mathcal{H}_{Q_2}$ shown in Figure 4. We have already observed that this hypergraph is not acyclic because it has no join tree. However, the connectedness condition can be fulfilled if one inserts additional nodes into the decomposition bags, possibly taken from multiple hyperedges. For instance, this behavior can be observed in the tree decomposition shown in the left part of Figure 4. Note that the width of this decomposition is 7, because the notion of treewidth is based on the number of nodes used in each label. However, this hypergraph is evidently quasi-acyclic. In fact, the hypertree width of $\mathcal{H}_{Q_2}$ is at most 2, because all sets of nodes used in the labels may be covered by two hyperedges at most, as witnessed by the hypertree decomposition in the bottom part of Figure 4. To complete the picture, note that the hypertree width of $\mathcal{H}_{Q_2}$ is precisely 2, because the fact that it is cyclic entails $hw(\mathcal{H}_{Q_2}) > 1$. ◁

3.3 Complexity of Computing HDs

Choosing a decomposition tree and suitable labelings χ and λ in order to get a hypertree decomposition of width $\leq k$ is not that easy, and it is definitely more difficult than computing a bounded-width tree decomposition. While, as already said, deciding whether a hypergraph has generalized hypertree width at most k is NP-complete for any fixed $k \geq 3$ [76], the problem is fixed-parameter tractable if the parameter is the maximum hyperedge size (this follows from a more general result on tree projections [88]).

For those instances where the number of hyperedges and their cardinality is not small, it is convenient to look for width-k hypertree decompositions that, due to the Descendant Condition, can be computed in polynomial time [74]. Moreover, as observed above, they can differ from the best generalized hypertree decomposition of a very small constant factor. In order to get an intuition on this, it is convenient to recall here a useful characterization of hypertree decompositions in terms of the *robber and marshals game* [75].

The game is played by one robber and a number of marshals on a hypergraph. The robber moves on nodes, while marshals move on hyperedges. At each step, any marshal controls an entire hyperedge. During a move of the marshals from the set of hyperedges E to the set of hyperedges E', the robber cannot pass through the nodes in $B = (\cup E) \cap (\cup E')$, where, for a set of hyperedges F, $\cup F$ denotes the union of all hyperedges in F. Intuitively, the vertices in B are those not released by the marshals during their move from E to E'. The game is won by the marshals if they corner and capture the robber somewhere in the hypergraph, by monotonically shrinking the moving space of the robber. A hypergraph $\mathcal{H}$ has k-bounded hypertree width if, and only if, k marshals win the robber and marshals game on $\mathcal{H}$ [75]. In particular, $\mathcal{H}$ is acyclic if, and only if, one marshal wins on $\mathcal{H}$.

It can be seen that winning strategies for the marshals correspond to certain *normal form* decompositions, where the escape spaces of the robber correspond to certain components of the hypergraph and redundancies are ruled out. In such decompositions, formally defined in [74] and later strengthened in [130, 87], the number of vertices cannot exceed the number of variables of $\mathcal{H}$, and is typically much smaller. Moreover, $\mathcal{H}$ has a hypertree decomposition of width w if, and only if, it has a normal-form hypertree decomposition of the same width w. It follows that the game characterization immediately gives to us an algorithm for computing hypertree decompositions. Indeed, one can use a logspace alternating Turing machine (ATM) for deciding whether the marshals win the game in a quite natural way. It can be shown that such a machine has always polynomial space computation trees for "yes" instances, which entails that, for any fixed $k \geq 1$, deciding whether a given hypergraph has hypertree width at most k is in LOGCFL. For the sake of completeness, we recall here that the class LOGCFL consists of all decision problems that are logspace reducible to a context-free language. Note that, since LOGCFL $\subseteq$ AC^1 $\subseteq$ NC^2, deciding if a given hypergraph has hypertree width at most k is a highly parallelizable problem. Correspondingly, the problem of computing a k-bounded hypertree decomposition belongs to the functional version of LOGCFL [74].

3.4 Algorithms for Computing HDs

Based on the intuitions discussed above, a number of algorithms for computing hypertree decompositions have been proposed. The (historically) first implementation is based on dynamic programming and is a quite direct simulation of the method suggested by the game, where we evaluate a bipartite graph whose nodes encodes the possible choices of the k marshals (i.e., k-uples of hyperedges) and the possible escape spaces for the robber (i.e., components of nodes determined by the moves of the marshals) [71]. This algorithm, called `opt-`k`-decomp`, is good if we would like to compute an optimal hypertree decomposition, that is, a decomposition whose width is precisely the hypertree width of the given hypergraph. Whenever we are instead satisfied with any (possibly not optimal) width-k hypertree decomposition for some a-priori chosen width k, it may be better to tackle an opposite top-down approach, as done in the algorithm proposed in [77], which exhibits good performances by using heuristic approaches to accelerate the search for a (generalized) hypertree decomposition of width at most k (not necessarily the minimal one). The resulting algorithm `det-`k`-decomp` is based on backtracking, and can also be implemented for parallel executions.

We note however that, if the threshold width k is not very small and the hypergraph has many hyperedges, then exact algorithms may require too much time. In fact, unlike treewidth, the problem of deciding whether $hw(\mathcal{H}) \leq k$ is fixed-parameter intractable (more precisely, W[2]-hard) if the threshold k is used as parameter [66]. Therefore, unless some unlikely collapse occurs in the classes defined by the fixed-parameter complexity theory, an exponential dependency on k of the form $O(f_1(n)^{f_2(k)})$ is unavoidable. This is usually not a problem for database applications, but large hypergraphs may occur in other applications of hypertree decompositions, such as constraint satisfaction problems or those that we mention in Section 7.

For dealing with large instances, a number of algorithms based on heuristics have been proposed, in order to find efficiently (generalized) hypertree decompositions of a given hypergraph $\mathcal{H}$, without a bounded guarantee on their width. A widely used technique is to look for some tree decomposition of $\mathcal{H}$, e.g., by using the bucket elimination method by [44] or any other algorithm developed for tree decompositions [22], and then cover the bags of nodes with as few hyperedge as possible (by using set covering heuristics) in order to get a generalized hypertree decomposition. As proposed in [47, 109], this can be improved by using techniques to find good separators, that is, suitable sets of nodes that are able to partition the hypergraph to deal with the components to be decomposed—think of them as the possible escape spaces for the robber.

Other heuristic approaches aim at finding some restricted kinds of decompositions, such as the *connected* hypertree decompositions of [132]. Another technique has been discussed in [127], which shows how to use the *branch-decomposition* approach for ordinary graphs [38] for the heuristic construction of generalized

hypertree decompositions (based on the fact that every branch decomposition of width k can be transformed into a tree decomposition of width at most $3k/2$). The use of tabu search for computing (generalized) hypertree decompositions was considered in [116].

A list of available implementations of algorithms for computing hypertree decompositions (and heuristics) is available at the Hypertree Decomposition HomePage [128].

4. HOW TO ANSWER A QUERY VIA HYPERTREE DECOMPOSITIONS?

There are different ways to use (generalized) hypertree decompositions for answering a query (as well as, for instance, to check query containment). We here review some of them, starting with the most natural one.

4.1 HDs and Query Plans

Let Q be a conjunctive query over a database instance DB, and let $HD = \langle T, \chi, \lambda \rangle$ be a generalized hypertree decomposition for $\mathcal{H}_Q$ whose width is k. The basic idea is to use HD as a guide for obtaining a logical query plan that allows us to answer the query in a similar way we can do for acyclic queries. A high level view of this approach is given by the following two steps.

First, for each vertex p in the decomposition tree T, we compute a fresh atom such that: its set of variables is $\chi(p)$; its relation is obtained by projecting on $\chi(p)$ the join of all relations associated with the query atoms whose sets of variables are the hyperedges in $\lambda(p)$. After that, because the χ labeling encodes a tree decomposition, it can be seen that the conjunction of these fresh atoms forms an acyclic conjunctive query, say Q', equivalent to Q. Indeed, the decomposition tree T represents a join tree of $\mathcal{H}_{Q'}$; all fresh atoms encodes subqueries of Q (hence cannot be more restrictive than the original query); and all query atoms are included in some of these subqueries (so that Q' cannot be more liberal than Q). Moreover, because the hypertree width is at most k, each subquery consists of k atoms at most and can be answered in polynomial time.

EXAMPLE 4.1. Consider the query Q_2 we have introduced in Example 2.2 and the hypertree decomposition for it depicted at the bottom of Figure 4. The root, say p_1, of the hypertree covers the hyperedges $\{H, I\}$ and $\{B, M\}$, which come from the query atoms $r_3(H, I)$ and $r_7(B, M)$. Moreover, $\chi(p_1) = \{H, I, B, M\}$, that is, all variables of these hyperedges are included in $\chi(p_1)$. Therefore, in the first phase we associate with p_1 the following query:

$$ans_1(H, I, B, M) \leftarrow r_3(H, I) \wedge r_7(B, M).$$

Similarly, for the remaining three vertices, we get:

$$\begin{aligned} ans_2(A, B, C, D, E, F, G, H) &\leftarrow r_1(A, B, C) \wedge \\ &\quad\ r_2(A, C, D, E, F, G, H), \\ ans_3(I, J, K, M) &\leftarrow r_4(I, J, K) \wedge r_7(B, M), \\ ans_4(K, L, M) &\leftarrow r_5(K, L) \wedge r_6(L, M). \end{aligned}$$

Then, it can be verified immediately that the original query Q_2 is equivalent to the query Q_2' defined as follows:

$$\begin{aligned} ans() \leftarrow\ & ans_1(H, I, B, M) \wedge \\ & ans_2(A, B, C, D, E, F, G, H) \wedge \\ & ans_3(I, J, K, M) \wedge \\ & ans_4(K, L, M). \end{aligned}$$

The query Q_2' is now acyclic. Indeed, by considering the χ-labeling of the hypertree for $\mathcal{H}_{Q_2}$, we immediately get a join tree of $\mathcal{H}_{Q_2'}$. This means that, with the results for $ans_1, ..., ans_4$ at hand, we can answer Q_2 by using any algorithm for answering acyclic queries over Q_2'. ◁

The first phase of the method is actually a polynomial-time reduction to the problem of answering an acyclic query. Indeed, from the given instance (Q, DB), we derive the acyclic instance (Q', DB') by using HD. At this point, as the second step, we can easily answer the acyclic Q' as discussed in Section 2.

Concerning the running time, let r be the size of the largest relation of the database instance DB and let m be the number of vertices of the decomposition tree, which in normal form decompositions cannot exceed the number of variables. The first phase is feasible in $O(m \cdot r^k)$, with each relation in the new database DB$'$ having at most r^k tuples. Let $r' \leq r^k$ be the actual size of the largest relation of the database DB$'$. The overall complexity immediately follows by adding the cost of evaluating the new acyclic instance, which actually dominates the overall cost: the worst-case upper bound is $O(m \cdot (r' + s) \cdot \log(r' + s))$ time and $O(m \cdot (r' + s))$ space, where s is the size of the output.

Note that the above algorithm works as well with the following *projection-free variant*: In the first phase, at each vertex p, we simply omit the projection over the $\chi(p)$ variables, and keep the full join of all the relations associated with hyperedges in $\lambda(p)$.

4.1.1 More Accurate Bounds

The above bounds are largely determined by the output size of the subqueries associated with the vertices of the decomposition tree, for which the estimate $r' = r^k$ is a very rough upper bound. In fact, a number of works recently investigated in detail how to derive more accurate bounds. For each vertex p of the decomposition tree, let Q_p be the conjunctive subquery associated with p, that is, the join of the relations associated with hyperedges in $\lambda(p)$, projected over its "local output variables" $\chi(p)$. In [67], the notion of *coloring number* $C(Q_p)$ of such a query is defined and shown to provide a tight bound, namely $r_p^{C(Q_p)}$, for the output size of Q_p, where r_p is the size of the largest relation occurring in the input database for Q_p. By using the fact that we are interested only in the output variables of the subquery Q_p, this bound improves an earlier defined bound, called *AGM bound* after the authors of [14]. This bound is based on the notion of fractional cover [90, 14] (see also Section 8). Moreover, the coloring number is able to take into account certain kinds of functional dependencies, such as keys, and can be used to provide a tight bound for the output size of the whole query Q, too.

Similar size bounds were also obtained for the factorised representations of query results [119], where relations are succinctly represented through relational algebra expressions comprising Cartesian products, relations with singleton tuples, and their unions.

The above bounds are based on structural information only, without considering (besides keys) auxiliary information about data such as relation sizes, histograms, or attributes selectivities. By using such information on the input database, yet more refined bounds can be obtained, which can also guide the search for a "good" decomposition for the given database, as we shall see later in Section 6. Besides classical estimates used in database optimization (see, e.g., [54]), we mention here the recent DBP bound, based on the notion of *degree-based packing* [97], which exploits the information on the number of tuples in which any value occurs in a relation (degree of the value).

4.1.2 Hypertree-Based Plans for Multiway Joins

Recent works have shown that traditional query optimizers are provably suboptimal on large classes of queries, and worst-case optimal algorithms have been developed [118, 136]. Such algorithms, based on a multiway join approach that may look at all atoms at once, have been implemented, e.g., in the LogicBlox system [11]

and in the EmptyHeaded relational engine [1, 2]. Unfortunately, these algorithms are not able to recover the polynomial-time worst-case bounds for queries having bounded hypertree width. For instance, they may require exponential time for acyclic queries with an empty output (as long as, in principle, such queries could have an exponential number of answers). To overcome these troubles, EmptyHeaded additionally features a query compiler based on hypertrees: it searches for a GHD having the minimum possible estimated size for the intermediate results (by using fractional covers and the AGM bound), and then uses this information to determine the order of attributes to be used in the multiway joins. Such an order is also exploited for the multi-level data structures, called *tries*, used to store input and output relations, and to perform the joins efficiently. For experimental results, see Section 9. We also mention a different approach designed for the Leapfrog Trie Join algorithm [136], where (hyper)tree decompositions are used to guide a flexible caching of intermediate results [98]. The algorithm described in [103] considers also possible functional dependencies, by using the coloring number bound of [67].

Further algorithms based on multiway joins have been defined in order to guarantee the worst-case upper bound that can be obtained by using generalized hypertree decompositions of the given query (more precisely, the bound determined by its fractional hypertree width—see Section 8), without using a dynamic programming approach á la Yannakakis. This is the case in [104], where a notion of geometric resolution is defined to support different kinds of indices and even multiple indices per table. By performing such resolutions, the proposed algorithm covers the whole multidimensional (tuple) space by distinguishing the output tuples (if any) and the other infeasible (non-matching) tuples. As opposed to the standard bottom-up computation, this method can be viewed as a backtracking algorithm with memoization.

4.1.3 Parallel and Distributed Evaluation

From a computational complexity viewpoint, evaluating Boolean queries having bounded hypertree-width is LOGCFL-complete (even for binary acyclic queries) [72]. Combining the result with the techniques discussed by [73], it can be seen easily that the corresponding computation problem (output an answer) is in (functional) LOGCFL. It is known that this class contains highly parallelizable problems: Any problem in LOGCFL is solvable in logarithmic time by a concurrent-read concurrent-write *parallel random access machine* (CRCW PRAM) with a polynomial number of processors, or in $\log^2$-time by an exclusive-read exclusive-write (EREW) PRAM with a polynomial number of processors. In fact, efforts have been spent in the literature to translate these theoretical results into practical implementation of algorithms for parallel (and distributed) query evaluation.

A parallel algorithm, called DB-SHUNT, has been defined in [72] for answering Boolean acyclic queries—an extension, called ACQ, which is able to deal with acyclic queries with output variables is discussed in [68]. This algorithm is well-suited for bounded hypertree-width queries once the transformation process in Section 4.1 has been performed. The transformation itself can be done by exploiting parallelism. For instance, the data storage in the EmptyHeaded system is specifically designed to leverage SIMD parallel architectures, in particular, for the execution of the multiway joins to be performed at every vertex of the decomposition tree. Having the (equivalent) acyclic query at hand, DB-SHUNT (and ACQ) uses a special *shunt* operation based on relational algebra for contracting a join tree, akin the well-known shunt operation used for the parallel evaluation of arithmetic expressions [100]. This way, any tree can be contracted to a single node by a logarithmic number

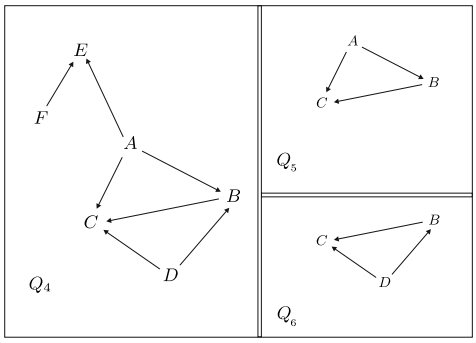

Figure 5: The (hypergraph of the) query Q_4.

of parallel steps, independently of the shape of the tree (e.g., even if the tree is a highly unbalanced chain).

More in detail, the decomposition tree T is preliminary made strictly binary, and its leaves are numbered from left to right. At each iteration, the shunt operation is applied in parallel to all odd numbered leaves of T. To avoid concurrent changes on the same relation, left and right leaves are processed in two distinct steps. Thus, after each iteration, the number of leaves is halved, and the tree-contraction ends within $2 \log m$ parallel steps by using $O(m)$ processors and $O(m)$ intermediate relations having size at most $O(r^2)$, where r is the size of the largest relation of the database instance and m is the number of vertices of T. It can be shown that, if we have a fixed number of processors $c < m/2$, then the number of parallel shunt operations becomes $2(\lceil \log\ c \rceil + 2\lceil m/(4c) \rceil)$. Here, transformations of query atoms and input other than relations are considered costless as long as they are polynomial. Moreover, the above cost refers to the first bottom-up processing of the decomposition tree, which is enough for the evaluation of a Boolean query. For computing the answers of a non-Boolean query, two further processing of the tree are needed, with the same cost (but for the space consumption of the final ascending phase, where we have to consider an additional cost that is linear in the output size).

An implementation of the above strategy has been recently described in [7] for the Valiant's bulk synchronous parallel (BSP) computational model [135], which can simulate the PRAM model. In this model, there are a set of connected machines that do not share any memory. The computation consists in general in a series of rounds, where machines perform some local computation in parallel and communicate messages over the network. Moreover, the same authors use generalized hypertree decompositions for parallel query answering in their GYM algorithm [7], which is a distributed and generalized version of Yannakakis' algorithm specifically designed for MapReduce [43] (as well as for other BSP-based frameworks). In fact, especially after the introduction of Google MapReduce, the problem of evaluating joins efficiently in distributed environments has been attracting much attention in the literature. In a BSP distributed environment, algorithms should mainly deal with the communication costs between the machines, and the number of global synchronizations that are needed to take place between the machines, in particular, the number of rounds of MapReduce jobs that need to be executed.

4.2 Answering Queries without Decompositions

The above approaches to answer a query need to exploit the knowledge of a suitable decomposition tree. However, approaches have been proposed in the literature that do not need this knowledge and that can answer the query by means of "local" computations

only. The basic idea is, again, inspired by an interesting property of acyclic queries: local consistency entails global consistency.

4.2.1 Answering Acyclic Queries without Join Trees

Let DB be a database instance, let Q be a Boolean acyclic query, and let $\mathcal{V}_Q$ denote the set of relations associated with the query atoms of Q. Recall that, according to Yannakakis' algorithm [137], we can answer Q on DB by processing any join tree bottom-up, by performing upward semijoins on the relations in $\mathcal{V}_Q$.

Consider now a different approach that, at the first step, performs the semijoin between *all* pairs of relations in $\mathcal{V}_Q$, without any specific ordering. Note that the semijoin that Yannakakis' algorithm would perform as the first one is also performed at the first step of this novel approach, even though interleaved with other semijoins. The crucial observation is that these additional semijoins preserve the semantics of the query, and they can possibly help to speed up the computation: If some relation becomes empty during the process, then the answer of the query is clearly empty, too.

Let us iterate the method, by performing again semijoins between *all* pairs of relations. Of course, the semijoin that Yannakakis' algorithm would perform as the second one is also performed at this second step, even though interleaved with other semijoins that might have filtered out only tuples that will never contribute to the answer of Q.

By repeating this method for $|\mathcal{V}_Q|$ steps, since any join tree contains $|\mathcal{V}_Q|$ nodes at most, it is now clear that all semijoins that Yannakakis' algorithm would perform are also performed in the same order, just being interleaved with other semijoins—irrelevant as far as the correctness is concerned. Hence, we derive that the answer of Q is empty if, and only if, some relation becomes empty at the end of the process. Therefore, the query Q has been answered without the knowledge of any join tree, but only under the guarantee (promise) that it is actually an acyclic query. In fact, this approach involves computing $O(|\mathcal{V}_Q|^3)$ semijoins in the worst case. Hence, it is hardly useful in practice for dealing with acyclic queries, since computing a join tree is feasible in linear time [133] and since, based on it, we can answer the query with $O(|\mathcal{V}_Q|)$ semijoins only.

Actually, the approach described above is a correct query answering method not only for acyclic queries, but also for queries whose *cores* are acyclic [82, 84]. For the sake of completeness, let us recall and exemplify the concept of core. Let Q' be such that $atoms(Q') \subseteq atoms(Q)$, i.e., Q' is a subquery of Q, and assume there is a homomorphism from Q to Q'. Then, Q and Q' are *homomorphically equivalent*. Moreover, if Q' is a minimal homomorphically equivalent subquery of Q, then Q' is a *core* of Q. All cores are *isomorphic* with each other.

EXAMPLE 4.2. Consider the following query Q_4:

$$ans() \leftarrow \quad r(A,B) \wedge r(B,C) \wedge r(A,C) \wedge r(D,C) \wedge r(D,B) \wedge r(A,E) \wedge r(F,E),$$

Its associated (hyper)graph is reported in Figure 5, where edge orientation just reflects the position of the variables in query atoms. Moreover, the figure reports the hypergraphs associated with the following two queries

$$Q_5 : \ ans() \leftarrow r(A,B) \wedge r(B,C) \wedge r(A,C)$$
$$Q_6 : \ ans() \leftarrow r(D,B) \wedge r(B,C) \wedge r(D,C).$$

Note that Q_5 and Q_6 are two (isomorphic) cores of Q_4. ◁

Note that Yannakakis' algorithm can be still applied on queries with acyclic cores, in particular by building a join tree associated with a core. However, computing a core is a NP-hard task so that, for large queries involving many atoms or in the equivalent setting of constraint satisfaction (see Section 7), the above computation approach that does not need the knowledge of a join tree might be a more viable option. Moreover and somehow surprisingly, acyclicity of a core is a necessary condition for the approach to work. Indeed, the approach is a sound and complete query answering method for a query Q on every database instance DB if, and only if, Q has an acyclic core [82, 84].

Note that having acyclic cores characterizes the (Boolean) *semantic acyclic queries*, that is, those conjunctive queries that are equivalent to acyclic ones [17] (see also [16]). Putting it all together, we get that a Boolean conjunctive query *is a semantic acyclic query if, and only if, it can be answered by just enforcing local consistency*. However, this result stated on classes defined via "cores" is a tractability result for the *promise* version of the problem, that is, the method can be applied only if we have a guarantee on the fact that Q is a semantic acyclic query. In fact, it has been observed [129, 27] that, unless P = NP, there is no efficient way to distinguish whether a non-empty answer means that the query has a true non-empty answer, or that Q has no acyclic core.

4.2.2 Answering Low-Width Queries without HDs

Using algorithms for answering queries that are not acyclic, without the knowledge of a decomposition, attracted much attention in the literature (e.g., [32, 41, 13, 61]). In fact, the precise relationship between generalized hypertree decompositions and such algorithms has recently been clarified.

For any fixed natural number $k > 0$, let $\mathcal{V}_{Q,k}$ be the set of relations containing, for each k-tuple of query atoms, one relation obtained as the join of the corresponding k database relations. In this case, the idea is to repeatedly perform semijoins between all pairs of relations in $\mathcal{V}_{Q,k}$ until a fixpoint is reached. At this point, either some relation is empty or we got the so-called *k-local consistency* property. In the former case, we are sure that the answer of Q on the given database is empty. We say that a Boolean query Q can be answered by enforcing *k-local consistency* if, for any given database, its answer is non-empty if, and only if, the above procedure ends without getting empty relations (that is, we achieve k-local consistency). It has been shown that this happens if, and only if, Q has a core whose generalized hypertree width is at most k [82, 84]. Therefore, similarly to the case of acyclic queries, we can answer the query Q neither knowing the core nor knowing any generalized hypertree decomposition $\langle T, \chi, \lambda \rangle$ of $\mathcal{H}_Q$. In this extension, the semijoin operations are defined with respect to all variables occurring in the k-tuples of query atoms, that is, without considering the key role played by the variables occurring in the χ labeling in any decomposition tree. The correctness of this approach follows immediately from the correctness of the projection-free variant for answering width-k queries described in Section 4.1.

Let v be the number of variables occurring in the core of Q. Assume the core has a generalized hypertree decomposition of width k. From the results in [87] it follows that it has a decomposition of the same width which has at most v decomposition vertices. Therefore, the fixpoint for enforcing k-local consistency must end after at most $2 \cdot v$ rounds of semijoins. In particular this entails that, if a fixpoint is not reached after $2 \cdot vars(Q)$ rounds, then we are sure that the generalized hypertree width of Q (and of its core) is greater than k. Again, the notion of core plays a crucial role, so that the approach might be viewed as a viable query answering procedure in the presence of large queries for which computing a core—and a generalized hypertree decomposition (both NP-hard tasks)—may be practically infeasible. However, the tractability result holds for the promise version of the problem.

A related characterization is also known for queries that are not Boolean. In this case, k-local consistency is a sound and complete method to compute the answer of a query Q over a set W of output variables if, and only if, variables in W are covered by some vertex of a width-k generalized hypertree decomposition of some query Q' that is homomorphically equivalent to Q [82, 84]. In particular, for $k = 1$ this provides a precise characterization of the relationship between (1-)local consistency and global consistency that was missing in the classical results about acyclic queries: *local consistency entails global consistency* if, and only if, the variables occurring in each atom p of Q are covered by a join tree of some (acyclic) query homomorphically equivalent to Q. Note that we do not require the existence of a join tree covering all atoms (this would mean that Q is acyclic). Instead, we can use different equivalent queries (with their join trees/decompositions) for different atoms.

5. CAN WE GO BEYOND CONJUNCTIVE QUERIES?

Aggregation operators are very often at the basis of decision support systems examining large volumes of data, in order to obtain business intelligence. We briefly review some results based on GHDs for such queries.

5.1 Counting Query Answers

Let Q be a conjunctive query and let DB be a database instance. So far, we have focused on the problem of computing the set Q^{DB}, but in many cases we might just be interested in computing the cardinality of this set. This *counting* problem is, indeed, a natural abstraction for SQL queries specifying "COUNT" aggregates (see, e.g, [26, 28]). The challenge in counting problems is to compute the right number in polynomial time without actually computing the (possibly exponentially-many) query answers. In particular it is crucial, both in theory and in practice, to deal with queries where output variables can be specified, so that we are only interested in counting the answers projected on them. Technically, such distinguished variables are free, while all the other variables are existentially quantified. As a matter of fact, in almost all practical applications, there are many of such "auxiliary" (existentially quantified) variables whose instantiations must not be counted.

Whenever all variables are free, having bounded generalized hypertree width is a sufficient condition for the tractability of the counting problem [121]. Moreover, on fixed-arity queries, this condition is also necessary (under widely believed assumptions in parameterized complexity) [40].[3] However, in presence of projections, classical decomposition methods are not helpful. Indeed, even for acyclic queries [121], counting answers is #P-hard in this case. An algorithm counting the answers of an acyclic query Q in $O(m \cdot r^2 \cdot 4^r)$, with r being the size of the largest database relation and m the number of atoms, has been exhibited in [121]. Therefore, the evaluation is tractable over acyclic instances w.r.t. query complexity (where the database is fixed and not part of the input). The technique can be extended easily to queries having generalized hypertree width at most k, for some fixed number k (cf. [86]). Thus, to get more powerful structural results, we should distinguish the hypergraph nodes associated with free and existentially quantified variables. A sufficient condition for tractability is the existence of a homomorphically equivalent subquery Q' of Q that includes all free variables, and such that there is a width-k generalized hypertree decomposition for Q' which covers all frontiers of the free variables with the existential variables (see [86] for more information). Interestingly, it turns out that for fixed-arity queries this structural condition precisely characterizes the queries where the counting problem is tractable [33].

[3]Recall that for queries having a fixed maximum arity, bounded (generalized) hypertree width entails bounded treewidth (and vice versa), so that the two notions are interchangeable.

5.2 Further Aggregations

Answering ORDER BY queries is intimately related to the problem of computing the best solution according to the given ordering, which is incidentally another important aggregate (MAX) in SQL queries. This was first given in the context of ranking solutions to discrete optimization problems [110]. In particular, whenever the MAX problem of computing an optimal solution is feasible in polynomial time, then we can solve the problem of returning all the solutions in the ranked order with polynomial delay, as well as the more general problem of returning the best K-ranked solutions over all solutions, i.e., the top-K query evaluation problem [94].

A number of structural tractability results for MAX (so, for top-K and ORDER BY) queries are known in the literature (see, e.g., [64]). For *monotone* functions built on top of one binary aggregation operator (such as the standard $+$ and $\times$), MAX is feasible in polynomial time over queries that have bounded treewidth [45], and over queries that have bounded generalized hypertree width [106, 60]. Structural tractability results for extensions to certain (possibly) *non-monotone* functions which manipulate "small" (in fact, polynomially-bounded) values, called *smooth evaluation functions*, have also been studied [83, 64].

An implementation of some aggregation operators based on hypertree decompositions is described in [56, 57]. It can be used with any system at a logical level. For the sake of efficiency, a semijoin operator that supports the execution of these operators over decompositions is implemented in the open-source DBMS PostgreSQL. Further operators are considered in [96] for the so-called AJAR queries, that is, queries with annotated relations over which (possibly multiple) aggregations can be used. Technically, aggregations are modeled by means of semiring quantifiers that "sum over" or "marginalize out" values. It is argued that such queries can be used to extend conjunctive queries with most SQL-like aggregation operators, as well as to capture data processing problems such as probabilistic inference via message passing on graphical models [101]. The problem with these queries is finding a variable ordering that allows us to manage the aggregations by using a suitable generalized hypertree decompositions ("compatible" with the ordering), together with standard join algorithms. The proposed algorithm can efficiently be implemented in the parallel framework, by using GYM [7] and the degree-based MapReduce algorithm in [97]. Answering AJAR queries is equivalent to answering Functional Aggregate Queries [105].

6. CAN WE COMBINE STRUCTURAL AND DATA PROPERTIES?

The basic notion of hypertree decomposition ignores the quantitative aspects of data, such as attributes selectivity and degree, and cardinality of relations, as well as information on functional dependencies such as primary or external keys, and so on. Clearly enough, such knowledge, in the practice of query evaluation, may dramatically speed-up the evaluation time. In general there is an exponential number of hypertree decompositions of a given width, and every decomposition encodes a different way of aggregating groups of atoms and arranging them in a tree-like fashion. Now, as far as the theoretical tractability is concerned, we may be happy with any minimum-width decomposition, possibly measured according to the accurate bounds in Section 4. However, in practical

real-world applications we have to exploit all information available about the database, and compute a decomposition that will eventually lead to the best possible evaluation performances. For instance, by using information on functional dependencies and their closure, it is possible to obtain a useful generalization of hypertree decompositions [5], while further information on the data have been used in the algorithms in [97, 2].

In order to express preferences about hypertree decompositions of a given query, a notion of *weighted* hypertree decomposition has been proposed by [130]. In a nutshell, the notion is based on fixing a weighting function that equips each (generalized) hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ of $\mathcal{H}$ with a rational number, and then on looking for a decomposition minimizing this value over all possible decompositions of some desired width.

In particular, an interesting weighting function can be obtained by estimating the cost associated with the operations required to answer Q according to the algorithm discussed in Section 4.1. Formally, if p is a vertex of T, then we define $v^*(p)$ as the cost of computing the join of all atoms associated with hyperedges in $\lambda(p)$, by eventually projecting the result over $\chi(p)$. Similarly, if p' is a child of p, then we define $e^*(p, p')$ as the cost of evaluating the semi-join between the reletion hereby computed for p and p'. These costs can be estimated via standard techniques (see, e.g., [55]), possibly combined with recent proposals [35].

Based on the functions v^* and e^*, every decomposition HD can be then associated with the weight: $\omega^*(HD) = \sum_p v^*(p) + \sum_{(p,p')} e^*(p, p')$. In [130], it has been observed that, for any fixed natural number $k > 0$, computing a normal form width-k hypertree decomposition whose ω^*-weight is the minimum possible is feasible in polynomial time. For arbitrary weighting functions, the problem has been shown to be intractable.

Further hybrid approaches, specifically designed to deal with counting problems, are discussed in [86].

7. ARE THERE APPLICATIONS OUTSIDE THE DATABASE AREA?

While originally tailored for database applications, hypertree decompositions have emerged over the years as a powerful approach to identify islands of tractability in a number of different areas of research. A few applications are discussed below. Further applications include, e.g., matrix reordering [53] and circuit design [109].

7.1 CSPs and Homomorphisms

An instance of a *constraint satisfaction problem (CSP)* (also *constraint network*) (e.g., [44]) is a triple $I = (Var, U, \mathcal{C})$, where Var is a finite set of variables, U is a finite domain of values, and $\mathcal{C} = \{C_1, C_2, \ldots, C_q\}$ is a finite set of constraints. Each constraint C_i is a pair (S_i, r_i), where S_i is a list of variables of length m_i called the *constraint scope*, and r_i is an m_i-ary relation over U, called the *constraint relation*. In fact, the tuples of r_i indicate the allowed combinations of simultaneous values for the variables S_i. A *solution* to a CSP instance is a substitution $\theta : Var \longrightarrow U$, such that for each $1 \leq i \leq q$, $S_i\theta \in r_i$. The problem of deciding whether a CSP instance has any solution is called *constraint satisfiability*. This problem is in general NP-complete.

The structure of a CSP instance I can be represented by its associated hypergraph $\mathcal{H}(I) = (V, H)$, where $V = Var$ and $H = \{S \mid (S, r) \in \mathcal{C}\}$. Constraint satisfiability is feasible in polynomial time on classes of instances having bounded hypertree width [74]. The result is the natural consequence of the fact that constraint satisfiability is essentially the same problem of answering conjunctive queries, reformulated in a different context and with a different syntax (see, e.g., [10, 93] for a recent implementation in the CSP area). Indeed, as said in Section 1.7, the two settings (CSPs and conjunctive queries) can be abstractly viewed as special instances of the *homomorphism problem*, which takes as input two finite relational structures $\mathcal{A}$ and $\mathcal{B}$, and asks whether there is a homomorphism from $\mathcal{A}$ to $\mathcal{B}$ [108]. In the CSP setting, $\mathcal{A}$ models the variables and the scopes of the constraints, while $\mathcal{B}$ models the relations associated with constraints; in the database setting, instead, $\mathcal{A}$ models the conjunctive query and $\mathcal{B}$ models the database instance.

Structural methods are used to support different techniques, too. For instance, hypertree decompositions are used for improving the performance of AND/OR search algorithms, by producing tighter bounds on the search space explored by these algorithms [120]. Some approaches also use structural methods to improve performances of algorithms based on enforcing high levels of consistencies localized to the decomposition clusters (see, e.g., [99]).

Note that the tractability results discussed in this paper refer to the so-called *uniform* CSP (homomorphism) problem, where both structures are part of the input, i.e., nothing is fixed. For completeness we recall that an interesting line of research in the CSP community looks at the so-called *non-uniform* problem, where $\mathcal{B}$ is a fixed structure (thus, any instance of such a problem just consists of some structure $\mathcal{A}$). In this case, the challenge is identifying any relational structure $\mathcal{B}$ such that all instances over this fixed structure is tractable, for any input structure $\mathcal{A}$ (see. e.g. [36]).

7.2 Strategic Games and Nash Equilibria

The theory of strategic games has important applications in economics and decision making [117]. Nash equilibria are the most prominent solution concept therein, and problems related to their computation have attracted much research in computer science [42, 34]. In general, Nash equilibria admit strategies played according to probability distributions, but in many cases it is useful to focus on *pure* strategies, which have to be played deterministically [49].

A *strategic game* [117] $\mathcal{G}$ is a tuple $\langle P, Neigh, Act, U \rangle$, where P is a non-empty set of distinct players, $Neigh : P \longrightarrow 2^P$ is a function such that for each $p \in P$, $Neigh(p) \subseteq P - \{p\}$ contains all players of interest for p, $Act : P \longrightarrow A$ is a function returning for each player p a set of possible actions $Act(p)$, and U associates a utility function $u_p : Act(p) \times_{j \in Neigh(p)} Act(j) \rightarrow \Re$ to each player p. For a player p, p_a denotes her *strategy* to play the action $a \in Act(p)$. A global strategy $\mathbf{x}$ is a set containing a strategy for each player $p \in P$ and, with a small abuse of notation, $u_p(\mathbf{x})$ denotes the output of u_p on the projection of $\mathbf{x}$ to the domain of u_p, i.e., the output of the function u_p applied to the actions played by p and her neighbors according to the strategy $\mathbf{x}$. Moreover, we denote by $\mathbf{x}_{-p}$ the set obtained by removing from $\mathbf{x}$ the strategy associated with player p. Then, we say that a global strategy $\mathbf{x}$ is a pure Nash Equilibrium for $\mathcal{G}$ if, for every player $p \in P$, there is no strategy p_a such that $u_p(\mathbf{x}) < u_p(\mathbf{x}_{-p} \cup \{p_a\})$. A game $\mathcal{G}$ is naturally associated with the *strategic dependency hypergraph* $\mathcal{H}(\mathcal{G})$, whose vertices are the players P and whose set of hyperedges is $\{\{p\} \cup Neigh(p) \mid p \in P\}$. Deciding the existence of pure Nash equilibria (and compute one, if any) is NP-hard on arbitrary strategic games. However, the problem becomes tractable on classes of games whose associated have bounded hypertree width [63]. Tractability results also hold on such structurally-restricted games, for related solution concepts [102, 80].

Decomposition methods have also been used in the related setting of *coalitional* games (see, e.g., [79, 29]), where players are not selfish and the main goal is to compute worth distributions among the agents that can be perceived as fair and stable by all of them.

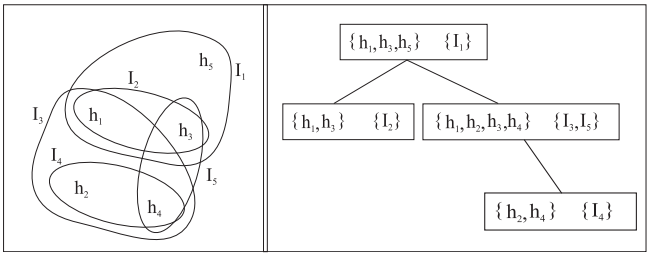

Figure 6: A dual hypergraph for a combinatorial auction, with a with-2 hypertree decomposition.

7.3 Combinatorial Auctions

Combinatorial auctions are well-known mechanisms for resource and task allocation where bidders are allowed to simultaneously bid on combinations of items. Such mechanisms apply to those situations where the bidders' valuations of bundles of items are not equal to the sum of their valuations of individual items [39].

A *combinatorial auction* is a pair $\langle \mathcal{I}, \mathcal{B} \rangle$, where $\mathcal{I} = \{I_1, ..., I_m\}$ is the set of items the auctioneer has to sell, and $\mathcal{B} = \{B_1, ..., B_n\}$ is the set of bids from the buyers interested in the items in $\mathcal{I}$. Each bid B_i has the form $\langle item(B_i), pay(B_i) \rangle$, where $pay(B_i)$ is a rational number denoting the price a buyer offers for the items in $item(B_i) \subseteq \mathcal{I}$. An outcome for $\langle \mathcal{I}, \mathcal{B} \rangle$ is a subset $\mathbf{b}$ of $\mathcal{B}$ such that $item(B_i) \cap item(B_j) = \emptyset$, for each pair B_i and B_j of bids in $\mathbf{b}$ with $i \neq j$. Bidder interaction in a combinatorial auction $\langle \mathcal{I}, \mathcal{B} \rangle$ can be represented by the so-called *dual hypergraph* $\overline{\mathcal{H}}(\langle \mathcal{I}, \mathcal{B} \rangle)$, whose nodes are the various bids in the auction and hyperedges represent items, that is, each item $\mathrm{I} \in \mathcal{I}$ is associated with a hyperedge consisting of the set of bids that contain I.

For example, the hypergraph in Figure 6, on the left, encodes an auction over items $\{\mathrm{I}_1, ..., \mathrm{I}_5\}$ and where bids are on the following bundles of items: h_1:$\{\mathrm{I}_1\}$, h_2:$\{\mathrm{I}_1, \mathrm{I}_2, \mathrm{I}_3\}$, h_3:$\{\mathrm{I}_1, \mathrm{I}_2, \mathrm{I}_5\}$, h_4:$\{\mathrm{I}_3, \mathrm{I}_4\}$, and h_5:$\{\mathrm{I}_3, \mathrm{I}_4, \mathrm{I}_5\}$. Note that this hypergraph is not acyclic and its hypertree width 2, as it is witnessed by the decomposition reported in Figure 6.

A crucial problem for combinatorial auctions is the *winner determination problem* of determining the outcome $\mathbf{b}^*$ that maximizes the sum of the accepted bid prices (i.e., $\sum_{B_i \in \mathbf{b}^*} pay(B_i)$) over all possible outcomes. The problem is in general NP-hard [125]. However, it is tractable on auctions with bounded hypertree-width dual hypergraphs [60] (see also [51]).

8. CAN WE GO BEYOND HYPERTREES?

Consider a generalized hypertree decomposition HD=$\langle T, \chi, \lambda \rangle$. The cyclicity measure used in standard generalized hypertree decompositions is based on the cardinality of the set $\lambda(p)$ in charge of covering $\chi(p)$, for each vertex p of the decomposition tree. It has been observed that a more general notion can be obtained by allowing the use of more general functions $f_{HD}(\cdot)$, to measure the weight $f_{HD}(p)$ of such a vertex, and hence to define the width of a decomposition. This is formalized in [4], where possibly infinite hypergraphs are considered, too.

In Section 1.8 a number of restrictions of GHDs, such as the *greedy hypertree decompositions* [88] and the *subset-based decompositions* [76], have been mentioned. The notion of width in such methods was however still based on the cardinality of the λ labelings. We recall some further decompositions based on more general width functions.

8.1 Fractional HDs

A *fractional hypertree decomposition* [90] of a hypergraph $\mathcal{H}$ is a pair $FHD = \langle HD, \gamma \rangle$, where $HD = \langle T, \chi, \lambda \rangle$ is a generalized hypertree decomposition of $\mathcal{H}$, and γ is a mapping associating each vertex p of T with a function $\gamma_p : \lambda(p) \mapsto \mathbb{R}$ mapping hyperedges to non-negative real numbers. For each vertex p in T, γ_p encodes a *fractional cover* of the nodes in $\chi(p)$: for each $v \in \chi(p)$, $\sum_{h \in \lambda(p), v \in h} \gamma_p(h) \geq 1$. Define $\rho(p) = \sum_{h \in \lambda(p)} \gamma_p(h)$. The *width* of FHD is the maximum of $\rho(p)$ over all vertices p in T. The *fractional hypertree width* of $\mathcal{H}$, denoted by $fhw(\mathcal{H})$, is the minimum width over all its fractional hypertree decompositions.[4]

Queries whose fhw is bounded by some constant k can be answered in polynomial time by applying the transformation of the given query Q to an acyclic query, based on the generalized hypertree decomposition HD at hand (see Section 4.1). Indeed, the maximum number of answers of the subquery Q_p associated with each vertex p is at most $|r_p|^{\rho(p)}$, where r_p is the largest database relation associated with the hyperedges in $\lambda(p)$ [90, 14]. Generalized hypertree width is obtained if in the definition of fractional hypertree decomposition we restrict the codomain of each γ_p to be integral. It is thus not surprising that there are hypergraphs where the fractional hypertree width is smaller than the (generalized) hypertree width—see Figure 2.

EXAMPLE 8.1 ([90]). For any value of $n \geq 1$, consider the hypergraph $\mathcal{H}_n$ built as follows: $\mathcal{H}_n$ has a node v_S, for each subseteq $S \subseteq \{1, ..., 2n\}$; furthermore, for every $i \in \{1, ..., 2n\}$, $\mathcal{H}_n$ has a hyperedge $h_i = \{v_S \mid i \in S\}$. It can be checked that $fhw(\mathcal{H}_n) = 2$. This is witnessed by the fractional decomposition $FHD = \langle T, \chi, \gamma \rangle$ consisting of one vertex p only and such that $\gamma_p(h_i) = 1/n$, for each hyperedge $h_i \in edges(\mathcal{H}_n)$. On the other hand, the generalized hypertree width of $\mathcal{H}_n$ is n. ◁

The complexity of checking whether a hypergraph has fractional hypertree width at most w, for some fixed constant $w \geq 0$, is an open problem. However, there is a polynomial-time algorithm for deciding whether the fractional hypertree width is $f(w)$ for a function f in $O(w^3)$ [113].

8.2 Tree Projections

All the known purely-structural decomposition methods, where decompositions are only based on the (hyper)graph structure of the given query, are in fact specializations of the general and abstract framework of *tree projections* [126]. In this framework, a query Q is given together with a set $\mathcal{V}$ of atoms, called views, which are defined over the variables in Q. The question is whether (parts of) the views can be arranged as to form a tree projection (playing the role of a decomposition tree), i.e., a novel acyclic query that still "covers" Q. By representing Q and $\mathcal{V}$ via the hypergraphs $\mathcal{H}_Q$ and $\mathcal{H}_\mathcal{V}$, where hyperedges one-to-one correspond with query atoms and views, respectively, the tree projection problem reveals its graph-theoretic nature. For two hypergraphs $\mathcal{H}_1$ and $\mathcal{H}_2$, let $\mathcal{H}_1 \leq \mathcal{H}_2$ denote that each hyperedge of $\mathcal{H}_1$ is contained in some hyperedge of $\mathcal{H}_2$. Then, a tree projection of $\mathcal{H}_Q$ w.r.t. $\mathcal{H}_\mathcal{V}$ is any acyclic hypergraph $\mathcal{H}_a$ such that $\mathcal{H}_Q \leq \mathcal{H}_a \leq \mathcal{H}_\mathcal{V}$.

The existence of a tree projection is often a key to establish tractability results for decision problems [126] and for enumeration ones [85]. Moreover, it guarantees that interesting consistency properties [82] and game-theoretic characterizations [87] hold. Indeed, according to this unifying view [62], differences among the various (purely) structural decomposition methods just come in the

[4]The equivalent (original) definition in [90] does not use the λ-labeling, so that γ_p weighs all hyperedges. In practical applications, it can be convenient to consider λ, possibly for restricting the search space at each vertex. Our definition is used, e.g., in [1].

way the resource hypergraph $\mathcal{H}_2$ is defined. In particular, for a fixed natural number k, the treewidth method is obtained by considering as available hyperedges in the resource hypergraph all combinations of k nodes of $\mathcal{H}_1$; while the generalized hypertree-width method is obtained by considering, as resource hypergraph $\mathcal{H}_2$, the hypergraph $\mathcal{H}_1^k$ such that each of its hyperedge is the union of k hyperedges of $\mathcal{H}_1$. However, note that the notion of tree projection is more general then both treewidth and hypertree width, because the hyperedges of the "resource" hypergraph $\mathcal{H}_2$ may model arbitrary subproblems of the given instance whose solutions are easy to compute, or already available from previous computations (for instance, materialized views while answering database queries).

Deciding the existence of tree projections is NP-hard [76], however the problem is fixed parameter tractable, if the cardinality of the largest hyperedge of $\mathcal{H}_2$ is used as parameter [87]. A greedy technique has been proposed by [88] to identify tractable instances. This technique, when $\mathcal{H}_2 = \mathcal{H}_1^k$, provides the notion of greedy hypertree decomposition [88].

8.3 Submodular Width

While the frontier of polynomial-time tractability for conjunctive queries is still unknown, some important advances have been made with regard to a weaker notion of tractability based on parameterized complexity. Let p-CQ be the problem of answering a Boolean conjunctive query Q, parameterized by the hypergraph $\mathcal{H}_Q$. Given a set $\mathbf{H}$ of hypergraphs, let $CQ(\mathbf{H})$ denote the set of all Boolean conjunctive queries, whose associated hypergraphs belong to $\mathbf{H}$. It has been shown that, for each recursively enumerable class of hypergraphs $\mathbf{H}$, the problem p-CQ over the class of queries $CQ(\mathbf{H})$ is fixed-parameter tractable if, and only if, $\mathbf{H}$ has bounded *submodular width* (unless the Exponential Time Hypothesis [95] fails) [115]. Consider such a class $\mathbf{H}$ and assume that the submodular width of all hypergraphs in the class is at most w. Then, the result guarantees that there are two computable functions f_1 and f_2 such that each query Q in $CQ(\mathbf{H})$ can be answered over a database DB in time $O(f_1(\mathcal{H}_Q) \cdot |\mathrm{DB}|^{f_2(w)})$. In the algorithm described in [115], f_1 has a term of the form $2^{2^{O(|V|)}}$, where V is the set of nodes in $\mathcal{H}_Q$. For the submodular width, the decompositions used to answer the query depend not only on the query hypergraph, but on the actual database DB, too. It follows that, unlike for all methods based on generalized hypertree decompositions (and tree projections), we cannot amortize the time spent in computing a good query plan beforehand through the time saved later, when Q is executed multiple times. Rather, at each query execution, we incur the same overhead exponential in the size of $\mathcal{H}_Q$.

Submodular width is more general than the fractional (and hence the generalized) hypertree width. Indeed, it has been shown that there are classes of hypergraphs having bounded submodular width, but unbounded fractional hypertree width. This follows from the results on the adaptive width in [114], and from its equivalence with the submodular width [115].

9. WHAT DO EXPERIMENTS SHOW?

Experimental evidence of the benefits gained from structural decomposition methods in query optimization (and CSP solving) has been provided in the literature by a number of different authors and in different application domains.

As already mentioned, the EmptyHeaded relational engine uses generalized hypertree decompositions in its query planner [1]. The system, supporting a rich datalog-like query language, has been tested on popular graphs datasets (Google+, Higgs, LiveJournal, Orkuts, Patent, Twitter). It emerged that graph engines supporting the same kind of high-level query language are outperformed by EmptyHeaded, which is even an order of magnitude faster than many low-level graph engines on such queries. The system has been also tested in the context of RDF processing [2], by comparing its performances with those of state-of-the-art specialized RDF engines. Experiments have been performed on the LUBM benchmark [91]. The results show that EmptyHeaded outperforms all other engines on many cyclic queries, while remaining competitive with the specialized RDF engines on the others.

In [98], similar techniques based on structural decompositions (but using a different width function) have been used to guide a flexible caching of intermediate results in the context of computing multiway joins. Experimental validation has been conducted on the SNAP collection (*https://snap.stanford.edu/data*) showing benefits over both earlier ad-hoc algorithms and standard DBMSs.

In [56, 57], a query optimizer using hypertree decompositions and taking into account the cost model discussed in Section 6 has been implemented. The optimizer can be put on top of any existing database management system supporting JDBC technology, by transparently replacing its standard optimization module. Moreover, to exploit the potential of HDs in a proper way, a special semijoin operator supporting some aggregation operators has been implemented within the PostgreSQL DBMS. The resulting prototype has been compared with standard PostgreSQL and with a well-known commercial DBMS on the TPC-H (*www.tpc.org/tpch*) benchmark, consisting of queries designed for simulating classical tasks in decision support systems. The results demonstrate a significant gain obtained by using query plans based on hypertree decompositions on queries involving more than two query atoms.

In [10], a CSP solving technique based on generalized hypertree decompositions and using compressed representations for the relations has been proposed, and its scalability has been assessed over well-known CPS benchmarks (Large bdd, VarDimacs, Modified Renault, Pret, Dubois). Another implementation focusing on large (CSP) instances is described in [112] that, in particular, focused on the so-called Modified Renault benchmark,[5] which comes from a real application concerning the configuration of a car (the Renault Megane) [9, 12]. The considered instances consist of about 150 atoms/constraints and 110 variables, with database instances where attributes have degree at most 42, and the largest constraint relation contains 48721 tuples. The generalized hypertree width is 3 for most instances (with a maximum of 4). The total number of solutions is about $2 \cdot 10^{12}$, however this information is not very meaningful, because of the many auxiliary variables occurring in the problem. Rather, by using the algorithms based on generalized hypertree decompositions, it is possible to compute the solutions of these instances (or just their number) over the actual variables of interest. None of the other available engines that we know, either in the database or in the CSP community, were able to compute such a result for those large instances. For more information, we refer to the Hypertree Decomposition Web-page [128].

10. CONCLUSION & FUTURE RESEARCH

The aim of decomposition methods is to extend the nice computational properties of acyclic queries to larger classes of queries. Hypertree decompositions serve this aim very well. They fulfill three important criteria postulated for decompositions: (1) Generalization of Acyclicity, (2) Tractable Recognizability, and (3) Tractable Query-Answering. There has been a large body of research on HDs since their introduction in 1999. However, there are still

[5] http://www.cril.univ-artois.fr/ lecoutre/benchmarks.html

many interesting questions that deserve further research, some of purely theoretical nature, some of more practical relevance.

We conclude this paper by listing those research issues which we deem most important.

- Are there substantially different decomposition methods generalizing hypertree decompositions, which still fulfill criteria 1-3?
- Are there more efficient exact algorithms for computing HDs? The best known algorithm for checking for hypertree width k runs in time exponential in $2k$. Given that the problem is fixed-parameter intractable in parameter k, we most likely cannot get rid of the constant k in the exponent, but is it maybe possible to get rid of the factor 2 in front of k?
- Further research on heuristic methods for computing HDs of "reasonably" low width would be beneficial, especially in the context of constraint satisfaction.
- In order to suitably integrate the HD method into query optimizers of existing and future DBMSs, research is needed on how this method can be effectively combined with the standard methods of logical and physical query optimization. Such combined methods could, for example, use weighted hypertree decompositions [130], see Section 6.
- It has been shown [89] (under a standard complexity assumption) that Boolean queries in a recursively enumerable class $\mathbf{Q}$ can be answered in polynomial time if, and only if, the cores of the queries in $\mathbf{Q}$ have bounded treewidth (or bounded hypertree width, as boundedness for tw and hw coincides in case of bounded arity). Is there a similar characterization for tractability for the case of unbounded arities (see also Section 8.3)? A related open question is to chart the tractability frontier for arbitrary queries, i.e., not necessarily Boolean [27, 85, 131, 15].
- Find new applications that profit from the technique of hypertree decompositions. As an example, it would be interesting to see if hypertree decompositions could be used profitably for speeding up Datalog engines (see [78, 134]), and, more specifically, widely used Datalog-based answer set programming (ASP) systems such as e.g. DLV [111]. In such systems, Datalog rules bodies essentially consist of conjunctive queries.

Acknowledgments

Georg Gottlob's work was supported by the EPSRC Programme Grant EP/M025268/"VADA: Value Added Data Systems – Principles and Architecture". The work of Gianluigi Greco and Nicola Leone was supported by the Italian Ministry of University and Research under PON project "Ba2Know (Business Analytics to Know) Service Innovation - LAB", No. PON03PE_00001_1. Gianluigi Greco's work was also supported by a Kurt Gödel Research Fellowship, awarded by the Kurt Gödel Society.

We thank Andrea Calì for comments on a draft of this paper.

11. REFERENCES

[1] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. EmptyHeaded: A Relational Engine for Graph Processing. In *Proc. of SIGMOD'16*, 2016.

[2] C. R. Aberger, S. Tu, K. Olukotun, and C. Ré. Old techniques for new join algorithms: A case study in RDF processing. *CoRR*, abs/1602.03557, 2016.

[3] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases: The Logical Level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[4] I. Adler. *Width functions for hypertree decompositions*. PhD thesis, Albert-Ludwigs-Universität Freiburg, 2006.

[5] I. Adler. Tree-width and functional dependencies in databases. In *Proc. of PODS'08*, pages 311–320, 2008.

[6] I. Adler, G. Gottlob, and M. Grohe. Hypertree width and related hypergraph invariants. *European Journal on Combinatorics*, 28(8):2167–2181, 2007.

[7] F. N. Afrati, M. Joglekar, C. Ré, S. Salihoglu, and J. D. Ullman. GYM: A multiround join algorithm in mapreduce. *CoRR*, abs/1410.4156, 2014.

[8] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. DBToaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, 5(10):968–979, 2012.

[9] J. Amilhastre, H. Fargier, and P. Marquis. Consistency restoriation and explanations in dynamic CSPs - application to configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002.

[10] K. Amroun, Z. Habbas, and W. Aggoune-Mtalaa. A compressed generalized hypertree decomposition-based solving technique for non-binary constraint satisfaction problems. *AI Communications*, 29(2):371–392, 2016.

[11] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. Design and implementation of the logicblox system. In *Proc. of SIGMOD'15*, pages 1371–1382, 2015.

[12] J.-M. Astesana, L. Cosserat, and H. Fargier. Constraint-based vehicle configuration: A case study. In *Proc. of ICTAI'10*, pages 68–75, 2010.

[13] A. Atserias, A. Bulatov, and V. Dalmau. On the power of k-consistency. In *Proc. of ICALP'07*, pages 279–290, 2007.

[14] A. Atserias, M. Grohe, and D. Marx. Size bounds and query plans for relational joins. *SIAM Journal on Computing*, 42(4):1737–1767, 2013.

[15] G. Bagan, A. Durand, and E. Grandjean. On acyclic conjunctive queries and constant delay enumeration. In *Proc. of CSL/EACSL'07*, pages 208–222, 2007.

[16] P. Barceló, G. Gottlob, and A. Pieris. Semantic acyclicity under constraints. In *Proc. of PODS'16*, 2016.

[17] P. Barceló, M. Romero, and M. Y. Vardi. Semantic acyclicity on graph databases. In *Proc. of PODS'13*, pages 237–248, 2013.

[18] C. Beeri, R. Fagin, D. Maier, A. Mendelzon, J. Ullman, and M. Yannakakis. Properties of acyclic database schemes. In *Proc. of STOC'81*, pages 355–362, 1981.

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

[20] P. A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

[21] U. Bertelé and F. Brioschi. *Nonserial Dynamic Programming*. Academic Press, 1972.

[22] H. Bodlaender. Treewidthlib. http://www.cs.uu.nl/research/projects/treewidthlib/.

[23] H. L. Bodlaender. A Linear Time Algorithm for Finding Tree-decompositions of Small Treewidth. In *Proc. of STOC'93*, pages 226–234, 1993.

[24] H. L. Bodlaender, F. V. Fomin, A. M. C. A. Koster, D. Kratsch, and D. M. Thilikos. On exact algorithms for

treewidth. *ACM Transactions on Algorithms*, 9(1):12:1–12:23, 2012.

[25] L. Bordeaux, Y. Hamadi, and P. Kohli, editors. *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 2014.

[26] A. A. Bulatov. The complexity of the counting constraint satisfaction problem. *Journal of the ACM*, 60(5):34:1–34:41, 2013.

[27] A. A. Bulatov, V. Dalmau, M. Grohe, and D. Marx. Enumerating homomorphisms. *Journal of Computer and System Sciences*, 78(2):638–650, 2012.

[28] A. A. Bulatov, M. Dyer, L. A. Goldberg, M. Jerrum, and C. Mcquillan. The expressibility of functions on the boolean domain, with applications to counting CSPs. *Journal of the ACM*, 60(5):32:1–32:36, 2013.

[29] G. Chalkiadakis, G. Greco, and E. Markakis. Characteristic function games with restricted agent interactions: Core-stability and coalition structures. *Artificial Intelligence*, 232:76–113, 2016.

[30] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. of STOC'77*, pages 77–90, 1977.

[31] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Theoretical Computer Science*, 239(2):211–229, 2000.

[32] H. Chen and V. Dalmau. Beyond hypertree width: Decomposition methods without decompositions. In *Proc. of CP'05*, pages 167–181, 2005.

[33] H. Chen and S. Mengel. A trichotomy in the complexity of counting answers to conjunctive queries. In *Proc. of ICDT'15*, pages 110–126, 2015.

[34] X. Chen, X. Deng, and S.-H. Teng. Settling the complexity of computing two-player Nash equilibria. *Journal of the ACM*, 56(3):14:1–14:57, 2009.

[35] S. Chu, M. Balazinska, and D. Suciu. From theory to practice: Efficient join query evaluation in a parallel database system. In *Proc. of SIGMOD'15*, pages 63–78, 2015.

[36] D. A. Cohen and P. Jeavons. The complexity of constraint languages. In *Handbook of Constraint Programming*, pages 245–280. 2006.

[37] D. A. Cohen, P. Jeavons, and M. Gyssens. A unified theory of structural tractability for constraint satisfaction problems. *Journal of Computer and System Sciences*, 74(5):721–743, 2008.

[38] W. Cook and P. Seymour. Tour merging via branch-decomposition. *INFORMS Joural on Computing*, 15(3):233–248, 2003.

[39] P. Cramton, Y. Shoham, and R. Steinberg, editors. *Combinatorial Auctions*. MIT Press, 2006.

[40] V. Dalmau and P. Jonsson. The complexity of counting homomorphisms seen from the other side. *Theory of Computing Systems*, 329(1-3):315–323, 2004.

[41] V. Dalmau, P. G. Kolaitis, and M. Y. Vardi. Constraint satisfaction, bounded treewidth, and finite-variable logics. In *Proc. of CP'02*, pages 310–326, 2002.

[42] C. Daskalakis, P. W. Goldberg, and C. H. Papadimitriou. The complexity of computing a Nash equilibrium. In *Proc. of STOC'06*, pages 71–78, 2006.

[43] J. Dean and S. Ghemawat. Mapreduce: a flexible data processing tool. *Commununications of the ACM*, 53(1):72–77, 2010.

[44] R. Dechter. *Constraint Processing*. Morgan Kaufmann Publishers Inc., 2003.

[45] R. Dechter, N. Flerova, and R. Marinescu. Search Algorithms for M Best Solutions for Graphical Models. In *Proc. of AAAI'12*, pages 1895–1901, 2012.

[46] R. Dechter and J. Pearl. Tree Clustering for Constraint Networks. *Artificial Intelligence*, 38(3):353–366, 1989.

[47] A. Dermaku, T. Ganzow, G. Gottlob, B. McMahan, N. Musliu, and M. Samer. Heuristic methods for hypertree decomposition. In *Proc. of MICAI'08*, pages 1–11, 2008.

[48] M. Elberfeld, A. Jakoby, and T. Tantau. Logspace Versions of the Theorems of Bodlaender and Courcelle. In *Proc. of FOCS'10*, pages 143–152, 2010.

[49] A. Fabrikant, C. Papadimitriou, and K. Talwar. The complexity of pure Nash equilibria. In *Proc. of STOC'04*, pages 604–612, 2004.

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

[51] V. Fionda and G. Greco. The complexity of mixed multi-unit combinatorial auctions: Tractability under structural and qualitative restrictions. *Artificial Intelligence*, 196:1–25, 2013.

[52] E. C. Freuder. A sufficient condition for backtrack-bounded search. *Journal of the ACM*, 32(4):755–761, 1985.

[53] W. Gansterer and T. Korimort. Matrix reordering by hypertree decomposition. *AURORA TR2003-19*, 2003.

[54] H. Garcia-Molina, J. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice-Hall, Inc., 2008.

[55] H. Garcia-Molina, J. Widom, and J. D. Ullman. *Database System Implementation*. Prentice-Hall, Inc., 1999.

[56] L. Ghionna, L. Granata, G. Greco, and F. Scarcello. Hypertree decompositions for query optimization. In *Proc. of ICDE'07*, pages 36–45, 2007.

[57] L. Ghionna, G. Greco, and F. Scarcello. H-DB: A Hybrid Quantitative-structural SQL Optimizer. In *Proc. of CIKM '11*, pages 2573–2576, 2011.

[58] N. Goodman and O. Shmueli. Characterizations of tree database schemas. *TR, Harvard University*, 1981.

[59] N. Goodman and O. Shmueli. Tree queries: A simple class of relational queries. *ACM Transactions on Database Systems*, 7(4):653–677, 1982.

[60] G. Gottlob and G. Greco. Decomposing combinatorial auctions and set packing problems. *Journal of the ACM*, 60(4):24:1–24:39, 2013.

[61] G. Gottlob, G. Greco, and B. Marnette. Hyperconsistency width for constraint satisfaction: Algorithms and complexity results. In *Graph Theory, Computational Intelligence and Thought*, pages 87–99, 2009.

[62] G. Gottlob, G. Greco, Z. Miklós, F. Scarcello, and T. Schwentick. Tree projections: Game characterization and computational aspects. In *Graph Theory, Computational Intelligence and Thought*, pages 217–226, 2009.

[63] G. Gottlob, G. Greco, and F. Scarcello. Pure Nash equilibria: hard and easy games. *Journal of Artificial Intelligence Research*, pages 357–406, 2005.

[64] G. Gottlob, G. Greco, and F. Scarcello. Tractable optimization problems through hypergraph-based structural restrictions. In *Proc. of ICALP'09*, pages 16–30, 2009.

[65] G. Gottlob, G. Greco, and F. Scarcello. Treewidth and Hypertree Width. In L. Bordeaux, Y. Hamadi, and P. Kohli, editors, *Tractability: Practical Approaches to Hard Problems*. 2012.

[66] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree decompositions: Structure, algorithms, and applications. In *Proc. of WG'05*, pages 1–15, 2005.

[67] G. Gottlob, S. T. Lee, G. Valiant, and P. Valiant. Size and Treewidth Bounds for Conjunctive Queries. *Journal of the ACM*, 59(3):1–35, 2012.

[68] G. Gottlob, N. Leone, and F. Scarcello. Advanced parallel algorithms for acyclic conjunctive queries. Technical Report DBAI-TR-98/18, Technical University of Vienna, 1998.

[69] G. Gottlob, N. Leone, and F. Scarcello. On tractable queries and constraints. In *Proc. of DEXA'99*, pages 1–15, 1999.

[70] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artificial Intelligence*, 124(2):243–282, 2000.

[71] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions: A survey. In *Proc. of MFCS'01*, pages 37–57, 2001.

[72] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498, 2001.

[73] G. Gottlob, N. Leone, and F. Scarcello. Computing LOGCFL certificates. *Theoretical Computer Science*, 270(1-2):761–777, 2002.

[74] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *Journal of Computer and System Sciences (conference version has appeared in PODS'99)*, 64(3):579–627, 2002.

[75] G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals, and guards: Game theoretic and logical characterizations of hypertree width. *Journal of Computer and System Sciences*, 66(4):775–808, 2003.

[76] G. Gottlob, Z. Miklós, and T. Schwentick. Generalized hypertree decompositions: NP-hardness and tractable variants. *Journal of the ACM*, 56(6):30:1–30:32, 2009.

[77] G. Gottlob and M. Samer. A backtracking-based algorithm for hypertree decomposition. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[78] G. Gottlob, L. Tanca, and S. Ceri. *Logic Programming and Databases*. Springer-Verlag, 1990.

[79] G. Greco, F. Lupia, and F. Scarcello. Structural tractability of Shapley and Banzhaf values in allocation games. In *Proc. of IJCAI'15*, pages 547–553, 2015.

[80] G. Greco and F. Scarcello. On the complexity of constrained Nash equilibria in graphical games. *Theoretical Computer Science*, 410(38-40):3901–3924, 2009.

[81] G. Greco and F. Scarcello. On the power of structural decompositions of graph-based representations of constraint problems. *Artificial Intelligence*, 174(5-6):382–409, 2010.

[82] G. Greco and F. Scarcello. The power of tree projections: Local consistency, greedy algorithms, and larger islands of tractability. In *Proc. of PODS'10*, pages 327–338, 2010.

[83] G. Greco and F. Scarcello. Structural tractability of constraint optimization. In *Proc. of CP'11*, pages 340–355, 2011.

[84] G. Greco and F. Scarcello. Tree projections and structural decomposition methods: The power of local consistency. *CoRR*, abs/1205.3321, 2012.

[85] G. Greco and F. Scarcello. Structural tractability of enumerating CSP solutions. *Constraints*, 18(1):38–74, 2013.

[86] G. Greco and F. Scarcello. Counting solutions to conjunctive queries: Structural and hybrid tractability. In *Proc. of PODS'14*, pages 132–143, 2014.

[87] G. Greco and F. Scarcello. Tree projections and structural decomposition methods: Minimality and game-theoretic characterization. *Theoretical Computer Science*, 522:95–114, 2014.

[88] G. Greco and F. Scarcello. Greedy strategies and larger islands of tractability for conjunctive queries and constraint satisfaction problems. *CoRR*, abs/1603.09617, 2016.

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

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

[91] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Web Semantics*, 3(2-3):158–182, 2005.

[92] M. Gyssens, P. G. Jeavons, and D. A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66(1):57–89, 1994.

[93] Z. Habbas, K. Amroun, and D. Singer. A forward-checking algorithm based on a generalised hypertree decomposition for solving non-binary constraint satisfaction problems. *Journal of Experimental and Theoretical Artificial Intelligence*, 27(5):649–671, 2015.

[94] I. F. Ilyas, G. Beskales, and M. A. Soliman. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Computing Surveys*, 40(4):11:1–11:58, 2008.

[95] R. Impagliazzo, R. Paturi, and F. Zane. Which problems have strongly exponential complexity? *Journal of Computer and System Sciences*, 63(4):512–530, 2001.

[96] M. Joglekar, R. Puttagunta, and C. Ré. Aggregations over generalized hypertree decompositions. In *Proc. of PODS'16*, 2016.

[97] M. R. Joglekar and C. M. Ré. It's All a Matter of Degree: Using Degree Information to Optimize Multiway Joins. In *Proc. of ICDT'16*, volume 48, pages 11:1–11:17, 2016.

[98] O. Kalinsky, Y. Etsion, and B. Kimelfeld. Flexible caching in trie joins. *CoRR*, abs/1602.08721, 2016.

[99] S. Karakashian, R. J. Woodward, and B. Y. Choueiry. Improving the performance of consistency algorithms by localizing and bolstering propagation in a tree decomposition. In *Proc. of AAAI'13*, 2013.

[100] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science (Vol. A)*, pages 869–941. MIT Press, 1990.

[101] K. Kask, R. Dechter, J. Larrosa, and A. Dechter. Unifying tree decompositions for reasoning in graphical models. *Artificial Intelligence*, 166(1-2):165–193, 2005.

[102] M. J. Kearns, M. L. Littman, and S. P. Singh. Graphical models for game theory. In *Proc. of UAI'01*, pages 253–260, 2001.

[103] M. A. Khamis, H. Ngo, and D. Suciu. Worst-case optimal algorithms for conjunctive queries with functional dependencies. In *Proc. of PODS'16*, 2016.

[104] M. A. Khamis, H. Q. Ngo, C. Ré, and A. Rudra. Joins via geometric resolutions: Worst-case and beyond. In *Proc. of PODS'15*, pages 213–228, 2015.

[105] M. A. Khamis, H. Q. Ngo, and A. Rudra. FAQ: questions asked frequently. In *Proc. of PODS'16*, 2016.

[106] B. Kimelfeld and Y. Sagiv. Incrementally computing ordered answers of acyclic conjunctive queries. In *Proc. of NGITS'06*, pages 141–152, 2006.

[107] P. G. Kolaitis. Constraint satisfaction, databases, and logic. In *Proc. of IJCAI'03*, pages 1587–1595, 2003.

[108] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *Journal of Computer and System Sciences*, 61(2):302–332, 2000.

[109] T. Korimort. *Heuristic Hypertree Decomposition*. PhD thesis, Technical University of Vienna, Vienna, 2003.

[110] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[111] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logics*, 7(3):499–562, 2006.

[112] F. Lupia. *Constraint Satisfaction: Algorithms, Complexity Results, and Applications*. PhD thesis, University of Calabria, 2015.

[113] D. Marx. Approximating fractional hypertree width. *ACM Transactions on Algorithms*, 6(2):29:1–29:17, 2010.

[114] D. Marx. Tractable structures for constraint satisfaction with truth tables. *Theory of Computing Systems*, 48(3):444–464, 2011.

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

[116] N. Musliu. Generation of tree decompositions by iterated local search. In *Proc. of EvoCOP'07*, pages 130–141, 2007.

[117] J. Nash. Non-cooperative games. *Annals of Mathematics*, 54(2):286–295, 1951.

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

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

[120] L. Otten and R. Dechter. Bounding Search Space Size via (Hyper) tree Decompositions. In *Proc. of UAI'08*, 2008.

[121] R. Pichler and S. Skritek. Tractable counting of the answers to conjunctive queries. *Journal of Computer and System Sciences*, 79(6):984–1001, 2013.

[122] O. Reingold. Undirected connectivity in log-space. *Journal of the ACM*, 55(4):17:1–17:24, 2008.

[123] N. Robertson and P. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, 1986.

[124] N. Robertson and P. D. Seymour. Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, 36(1):49–64, 1984.

[125] M. H. Rothkopf, A. Pekec, and R. M. Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44:1131–1147, 1998.

[126] Y. Sagiv and O. Shmueli. Solving queries by tree projections. *ACM Transactions on Database Systems*, 18(3):487–511, 1993.

[127] M. Samer. Hypertree-decomposition via branch decomposition. In *Proc. of IJCAI'05*, pages 1535–1536, 2005.

[128] F. Scarcello. Hypertree decompositions homepage. http://www.dimes.unical.it/scarcello/Hypertrees/.

[129] F. Scarcello, G. Gottlob, and G. Greco. Uniform constraint satisfaction problems and database theory. In *Complexity of Constraints*, pages 156–195. Springer-Verlag, 2008.

[130] F. Scarcello, G. Greco, and N. Leone. Weighted hypertree decompositions and optimal query plans. *Journal of Computer and System Sciences*, 73(3):475–506, 2007.

[131] L. Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Records*, 44(1):10–17, 2015.

[132] S. Subbarayan and H. R. Andersen. Backtracking procedures for hypertree, hyperspread and connected hypertree decomposition of CSPs. In *Proc. of IJCAI'07*, pages 180–185, 2007.

[133] R. E. Tarjan and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM Journal on Computing*, 13(3):566–579, 1984.

[134] J. D. Ullman. *Principles of Database and Knowledge-Base Systems: Volume II: The New Technologies*. W. H. Freeman & Co., 1990.

[135] L. G. Valiant. A bridging model for parallel computation. *Commununications of the ACM*, 33(8):103–111, 1990.

[136] T. L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *Proc. of ICDT'14*, pages 96–106, 2014.

[137] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. of VLDB'81*, pages 82–94, 1981.