Structural Decomposition Methods: Key Notions and Database Applications

G. Greco, N. Leone, F. Scarcello and G. Terracina

Abstract Many difficult problems that are tractable when restricted to acyclic instances are good candidates to be solved efficiently whenever their structure is not precisely acyclic, but not far from that. This is the case for fundamental database problems such as answering conjunctive queries or counting the number of answers (without actually computing them). The chapter describes structural decomposition methods that guarantee tractability for all such problem instances whose associated hypergraphs have a small degree of cyclicity, called width. In particular, it focuses on the notion of hypertree width, by describing its properties and its applications to the database field, and covering queries with aggregate operators and some recent parallel and distributed implementations.

1 Introduction

Answering conjunctive queries to relational databases is a basic problem in database theory, and it is equivalent to many other fundamental problems, such as conjunctive query containment and constraint satisfaction. Recall that conjunctive queries are defined through conjunctions of atoms (without negation), and are known to be equivalent to Select-Project-Join queries. The problem of evaluating such queries is NP-hard in general, but it is feasible in polynomial time on the class of acyclic queries, which was the subject of many seminal research works since the early ages of database theory. This class contains all queries Q whose associated query hypergraph $\mathcal{H}_Q$ is *acyclic*, where the hypergraph $\mathcal{H}_Q$ associated to Q has as vertices

G. Greco (✉) · N. Leone · F. Scarcello · G. Terracina
University of Calabria, 87036 Rende, Italy
e-mail: ggreco@mat.unical.it

N. Leone
e-mail: leone@mat.unical.it

F. Scarcello
e-mail: scarcello@dimes.unical.it

G. Terracina
e-mail: terracina@mat.unical.it

S. Flesca et al. (eds.), *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years*, Studies in Big Data 31, DOI 10.1007/978-3-319-61893-7_15

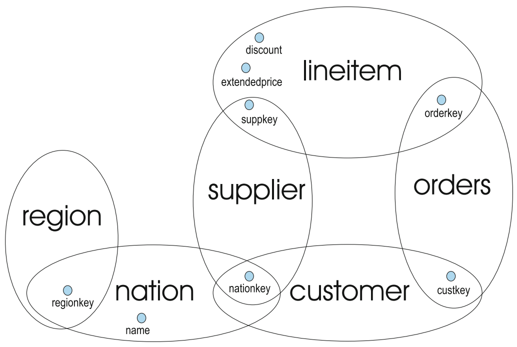

Fig. 1 Hypegraph associated with the query in Example 1

its variables and has, for each query atom, a hyperedge consisting of the set of all variables appearing in that atom.

Example 1 Consider the following conjunctive query Q, adapted from a benchmark SQL query taken from the TPC-H specifications (see [21]):

$$\begin{array}{l}\mathsf{customer}(\mathsf{CustKey}, \mathsf{NationKey}) \ \wedge \ \mathsf{orders}(\mathsf{OrdKey}, \mathsf{NationKey}) \\ \wedge\ \mathsf{lineitem}(\mathsf{SuppKey}, \mathsf{OrdKey}, \mathsf{ExtendedPrice}, \mathsf{Discount}) \\ \wedge \mathsf{supplier}(\mathsf{SuppKey}, \mathsf{NationKey}) \ \wedge \ \mathsf{region}(\mathsf{RegionKey}) \\ \wedge \mathsf{nation}(\mathsf{Name}, \mathsf{NationKey}, \mathsf{RegionKey}),\end{array}$$

The query consists of the conjunction of 6 atoms and its associated hypergraph $\mathcal{H}(Q)$ is the one shown in Fig. 1. ◁

To be precise, there are several notions of hypergraph acyclicity, among which this chapter focuses on the most liberal one, known as α-acyclicity (cf. [18]). According to this notion [7], a hypergraph $\mathcal{H}$ is acyclic if it has a *join tree* $JT(\mathcal{H})$, that is a tree whose vertices are the hyperedges of $\mathcal{H}$ and such that, whenever the same node X 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 $JT(\mathcal{H})$. Note that deciding whether a hypergraph is acyclic is feasible in linear time [61], and also in deterministic logspace. This latter property follows from the fact that hypergraph acyclicity belongs to symmetric logspace [30], and that this class is equal to deterministic logspace [58].

It is well-known that *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 processing the query efficiently using a smart algorithm, such as Yannakakis' algorithm [65], 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. 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 [51]—and for equivalent problems, such as conjunctive query containment, constraint satisfaction problems, and so on [30].

As a matter of fact, however, conjunctive queries arising in practical applications are not properly acyclic, in many cases. For instance, the query introduced in Example 1 is not acyclic, as it can be checked easily by looking at its hypergraph in Fig. 1. Therefore, significant efforts have been made since the nineties to define appropriate notions of "quasi-acyclicity", leading to identify classes of conjunctive queries over which efficient evaluation algorithms can still be singled out. In this context, the degree of cyclicity of a hypergraph (or of a corresponding query) is usually referred to as its *width* and, for each fixed width $k \geq 1$, one seeks notions enjoying the following fundamental three conditions:

(i) **Generalization of Acyclicity**: Queries of width k include the acyclic ones.
(ii) **Tractable Recognizability**: Queries of width k can be recognized in polynomial time.
(iii) **Tractable Query-Answering**: Queries of width k can be answered in input–output polynomial time (that is, with respect to the size of the input and the output). Moreover, tractability holds even by considering the so-called combined complexity, where both the query and the database are taken into account, and nothing is assumed to be fixed (or small).

The rest of the chapter is devoted to illustrate *structural decomposition methods* proposed in the literature to generalize acyclic queries, in particular, by focusing on *tree decompositions* and *hypertree decompositions*. Moreover, applications in the database area are discussed, and some recent advances and directions for future work are illustrated.

2 Tree Decompositions

The notion of tree decomposition [59] represents a significant success story in Computer Science (see, e.g., [25]). The associated notion of *treewidth* was meant to provide a measure of the degree of cyclicity in graphs and hypergraphs.[1]

Formally, a *tree decomposition* [59] of a hypergraph $\mathcal{H}$, where $nodes(\mathcal{H})$ and $edges(\mathcal{H})$ denotes its set of nodes and of edges, respectively, 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 the hypergraph $\mathcal{H}$, denoted by $tw(\mathcal{H})$, is the minimum width over all its tree decompositions.

[1]Some notions strongly related to the treewidth appeared even before the 80's in the literature. For a detailed story, we refer to [16].

The notion of treewidth enjoys some desirable properties. First, it is efficiently recognizable (cf. condition *(ii)* in the Introduction). Indeed, for any constant k, determining whether a hypergraph has treewidth k is feasible in linear time [8]. Moreover, condition *(iii)* is satisfied, too. In particular, Boolean conjunctive queries (whose associated hypergraphs are) of treewidth k can be answered 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 the given database.

However, treewidth is not a proper generalization of hypergraph acyclicity, that is, the notion does not satisfy condition *(i)*. Indeed, the notion of treewidth is essentially aimed at characterizing nearly-acyclic *graphs*, rather than hypergraphs. In fact, it is easy to check that a graph is acyclic if, and only if, it has treewidth 1. More in detail, 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 $edges(\mathcal{H})$. So, the tree decomposition method obscures, in many cases, the actual degree of cyclicity of the query hypergraph. For instance, by looking at the primal graph of $\mathcal{H}$, there is no way to understand whether a given clique over three variables comes from one atom having arity 3 in the original query or, instead, it comes from the interaction of three binary atoms.

3 Hypertree Decompositions

The notion of hypertree decomposition has been proposed in the literature as a generalization of the tree decomposition method, specifically designed to deal with query hypergraphs [32, 33]. The idea is to use the power of hyperedges, which may involve many variables at once, contrasted with tree decompositions, which are based just on single variables.

3.1 Basic Notions

In order to define hypertree decompositions, a more general related notion is first introduced.

A *generalized hypertree decomposition* [32] 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.

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 $ghw(\mathcal{H})$, is the minimum width over all its generalized hypertree decompositions. A class of hypergraphs has bounded generalized hypertree

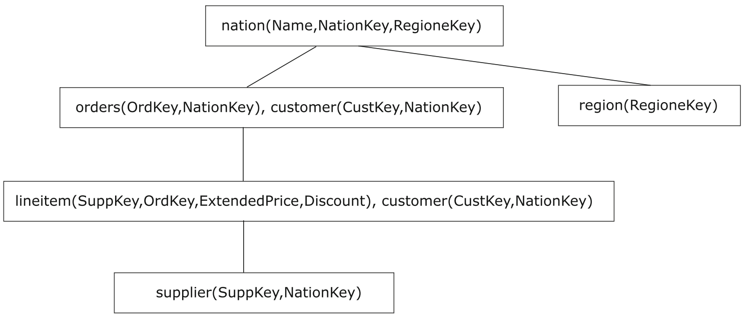
nation(Name,NationKey,RegioneKey)
orders(OrdKey,NationKey), customer(CustKey,NationKey)
region(RegioneKey)
lineitem(SuppKey,OrdKey,ExtendedPrice,Discount), customer(CustKey,NationKey)
supplier(SuppKey,NationKey)

Fig. 2 A hypertree decomposition of $\mathcal{H}(Q)$

width if every hypergraph in the class has generalized hypertree width at most k, for some (finite) natural number k.

Example 2 Consider again the query Q in Example 1 and the generalized hypertree decomposition that is depicted in Fig. 2. We use an intuitive graphical notation: for each vertex p of the decomposition, the figure reports the atoms corresponding to the hyperedges in $\lambda(p)$. It can be checked that this is indeed a generalized hypertree decomposition and its width is 2.

In the example, $\chi(p) = \bigcup_{h \in \lambda(v)} h$, that is, each variable of any atom occurring in p is included in $\chi(p)$, too. If this is not true, an anonymous variable "__" is used in place of any variable occurring in some atom in $\lambda(p)$, but not occurring in the set of *relevant* variables $\chi(p)$. For instance, an alternative decomposition may be obtained by replacing customer(CustKey,NationKey) with customer(__,NationKey) in the vertex covering the atoms lineitem and customer. $\lhd$

The notion of generalized hypertree width is a true generalization of acyclicity, as the acyclic hypergraphs are precisely those hypergraphs having generalized hypertree width 1 [33]. Hence, condition *(i)* is satisfied. Moreover, given a query Q and a width-k generalized hypertree decomposition of its hypergraph, Q can be answered in polynomial time [32]. In particular, if Q is Boolean than it 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. In the general case of queries with output variables (whose results may consist of exponentially many tuples), the input–output polynomial time bound is $O(v \cdot (r^k + s) \cdot \log(r + s))$, where s is the number of output tuples. Note that the exponent of the polynomial in the upper bound involves k as a factor, so that in practice the method can be used only for small degrees of cyclicity. With this respect, it is worthwhile noting that the factor k in the exponent cannot be avoided, unless some unlikely collapse occurs in parameterized complexity theory.

However, condition *(ii)* is not satisfied by this method, since it is NP-hard to decide whether a given query has generalized hypertree width bounded by a fixed constant k, even for $k = 2$ [19]. In fact, an additional restriction has to be added in the definition of generalized hypertree decomposition, in order to get a tractable notion.

A *hypertree decomposition* [32] of a hypergraph $\mathcal{H}$ is a generalized hypertree decomposition of $\mathcal{H}$ that satisfies the following additional 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 $\chi(T_p)$ is the set of all variables occurring in the χ labeling of this subtree. Then, the *hypertree width* of $\mathcal{H}$, denoted by $hw(\mathcal{H})$, is naturally defined as the minimum width over all its hypertree decompositions.

Intuitively, the above technical condition forces variables to be included in a χ label the first time (looking top-down) some atom where they occur is used in the decomposition. A very important property of this notion is that it is not very far apart from the notion of generalized hypertree width. Indeed, for each hypergraph $\mathcal{H}$, $ghw(\mathcal{H}) \leq hw(\mathcal{H}) \leq 3 \cdot ghw(\mathcal{H}) + 1$ [3]. In particular, this entails that a class of queries has bounded generalized hypertree width if, and only if, it has bounded hypertree width.

Example 3 Since $\chi(p) = \bigcup_{h \in \lambda(v)} h$ holds for each vertex p in the generalized hypertree decomposition of Fig. 2, we trivially derive that this also a hypertree decomposition of the (hypergraph associated with the) query in Example 1. $\lhd$

3.2 Desirable Properties of Hypertree Decompositions

Since their introduction, hypertree decompositions have received considerable attention in the literature, due to the fact that they satisfy the fundamental conditions stated in the Introduction (see also, [23, 39]):

(i) Hypertree width properly generalizes hypergraph acyclicity. In particular, a hypergraph $\mathcal{H}$ is acyclic if, and only if, $hw(\mathcal{H}) = 1$ [32].

(ii) For a fixed constant k, it can be checked in polynomial time whether (the hypergraph associated with) a conjunctive query Q has hypertree width $k' \leq k$. Moreover, if so, a hypertree decomposition of width k' can be computed in polynomial time, more precisely in $O(m^{2k} v^2)$, where m and v are the number of atoms and the number of variables in Q, respectively [29]. The decision and computation problems are, moreover, at a very low level of computational complexity and are highly parallelizable, as they belong to LOGCFL (see Sect. 4.2 for some properties of this class). Again, it is deemed very unlikely that we can get rid of the factor k in the exponent, because deciding whether a query has hypertree width k is fixed-parameter intractable [26].

(iii) Queries having bounded hypertree width can be answered in input–output polynomial time. This follows immediately from the same property of generalized hypertree width, however in this case it is not necessary that the decomposition is provided in input with the query, because it can be computed easily, by property (ii).

4 Applications of Hypertree Decompositions

In this section, we overview a number of database applications where the notion of hypertree decomposition has been used profitably.

4.1 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 [56, 64]. 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 [5] 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 deal with this issue, EmptyHeaded additionally features a query compiler based on hypertrees: it searches for a generalized hypertree decomposition having the minimum possible estimated size for the intermediate results, 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.

We also mention a different approach designed for the Leapfrog Trie Join algorithm [64], where (hyper)tree decompositions are used to guide a flexible caching of intermediate results [44]. The algorithm described in [47] considers also possible functional dependencies, by using the coloring number bound of [27].

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, without using a dynamic programming approach á la Yannakakis. This is the approach described in [48], 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.

4.2 Parallel and Distributed Evaluation

From a computational complexity viewpoint, evaluating Boolean queries having bounded hypertree-width is LOGCFL-complete (even for binary acyclic queries) [30]. Combining the result with the techniques discussed by [31], 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.

Several efforts have been spent in the literature to translate the above theoretical results into practical implementation of algorithms for parallel (and distributed) query evaluation. In particular, a parallel algorithm, called DB-SHUNT, has been defined in [30] for answering Boolean acyclic queries—an extension, called ACQ, which is able to deal with acyclic queries with output variables is discussed in [28]. This algorithm is well-suited for bounded hypertree-width queries after a suitable preprocessing step where they are transformed in acyclic queries. In this step, for each vertex p in the decomposition tree T, a fresh atom is computed such that its set of variables is $\chi(p)$, and 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. Then, 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 [45]. This way, any tree can be contracted to a single node by a logarithmic number 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 having a fixed number of processors $c < m/2$, 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 [4] for the Valiant's bulk synchronous parallel (BSP) computational model [63], 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 [4], which is a distributed and generalized version of Yannakakis' algorithm specifically designed for MapReduce [15] (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 to be executed.

4.3 Counting Query Answers

So far, the chapter focused on the use of hypertree decompositions for evaluating conjunctive queries, that is, Select-Project-Join queries. The method, however, found applications also in the evaluation of further kinds of queries involving more complex constructs. A noticeable example is the use of hypertree decompositions to deal with "COUNT" aggregates (see, e.g., [9, 10]).

In this context, the query evaluation problem can be abstractly reformulated as *counting* the number of answers of a given conjunctive query. The challenge is then 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 [57]. Moreover, on fixed-arity queries, this condition is also necessary (under widely believed assumptions in parameterized complexity) [14]. However, in presence of projections, classical decomposition methods are not helpful. Indeed, even for acyclic queries [57], 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 [57]. 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. [37]). 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 [37] 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 [12].

4.4 Further Aggregations

When evaluating queries, database management systems list answers according to some arbitrary order, which reflects the algorithms internally used for optimization purposes. However, this is often unsatisfactory, so that users often write ORDER BY clauses to force answers to be listed according to their preferences. For instance, interactive queries are typically ORDER BY queries, where users are interested in a few best answers (to be returned very fast). In this context, a crucial observation is that 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 operator (MAX) in SQL queries. This was first pointed out in the context of ranking solutions to discrete optimization problems [52]. 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 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 [41].

A number of structural tractability results for MAX (so, for top-K and ORDER BY) queries are known in the literature (see, e.g., [24]). 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 [17], and over queries that have bounded generalized hypertree width [22, 50]. Structural tractability results for extensions to certain *non-monotone* functions which manipulate "small" (in fact, polynomially-bounded) values have been studied, too [24, 36].

An implementation of some aggregation operators based on hypertree decompositions is described in [20, 21]. It can be used with any system at a logical level. For the sake of efficiency, a semi joinoperator that supports the execution of these operators over decompositions is implemented in a prototypal extension of the open-source DBMS PostgreSQL. Further operators are considered in [42] 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 [46]. A crucial step 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 [4] and the degree-based

MapReduce algorithm in [43]. Answering AJAR queries is equivalent to answering so-called Functional Aggregate Queries [49].

4.5 Parallel and Distributed Querying of Deductive Databases

This overview of applications of hypertree decompositions closes by focusing on problems related to querying distributed deductive databases, in particular, by considering a scenario where data *natively* resides on different *autonomous* sources, and it is necessary to deal with reasoning tasks via logic programming. This scenario has been considered in [6] and differs from the classic parallel evaluation of queries (e.g., [28, 30]) in several aspects: *(i)* the focus is the optimized evaluation of (normal, stratified) logic programs [55]; *(ii)* the input data are physically distributed over several databases, and this distribution must be considered in the optimization process; *(iii)* the integration of the reasoning engine responsible of orchestrating the querying process with the distributed database management systems must be tight enough to allow efficient interactions, but general enough to avoid limitations in the kind and location of databases.

In this scenario, the notion of hypertree decomposition has been shown to provide a powerful tool to address issue *(ii)*, whereas the reasoning engine DLV^{DB} presented in [62] has been used to transparently evaluate logic programs directly on generic database systems. In order to describe the optimization process discussed by [6], let us first consider programs composed of one rule.

A single logic rule r can be seen as a conjunctive query (possibly with negation), whose result must be stored in the head predicate. The optimized evaluation of r starts from the computation of a hypertree decomposition *HD* for this query, which is interpreted as a distributed query plan. To this end and, in particular, in order to take into account data distribution, each node p of the hypertree *HD* is labelled with the database where the partial data associated with it reside. Formally, let $Site(p)$ denote the site associated with the node p in *HD* and let $net(Site(p), Site(p'))$ be the unitary data transfer cost from $Site(p)$ to $Site(p')$. Let $\lambda(p)$ be the set of atoms referred by p, and $\chi(p)$ the variables covered by p. Then, $Site(p)$ is chosen among the databases where the relations in $\lambda(p)$ reside by computing:

$h_m = arg\ min_{h_i \in \lambda(p)}\ \{\Sigma_{h_j \in \lambda(p)} |rel(h_j)| \times net(Site(h_j), Site(h_i))\}$.

Thus, $Site(p)$ is chosen to be the site where the database with h_m is stored, and this choice minimizes the cost of transferring all data required to evaluate the subproblem associated with $\lambda(p)$.

Eventually, in order to optimize the distribution of rule evaluation, the adopted cost function is further refined in [6] by taking into account other aspects, such as the estimated cardinality of join results. The query decomposition strategy, in addition, includes options like moving data from one site to another, and changing the order of joins based on data localization. Once the hypergraph $\mathcal{H}_r$ for r is obtained and the above cost function is defined, a hypertree decomposition HD_r minimizing such function is computed via the algorithm discussed in [60]. Then, a distributed plan for r is composed accordingly. Intuitively, the plan is built in such a way that it evaluates joins bottom-up, from the leaves to the root, by suitably transferring data if the sites of a child node and its father are different. Independent sub-trees are executed in parallel processes.

When the program is composed of several rules, three further kind of optimizations can be applied: *(i)* rule unfolding optimization, *(ii)* inter-components optimization, *(iii)* intra-component optimization. Since the rule optimization strategy is particularly suited for rules having long bodies, in presence of filters (or queries) in the program, an unfolding [55] optimization step is carried out to make rule bodies as long as possible, and to reduce their number. In particular, the inter-components optimization [11] is in charge of dividing the input (possibly unfolded) program $\mathcal{P}$ into subprograms, according to the dependencies among the predicates occurring in it, and by identifying which of them can be evaluated in parallel. This allows for concurrently evaluating rules involved within the same component.

5 Further Methods and Conclusions

The chapter describes structural decomposition methods aimed at defining suitable generalizations of hypergraph acyclicity, by focusing in particular on hypertree decompositions. The natural research question to be addressed is whether it is possible to go beyond this method, by defining more general notions the still retain the desirable conditions pointed out in the Introduction.

Observe that in the λ labelling of hypertree decompositions one could use as available resources all subsets of the hyperedges of a hypergraph $\mathcal{H}$, instead of using just the original hyperedges. This way, it is possible to recover the full power of generalized hypertree decompositions, but with an exponential blow-up. If, on the other hand, only polynomially many subsets are used, it is possible to obtain a number of variants of hypertree decompositions that still fulfil conditions *(i)*, *(ii)*, and *(iii)*. This is precisely the idea of subset-based decompositions as defined in [34]. Specific subset-based decompositions are the *component decompositions* defined in [34] and the *spread-cut* decompositions defined in [13], while an interesting way of computing a subset-based decomposition *dynamically* provides the new notion of *greedy hypertree decomposition* [35, 38].

A more radical approach to improve HDs, and even GHDs, was taken by Grohe and Marx [40], who defined *fractional hypertree decompositions (FHDs)*, giving rise to the notion of *fractional hypertree width* $fhw(\mathcal{H})$ of a hypergraph $\mathcal{H}$ (or of a query). The fractional hypertree width width is more general than the generalized hypertree width, and hence $fhw(\mathcal{H}) \leq ghw(\mathcal{H})$. Moreover, there are classes of hypergraphs having unbounded ghw, but bounded fhw. A problem with fractional hypertree decompositions is that they are intractable [19]. However, an approximate decomposition whose width is bounded by a cubic function of the fractional hypertree width can be computed in polynomial time [53]. A yet more general width-concept is the *submodular width* [54]. Unfortunately, recognizing queries of bounded submodular width is not known to be tractable. Moreover, having bounded submodular width does not guarantee a polynomial-time combined complexity as for hypertree width, but just fixed-parameter tractability where the query hypergraph is used as a parameter.

Acknowledgements The work was supported by project "Ba2Know (Business Analytics to Know) Service Innovation - LAB", No. PON03PE_00001_1 funded by the Italian Ministry of University and Research (MIUR), and by project "Smarter Solutions in the Big Data World (S2BDW)", funded by the Italian Ministry for Economic Development (MISE) within the programme PON "Imprese e competitivitá" 2014–2020.

References

1. C.R. Aberger, S. Tu, K. Olukotun, C. Ré, EmptyHeaded: A relational engine for graph processing, in *Proceedings of SIGMOD 2016* (2016)
2. C.R. Aberger, S. Tu, K. Olukotun, C. Ré, Old techniques for new join algorithms: A case study in RDF processing, in *CoRR*, arXiv:abs/1602.03557 (2016)
3. I. Adler, G. Gottlob, M. Grohe, Hypertree width and related hypergraph invariants. Eur. J. Comb. **28**(8), 2167–2181 (2007)
4. F.N. Afrati, M. Joglekar, C. Ré, S. Salihoglu, J.D. Ullman, GYM: A multiround join algorithm in mapreduce, in *CoRR*, arXiv:abs/1410.4156 (2014)
5. M. Aref, B. ten Cate, T.J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T.L. Veldhuizen, G. Washburn, Design and implementation of the logicblox system, in *Proceedings of SIGMOD 2015* (2015), pp. 1371–1382
6. R. Barilaro, F. Ricca, G. Terracina, Optimizing the distributed evaluation of stratified programs via structural analysis, in *Proceeding of 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011), Vancouver, Canada, 2011*, Lecture Notes in Computer Science (Springer, Heidelberg, 2011), pp. 217–222
7. P.A. Bernstein, N. Goodman, Power of natural semijoins. SIAM J. Comput. **10**(4), 751–771 (1981)
8. H.L. Bodlaender, A linear time algorithm for finding tree-decompositions of small treewidth, in *Proceeding of STOC 1993* (1993), pp. 226–234
9. A.A. Bulatov, The complexity of the counting constraint satisfaction problem. J. ACM **60**(5), 34:1–34:41 (2013)
10. A.A. Bulatov, M. Dyer, L.A. Goldberg, M. Jerrum, C. Mcquillan, The expressibility of functions on the boolean domain, with applications to counting CSPs. J. ACM **60**(5), 32:1–32:36 (2013)
11. F. Calimeri, S. Perri, F. Ricca, Experimenting with parallelism for the instantiation of ASP programs. J. Algorithms Cogn. Inf. Log. **63**(1–3), 34–54 (2008)
12. H. Chen, S. Mengel, A trichotomy in the complexity of counting answers to conjunctive queries, in *Proceeding of ICDT 2015* (2015), pp. 110–126
13. D.A. Cohen, P. Jeavons, M. Gyssens, A unified theory of structural tractability for constraint satisfaction problems. J. Comput. Syst. Sci. **74**(5), 721–743 (2008)
14. V. Dalmau, P. Jonsson, The complexity of counting homomorphisms seen from the other side. Theory Comput. Syst. **329**(1–3), 315–323 (2004)
15. J. Dean, S. Ghemawat, Mapreduce: A flexible data processing tool. Commun. ACM **53**(1), 72–77 (2010)
16. R. Dechter, *Constraint Processing* (Morgan Kaufmann Publishers Inc., 2003)
17. R. Dechter, N. Flerova, R. Marinescu, Search algorithms for M Best solutions for graphical models, in *Proceeding of AAAI 2012* (2012), pp. 1895–1901

18. R. Fagin, Degrees of acyclicity for hypergraphs and relational database schemes. J. ACM **30**(3), 514–550 (1983)
19. W. Fischl, G. Gottlob, R. Pichler, General and fractional hypertree decompositions: Hard and easy cases, in *CoRR*, arXiv:abs/1611.01090 (2016)
20. L. Ghionna, L. Granata, G. Greco, F. Scarcello, Hypertree decompositions for query optimization, in *Proceeding of ICDE 2007* (2007), pp. 36–45
21. L. Ghionna, G. Greco, F. Scarcello, H-DB: A hybrid quantitative-structural sql optimizer, in *Proceeding of CIKM 2011* (2011), pp. 2573–2576
22. G. Gottlob, G. Greco, Decomposing combinatorial auctions and set packing problems. J. ACM **60**(4), 24:1–24:39 (2013)
23. G. Gottlob, N. Greco, N. Leone, F. Scarcello, Hypertree decompositions: Questions and answers, in *Proceeding of PODS 2016* (2016), pp. 57–74
24. G. Greco, F. Scarcello, Tractable optimization problems through hypergraph-based structural restrictions, in *Proceeding of ICALP 2009* (2009), pp. 16–30
25. G. Gottlob, G. Greco, F. Scarcello, Treewidth and hypertree width, in *Tractability: Practical Approaches to Hard Problems*, ed. by L. Bordeaux, Y. Hamadi, P. Kohli (2012)
26. G. Gottlob, M. Grohe, N. Musliu, M. Samer, F. Scarcello, Hypertree decompositions: Structure, algorithms, and applications, in *Proceeding of WG 2005* (2005), pp. 1–15
27. G. Gottlob, S.T. Lee, G. Valiant, P. Valiant, Size and treewidth bounds for conjunctive queries. J. ACM **59**(3), 1–35 (2012)
28. G. Gottlob, N. Leone, F. Scarcello, Advanced parallel algorithms for acyclic conjunctive queries. Technical Report DBAI-TR-98/18, Technical University of Vienna (1998)
29. G. Gottlob, N. Leone, F. Scarcello, On tractable queries and constraints, in *Proceeding of DEXA 1999* (1999), pp. 1–15
30. G. Gottlob, N. Leone, F. Scarcello, The complexity of acyclic conjunctive queries. J. ACM **48**(3), 431–498 (2001)
31. G. Gottlob, N. Leone, F. Scarcello, Computing LOGCFL certificates. Theor. Comput. Sci. **270**(1–2), 761–777 (2002)
32. G. Gottlob, N. Leone, F. Scarcello, Hypertree decompositions and tractable queries. J. Comput. Syst. Sci. (Conference Version has Appeared in PODS 1999) **64**(3), 579–627 (2002)
33. G. Gottlob, N. Leone, F. Scarcello, Robbers, marshals, and guards: Game theoretic and logical characterizations of hypertree width. J. Comput. Syst. Sci. **66**(4), 775–808 (2003)
34. G. Gottlob, Z. Miklós, T. Schwentick, Generalized hypertree decompositions: NP-hardness and tractable variants. J. ACM **56**(6), 30:1–30:32 (2009)
35. G. Greco, F. Scarcello, The power of tree projections: Local consistency, greedy algorithms, and larger islands of tractability, in *Proceeding of PODS 2010* (2010), pp. 327–338
36. G. Greco, F. Scarcello, Structural tractability of constraint optimization, in *Proceeding of CP 2011* (2011), pp. 340–355
37. G. Greco, F. Scarcello, Counting solutions to conjunctive queries: Structural and hybrid tractability, in *Proceeding of PODS 2014* (2014), pp. 132–143
38. G. Greco, F. Scarcello, Greedy strategies and larger islands of tractability for conjunctive queries and constraint satisfaction problems. Inf. Comput. **252**, 201–220 (2017)
39. G. Greco, F. Scarcello, The power of local consistency in conjunctive queries and constraint satisfaction problems. SIAM J. Comput. (2017)
40. M. Grohe, D. Marx, Constraint solving via fractional edge covers. ACM Trans. Algorithms **11**(1), 4:1–4:20 (2014)
41. I.F. Ilyas, G. Beskales, M.A. Soliman, A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. **40**(4), 11:1–11:58 (2008)
42. M. Joglekar, R. Puttagunta, C. Ré, Aggregations over generalized hypertree decompositions, in *Proceeding of PODS 2016* (2016)
43. M.R. Joglekar, C.M. Ré, It's all a matter of degree: Using degree information to optimize multiway joins, in *Proceeding of ICDT 2016* (2016), pp. 11:1–11:17
44. O. Kalinsky, Y. Etsion, B. Kimelfeld, Flexible caching in trie joins, in *CoRR*, arXiv:abs/1602.08721 (2016)

45. R.M. Karp, V. Ramachandran, Parallel algorithms for shared-memory machines, in *Handbook of Theoretical Computer Science*, vol. A (MIT Press, 1990), pp. 869–941
46. K. Kask, R. Dechter, J. Larrosa, A. Dechter, Unifying tree decompositions for reasoning in graphical models. Artif. Intell. **166**(1–2), 165–193 (2005)
47. M.A. Khamis, H. Ngo, D. Suciu, Worst-case optimal algorithms for conjunctive queries with functional dependencies, in *Proceeding of PODS 2016* (2016)
48. M.A. Khamis, H.Q. Ngo, C. Ré, A. Rudra, Joins via geometric resolutions: Worst-case and beyond, in *Proceeding of PODS 2015* (2015), pp. 213–228
49. M.A. Khamis, H.Q. Ngo, A. Rudra. FAQ: Questions asked frequently, in *Proceedings of PODS 2016* (2016)
50. B. Kimelfeld, Y. Sagiv, Incrementally computing ordered answers of acyclic conjunctive queries, in *Proceedings of NGITS 2006* (2006), pp. 141–152
51. P.G. Kolaitis, Constraint satisfaction, databases, and logic, in *Proceedings of IJCAI 2003* (2003), pp. 1587–1595
52. E.L. Lawler, A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. Manag. Sci. **18**(7), 401–405 (1972)
53. D. Marx, Approximating fractional hypertree width. ACM Trans. Algorithms **6**(2), 29:1–29:17 (2010)
54. D. Marx, Tractable hypergraph properties for constraint satisfaction and conjunctive queries. J. ACM **60**(6), 42:1–42:51 (2013)
55. J. Minker (ed.), *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann Publishers Inc., Washington DC, 1988)
56. H.Q. Ngo, C. Ré, A. Rudra, Skew strikes back: New developments in the theory of join algorithms. SIGMOD Rec. **42**(4), 5–16 (2013)
57. R. Pichler, S. Skritek, Tractable counting of the answers to conjunctive queries. J. Comput. Syst. Sci. **79**(6), 984–1001 (2013)
58. O. Reingold, Undirected connectivity in log-space. J. ACM **55**(4), 17:1–17:24 (2008)
59. N. Robertson, P. Seymour, Graph minors. II. Algorithmic aspects of tree-width. J. Algorithms **7**(3), 309–322 (1986)
60. F. Scarcello, G. Greco, N. Leone, Weighted hypertree decompositions and optimal query plans. J. Comput. Syst. Sci. **73**(3), 475–506 (2007)
61. R.E. Tarjan, M. Yannakakis, Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. SIAM J. Comput. **13**(3), 566–579 (1984)
62. G. Terracina, N. Leone, V. Lio, C. Panetta, Experimenting with recursive queries in database and logic programming systems. Theory Pract. Log. Program. (TPLP) **8**(2), 129–165 (2008)
63. L.G. Valiant, A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (1990)
64. T.L. Veldhuizen, Triejoin: A simple, worst-case optimal join algorithm, in *Proceedings of ICDT 2014* (2014), pp. 96–106
65. M. Yannakakis, Algorithms for acyclic database schemes, in *Proceedings of VLDB 1981* (1981), pp. 82–94