Journal of Computer and System Sciences **64**, 579–627 (2002)
doi:10.1006/jcss.2001.1809

Hypertree Decompositions and Tractable Queries[1]

Georg Gottlob

Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria
E-mail: gottlob@dbai.tuwien.ac.at

Nicola Leone

Department of Mathematics, University of Calabria, I-87030 Rende, Italy
E-mail: leone@unical.it

and

Francesco Scarcello

D.E.I.S., University of Calabria, I-87030 Rende, Italy
E-mail: scarcello@unical.it

Received November 3, 1999; revised October 8, 2000

Several important decision problems on conjunctive queries (CQs) are NP-complete in general but become tractable, and actually highly parallelizable, if restricted to acyclic or nearly acyclic queries. Examples are the evaluation of Boolean CQs and query containment. These problems were shown tractable for conjunctive queries of bounded treewidth (Ch. Chekuri and A. Rajaraman, *Theoret. Comput. Sci.* **239** (2000), 211–229), and of bounded degree of cyclicity (M. Gyssens *et al.*, *Artif. Intell.* **66** (1994), 57–89; M. Gyssens and J. Paredaens, in "Advances in Database Theory," Vol. 2, pp. 85–122, Plenum Press, New York, 1984). The so far most general concept of nearly acyclic queries was the notion of queries of bounded query-width introduced by Chekuri and Rajaraman (2000). While CQs of bounded query-width are tractable, it remained unclear whether such queries are efficiently recognizable. Chekuri and Rajaraman (2000) stated as an open problem whether for each constant k it can be determined in polynomial time if a query has query-width at most k. We give a negative answer by proving the NP-completeness of this problem (specifically, for $k = 4$). In order to

[1] A preliminary version of this paper appeared in the "Proceedings of the Eighteenth ACM Symposium on Principles of Database Systems (PODS'99)," pp. 21–32, Philadelphia, May 1999. Research supported by *FWF (Austrian Science Funds)* under the Project Z29-INF. Part of the work of Francesco Scarcello has been carried out while visiting the Technische Universität Wien. Part of the work of Nicola Leone has been carried out while he was with the Technische Universität Wien.

0022-0000/02 $35.00

circumvent this difficulty, we introduce the new concept of hypertree decomposition of a query and the corresponding notion of hypertree-width. We prove: (a) for each k, the class of queries with query-width bounded by k is properly contained in the class of queries whose hypertree-width is bounded by k; (b) unlike query-width, constant hypertree-width is efficiently recognizable; and (c) Boolean queries of bounded hypertree-width can be efficiently evaluated.

1. INTRODUCTION AND OVERVIEW OF RESULTS

1.1. Conjunctive Queries and Join Trees

One of the simplest but also one of the most important classes of database queries is the class of *conjunctive queries (CQs)*. In this paper we adopt the logical representation of a relational database [40, 1], where data tuples are identified with logical ground atoms, and conjunctive queries are represented as datalog rules. We will, in the first place, deal with *Boolean* conjunctive queries (BCQs) represented by rules whose heads are variable-free, i.e., propositional (see Example 1.1 below). From our results on Boolean queries, we are able to derive complexity results on important database problems concerning general (not necessarily Boolean) conjunctive queries.

EXAMPLE 1.1. Consider a relational database with the following relation schemas:

```
enrolled(Pers#,Course#,Reg_Date)
teaches(Pers#,Course#,Assigned)
parent(Pers1, Pers2)
```

The BCQ Q_1 below checks whether some student is enrolled in a course taught by his/her parent.

$$Q_1\colon ans \leftarrow \texttt{enrolled}(S, C, R) \wedge \texttt{teaches}(P, C, A) \wedge \texttt{parent}(P, S).$$

The following query Q_2 asks: Is there a professor who has a child enrolled in some course?

$$Q_2\colon ans \leftarrow \texttt{teaches}(P, C, A) \wedge \texttt{enrolled}(S, C', R) \wedge \texttt{parent}(P, S).$$

Decision problems such as the *evaluation problem* of Boolean CQs, the *tuple-of-query* problem (i.e., checking whether a given tuple belongs to a CQ), and the *containment problem* for CQs have been studied intensively. (For recent references, see [29, 9].) These problems—which are all equivalent via simple logspace transformations (see [19])—are NP-complete in the general setting but are polynomially solvable for a number of syntactically restricted subclasses.

Most prominent among the polynomial cases is the class of *acyclic queries* or *tree queries* [44, 4, 18, 45, 10, 13, 14, 31]. These queries can be characterized in terms of

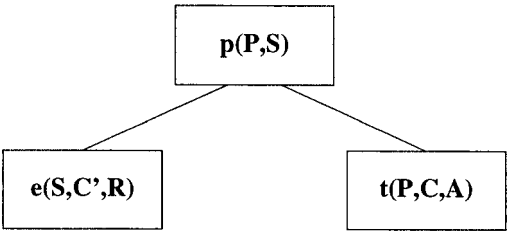

FIG. 1. A join tree of Q_2.

join trees: A query Q is *acyclic* iff it has a join tree [4, 3]. A *join tree* $JT(Q)$ for a conjunctive query Q is a tree whose vertices are the atoms in the body of Q such that whenever the same variable X occurs in two atoms A_1 and A_2, then A_1 and A_2 are connected in $JT(Q)$, and X occurs in each atom on the unique path linking A_1 and A_2. In other words, the set of nodes in which X occurs induces a (connected) subtree of $JT(Q)$. We will refer to this condition as the *Connectedness Condition* of join trees.

EXAMPLE 1.2. While query Q_1 of example 1.1 is cyclic and admits no join tree, query Q_2 is acyclic. A join tree for Q_2 is shown in Fig. 1 (note that predicate names are abbreviated by their first letter in the figure).

Acyclic Boolean queries can be efficiently evaluated. Intuitively, this is due to the fact that they can be evaluated by processing the join tree bottom-up by performing upward semijoins, thus keeping small the size of the intermediate relations (that could become exponential if regular joins were performed). This method is the Boolean version of Yannakakis' evaluation algorithm for general conjunctive queries [44]. Actually, this evaluation can be also performed in a highly parallel fashion, independently of the join tree shape [19].

1.2. Queries of Bounded Width

The tremendous speed-up obtainable in the evaluation of acyclic queries stimulated several research efforts towards the identification of wider classes of queries having the same desirable properties as acyclic queries. These studies identified a number of relevant classes of cyclic queries which are close to acyclic queries, because they can be decomposed via low width decompositions to acyclic queries. Thus, any such method M is characterized by some notion of M-width. We say that a query has *bounded width* according to M if its M-width is bounded by some fixed constant k.

The main classes of polynomially solvable bounded-width queries considered in database theory are:

- **The queries of bounded treewidth** [9] (see also [29, 19]). These are queries, whose variable-atom incidence graph has treewidth bounded by a constant. The treewidth of a graph is a well-known measure of its tree-likeness introduced by Robertson and Seymour in their work on graph minors [34]. This notion plays a central role in algorithmic graph theory as well as in many subdisciplines of

Computer Science. We omit a formal definition. It is well-known that checking that a graph has treewidth at most k for a fixed constant k, and in the positive case, computing a k-width tree decomposition is feasible in linear time [6]. In [29], another notion of treewidth of a query has been considered. This notion is equivalent to the treewidth of the Gaifman graph of the query, i.e., the graph linking two variables by an edge if they occur together in a query-atom.

• **Queries of bounded degree of cyclicity** [26, 25]. This is an interesting class of queries which encompasses the class of acyclic queries. (See [26, 25] for a formal definition.) For each constant k, checking whether a query has degree of cyclicity at most k is feasible in polynomial time [26, 25].

• **Queries of bounded query-width** [9]. The notion of bounded query-width is based on the concept of *query decomposition* [9]. Roughly, a query decomposition of a query Q consists of a tree each vertex of which is labeled by a set of atoms and/or variables. Each variable and atom induces a connected subtree (*connectedness condition*). Each atom occurs in at least one label. The width of a query decomposition is the maximum of the cardinalities of its vertices. The *query-width* $qw(Q)$ of Q is the minimum width over all its query decompositions. A formal definition is given in Section 3.1; Fig. 2 shows a 2-width query-decomposition for the cyclic query Q_1 of Example 1.1. This class is the widest of the three classes: Each query of bounded treewidth or of bounded degree of cyclicity k has also bounded query-width k, but for some queries the converse does not hold [9, 19]. In fact, there are even classes of queries with bounded query-width but with unbounded treewidth and unbounded degree of cyclicity. Note, however, that no polynomial algorithm for checking whether a query has width at most k was known.

Intuitively, a vertex of a k-width query decomposition stands for the natural join of (the relations of) its elements—the size of this join is $O(n^k)$, where n is the size of the input database. Once these joins have been done, the query decomposition can be treated exactly like a join tree of an acyclic query, and permits to evaluate the query in time polynomial in n^k [9]. This notion is a true generalization of the basic concept of acyclicity: A query is acyclic iff it has query-width 1.

The problem BCQ (evaluation of Boolean conjunctive queries) and the bounded query-width versions of all mentioned equivalent problems, e.g. query-containment $Q_1 \subseteq Q_2$, where the query-width of Q_2 is bounded, can be efficiently solved *if a k-width query decomposition of the query is given as (additional) input*. Chekuri and

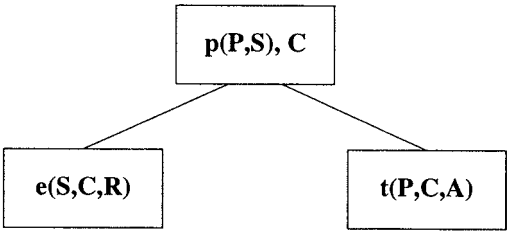

FIG. 2. A 2-width query decomposition of query Q_1.

Rajamaran provided a polynomial-time algorithm for this problem [9]; Gottlob *et al.* [19] later pinpointed the precise complexity of the problem by proving its LOGCFL-completeness.

1.3. A Negative Result

Unfortunately, unlike for acyclicity, for bounded treewidth, or for bounded degree of cyclicity, no efficient method for checking bounded query-width is known, and a k-width query decomposition, which is required for the efficient evaluation of a bounded-width query, is not known to be polynomial-time computable.

Chekuri and Rajaraman [9] state this as an open problem. This problem is the first question we address in the present paper.

The fact that treewidth k can be checked in linear time suggests that an analogous algorithm may work for query-width, too. Chekuri and Rajaraman [9] write: "*it would be useful to have an efficient algorithm that produces query decompositions of small width, analogous to the algorithm of Bodlaender [6] for decompositions of small treewidth.*" Kolaitis and Vardi [29] who also address this issue write: "*there is an important advantage of the concept of bounded treewidth over the concept of bounded query-width. Specifically, as seen above, the classes of structures of bounded treewidth are polynomially recognizable, whereas it is not known whether the same holds true for the classes of queries of bounded query-width.*"

Unfortunately, there is bad news:

Determining whether the query-width of a conjunctive query is at most 4 is NP-complete.

The NP-completeness proof is rather involved. We give some intuition in Section 3.3, and defer the technical proof to Section 7. As shown in Section 3.3, NP-hardness is intuitively due to the fact that the definition of query decomposition implicitly requires that certain sets of variables occurring in subtrees of the decomposition be *precisely* covered by query atoms. This requirement of precise covering is reminiscent of various covering problems known to be NP-complete. In fact, in our NP-completeness proof (given in Section 7), we succeeded to reduce the problem of `EXACT COVER BY 3-SETS` to the query-width problem. The proof led us to a better intuition about (i) why the problem is NP-complete, and (ii) how this could be redressed by adopting a different notion of width.

1.4. Hypertree Decompositions: Positive Results

To circumvent the high complexity of query decompositions, we introduce a new concept of decomposition and its associated notion of width, which we call *hypertree decomposition* and *hypertree-width*, respectively. The definition of hypertree decomposition (see Section 4) corresponds to a more liberal notion of "covering," which is computationally tractable.

We denote the query-width of a query by $qw(Q)$ and its hypertree-width by $hw(Q)$. We shall prove the following results:

1. For each conjunctive query Q it holds that $hw(Q) \leqslant qw(Q)$.
2. There exist queries Q such that $hw(Q) < qw(Q)$.
3. For each fixed constant k, the problems of determining whether $hw(Q) \leqslant k$ and of computing (in the positive case) a hypertree decomposition of width at most k are feasible in polynomial time.
4. For fixed k, evaluating a Boolean conjunctive query Q with $hw(Q) \leqslant k$ is feasible in polynomial time.
5. The result of a (non-Boolean) conjunctive query Q of bounded hypertree-width can be *computed* in time polynomial in the combined size of the input instance and of the output relation.
6. Tasks 3 and 4 are not only polynomial, but are highly parallelizable. In particular, for fixed k, checking whether $hw(Q) \leqslant k$ is in the parallel complexity class LOGCFL; computing a hypertree decomposition of width k (if any) is in functional LOGCFL, i.e., is feasible by a logspace transducer that uses an oracle in LOGCFL; evaluating Q where $hw(Q) \leqslant k$ on a database is complete for LOGCFL under logspace reductions.

Similar results hold for the equivalent problem of conjunctive query containment $Q_1 \subseteq Q_2$, where $hw(Q_2) \leqslant k$, and for all other of the aforementioned equivalent problems.

Let us comment on these results. By statements 1 and 2, the concept of hypertree-width is a proper generalization of the notion of query width. By statement 3, bounded hypertree-width is efficiently checkable, and by statement 4, queries of bounded hypertree-width can be efficiently evaluated. In summary, this is truly good news. It means that the notion of bounded hypertree-width not only shares the desirable properties of bounded query-width, it also does *not* share the bad properties of the latter, and, in addition, is a more general concept.

It thus turns out that the high complexity of determining bounded query-width is not, as one would usually expect, the price for the generality of the concept. Rather, it is due to some peculiarity in its definition related to the exact covering paradigm. In the definition of hypertree width we succeeded to eliminate these problems without paying any additional charge, i.e., hypertree-width comes as a freebie!

Furthermore, Statement 6 asserts that the main algorithmic tasks related to bounded hypertree-width are in the very low complexity class LOGCFL, and thus are highly parallelizable. (See Section 2.2).

The definitions of hypertree decomposition and hypertree width given below (in Section 4) are quite technical. However, in a recent paper [23], we were able to give extremely natural characterizations of the classes of queries (or hypergraphs) of bounded hypertree width, both in terms of games and in terms of suitable fragments of first order logic. From the results in [23], it follows that the concept of hypertree decomposition is a natural generalization of the concept of tree decomposition [34] (which is defined for graphs only) to hypergraphs.

1.5. Structure of the Paper

The rest of this paper is structured as follows. In Section 2, we give some basic notions of database and complexity theory. In Section 3, we formally define the query decompositions and provide some intuition on why finding (even small) query decompositions is NP-hard. The new notions of hypertree decomposition and hypertree-width are formally defined in Section 4, where also some examples are given, and it is shown that queries having bounded hypertree-width are efficiently evaluable. In Section 5, we present the alternating algorithm `k-decomp` that checks whether a query has hypertree-width at most k, where k is a fixed constant. This algorithm is shown to run on a logspace ATM having polynomially-sized accepting computation-trees, thus the problem is actually in LOGCFL. In Section 6, hypertree decomposition is compared to related notions and, in particular, it is shown that hypertree decomposition properly generalizes the notion of query decomposition. In Section 7 we give the full NP-completeness proof of the problem of deciding bounded query-width.

This paper has two appendices. In Appendix 1 we show how the concepts of hypertree decomposition and hypertree width can be defined for hypergraphs rather than for conjunctive queries, and we show how the two settings are related. In Appendix 2, we present a deterministic polynomial time algorithm (in form of a Datalog program) for checking whether a query has hypertree width at most k.

Moreover, we maintain the hypertree decompositions' homepage [36], containing further information and a download section with a program for computing hypertree decompositions and other useful tools.

2. PRELIMINARIES

2.1. Databases and Queries

For a background on databases, conjunctive queries, etc., see [40, 1, 30]. We define only the most relevant concepts here.

A relation schema R consists of a name (name of the relation) r and a finite ordered list of attributes. To each attribute A of the schema, a countable domain $Dom(A)$ of atomic values is associated. A *relation instance* (or simply, a *relation*) over schema $R = (A_1, \ldots, A_k)$ is a finite subset of the cartesian product $Dom(A_1) \times \cdots \times Dom(A_k)$. The elements of relations are called *tuples*. A database schema DS consists of a finite set of relation schemas. A *database instance*, or simply *database*, **DB** over database schema $DS = \{R_1, \ldots, R_m\}$ consists of relation instances $r_1, \ldots, r_m$ for the schemas $R_1, \ldots, R_m$, respectively, and a finite universe $U \subseteq \bigcup_{R_i(A_1^i, \ldots, A_{k_i}^i) \in DS}(Dom(A_1^i) \cup \cdots \cup Dom(A_{k_i}^i))$ such that all data values occurring in **DB** are from U.

In this paper we will adopt the standard convention [1, 40] of identifying a relational database instance with a logical theory consisting of ground facts. Thus, a tuple $\langle a_1, \ldots a_k \rangle$, belonging to relation r, will be identified with the ground atom $r(a_1, \ldots, a_k)$. The fact that a tuple $\langle a_1, \ldots, a_k \rangle$ belongs to relation r of a database instance **DB** is thus simply denoted by $r(a_1, \ldots, a_k) \in \mathbf{DB}$.

A (rule based) *conjunctive query* Q on a database schema $DS = \{R_1, \ldots, R_m\}$ consists of a rule of the form

$$Q : ans(\mathbf{u}) \leftarrow r_1(\mathbf{u_1}) \wedge \cdots \wedge r_n(\mathbf{u_n}),$$

where $n \geqslant 0$, $r_1, \ldots, r_n$ are relation names (not necessarily distinct) of DS; *ans* is a relation name not in DS; and $\mathbf{u}, \mathbf{u_1}, \ldots, \mathbf{u_n}$ are lists of terms (i.e., variables or constants) of appropriate length. The set of variables occurring in Q is denoted by $var(Q)$. The set of atoms contained in the body of Q is referred to as $atoms(Q)$. Similarly, for any atom $A \in atoms(Q)$, $var(A)$ denotes the set of variables occurring in A; and for a set of atoms $R \subseteq atoms(Q)$, define $var(R) = \bigcup_{A \in R} var(A)$.

The *answer* of Q on a database instance **DB** with associated universe U, consists of a relation *ans* whose arity is equal to the length of $\mathbf{u}$, defined as follows. *ans* contains all tuples $ans(\mathbf{u})\,\vartheta$ such that ϑ: $var(Q) \rightarrow U$ is a substitution replacing each variable in $var(Q)$ by a value of U and such that for $1 \leqslant i \leqslant n$, $r_i(\mathbf{u_i})\,\vartheta \in \mathbf{DB}$. (For an atom A, $A\vartheta$ denotes the atom obtained from A by uniformly substituting $\vartheta(X)$ for each variable X occurring in A.)

The conjunctive query Q is a *Boolean conjunctive query (BCQ)* if its head atom $ans(\mathbf{u})$ does not contain variables and is thus a purely propositional atom. Q evaluates to *true* if there exists a substitution ϑ such that for $1 \leqslant i \leqslant n$, $r_i(\mathbf{u_i})\,\vartheta \in \mathbf{DB}$; otherwise the query evaluates to *false*.

The head literal in Boolean conjunctive queries is actually inessential, therefore we may omit it when specifying a Boolean conjunctive query.

Note that conjunctive queries as defined here correspond to conjunctive queries in the more classical setting of relational calculus, as well as to SELECT-PROJECT-JOIN queries in the setting of relational algebra, or to simple SQL queries of the type

$$\texttt{SELECT}\ R_{i_1}.A_{j_1}, \ldots R_{i_k}.A_{j_k}\ \texttt{FROM}\ R_1, \ldots R_n\ \texttt{WHERE}\ cond,$$

such that *cond* is a conjunction of conditions of the form $R_i.A = R_j.B$ or $R_i.A = c$, where c is a constant.

A query Q is *acyclic* [3, 4] if its associated hypergraph $H(Q)$ is acyclic, otherwise Q is cyclic. The vertices of $H(Q)$ are the variables occurring in Q. Denote by $atoms(Q)$ the set of atoms in the body of Q, and by $var(A)$ the variables occurring in any atom $A \in atoms(Q)$. The hyperedges of $H(Q)$ consist of all sets $var(A)$, such that $A \in atoms(Q)$. We refer to the standard notion of cyclicity/acyclicity in hypergraphs used in database theory [30, 40, 1].

A *join tree* $JT(Q)$ for a conjunctive query Q is a tree whose vertices are the atoms in the body of Q such that whenever the same variable X occurs in two atoms A_1 and A_2, then A_1 and A_2 are connected in $JT(Q)$, and X occurs in each atom on the unique path linking A_1 and A_2. In other words, the set of nodes in which X occurs induces a (connected) subtree of $JT(Q)$ (*connectedness condition*).

Acyclic queries can be characterized in terms of join trees: A query Q is *acyclic* iff it has a join tree [4, 3].

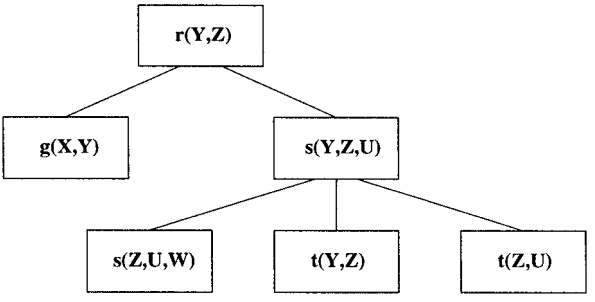

FIG. 3. A join tree of Q_3.

EXAMPLE 2.1. While query Q_1 of example 1.1 is cyclic and admits no join tree, query Q_2 is acyclic. A join tree for Q_2 is shown in Fig. 1.

Consider the following query Q_3:

$$ans \leftarrow r(Y, Z) \wedge g(X, Y) \wedge s(Y, Z, U) \wedge s(Z, U, W) \wedge t(Y, Z) \wedge t(Z, U).$$

A join tree for Q_3 is shown in Fig. 3.

Acyclic conjunctive queries have highly desirable computational properties:

1. The problem BCQ of evaluating a Boolean conjunctive query can be efficiently solved if the input query is acyclic. Yannakakis provided a (sequential) polynomial time algorithm solving BCQ on acyclic conjunctive queries [43].[2] The authors of the present paper have recently shown that BCQ is highly parallelizable on acyclic queries, as it is complete for the low complexity class LOGCFL [19].

2. Acyclicity is efficiently recognizable, and a join tree of an acyclic query is efficiently computable. A linear-time algorithm for computing a join tree is shown in [39]; an L^{SL} method has been provided in [19].

3. The result of a (non-Boolean) acyclic conjunctive query Q can be *computed* in time polynomial in the combined size of the input instance and of the output relation [44].

Acyclicity is a key-property responsible for the polynomial solvability of problems that are in general NP-hard such as BCQ [8] and other equivalent problems such as Conjunctive Query Containment [33, 9], Clause Subsumption, and Constraint Satisfaction [29, 19]. (For a survey and detailed treatment see [19].)

2.2. The Class LOGCFL

LOGCFL consists of all decision problems that are logspace reducible to a context-free language. An obvious example of a problem complete for LOGCFL is Greibach's hardest context-free language [24]. There are a number of very

[2] Note that, since both the database **DB** and the query Q are part of an input-instance of BCQ, what we are considering is the *combined complexity* of the query [43].

interesting natural problems known to be LOGCFL-complete (see, e.g. [19, 38, 37]). The relationship between LOGCFL and other well-known complexity classes is summarized in the following chain of inclusions:

$$AC^0 \subseteq NC^1 \subseteq L \subseteq SL \subseteq NL \subseteq LOGCFL \subseteq AC^1 \subseteq NC^2 \subseteq P \subseteq NP$$

Here L denotes logspace, AC^i and NC^i are *logspace-uniform* classes based on the corresponding types of Boolean circuits, SL denotes symmetric logspace, NL denotes nondeterministic logspace, P is polynomial time, and NP is nondeterministic polynomial time. For the definitions of all these classes, and for references concerning their mutual relationships, see [28].

Since—as mentioned in the introduction—$LOGCFL \subseteq AC^1 \subseteq NC^2$, the problems in LOGCFL are all highly parallelizable. In fact, they are solvable in logarithmic time by a concurrent-read-concurrent-write (CRCW) parallel random-access-machine (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 this paper, we will use an important characterization of LOGCFL by Alternating Turing Machines. We assume that the reader is familiar with the *alternating Turing machine (ATM)* computational model introduced by Chandra et al. [7]. Here we assume without loss of generality that the states of an ATM are partitioned into existential and universal states.

As in [35], we define a *computation tree* of an ATM M on an input string w as a tree whose nodes are labeled with configurations of M on w, such that the descendants of any non-leaf labeled by a universal (existential) configuration include all (resp. one) of the successors of that configuration. A computation tree is *accepting* if the root is labeled with the initial configuration, and all the leaves are accepting configurations.

Thus, an accepting tree yields a certificate that the input is accepted. A complexity measure considered by Ruzzo [35] for the alternating Turing machine is the tree-size, i.e. the minimal size of an accepting computation tree.

DEFINITION 2.2 [35]. A decision problem $\mathscr{P}$ is solved by an alternating Turing machine M within *simultaneous* tree-size and space bounds $Z(n)$ and $S(n)$ if, for every "yes" instance w of $\mathscr{P}$, there is at least one accepting computation tree for M on w of size (number of nodes) $\leqslant Z(n)$, each node of which represents a configuration using space $\leqslant S(n)$, where n is the size of w. (Further, for any "no" instance w of $\mathscr{P}$ there is no accepting computation tree for M.)

Ruzzo [35] proved the following important characterization of LOGCFL :

PROPOSITION 2.3 [35]. *LOGCFL coincides with the class of all decision problems recognized by ATMs operating simultaneously in tree-size* $O(n^{O(1)})$ *and space* $O(\log n)$.

3. QUERY DECOMPOSITIONS

In this section, we first give the formal definitions of query-width and query decomposition. Then, we provide some intuition of why deciding whether a query has bounded query-width is NP-hard.

3.1. Bounded Query-Width and Bounded Query-Decompositions

The following definition of query decomposition is a slight modification of the original definition given by Chekuri and Rajaraman [9]. Our definition is a bit more liberal because, for any conjunctive query Q, we do not care about the atom $head(Q)$, as well as of the constants possibly occurring in Q. However, in this paper, we will only deal with Boolean conjunctive queries without constants, for which the two notions coincide.

DEFINITION 3.1. A *query decomposition* of a conjunctive query Q is a pair $\langle T, \lambda \rangle$, where $T = (N, E)$ is a tree, and λ is a labeling function which associates to each vertex $p \in N$ a set $\lambda(p) \subseteq (atoms(Q) \cup var(Q))$, such that the following conditions are satisfied:

1. for each atom A of Q, there exists $p \in N$ such that $A \in \lambda(p)$;
2. for each atom A of Q, the set $\{p \in N \mid A \in \lambda(p)\}$ induces a (connected) subtree of T;
3. for each variable $Y \in var(Q)$, the set

$$\{p \in N \mid Y \in \lambda(p)\} \cup \{p \in N \mid Y \text{ occurs in some atom } A \in \lambda(p)\}$$

induces a (connected) subtree of T.

The *width* of the query decomposition $\langle T, \lambda \rangle$ is $max_{p \in N} |\lambda(p)|$. The *query-width* $qw(Q)$ of Q is the minimum width over all its query decompositions. A query decomposition for Q is *pure* if, for each vertex $p \in N$, $\lambda(p) \subseteq atoms(Q)$.

Note that Condition 3 above is the analogue of the connectedness condition of join trees and thus we will refer to it as the *Connectedness Condition*, as well.

EXAMPLE 3.2. Figure 2 shows a 2-width query decomposition for the cyclic query of Example 1.1.

Consider the following query Q_4:

$$ans \leftarrow s(Y, Z, U) \wedge g(X, Y) \wedge t(Z, X) \wedge s(Z, W, X) \wedge t(Y, Z)$$

Q_4 is a cyclic query, and its query-width equals 2. A 2-width decomposition of Q_4 is shown in Fig. 4. Note that this query decomposition is pure.

The next proposition, which is proved elsewhere [19], shows that we can focus our attention on pure query decompositions.

PROPOSITION 3.3 [19]. *Let Q be a conjunctive query and $\langle T, \lambda \rangle$ a c-width query decomposition of Q. Then*

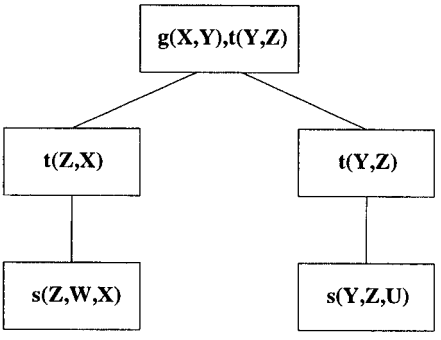

FIG. 4. A 2-width query decomposition of query Q_4.

1. *there exists a pure c-width query decomposition* $\langle T, \lambda' \rangle$ *of* Q;
2. $\langle T, \lambda' \rangle$ *is logspace computable from* $\langle T, \lambda \rangle$.

Thus, by Proposition 3.3, for any conjunctive query Q, $qw(Q) \leqslant k$ if and only if Q has a pure c-width decomposition, for some $c \leqslant k$.

k-bounded-width queries are queries whose query-width is bounded by a fixed constant $k > 0$. The notion of bounded query-width generalizes the notion of acyclicity [9]. Indeed, acyclic queries are exactly the conjunctive queries of query-width 1, because any join tree is a query decomposition of width 1.

Bounded-width queries share an important computational property with acyclic queries: BCQ can be efficiently solved on queries of k-bounded query-width, if a k-width query decomposition of the query is given as (additional) input. Chekuri and Rajamaran provided a polynomial time algorithm for this problem [9]; while Gottlob et al. [19] pinpointed that the precise complexity of the problem is LOGCFL-complete.

Unfortunately, unlike for acyclicity, no efficient method for checking bounded query-width is known. In fact, we will show that deciding whether a conjunctive query has a bounded-width query decomposition is NP-complete. The proof is quite involved. We will give an intuition of the source of complexity in Section 3.3, while Section 7 at the end of the paper is devoted to the full NP-hardness proof. Before proceeding with NP-completeness issues in Section 3.3, we define a number of important concepts.

3.2. Important Definitions for Hypergraph-Based Decompositions

The concepts we are going to define here are not only relevant to the context of query decompositions but will be used in later sections, too.

Let $V \subseteq var(Q)$ be a set of variables, and $X, Y \in var(Q)$ a pair of variables occurring in a query Q, then X is $[V]$-adjacent to Y if there exists an atom $A \in atoms(Q)$ such that $\{X, Y\} \subseteq (var(A) - V)$. A $[V]$-path π from X to Y consists of a sequence $X = X_0, \ldots, X_h = Y$ of variables and a sequence of atoms $A_0, \ldots, A_{h-1}$ ($h \geqslant 0$) such that: X_i is $[V]$-adjacent to X_{i+1} and $\{X_i, X_{i+1}\} \subseteq var(A_i)$, for each $i \in [0 \ldots h-1]$. We denote by $var(\pi)$ (resp. $atoms(\pi)$) the set of variables (atoms) occurring in the sequence $X_0, \ldots, X_h$ ($A_0, \ldots, A_{h-1}$).

Let $V \subseteq var(Q)$ be a set of variables occurring in a query Q. A set $W \subseteq var(Q)$ of variables is $[V]$-connected if $\forall X, Y \in W$ there is a $[V]$-path from X to Y.

A $\{[V]$-*component* is a maximal $[V]$-connected non-empty set of variables $W \subseteq (var(Q) - V)$.

Note that the variables in V do not belong to any $[V]$-component (i.e., $V \cap C = \varnothing$ for each $[V]$-component C).

Let C be a $[V]$-component for some set of variables V. We define:

$$atoms(C) := \{A \in atoms(Q) \mid var(A) \cap C \neq \varnothing\}.$$

Note that for any set V of variables and for every atom $A \in atoms(Q)$ such that $var(A) \not\subseteq V$, there exists exactly one $[V]$-component C of Q such that $A \in atoms(C)$.

3.3. NP-completeness of Bounded Query-Width (Outline)

As mentioned in the Introduction, Chekuri and Rajaraman [9] stated as an open problem the tractability of deciding whether the width of a given query is bounded by some fixed constant. We next give a negative answer to their question.

THEOREM 3.4. *Deciding whether the query-width of a conjunctive query is at most* 4 *is NP-complete.*

For the sake of completeness, it is worthwhile noting that this result holds for the original definition of query decomposition given in [9], too. Indeed, the NP-hardness of Boolean conjunctive queries without constants (that we consider in this paper) clearly entails the NP-hardness of the general case, and hence of the notion given in [9]. Moreover, the membership in NP is routine, and it is already discussed in [9].

The proof of the theorem above is rather involved and is deferred to Section 7. In the the rest of the present section, we make some observations on query-decompositions and give some intuitions about the source of NP-hardness of finding a small query decomposition. These intuitions provide some insight into the nature of query decompositions and give us a hint on how the high complexity could be redressed by suitably modifying the notion of query decomposition.

The following example will serve as running example for this and later sections.

EXAMPLE 3.5. Consider the following conjunctive query Q_5:

$$ans \leftarrow a(S, X, X', C, F) \wedge b(S, Y, Y', C', F') \wedge c(C, C', Z) \wedge d(X, Z) \wedge e(Y, Z) \wedge$$
$$\wedge f(F, F', Z') \wedge g(X', Z') \wedge h(Y', Z') \wedge j(J, X, Y, X', Y')$$

The query-width of Q_5 is 3. A query decomposition of Q_5 of width 3 is depicted in Fig. 5. Note that this is not the only one, as Q_5 admits several other possible query decompositions of width 3.

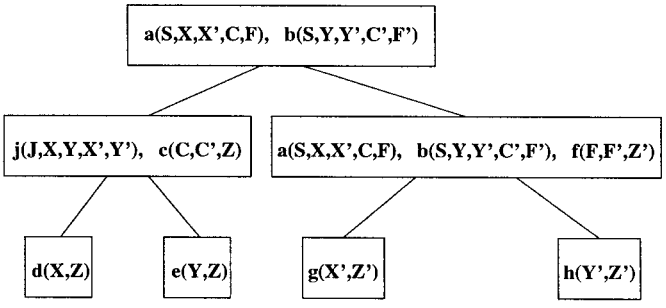

FIG. 5. A 3-width query decomposition of query Q_5.

For any vertex p of a query-decomposition $\langle T = (N, E), \lambda \rangle$ of some query Q, we denote by $var(p)$ the set $\bigcup_{A \in \lambda(p)} var(A)$ of all variables occurring in the atoms associated with p, by $T_p = (N_p, E_p)$ the subtree of T rooted at p, and by $var(T_p)$ the set of all variables covered by T_p, formally, $var(T_p) = \bigcup_{q \in N_p} var(q)$.

The following observation follows easily from Definition 3.1.

PROPOSITION 3.6. *Let $T = (N, E)$ be a pure query decomposition of a conjunctive query Q. Let p be any non-leaf vertex of T. Then*

$$var(T_p) = var(p) \cup \bigcup_{i \in I} C_i,$$

where I is some index set and each C_i is a $[var(p)]$-component.

For illustration, consider, e.g., the root vertex p_0 of the decomposition $QD_5 = \langle T, \lambda \rangle$ of query Q_5 depicted in Fig. 5. The set of the variables occurring in p_0 is $var(p_0) = \{S, X, X', C, F, Y, Y', C', F'\}$ and there are three $[var(p_0)]$-components: $C_1 = \{J\}$, $C_2 = \{Z\}$, and $C_3 = \{Z'\}$. We have: $var(T) = var(T_{p_0}) = var(p_0) \cup C_1 \cup C_2 \cup C_3$.

A query-decomposition of width k thus consists of a tree where the atoms of each subtree T_p rooted at any vertex p *precisely* cover the variables of p plus some $[var(p)]$-components. The atoms of T_p may *not* contain any additional variable which neither occurs in $var(p)$ nor in any of the chosen $[var(p)]$-components.

It is this requirement of *precise* covering which, intuitively, makes it so hard to compute a suitable query decomposition.

Very roughly, the source of NP-hardness can be pinpointed as follows. In order to find a query decomposition of width bounded by k, we can proceed as follows. At any step, the decomposition is guided by a set C of variables that still needs to be processed. Initially, i.e., at the root of the decomposition, C consists of *all* variables that occur in the query. We then choose as root of the decomposition tree a hypernode p_0 of at most k query atoms. By fixing this hypernode, we eliminate a set of variables, namely those which occur in the atoms of p_0. The remaining set of variables disintegrates into connected components. For instance, assume that we are looking for a query decomposition of width 2 for query Q_5 of our running

example. We choose to label the root p_0 of this query decomposition by atoms a, b,[3] as for the decomposition shown in Fig. 5. Thus, the variables in $\{X, X', Y, Y', S, C, C', F, F'\}$ are "fixed." As a consequence $\{J\}$, $\{Z\}$, and $\{Z'\}$ are now distinct $[\mathrm{p}_0]$-components, in that every path connecting any pair of these variables contains some variable occurring in $\{a, b\}$.

We now expand the decomposition tree by attaching children, and thus, in the long run, subtrees to p_0. By Proposition 3.6, each subtree rooted in any child p of p_0 must precisely contain (in its labels) all the variables in $var(p)$ and in some $[var(p)]$-components. Moreover, each of these $[var(p)]$-components occurs in exactly one subtree (otherwise the connectedness condition would be violated). For instance, variable J, which occurs in the left child of QD_5 cannot occur in the other subtree of the root, because J does not belong to $var(a) \cup var(b)$.

Moreover, each atom w should be eventually covered, that is, there must exist a vertex of the tree whose label contains all the variables in $var(w)$. For instance, the atom j, is covered by the left child of the root. This process goes on until all variables are eliminated and until all query atoms are eventually covered. As we observed above, the definition of query-width requires this covering be *exact*, i.e., each atom containing a variable of a certain component C occurs only in the subtree corresponding to C. (Again, the requirement of precise covering is due to the connectedness condition.) As a consequence, since we labeled the root p_0 by $\{a, b\}$, and then covered j in the left child of the root of QD_5, we cannot use this atom in the other child of the root, where we deal with the component $\{Z'\}$. Recall that our purpose was finding a query decomposition of width 2. However, covering atoms f, g, and h with only two atoms per label would require the use of atom j, because it contains both variables X' and Y' which occur in g and h, respectively. But j already appears in the left subtree of the root and, as said, cannot appear in the right subtree, too. Therefore, the query decomposition shown in Figure 5 has width 3, because in order to enforce the connectedness condition, both atoms a and b must be used again in the label of the right child of p_0. In fact, by checking all possible labelings, it can be shown that Q_5 has no query decomposition of width 2.

In general, suitable choices of labels at each step of the decomposition are crucial in order to meet the required bound k on the query-width.

Thus, the question is: how can one choose the right $k_p \leqslant k$ labels at each decomposition vertex p in order to cover every atom of the query and satisfying the connectedness condition? It turns out that this is a difficult task, and indeed we reduce the well-known NP-hard problem EXACT COVER BY 3-SETS [16] to the problem of finding a query decomposition of width at most 4 for some query Q. An instance of EXACT COVER BY 3-SETS consists of a pair $I = (R, \Delta)$ where R is a set of $r = 3s$ elements, and Δ is a collection of m 3-element subsets of R. The question is whether we can select s subsets out of Δ such that they form a partition of R. Roughly, in Section 7 we show that, given such an instance I, we build a query Q, depending on I, such that every query decomposition for Q of width at most 4 corresponds to an exact cover of R. In particular, any choice for the labels of some special vertices corresponds to the selection of some 3-element subset from Δ. If every labeling is

[3] For the sake of readability, whenever is possible, we identify atoms with their predicate symbols.

correct, one is able to find a query-decomposition for Q of width at most 4 (if any), and hence a solution for the covering-problem instance I.

4. CONJUNCTIVE QUERIES OF BOUNDED HYPERTREE-WIDTH

In Section 4.1 we formally introduce the notions of *hypertree decomposition* and *hypertree-width*. In Section 4.2, we prove that Boolean conjunctive queries whose hypertree-width is bounded by a constant can be answered in polynomial time. The problem of recognizing constant hypertree-width is deferred to Section 5.

4.1. Hypertree Decompositions and Hypertree-Width

The definition of hypertree decomposition eliminates the source of NP-completeness present in query decompositions by recurring to a more liberal notion of "covering".

Intuitively, when choosing a vertex p of the decomposition tree, we no longer want to require that the set of all variables in the atoms of T_p precisely coincide with the variables of $var(p)$ plus those of some $[var(p)]$-components (cf. Proposition 3.6). Instead, it shall be sufficient that the former set of variables be a superset of the latter. (Which, by the way, corresponds to a more standard notion of covering.)

In order to achieve this, we decouple the set of variables associated with a node in the decomposition tree from the set of query atoms associated with the same node. The set of variables associated with a node p is denoted by $\chi(p)$ while the set of atoms associated with p is denoted as before by $\lambda(p)$. In our definition below we will not require that $\chi(p) = var(\lambda(p))$ but only that $\chi(p) \subseteq var(\lambda(p))$.

Let Q be a conjunctive query. A *hypertree for* Q is a triple $\langle T, \chi, \lambda \rangle$, where $T = (N, E)$ is a rooted tree, and χ and λ are labeling functions which associate to each vertex $p \in N$ two sets $\chi(p) \subseteq var(Q)$ and $\lambda(p) \subseteq atoms(Q)$. If $T' = (N', E')$ is a subtree of T, we define $\chi(T') = \bigcup_{v \in N'} \chi(v)$. We denote the set of vertices N of T by $vertices(T)$, and the root of T by $root(T)$. Moreover, for any $p \in N$, T_p denotes (as before) the subtree of T rooted at p.

DEFINITION 4.1. A *hypertree decomposition* of a conjunctive query Q is a hypertree $\langle T, \chi, \lambda \rangle$ for Q which satisfies all the following conditions:

1. for each atom $A \in atoms(Q)$, there exists $p \in vertices(T)$ such that $var(A) \subseteq \chi(p)$;
2. for each variable $Y \in var(Q)$, the set $\{p \in vertices(T) \mid Y \in \chi(p)\}$ induces a (connected) subtree of T;
3. for each vertex $p \in vertices(T)$, $\chi(p) \subseteq var(\lambda(p))$;
4. for each vertex $p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

The *width* of the hypertree decomposition $\langle T, \chi, \lambda \rangle$ is $max_{p \in vertices(T)} |\lambda(p)|$. The *hypertree-width* $hw(Q)$ of Q is the minimum width over all its hypertree decompositions.

In analogy to join trees and query decompositions, we will refer to Condition 2 above as the *Connectedness Condition*. Note that the inclusion in Condition 4 is actually an equality, because Condition 3 implies the reverse inclusion. Moreover, by Condition 1, $\chi(T) = var(Q)$. Hence Condition 4 entails that, for $s_0 = root(T)$, $var(\lambda(s_0)) = \chi(s_0)$.

DEFINITION 4.2. A hypertree decomposition $\langle T, \chi, \lambda \rangle$ of Q is a *complete decomposition* of Q if, for each atom $A \in atoms(Q)$, there exists $p \in vertices(T)$ such that $var(A) \subseteq \chi(p)$ and $A \in \lambda(p)$.

Intuitively, the χ labeling selects the set of variables to be fixed in order to split the cycles and achieve acyclicity; $\lambda(p)$ "covers" the variables of $\chi(p)$ by a set of atoms. Thus, the relations associated to the atoms of $\lambda(p)$ restrict the range of the variables of $\chi(p)$. For the evaluation of query Q, each vertex p of the decomposition is replaced by a new atom whose associated database relation is the projection on $\chi(p)$ of the join of the relations in $\lambda(p)$. This way, we obtain a join tree JT of an acyclic query Q' over a database $\mathbf{DB}'$ of size polynomial in the original input database. All the efficient techniques available for acyclic queries can be then employed for the evaluation of Q'.

More technically, Condition 1 and Condition 2 in Definition 4.1 extend the notion of tree decomposition [34] from graphs to hypergraphs (the hypergraph of a query Q groups the variables of the same atom in one hyperedge [3]). Thus, the pair $\langle T, \chi \rangle$ of a hypertree decomposition $\langle T, \chi, \lambda \rangle$ of a conjunctive query Q, can be seen as the correspondent of a tree decomposition on the query hypergraph. However, the treewidth of $\langle T, \chi \rangle$ (i.e., the maximum cardinality of the χ-labels of the vertices of T) is not an appropriate measure of the width of the hypertree decomposition, because a set of m variables appearing in the same atom should count 1 rather than m for the width. Thus, $\lambda(p)$ provides a set of atoms which "covers" $\chi(p)$ and its cardinality gives the measure of the width of vertex p. It is worthwhile noting that $\langle T, \lambda \rangle$ may violate the classical connectedness condition usually imposed on the variables of the join trees, as it is allowed that a variable X appears in both $\lambda(p)$ and $\lambda(q)$ while it does not appear in $\lambda(s)$, for some vertex s on the path from p to q in T. However, this violation is not a problem, as the variables in $var(\lambda(p)) - \chi(p)$ are meaningless and can be projected out before starting the query evaluation process, because the role of $\lambda(p)$ is just that of providing a binding for the variables of $\chi(p)$.

EXAMPLE 4.3. The hypertree-width of the cyclic query Q_1 of Example 1.1 is 2; a (complete) 2-width hypertree decomposition of Q_1 is shown in Fig. 6a.

Figure 6b shows a (complete) hypertree decomposition HD_5 of query Q_5 of Example 3.5. Since Q_5 is a cyclic query and only acyclic queries have hypertree-width 1 (see Theorem 4.5 below), it follows that $hw(Q_5) = 2$.

An alternative representation, called *atom representation*, of the hypertree decomposition HD_5 of query Q_5 is depicted in Fig. 7. Each node p in the tree is labeled by a set of atoms representing $\lambda(p)$; $\chi(p)$ is the set of all variables, distinct from '_', appearing in these atoms. Thus, the anonymous variable '_' is in the place of the variables in $var(\lambda(p)) - \chi(p)$.

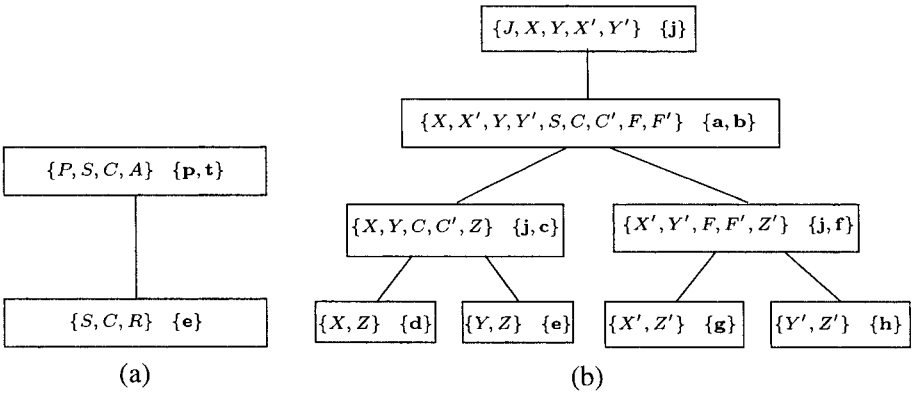

FIG. 6. A 2-width hypertree decomposition of (a) query Q_1; and (b) query Q_5.

Definition 4.1 does not require the presence of all query atoms in a decomposition HD, as it is sufficient that every atom is "covered" by some vertex p of HD (i.e., its variables are included in $\chi(p)$). However, every missing atom can be easily added to complete decompositions.

LEMMA 4.4. *Given a conjunctive query Q, every k-width hypertree decomposition HD of Q can be transformed in logspace into a k-width complete hypertree decomposition HD' of Q, whose size is $O(\|Q\| + \|HD\|)$.*

Proof. Let Q be a conjunctive query and $HD = \langle T, \chi, \lambda \rangle$ a hypertree decomposition of Q. We transform HD into a complete decomposition HD' as follows. For each atom $A \in atoms(Q)$ such that no vertex $q \in vertices(T)$ satisfies $var(A) \subseteq \chi(q)$ and $A \in \lambda(q)$, create a new vertex v_A with $\lambda(v_A) := \{A\}$ and $\chi(v_A) = var(A)$, and attach v_A as a new child of a vertex $p \in vertex(T)$ such that $var(A) \subseteq \chi(p)$. (By Condition 1 of Definition 4.1 such a p must exist.)

This transformation is obviously feasible in logspace. Moreover, the size of HD' differs only by $O(\|Q\|)$ from the size of HD given that HD' is obtained from HD by adding nodes corresponding to some atoms of Q. Thus, the size $\|HD'\|$ of HD' is $O(\|Q\| + \|HD\|)$. ∎

The acyclic queries are precisely the queries of hypertree-width one.

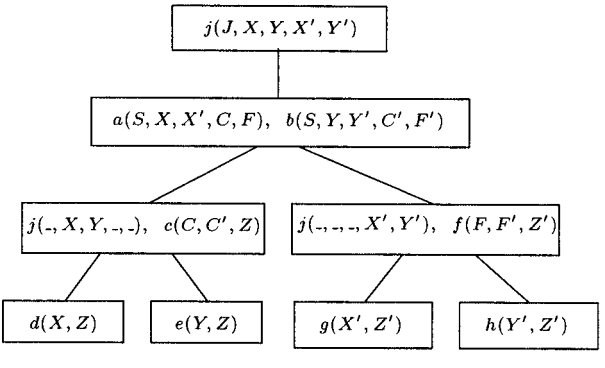

FIG. 7. Atom representation of hypertree decomposition HD_5.

THEOREM 4.5. *A conjunctive query Q is acyclic if and only if $hw(Q) = 1$.*

Proof. (*Only if part.*) If Q is an acyclic query, there exists a join tree $JT(Q)$ for Q. Let T be a tree, and f a bijection from $vertices(JT(Q))$ to $vertices(T)$ such that, for any $p, q \in vertices(JT(Q))$, there is an edge between p and q in $JT(Q)$ if and only if there is an edge between $f(p)$ and $f(q)$ in T. Moreover, let λ be the following labeling function: If p is a vertex of $JT(Q)$ and A is the atom of Q associated to p, then $\lambda(f(p)) = \{A\}$. For any vertex $p' \in vertices(T)$ define $\chi(p') = var(\lambda(p'))$. Then, $\langle T, \chi, \lambda \rangle$ is clearly a width 1 hypertree-decomposition of Q.

(*If part.*) Let $HD = \langle T, \chi, \lambda \rangle$ be a width 1 hypertree-decomposition of Q. Without loss of generality, assume that HD is a complete hypertree decomposition. Since HD has width 1, all the λ labels are singletons, i.e., λ associate one atom of Q to each vertex of T.

We next show how to trasform HD into a width 1 complete hypertree decomposition of Q such that, for any vertex $p \in vertices(T)$, $\chi(p) = var(A)$, where $\{A\} = \lambda(p)$, and p is the unique vertex labeled with the atom A.

Choose any total ordering $\prec$ of the vertices of T. For any atom $A \in atoms(Q)$, denote by $v(A)$ the $\prec$-least vertex of T such that $\chi(v(A)) = var(A)$ and $\lambda(v(A)) = \{A\}$. The existence of such a vertex is guaranteed by definition of complete hypertree decomposition and by the hypothesis that every λ label consists of exactly one atom.

For any atom $A \in atoms(Q)$, and for any vertex $p \neq v(A)$ such that $\lambda(p) = \{A\}$, perform the following actions. For any child p' of p, delete the edge between p and p' and let p' be a new child of $v(A)$, hence the subtree $T_{p'}$ is now attached to $v(A)$. Then, delete vertex p. By Condition 3 of Definition 4.1, $\chi(p) \subseteq var(\lambda(p))$. Since $var(\lambda(p)) = var(A) = \chi(v(A))$, we get $\chi(p) \subseteq \chi(v(A))$. Then, it is easy to see that the (transformed) tree T satisfies the connectedness condition.

Eventually, we obtain a new hypertree $H' = \langle T', \chi, \lambda \rangle$ such that $vertices(T') \subseteq vertices(T)$ and H' has the following properties: (i) for any $A \in atoms(Q)$, there exists exactly one vertex $p = v(A)$ of T' such that $\lambda(p) = \{A\}$ and $\chi(p) = var(A)$; (ii) for any vertex p of T', $p = v(A)$ holds, for some $A \in atoms(Q)$; (iii) H' satisfies the connectedness condition. Thus, H' clearly corresponds to a join tree of Q. ∎

4.2. Efficient Query Evaluation

In this section, we show that for any constant k, Boolean conjunctive queries with given hypertree decompositions of width k can be efficiently answered. In fact, we show that any such query is equivalent, via a logspace transformation, to an acyclic conjunctive query.

The following lemma shows that, starting from a hypertree decomposition of a query Q, we can efficiently build a join tree of an acyclic query Q' equivalent to Q.

LEMMA 4.6. *Let Q be a Boolean conjunctive query over a database* **DB**, *and HD a hypertree decomposition of Q of width k. Then, there exists Q',* **DB**′, *JT such that:*

1. *Q' is an acyclic (Boolean) conjunctive query evaluating to 'true' on database* **DB**′ *if and only if the answer of Q on* **DB** *is 'true'.*

2. *$\|\langle Q', \mathbf{DB}', JT \rangle\| = O((\|Q\| + \|HD\|) \cdot r^k)$, where r denotes the maximum relation-size over the relations in* **DB**.

3. *JT is a join tree of the query* Q'.
4. $\langle Q', \mathbf{DB}', JT\rangle$ *is logspace computable from* $\langle Q, \mathbf{DB}, HD\rangle$.

Proof. Let Q be a Boolean conjunctive query over a database **DB**, and HD a hypertree decomposition of Q of width k. Without loss of generality, we assume that Q does not contain any atom A such that $var(A) = \varnothing$. We first transform HD, using the construction shown in the proof of Lemma 4.4, into a k-width complete hypertree decomposition $\widehat{HD} = \langle \mathrm{T}, \chi, \lambda\rangle$ of Q. By Lemma 4.4, this transformation is feasible in logspace, and $\|\widehat{HD}\| = O(\|Q\| + \|HD\|)$.

Q evaluates to *true* on **DB** if and only if $\bowtie_{A \in atoms(Q)} rel(A)$ is a non-empty relation, where $rel(A)$ denotes the relation of **DB** associated to the atom A, and $\bowtie$ is the natural join operation (with common variables acting as join attributes).

For each vertex $p \in vertices(T)$ define a Boolean query $Q(p)$ and a database $\mathbf{DB}(p)$ as follows. For each atom $A \in \lambda(p)$:

- If $var(A) \subseteq \chi(p)$, then A occurs in $Q(p)$ and $rel(A)$ belongs to $\mathbf{DB}(p)$;
- if $var(A) \not\subseteq \chi(p)$ and $(var(A) \cap \chi(p)) \neq \varnothing$, then $Q(p)$ contains a new atom A' such that $var(A') = var(A) \cap \chi(p)$, and $\mathbf{DB}(p)$ contains the corresponding relation $rel(A')$, which is the projection of $rel(A)$ on the set of attributes corresponding to the variables in $var(A')$;

Now, consider the following query $\bar{Q}$ on the database $\overline{\mathbf{DB}} = \bigcup_{p \in vertices(T)} \mathbf{DB}(p)$:

$$\bar{Q}: \bigwedge_{p \in vertices(T)} Q(p).$$

By the associative and commutative properties of natural joins, and by the fact that $\widehat{HD}$ is a complete hypertree decomposition, it follows that $\bar{Q}$ on $\overline{\mathbf{DB}}$ is equivalent to Q on **DB**. To see this, note that $\overline{\mathbf{DB}}$ just contains some new relations which are projections of relations already occurring in **DB**, and $\bar{Q}$ contains the atoms corresponding to these relations. Thus, no tuple can be lost by taking the additional joins corresponding to these relations. It follows that if Q evaluates to *true* on **DB**, then also $\bar{Q}$ evaluates to *true* on $\overline{\mathbf{DB}}$. On the other hand, since $\widehat{HD}$ is a complete decomposition, every atom A of Q also occurs in $\bar{Q}$ (as A must occur in $\lambda(p)$ for some vertex p of T). Thus, if $\bar{Q}$ evaluates to *true* then also Q evaluates to *true*.

We build $\langle Q', \mathbf{DB}', JT\rangle$ as follows. JT has exactly the same tree shape of T. For each vertex p of T, there is precisely one vertex p' in JT, and one relation P' in $\mathbf{DB}'$. Then p' is an atom having $\chi(p)$ as arguments and its corresponding relation P' in $\mathbf{DB}'$ is the natural join of all atoms in $Q(p)$. $Q' = \bigwedge_{p \in JT} p'$ is the conjunction of all atoms corresponding to some vertex of JT.

Q' on $\mathbf{DB}'$ is clearly equivalent to $\bar{Q}$ on $\overline{\mathbf{DB}}$, and hence to Q on **DB**. JT is a join tree of Q', because the connectedness condition holds in the hypertree decomposition HD and thus in JT, by construction. Figure 8 shows the join tree JT_5 of the acyclic query Q'_5 corresponding to query Q_5 of our running example.

Each relation in **DB′** is the join of (at most) k relations of **DB**, and its size is $O(r^k)$, where r denotes the maximum relation-size over the relations in **DB**. Moreover, the number of relations in **DB′** is equal to the number of vertices of T. Therefore, $\|\mathbf{DB}'\| = O(\|\widehat{HD}\| \cdot r^k)$. Since $\|JT\| = O(\|\widehat{HD}\|)$ and $\|Q'\| = O(\|\widehat{HD}\|)$, we have $\|\langle Q', \mathbf{DB}', JT\rangle\| = O((\|Q\| + \|HD\|) \cdot r^k)$ (recall that $\|\widehat{HD}\| = O(\|Q\| + \|HD\|)$, where the term $\|Q\|$ comes from the "completion" of the decomposition HD—see above).

The described transformation mainly involves a join of at most k relations for each node of the tree. Since the join of a constant number of relations can be computed in logspace, the transformation is feasible in logspace. ▌

THEOREM 4.7. *Given a database* **DB**, *a Boolean conjunctive query* Q, *and a* k*-width hypertree-decomposition of* Q *for a fixed constant* $k > 0$, *deciding whether* Q *evaluates to true on* **DB** *is LOGCFL-complete.*

Proof. Membership in LOGCFL. From Lemma 4.6, it follows that the problem of deciding whether a Boolean conjunctive query Q evaluates to true on a database **DB**, given a k-width hypertree decomposition for Q (where k is fixed) is logspace reducible to the LOGCFL-complete problem of evaluating an acyclic conjunctive query Q' given a join tree for Q' [19].

Hardness for LOGCFL. Immediately follows from the fact that a join tree for an acyclic query Q trivially corresponds to a hypertree decomposition of Q of width 1, and from the above cited LOGCFL-hardness of the problem of evaluating an acyclic Boolean conjunctive query. ▌

Using the same construction as in the proof of Lemma 4.6 and well-known results on the sequential complexity of acyclic conjunctive-query evaluation [44], we get the following result for non-Boolean conjunctive queries.

THEOREM 4.8. *Given a database* **DB**, *a (non-Boolean) conjunctive query* Q, *and a* k*-width hypertree decomposition of* Q *for a fixed constant* $k > 0$, *the answer of* Q *on* **DB** *can be computed in time polynomial in the combined size of the input instance and of the output relation.*

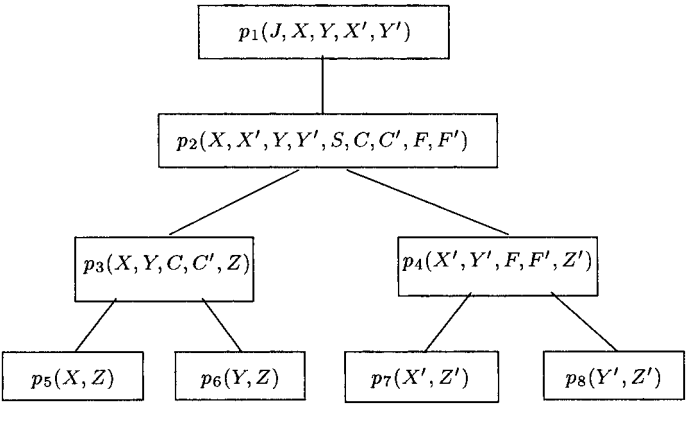

FIG. 8. Join tree JT_5 computed for query Q'_5.

Remark. In this section we demonstrated that k-bounded hypertree-width queries are efficiently computable, once a k-width hypertree decomposition of the query is given as (additional) input. In Section 5.2, we will strengthen these results showing that they remain true for queries of hypertree-width k even if a corresponding hypertree decomposition is not given. As will be seen, unlike query decompositions, bounded hypertree decompositions can be computed very efficiently.

5. BOUNDED HYPERTREE DECOMPOSITIONS ARE EFFICIENTLY COMPUTABLE

In this section we show that for any given query Q a bounded-width hypertree-decomposition for Q can be efficiently computed. In particular, we prove that this task is not only feasible in polynomial time, but it is also highly parallelizable. From the results given in the previous sections, it follows that evaluating a bounded hypertree-width query is tractable, and actually is LOGCFL-complete.

To prove these results, we first introduce a normal form for hypertree decompositions. Then, we give two algorithms for deciding whether a query has a bounded-width hypertree decomposition. The first one runs on an alternating Turing machine and proves the problem belongs to LOGCFL; the second one is a deterministic polynomial-time algorithm, implemented as a Datalog program. Another algorithm for the computation of hypertree decompositions has been recently presented in [22], and its implementation is available on the WEB [36].

5.1. Normal form

In this section we introduce a very useful normal form for hypertree decompositions.

Let $H = \langle T, \chi, \lambda \rangle$ be a hypertree of Q and $V \subseteq var(Q)$ a set of variables. We define $vertices(V, H) = \{p \in vertices(T) \mid \chi(p) \cap V \neq \varnothing\}$.

For any vertex v of T, we will often use v as a synonym of $\chi(v)$. In particular, $[v]$-*component* denotes $[\chi(v)]$-component; the term $[v]$-path is a synonym of $[\chi(v)]$-path; and so on.

DEFINITION 5.1. A hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ of a conjunctive query Q is in *normal form* (*NF*) if for each vertex $r \in vertices(T)$, and for each child s of r, all the following conditions hold:

1. there is (exactly) one $[r]$-component C_r such that $\chi(T_s) = C_r \cup (\chi(s) \cap \chi(r))$;
2. $\chi(s) \cap C_r \neq \varnothing$, where C_r is the $[r]$-component satisfying Condition 1;
3. $var(\lambda(s)) \cap \chi(r) \subseteq \chi(s)$.

Note that Condition 2 above entails that, for each vertex $r \in vertices(T)$, and for each child s of r, $\chi(s) \not\subseteq \chi(r)$. Indeed, $C_r \cap \chi(r) = \varnothing$, and s must contain some variable belonging to the $[r]$-component C_r.

Note that, according to the general definition, a hypertree decomposition can contain some redundancies, e.g., adjacent vertices labeled by the same set of variables and the same set of atoms. However, no such redundancies can occur in hypertree decompositions in normal form. As a consequence, given a query Q on some database **DB** and an NF hypertree decomposition for Q, the evaluation of Q on **DB** is much more efficient, because the upper bound on the size of the equivalent query Q' (see Lemma 4.6) can be improved. A detailed analysis of this issue has been carried out in [21].

Furthermore, we will prove some nice properties of NF hypertree decompositions that make these decompositions easy to compute, as well.

LEMMA 5.2. *Let $HD = \langle T, \chi, \lambda \rangle$ be a hypertree decomposition of a conjunctive query Q. Let r be a vertex of T, let s be a child of r, and let C be an $[r]$-component of Q such that $C \cap \chi(T_s) \neq \emptyset$. Then, $vertices(C, HD) \subseteq vertices(T_s)$.*

Proof. For any subtree T' of T, let $covered(T')$ denote the set of atoms $\{A \in atoms(Q) \mid var(A) \subseteq \chi(v)$ for some $v \in vertices(T')\}$.

We proceed by contradiction. Assume there exists some vertex $q \in vertices(C, HD)$ such that $q \notin vertices(T_s)$. Since $C \cap \chi(T_s) \neq \emptyset$, there exists a vertex $p \in vertices(T_s)$ which also belongs to $vertices(C, HD)$. By definition of $vertices(C, HD)$, there exists a pair of variables $\{X, Y\} \subseteq C$ such that $X \in \chi(p)$ and $Y \in \chi(q)$. Since $X, Y \in C$, there exists an $[r]$-*path* π from X to Y consisting of a sequence of variables $X = X_0, \ldots, X_i, X_{i+1}, \ldots, X_\ell = Y$, and a sequence of atoms $A_0, \ldots, A_i, A_{i+1}, \ldots, A_{\ell-1}$.

Note that $Y \notin \chi(T_s)$. Indeed, $Y \notin \chi(r)$, hence any occurrence of Y in $\chi(v)$, for some vertex v of T_s, would violate Condition 2 of Definition 4.1 (i.e., the connectedness condition). Similarly, X only occurs as a variable in $\chi(T_s)$. As a consequence, $A_0 \in covered(T_s)$ (by Condition 1 of Definition 4.1) and $A_{\ell-1} \notin covered(T_s)$, hence the $[r]$-path π leaves T_s, i.e., $atoms(\pi) \not\subseteq covered(T_s)$.

Assume without loss of generality that the atoms A_i, $A_{i+1} \in atoms(\pi)$ form the "frontier" of this path w.r.t. T_s, i.e., $A_i \in covered(T_s)$ and $A_{i+1} \notin covered(T_s)$, and consider the variable X_{i+1}, which occurs in both A_i and A_{i+1}. X_{i+1} belongs to C, hence it does not occur in $\chi(r)$, and this immediately yields a contradiction to Condition 2 of Definition 4.1. ∎

LEMMA 5.3. *Let $HD = \langle T, \chi, \lambda \rangle$ be a hypertree decomposition of a conjunctive query Q and $r \in vertices(T)$. If V is an $[r]$-connected set of variables in $var(Q) - \chi(r)$, then $vertices(V, HD)$ induces a (connected) subtree of T.*

Proof. We use induction on $|V|$.

Basis. If $|V| = 1$, then V is a singleton, and the statement follows from Condition 2 of Definition 4.1.

Induction Step. Assume the statement is established for sets of variables having cardinalities $c \leqslant h$. Let V be an $[r]$-connected set of variables such that $|V| = h+1$, and let $X \in V$ be any variable of V such that $V - \{X\}$ remains $[r]$-connected. (It is easy to see that such a variable exists.) By the induction hypothesis, $vertices(V - \{X\}, HD)$ induces a connected subtree of T. Moreover, $\{X\}$ is a singleton, thus $vertices(\{X\}, HD)$ induces a connected subtree of T, too.

Since $X \in V$, V is $[r]-connected$, and $|V| > 1$, there exists a variable $Y \in V - \{X\}$ which is $[r]-adjacent$ to X. Hence, there exists an atom $A \in atoms(Q)$ such that $\{X, Y\} \subseteq var(A)$. By Condition 1 of Definition 4.1, there exists a vertex $p \in vertices(T)$ such that $var(A) \subseteq \chi(p)$. Note that $vertices(V, HD) = vertices(V - \{X\}, HD) \cup vertices(\{X\}, HD)$, and p belongs to both $vertices(V - \{X\}, HD)$ and $vertices(\{X\}, HD)$. Then, both sets induce connected subgraphs of T that are, moreover, connected to each other via vertex p. Thus, $vertices(V, HD)$ induces a connected subgraph of T, and hence a subtree, because T is a tree. ∎

THEOREM 5.4. *For each k-width hypertree decomposition of a conjunctive query Q there exists a k-width hypertree decomposition of Q in normal form.*

Proof. Let $HD = \langle T, \chi, \lambda \rangle$ be any k-width hypertree decomposition of Q. We show how to transform HD into a k-width hypertree decomposition in normal form.

Assume there exist two vertices r and s such that s is a child of r, and s violates any condition of Definition 5.1. If s satisfies Condition 1, but violates Condition 2, then $\chi(s) \subseteq \chi(r)$ holds. In this case, simply eliminate vertex s from the tree as shown in Fig. 9. It is immediate to see that this transformation is correct.

Assume T_s does not meet Condition 1 of Definition 5.1, and let $C_1, \ldots, C_h$ be all the $[r]$-components containing some variable occurring in $\chi(T_s)$. Hence, $\chi(T_s) \subseteq (\bigcup_{1 \leqslant i \leqslant h} C_i \cup \chi(r))$. For each $[r]$-component C_i $(1 \leqslant i \leqslant h)$, consider the set of vertices $vertices(C_i, HD)$. Note that, by Lemma 5.3, $vertices(C_i, HD)$ induces a subtree of T, and by Lemma 5.2, $vertices(C_i, HD) \subseteq vertices(T_s)$, hence $vertices(C_i, HD)$ induces in fact a subtree of T_s.

For each vertex $v \in vertices(C_i, HD)$ define a new vertex $new(v, C_i)$, and let $\lambda(new(v, C_i)) = \lambda(v)$ and $\chi(new(v, C_i)) = \chi(v) \cap (C_i \cup \chi(r))$. Note that $\chi(new(v, C_i)) \neq \varnothing$, because by definition of $vertices(C_i, HD)$, $\chi(v)$ contains some variable belonging to C_i. Let $N_i = \{new(v, C_i) \mid v \in vertices(C_i, HD)\}$ and, for any C_i $(1 \leqslant i \leqslant h)$, let T_i denote the (directed) graph (N_i, E_i) such that $new(p, C_i)$ is a child of $new(q, C_i)$ iff p is a child of q in T. T_i is clearly isomorphic to the subtree of T_s induced by $vertices(C_i, HD)$, hence T_i is a tree, as well.

Now, transform the hypertree decomposition HD as follows. Delete every vertex in $vertices(T_s)$ from T, and attach to r every tree T_i for $1 \leqslant i \leqslant h$. Intuitively, we replace the subtree T_s by the set of trees $\{T_1, \ldots, T_h\}$. By construction, T_i contains a vertex $new(v, C_i)$ for each vertex v belonging to $vertices(C_i, HD)$ $(1 \leqslant i \leqslant h)$. Then, if we let $children(r)$ denote the set of children of r in the new tree T obtained after the transformation above, it holds that for any $s' \in children(r)$, there exists an $[r]$-component C of Q such that $vertices(T_{s'}) = vertices(C, HD)$, and $\chi(T_{s'}) \subseteq (C \cup \chi(r))$. Furthermore, it is easy to verify that all the conditions of Definition 4.1 are preserved during this transformation. As a consequence, Condition 2 of Definition 4.1 immediately entails that $(\chi(T_{s'}) \cap \chi(r)) \subseteq \chi(s')$. Hence, $\chi(T_{s'}) = C \cup (\chi(s') \cap \chi(r))$. Thus, any child of r satisfies both Condition 1 and Condition 2 of Definition 5.1.

Now, assume that some vertex $v \in children(r)$ violates Condition 3 of Definition 5.1. Then, add to the label $\chi(v)$ the set of variables $var(\lambda(v)) \cap \chi(r)$. Because variables in $\chi(r)$ induce connected subtrees of T, and $\chi(r)$ does not contain any

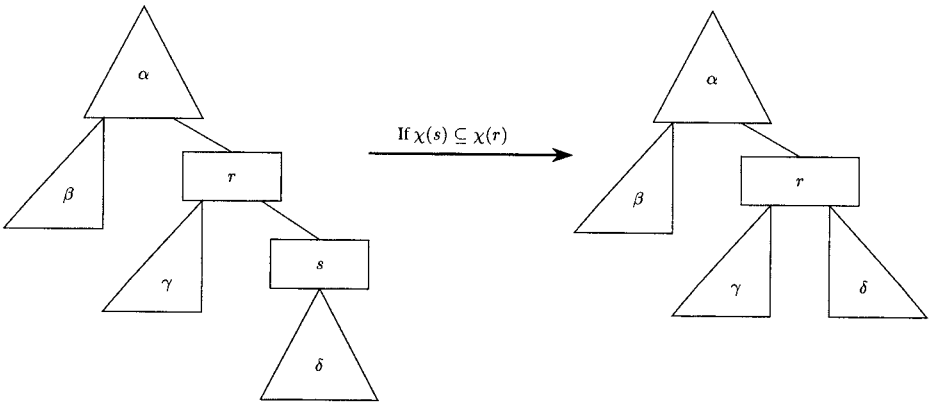

FIG. 9. Normalizing a hypertree decomposition.

variable occurring in some $[r]$-component, this further transformation never invalidates any other condition. Moreover, no new vertex is labeled by a set of atoms with cardinality greater than k, then we get in fact a legal k-width hypertree decomposition.

Note that $root(T)$ cannot violate any of the normal form conditions, because it has no parent in T. Moreover, the transformations above never change the parent r of a violating vertex s. Thus, if we apply such a transformation to the children of $root(T)$, and iterate the process on the new children of $root(T)$, and so on, we eventually get a new k-width hypertree decomposition $\langle T', \chi', \lambda' \rangle$ of Q in normal form. ▮

If $HD = \langle T, \chi, \lambda \rangle$ is an NF hypertree decomposition of a conjunctive query Q, we can associate a set $treecomp(s) \subseteq var(Q)$ to each vertex s of T as follows.

- If $s = root(T)$, then $treecomp(s) = var(Q)$;
- otherwise, let r be the parent of s in T; then, $treecomp(s)$ is the (unique) $[r]$-component C such that $\chi(T_s) = C \cup (\chi(s) \cap \chi(r))$.

Note that, since $s \in vertices(T_s)$, also $\chi(T_s) = C \cup \chi(s)$ holds.

LEMMA 5.5. *Let $HD = \langle T, \chi, \lambda \rangle$ be an NF hypertree decomposition of a conjunctive query Q, v a vertex of T, and $W = treecomp(v) - \chi(v)$. Then, for any $[v]$-componentC such that $(C \cap W) \neq \varnothing$, $C \subseteq W$ holds.*

Therefore, the set $\mathscr{C} = \{C' \subseteq var(Q) \mid C'$ is a $[v]$-component and $C' \subseteq treecomp(v)\}$ is a partition of $treecomp(v) - \chi(v)$.

Proof. Let C be a $[v]$-component such that $(C \cap W) \neq \varnothing$. We show that $C \subseteq W$. Assume this is not true, i.e., $C - W \neq \varnothing$. By definition of $treecomp(v)$, $\chi(T_v) = treecomp(v) \cup \chi(v)$. Hence, any variable $Y \in (C - W)$ only occurs in the χ label of vertices not belonging to $vertices(T_v)$. However, C is a $[v]$-component, therefore $C \cap \chi(v) = \varnothing$. As a consequence, $vertices(C, HD)$ induces a disconnected subgraph of T, and thus contradicts Lemma 5.3. ▮

LEMMA 5.6. *Let $HD = \langle T, \chi, \lambda \rangle$ be an NF hypertree decomposition of a conjunctive query Q, and r be a vertex of T. Then, $C = treecomp(s)$ for some child s of r if and only if C is an $[r]$-component of Q and $C \subseteq treecomp(r)$.*

Proof. (*If part.*) Assume C is an $[r]$-component of Q and $C \subseteq treecomp(r)$. Let $children(r)$ denote the set of the vertices of T which are children of r. Because $C \subseteq (treecomp(r) - \chi(r))$, C must be included in $\bigcup_{s \in children(r)} \chi(T_s)$. Moreover, for each subtree T_s of T such that $s \in children(r)$, there is a (unique) $[r]$-component $treecomp(s)$ such that $\chi(T_s) = treecomp(s) \cup (\chi(s) \cap \chi(r))$. Therefore, C necessarily coincides with one of these components, say $treecomp(\bar{s})$ for some $\bar{s} \in children(r)$.

(*Only if part.*) Assume $C = treecomp(s)$ for some child s of r, and let $C' = treecomp(r)$. By definition of $treecomp(s)$, C is an $[r]$-component, and hence $(C \cap \chi(r)) = \varnothing$. Since HD is in normal form, $\chi(T_s) = C \cup (\chi(s) \cap \chi(r))$ and $\chi(T_r) = (C' \cup \chi(r))$. Moreover, s is a child of r, then $vertices(T_s) \subseteq vertices(T_r)$ and thus $\chi(T_s) \subseteq \chi(T_r)$. Therefore, $C \cup (\chi(s) \cap \chi(r)) \subseteq \chi(T_r)$, and hence we immediately get $C \subseteq (C' \cup \chi(r))$. However, $(C \cap \chi(r)) = \varnothing$ and thus $C \subseteq C'$. ∎

LEMMA 5.7. *For any NF hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ of a query Q, $|vertices(T)| \leqslant |var(Q)|$ holds.*

Proof. Follows from Lemma 5.6, Lemma 5.5, and Condition 2 of the normal form, which states that, for any $v \in vertices(T)$, $\chi(v) \cap treecomp(v) \neq \varnothing$. Hence, $treecomp(v) - \chi(v) \subset treecomp(v)$ and thus, for any child s of v in T, $treecomp(s)$ is actually a proper subset of $treecomp(v)$. ∎

LEMMA 5.8. *Let $HD = \langle T, \chi, \lambda \rangle$ be an NF hypertree decomposition of a query Q, s a vertex of T, and C a set of variables such that $C \subseteq treecomp(s)$. Then, C is an $[s]$-component if and only if C is a $[var(\lambda(s))]$-component.*

Proof. Let $V = var(\lambda(s))$. By Condition 4 of Definition 4.1, $(V \cap \chi(T_s)) \subseteq \chi(s)$. Since HD is in normal form, V satisfies the following property.

$$(1) \quad (V \cap treecomp(s)) \subseteq \chi(s).$$

(*Only if part.*) Assume $C \subseteq treecomp(s)$ is an $[s]$-component. From Property 1 above, $C \cap V = \varnothing$ holds. As a consequence, for any pair of variables $\{X, Y\} \subseteq C$, X $[s]$-adjacent to Y entails X $[V]$-adjacent to Y. Hence, C is a $[V]$-connected set of variables. Moreover, $\chi(s) \subseteq V$. Then, any $[V]$-connected set which is a maximal [s]-connected set is a maximal $[V]$-connected set as well, and thus C is a $[V]$-component.

(*If part.*) Assume $C \subseteq treecomp(s)$ is a $[V]$-component. Since $\chi(s) \subseteq V$, C is clearly [s]-connected. Thus, $C \subseteq C'$, where C' is an $[s]$-component and, by Lemma 5.5, $C' \subseteq (treecomp(s) - \chi(s))$ holds. By the "only if" part of this lemma, C' is a $[V]$-component, therefore C cannot be a proper subset of C', and $C' = C$ actually holds. Thus, C is an $[s]$-component. ∎

5.2. A LOGCFL Algorithm Deciding k-Bounded Hypertree-Width

Figure 10 shows the algorithm `k-decomp`, deciding whether a given conjunctive query Q has a k-bounded hypertree-width decomposition. In that figure, we give a high level description of an alternating algorithm, to be run on an alternating Turing machine (ATM). The details of how the algorithm can be effectively implemented on a logspace ATM will be given later (see Lemma 5.15).

To each computation tree τ of `k-decomp` on input query Q, we associate a hypertree $\delta(\tau) = \langle T, \chi, \lambda \rangle$, called the *witness tree* of τ, defined as follows: For any existential configuration of τ corresponding to the "guess" of some set $S \subseteq atoms(Q)$ during the computation of *k-decomposable(C, R)*, for some $[var(R)]$-component C, (i.e., to Step 1 of `k-decomp`), T contains a vertex s. In particular, the vertex s_0 guessed at the initial call *k-decomposable(var(Q), ∅)*, is the root of T.

There is an edge between vertices r and s of T, where $s \neq s_0$, if S is guessed at Step 1 during the computation of *k-decomposable(C, R)*, for some $[var(R)]$-component C (S and R are the (guessed) sets of atoms of τ corresponding to s and r in T, respectively). We will denote C by *comp(s)*, and r by *parent(s)*. Moreover, for the root s_0 of T, we define $comp(s_0) = var(Q)$.

The vertices of T are labeled as follows. $\lambda(s) = S$ (i.e., $\lambda(s)$ is the guessed set S of atoms corresponding to s), for any vertex s of T. If $s_0 = root(T)$, let $\chi(s_0) = var(\lambda(s_0))$; for any other vertex s, let $\chi(s) = var(\lambda(s)) \cap (\chi(r) \cup C)$, where $r = parent(s)$ and $C = comp(s)$.

ALTERNATING ALGORITHM `k-decomp`
Input: A non-empty Query Q.
Result: "Accept", if Q has k-bounded hypertree-width; "Reject", otherwise.

Procedure *k-decomposable(C_R: SetOfVariables, R: SetOfAtoms)*
begin
1) **Guess** a set $S \subseteq atoms(Q)$ of k elements at most;
2) **Check** that all the following conditions hold:
   2.a) $\forall P \in atoms(C_R)$, $(var(P) \cap var(R)) \subseteq var(S)$ and
   2.b) $var(S) \cap C_R \neq \emptyset$
3) **If** the check above fails **Then Halt** and **Reject**; **Else**
   Let $\mathcal{C} := \{C \subseteq var(Q) \mid C \text{ is a } [var(S)]\text{-component and } C \subseteq C_R\}$;
4) **If, for each** $C \in \mathcal{C}$, *k-decomposable*(C, S)
   Then Accept
   Else Reject
end;

begin(* *MAIN* *)
   Accept if *k-decomposable*$(var(Q), \emptyset)$
end.

FIG. 10. A nondeterministic algorithm deciding k-bounded hypertree-width.

LEMMA 5.9. *For any given query Q such that $hw(Q) \leqslant k$,* `k-decomp` *accepts Q. Moreover, for any $c \leqslant k$, each c-width hypertree-decomposition of Q in normal form is equal to some witness tree for Q.*

Proof. Let $HD = \langle T, \chi, \lambda \rangle$ be a c-width NF hypertree decomposition of a conjunctive query Q, where $c \leqslant k$. We show that there exists an accepting computation tree τ for `k-decomp` on input query Q such that $\delta(\tau) = \langle T', \chi', \lambda' \rangle$ "coincides" with HD. Formally, there exists a bijection $f: vertices(T) \to vertices(T')$ such that, for any pair of vertices $p, q \in T$, p is a child of q in T iff $f(p)$ is a child of $f(q)$ in T', $\lambda(p) = \lambda'(f(p))$, $\lambda(q) = \lambda'(f(q))$, $\chi(p) = \chi'(f(p))$, and $\chi(q) = \chi'(f(q))$.

To this aim, we impose to `k-decomp` on input Q the following choices of sets S in Step 1:

(a) For the initial call *k-decomposable*$(var(Q), \varnothing)$, the set S chosen in Step 1 is $\lambda(root(T))$.

(b) Otherwise, for a call *k-decomposable*(C_R, R), if R is the label $\lambda(r)$ of some vertex r, and if r has a child s such that $treecomp(s) = C_R$, then choose $S = \lambda(s)$ in Step 1.

We use structural induction on trees to prove that, for any vertex $r \in vertices(T)$, if we denote $f(r)$ by r', the following equivalences hold: $\lambda(r) = \lambda'(r')$; $treecomp(r) = comp(r')$; and $\chi(r) = \chi'(r')$.

Basis: For $r = root(T)$, we set $f(root(T)) := root(T')$. Thus, by choosing $\lambda'(f(r)) = \lambda(r)$ as described in Item a) above, all the equivalences trivially hold.

Induction Step: Assume that the equivalence holds for some vertex $r \in vertices(T)$. Then, we will show that the statement also holds for every child of r. Let $r' \in vertices(T')$ denote $f(r)$, and let s be any child of r in T. By Lemma 5.6 and Lemma 5.8, the $[r]$-component $treecomp(s)$ coincides with some $[var(\lambda'(r'))]$-component $comp(s')$ corresponding to the call *k-decomposable*$(comp(s'), \lambda'(r'))$ that generated a child s' of r', which we define to be the image of s, i.e., we set $f(s) := s'$. Since HD is a k-width hypertree decomposition, and the induction hypothesis holds, it easily follows that, by choosing $\lambda(s) = \lambda'(s')$ as prescribed in Item b) above, no check performed in Step 2 of the call *k-decomposable*$(comp(s'), \lambda'(r'))$ can fail.

Next we show that $\chi(s) = \chi'(s')$. Let $C = comp(s') = treecomp(s)$, and $V = var(\lambda(s)) = var(\lambda'(s'))$. By Condition 4 of Definition 4.1, $V \cap \chi(T_s) \subseteq \chi(s)$ holds. Since HD is in normal form, we can replace $\chi(T_s)$ by $C \cup \chi(s)$ according to Condition 1 of Definition 5.1, and we get $V \cap (C \cup \chi(s)) \subseteq \chi(s)$. Hence, we obtain the following property

$$(1) \quad V \cap C \subseteq \chi(s).$$

Now, consider $\chi'(s')$. By definition of witness tree, $\chi'(s') = V \cap (\chi'(r') \cup C) = V \cap (\chi(r) \cup C)$. By Property (1) above, $V \cap C \subseteq \chi(s)$. Moreover, HD is in NF, and Condition 3 of the normal form entails that $(V \cap \chi(r)) \subseteq \chi(s)$. As a consequence, $\chi'(s') \subseteq \chi(s)$. We claim that this inclusionship cannot be proper. Indeed, by

definition of $\chi'(s')$, if $\chi'(s') \subset \chi(s)$, there exists a variable $Y \in \chi(s)$ which belongs neither to $\chi(r)$, nor to C. However, this entails that Y belongs to some other $[r]$-component and thus s violates Condition 1 of the normal form.

In summary, `k-decomp` accepts Q with the accepting computation tree τ determined by the choices described above, and its witness tree $\delta(\tau)$ is a c-width hypertree decomposition of Q in normal form. ▮

LEMMA 5.10. *Assume that* `k-decomp` *accepts an input query Q with an accepting computation tree τ and let $\delta(\tau) = \langle T, \chi, \lambda \rangle$ be the corresponding witness tree. Then, for any vertex s of T:*

(a) *if $s \neq root(T)$, then $comp(s)$ is a $[parent(s)]$-component;*

(b) *for any $C \subseteq comp(s)$, C is an $[s]$-component if and only if C is a $[var(\lambda(s))]$-component.*

Proof. We use structural induction on the tree T.

Basis: Both parts of the lemma trivially hold if s is the root of T. In fact, in this case, we have $\chi(s) = var(\lambda(s))$, by definition of witness tree.

Induction Step: Assume that the lemma holds for some vertex $r \in vertices(T)$. Then, we will show that both parts hold for every child of r. The induction hypothesis states that any $[var(\lambda(r))]$-component included in $comp(r)$ is an $[r]$-component included in $comp(r)$, and vice versa. Moreover, if $r \neq root(T)$, then $comp(r)$ is a $[\text{parent}(r)]$-component; otherwise, i.e., r is the root, $comp(r) = var(Q)$, by definition of witness tree. Let $s \in vertices(T)$ be a child of r.

(*Item a.*) The assertion of Item a. immediately follows by the definition of $comp(s)$ and by the induction hypothesis. Indeed, r is the parent of s and by the induction hypothesis any $[var(\lambda(r))]$-component included in $comp(r)$ is an $[r]$-component included in $comp(r)$. Thus, in particular, $comp(s)$ is an $[r]$-component.

(*Item b.*) Let $V = var(\lambda(s))$. We first observe that, by definition of the variable labeling χ of the witness tree, it follows that $var(\lambda(s)) \cap comp(s) \subseteq \chi(s)$. Hence, the following holds.

Fact 1: $(V - \chi(s)) \cap comp(s) = \varnothing$.

(*Only if part.*) Assume that a set of variables $C \subseteq comp(s)$ is an $[s]$-component. By Fact 1, $C \cap (V - \chi(s)) = \varnothing$ holds, and for any pair of variables $\{X, Y\} \subseteq C$, X $[s]$-adjacent to Y entails X $[V]$-adjacent to Y. Hence, C is a $[V]$-connected set of variables. Moreover, $\chi(s) \subseteq V$. Then, any $[V]$-connected set which is a maximal $[s]$-connected set is a maximal $[V]$-connected set as well, and thus C is a $[V]$-component.

(*If part.*) We proceed by contradiction. Assume $C \subseteq comp(s)$ is a $[V]$-component, but C is not an $[s]$-component, i.e., C is not a maximal $[s]$-connected set of variables. Since $\chi(s) \subseteq V$, C is clearly $[s]$-connected, then it is not maximal. That is, there exists a pair of variables $X \in C$ and $Y \notin C$ such that X is $[s]$-adjacent to Y, but X is not $[\text{var}(\lambda(s))]$-adjacent to Y. Let A be any atom proving their adjacency w.r.t. s, i.e., $\{X, Y\} \subseteq var(A) - \chi(s)$. Hence, because $X \in C$ and X is not $[V]$-adjacent to Y, it follows that $Y \in (V - \chi(s))$. By Fact 1, $(V - \chi(s)) \cap comp(s) = \varnothing$,

therefore $Y \notin comp(s)$. In summary, $X \in comp(s)$ and $Y \notin comp(s)$. Moreover, $comp(s) \subseteq comp(r)$, by Step 4 of k-decomp. Hence, by induction hypothesis, $comp(s)$ is an $[r]$-component and thus X is not $[r]$-adjacent to Y. Consider again the atom A. We get $\{X, Y\} \not\subseteq var(A) - \chi(r)$. Since $X \in comp(s)$, the variable Y must belong to $\chi(r)$. However, by definition of witness tree, $Y \in \chi(r)$ and $Y \in var(\lambda(s))$ entail that $Y \in \chi(s)$, which is a contradiction. ▮

LEMMA 5.11. *Assume that* k-decomp *accepts an input query Q with an accepting computation tree τ. Let $\delta(\tau) = \langle T, \chi, \lambda \rangle$ be the corresponding witness tree, and $s \in vertices(T)$. Then, for each vertex $v \in T_s$:*

$$\chi(v) \subseteq comp(s) \cup \chi(s)$$
$$comp(v) \subseteq comp(s).$$

Proof. We use induction on the distance $d(v, s)$ between any vertex $v \in vertices(T_s)$ and s. The basis is trivial, since $d(v, s) = 0$ means $v = s$.

Induction Step. Assume both statements hold for distance n. Let $v \in vertices(T_s)$ be a vertex such that $dist(v, s) = n+1$. Let v' be the parent of v in T_s. Clearly, $dist(v', s) = n$, thus

(a) $\chi(v') \subseteq (comp(s) \cup \chi(s))$; and

(b) $comp(v') \subseteq comp(s)$.

v is generated by some call $k\text{-}decomposable(comp(v), \lambda(v'))$. By the choice of v and the definition of witness tree, it must hold (a′) $\chi(v) \subseteq (comp(v) \cup \chi(v'))$, and by Step 4 of the call $k\text{-}decomposable(comp(v'), \lambda(parent(v')))$ we get (b′) $comp(v) \subseteq comp(v')$. By (a′) and (b′), we obtain (a″) $\chi(v) \subseteq (comp(v') \cup \chi(v'))$. By (a″), (b), and (a) we get $\chi(v) \subseteq (comp(s) \cup \chi(s))$. Moreover, (b) and (b′) yield $comp(v) \subseteq comp(s)$. ▮

LEMMA 5.12. *Let Q be a query such that* k-decomp *accepts Q, and $\langle T, \chi, \lambda \rangle$ the witness tree of an accepting computation tree of* k-decomp *on Q. Let s be any vertex of T, and let $C_r = comp(s)$. Then, for each $P \in atoms(C_r)$, it holds that $\forall A \in (atoms(Q) - atoms(C_r))$, $(var(P) \cap var(A)) \subseteq \chi(s)$.*

Proof. We use structural induction on the tree T.

Basis: The lemma trivially holds if s is the root of T. In fact, in this case, we have $C_r = comp(s) = var(Q)$ and hence $atoms(C_r) = atoms(Q)$.

Induction Step: Assume that the statement holds for some vertex $s \in vertices(T)$. Then, we will show that it also holds for every child of s. Let $C_r = comp(s)$ and $V = var(\lambda(s))$. The induction hypothesis states that $\forall P \in atoms(C_r)$ and $\forall A \notin atoms(C_r)$, $var(P) \cap var(A) \subseteq \chi(s)$. By Step 4 of k-decomp, for each $[V]$-component C such that $C \subseteq C_r$, T contains a vertex q such that $comp(q) = C$ and $parent(q) = s$. Moreover, by Lemma 5.10, C is an $[s]$-component. Let P' belong to $atoms(C)$. Since $C \subseteq C_r$, P' also belongs to $atoms(C_r)$. First note that, $\forall A \notin atoms(C)$, we have

$$(1) \quad var(P') \cap var(A) \subseteq \chi(s).$$

Indeed, if $var(A) \subseteq \chi(s)$, then (1) is trivial, and if A contains some variable belonging to another $[s]$-*component*, it immediately follows by definition of $[s]$-component.

Let s' be a child of s in T. Then, there exists a $[V]$-*component* $C_s \subseteq C_r$ such that $C_s = comp(s')$, i.e., Step 1 of *k-decomposable*$(C_s, \lambda(s))$ guessed the atoms in $\lambda(s')$ for decomposing the component C_s. Let P' an atom belonging to $atoms(C_s)$. Since $\langle T, \chi, \lambda \rangle$ is the witness tree of an accepting computation tree, $\forall B \in atoms(C_s)$, $(var(B) \cap var(\lambda(s))) \subseteq var(\lambda(s'))$. In particular, $(var(P') \cap var(\lambda(s))) \subseteq var(\lambda(s'))$. Because $\chi(s) \subseteq var(\lambda(s))$, this yields $(var(P') \cap \chi(s)) \subseteq (var(\lambda(s')) \cap \chi(s))$. By definition of witness tree, $(var(\lambda(s')) \cap \chi(s)) \subseteq \chi(s')$, hence we get $(var(P') \cap \chi(s)) \subseteq \chi(s')$. By combining this result with relationship (1) above, we get that $\forall A \notin atoms(C_s)\ (var(P') \cap var(A)) \subseteq (var(P') \cap \chi(s)) \subseteq \chi(s')$. ▌

LEMMA 5.13. *If* `k-decomp` *accepts an input query* Q*, then* $hw(Q) \leqslant k$*. Moreover, each witness tree for* Q *is a* c*-width hypertree-decomposition of* Q *in normal form, where* $c \leqslant k$*.*

Proof. Assume that τ is an accepting computation tree of `k-decomp` on input query Q. We show that $\delta(\tau) = \langle T, \chi, \lambda \rangle$ is an NF c-width hypertree decomposition of Q, for some $c \leqslant k$.

First, we will prove that $\delta(\tau)$ fulfils all the properties of Definition 4.1 and is thus a hypertree decomposition of Q.

Property 1. $\forall A \in atoms(Q)\ \exists v \in vertices(T)\ s.t.\ var(A) \subseteq \chi(v)$.
We first prove the following claim.

CLAIM A. *Let* s *be any vertex of* T*, and let* $C_r = comp(s)$*. Then, for each* $P \in atoms(C_r)$*, either* $var(P) \subseteq \chi(s)$ *or there exists an* $[s]$-component $C_s \subseteq C_r$ such that $P \in atoms(C_s)$.

Proof of Claim A. Let s be any vertex of T, let $C_r = comp(s)$, and let P belong to $atoms(C_r)$. Assume by contradiction that $var(P) \not\subseteq \chi(s)$ and that $P \in atoms(C'_s)$, where C'_s is an $[s]$-component not included in C_r, i.e., $C'_s \not\subseteq C_r$. Then, there exists a variable $Y \in C'_s$ such that $Y \notin C_r$ and there is an $[s]$-path from Y to each variable in $var(P) - \chi(s)$.

Let A be an atom belonging to both $atoms(C_r)$ and $atoms(C'_s)$. Then, $var(A) - \chi(r) \subseteq C_r$ and $var(A) - \chi(s) \subseteq C'_s$ hold. As a consequence, $(var(A) \cap C'_s) \subseteq C_r$. Indeed, if this is not true, there exists a variable $Z \in (var(A) - \chi(s))$ such that $Z \in \chi(r)$ and hence, by construction, $Z \in \lambda(r)$. By definition of the χ labeling of a witness tree, since $Z \in (var(A) \cap var(\lambda(r)))$ and we assumed $Z \notin \chi(s)$, it follows that Z does not belong to $var(\lambda(s))$. However, this contradicts the fact that A satisfies the condition checked at Step 2.a of `k-decomp`, because τ is an accepting computation tree. Note that, by assumption, both $P \in atoms(C_r)$ and $P \in atoms(C'_s)$ hold. Hence, there exists a variable $X \in var(P) - \chi(s)$ such that both $X \in C_r$ and $X \in C'_s$. From the latter condition, it follows that there exists an $[s]$-*path* π from Y to X.

Therefore, there exist two atoms $\{Q', P'\} \subseteq atoms(\pi)$ belonging to $atoms(C'_s)$ and adjacent in π such that $Q' \notin atoms(C_r)$, $P' \in atoms(C_r)$, and $var(Q') \cap var(P') \not\subseteq \chi(s)$. However, by Lemma 5.12, this is a contradiction. ▌

Note that, by Lemma 5.10, in the Step 4 of any call *k-decomposable*$(C, \lambda(r))$ of an accepting computation of k-decomp, $[s]$-components included in C and $[var(\lambda(s))]$-components included in C coincide. Thus, Property 1 follows by inductive application of Claim A. In fact, Claim A applied to the root s_0 of T states that, $\forall A \in atoms(Q)$, either $var(A) \subseteq \chi(s_0)$, or $A \in atoms(C_S)$ for some $[S]$-component C_S of Q that will be further treated in Step 4 of the algorithm. Thus, $var(A)$ is covered eventually by some chosen set of atoms S, i.e., there exists some vertex s of T, such that $\lambda(s) = S$, and $var(A) \subseteq \chi(s)$.

Property 2. *For each variable* $Y \in var(Q)$ *the set* $\{v \in vertices(T) \mid Y \in \chi(v)\}$ *induces a connected subtree of* T.

Assume that Property 2 does not hold. Then, there exists a variable $Y \in var(Q)$ and two vertices v_1 and v_2 of T such that $Y \in (\chi(v_1) \cap \chi(v_2))$ but the unique path from v_1 to v_2 in T contains a vertex w such that $Y \notin \chi(w)$. W.l.o.g, assume that v_1 is adjacent to w and that v_2 is a descendant of w in T, i.e., $v_2 \in vertices(T_w)$. There are two possibilities to consider:

- v_1 is a child of w and v_2 belongs to the subtree T_p of another child p of w. However, this would mean that, by Step 4 of k-decomp and by Lemma 5.11, the variables in sets $V_1 = (\chi(v_1) - \chi(w))$ and $V_2 = (\chi(v_2) - \chi(w))$ belong to distinct $[w]$-components. But this is not possible, because $Y \in (V_1 \cap V_2)$.
- w is a child of v_1 and v_2 belongs to the subtree T_w of T rooted at w. Then, $\lambda(w)$ was chosen as set S in Step 1 of k-decomposable$(C, \lambda(v_1))$, where C is a $[v_1]$-component. Note that $Y \in \chi(v_1)$ entails $Y \notin C$, by definition of $[v_1]$-component. Since v_2 belongs to the subtree T_w, by Lemma 5.11 it holds that $\chi(v_2) \subseteq (C \cup \chi(w))$. This is a contradiction, because $Y \in \chi(v_2)$, but Y belongs neither to $\chi(w)$, nor to C.

Property 3. $\forall p \in vertices(T)$, $\chi(p) \subseteq var(\lambda(p))$.
Follows by definition of the χ labeling of a witness tree.

Property 4. $\forall p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.
Let v be any vertex in T_p, and let $V = var(\lambda(p))$. By Lemma 5.11, $\chi(v) \subseteq comp(p) \cup \chi(p)$. Hence, $V \cap \chi(v) \subseteq (V \cap comp(p)) \cup \chi(p)$, because Property 3 holds for p. However, by definition of witness tree, $(V \cap comp(p)) \subseteq \chi(p)$, and thus $(V \cap \chi(v)) \subseteq \chi(p)$.

Thus, $\delta(\tau)$ is a hypertree decomposition of Q. Let c be the width of $\delta(\tau)$. Since Step 1 of k-decomp only chooses sets of atoms having cardinality bounded by k, $c \leqslant k$ holds.

Moreover, $\delta(\tau)$ is in normal form. Indeed, Condition 2 and Condition 3 of Definition 5.1 hold by Step 2.b of k-decomp, and by definition of the χ labeling of a witness tree. Finally, since $\delta(\tau)$ is a hypertree decomposition, by Lemma 5.2, Lemma 5.11, and the definition of the χ labeling of a witness tree, we get that Condition 1 holds for $\delta(\tau)$, too. ▮

By combining Lemma 5.9 and Lemma 5.13 we get:

THEOREM 5.14. k-decomp *accepts an input query* Q *if and only if* $hw(Q) \leqslant k$.

LEMMA 5.15. `k-decomp` *can be implemented on a logspace ATM having polynomially bounded tree-size.*

Proof. Let us refer to logspace ATMs with polynomially bounded tree-size as LOGCFL-ATMs. We will outline how the algorithm `k-decomp` can be implemented on a LOGCFL-ATM M.

We first describe the data-structures used by M. Instead of manipulating atoms directly, *indices* of atoms will be used in order to meet the logarithmic space bound. Thus the i-th atom occurring in the given representation of the input query Q will be represented by integer i. Sets of at most k atoms, k-sets for short, are represented by k-tuples of integers; since k is fixed, representing such sets requires logarithmic space only. Variables are represented as integers, too.

If R is a k-set, then a $[var(R)]$-component C is represented by a pair $\langle rep(R), first(C)\rangle$, where $rep(R)$ is the representation of the k-set R, and $first(C)$ is the smallest integer representing a variable of the component C. For example, the $\varnothing$-component $var(Q)$ is represented by the pair $\langle rep(\varnothing), 1\rangle$. It is thus clear that $[var(R)]$-components can be represented in logarithmic space, too.

The main data structures carried with each configuration of M consist of (the representations of):

- a k-set R,[4]
- a $[var(R)]$-component C_R,
- a k-set S, and
- a $[var(S)]$-component C.

Not all these items will contain useful data in all configurations. We do not describe further auxiliary logspace data structures that may be used for control tasks and for other tasks such as counting or for performing some of the SL subtasks described below.

We are now ready to give a description of the computation M performs on an input query Q.

To facilitate the description, we will specify some subtasks of the computation, that are themselves solvable in LOGCFL, as macro-steps without describing their corresponding computation (sub-)trees. We may imagine a macro-step as a special kind of configuration—termed *oracle configuration*—that acts as an oracle for the subtask to be solved.

Each oracle configuration can be *normal* or *converse*.

A *normal* oracle configuration has the following effect. If the subtask is negative, this configuration has no children and amounts to a REJECT. Otherwise, its value (ACCEPT or REJECT) is identical to the value of its unique successor configuration.

A *converse* oracle configuration has the following effect. If the subtask is negative, this configuration has no children and amounts to an ACCEPT. Otherwise, its

[4] The separate representation of R is actually slightly redundant, given that R also occurs in the description of the $[R]$-component C_R.

value (ACCEPT or REJECT) is identical to the value of its unique successor configuration.

From the definition of logspace ATMs with polynomial tree-size, it follows that any polynomially tree-sized logspace ATM M with LOGCFL oracle configurations (where an oracle configuration contributes 1 to the size of an accepting subtree) is equivalent to a standard logspace ATM having polynomial tree size.

M is started with R initialized to the empty set and C_R having value $var(Q)$.

We describe the evolution of M corresponding to a call to the procedure *k-decomposable*(C_R, R).

Instruction 1 is performed by guessing an arbitrary k-set S of atoms.

The "Guess" phase of Instruction 1 is implemented by an existential configuration of the ATM. (Actually, it is implemented by a subtree of existential configurations, given that a single existential configuration can only guess one bit; note however, that each accepting computation tree will contain only one branch of this subtree.)

Checking Step 2 is in symmetric logspace (SL). The most difficult task is to enumerate atoms of $atoms(C_R)$, which in turn—as most substantial subtask—requires to enumerate the variables of C_R. Remember that C_R is given in the form $\langle rep(R), i\rangle$ as described above. Thus, enumerating C_R amounts to cycling over all variables j and checking whether j is $[R]$-connected to i. The latter subtask is easily seen to be in SL because it essentially amounts to a connectedness-test of two vertices in an undirected graph. It follows that the entire checking-task of Instruction 2 is in SL. Since SL $\subseteq$ LOGCFL, this corresponds to a LOGCFL-subtask. We can thus assume that the checking-task is performed by some normal oracle configuration. If the oracle computation fails at some branch, the branch ends in a REJECT, otherwise, the guessed k-set S corresponding to that branch satisfies all the conditions checked by Step 2 of `k-decomp`.

Steps 4 intuitively corresponds to a "big" universal configuration that universally quantifies over all subtrees corresponding to the calls *k-decomposable*(C, S) for all $C \in \mathscr{C}$. This could be realized as follows. First, a subtree of universal configurations enumerates all candidates $C_i = \langle rep(S), i\rangle$ for $1 \leqslant i \leqslant |var(Q)|$, for $[var(S)]$-components. Each branch of this subtree (of polynomial depth) computes exactly one candidate C_i. Each such branch is expanded by a converse oracle configuration checking whether C_i is effectively a $[var(S)]$-component contained in C_R. Thus, branches that do not correspond to such a component are terminated with an ACCEPT configuration (they are of no interest), while all other branches are further expanded. Each branch C_i of the latter type is expanded by the subtree corresponding to the recursive call *k-decomposable*(C_i, S).

We have thus completely described a logspace ATM M with oracle configurations that implements `k-decomp`. It is easy to see that this machine has polynomial-size accepting computation trees. In fact, this is seen from the fact that there exists only a polynomial number of choices for set S in Step 1, and that no such set is chosen twice in any accepting computation tree. ▌

The above results entail that bounded hypertree-width queries are efficiently recognizable.

THEOREM 5.16. *Deciding whether a conjunctive query Q has k-bounded hypertree-width is in LOGCFL.*

Proof. Follows from Theorem 5.14, Lemma 5.15, and Proposition 2.3. ▮

In fact, the following proposition states that an accepting computation tree of a bounded-treesize logspace ATM can be *computed* in (the functional version of) LOGCFL.

PROPOSITION 5.17 [20]. *Let M be a bounded-treesize logspace ATM recognizing a language A. It is possible to construct an $\mathrm{L}^{\mathrm{LOGCFL}}$ transducer T which for each input $w \in A$ outputs a single (polynomially-sized) accepting tree for M and w.*

Thus, we immediately obtain that (bounded-width) hypertree decompositions are efficiently *computable*, as well.

THEOREM 5.18. *Computing a k-bounded hypertree decomposition (if any) of a conjunctive query Q is in $\mathrm{L}^{\mathrm{LOGCFL}}$, i.e., in functional* LOGCFL.

Proof. From Lemma 5.9 and Lemma 5.13, it follows that, for any given query Q, there exists a one-to-one correspondence between NF hypertree decompositions of Q having width at most k and witness trees of accepting computation trees of `k-decomp`. It is easy to see that a witness tree can be computed in logspace from an accepting computation tree of k-`decomp`. Moreover, $\mathrm{L} \circ \mathrm{L}^{\mathrm{LOGCFL}} = \mathrm{L}^{\mathrm{LOGCFL}}$ [20]. Thus, a hypertree decomposition of Q can be computed in $\mathrm{L}^{\mathrm{LOGCFL}}$, by Proposition 5.17. ▮

Since LOGCFL is closed under $\mathrm{L}^{\mathrm{LOGCFL}}$ reductions [20], the two following statements follow from the theorem above and Theorem 4.7 and Theorem 4.8, respectively.

COROLLARY 5.19. *Deciding whether a k-bounded hypertree-width query Q evaluates to true on a database* **DB** *is LOGCFL-complete.*

COROLLARY 5.20. *The answer of a (non-Boolean) k-bounded hypertree-width query Q can be computed in time polynomial in the combined size of the input instance and of the output relation.*

In Appendix 2, we also present a simple Datalog program for computing a hypertree decomposition of a given query. Another algorithm for computing hypertree decompositions has been recently presented in [22], and its implementation is available on the WEB [36]. Given a query Q and an integer $k > 0$, this algorithm returns an *optimal* hypertree decomposition for Q, i.e., a hypertree decomposition of width $hw(Q)$, if $hw(Q) \leqslant k$; otherwise, it answers that $hw(Q) > k$.

6. BOUNDED HYPERTREE-WIDTH VS RELATED NOTIONS

Many relevant cyclic queries are—in a precise sense—close to acyclic queries because they can be reduced via bounded-width decompositions to acyclic queries.

A similar phenomenon has been observed in artificial intelligence for *Constraint Satisfaction Problems (CSPs)*, and several decomposition methods have been developed for dealing with cyclic CSP instances. For a formal definition of CSPs, see, e.g., [27].

As pointed out by various authors [5, 25, 11, 29, 19], there is a tight relationship between CSPs and database problems such as the evaluation of conjunctive queries. Actually, the problems can be considered to be equivalent. Indeed, they can be also recast as the same fundamental algebraic problem of deciding whether, given two finite relational structures A and B, there exists a homomorphism $f : A \to B$ [29].

It follows that the CSP research area can profit from new developments on conjunctive queries and vice-versa. In [21] we compared the decomposition methods developed in AI to the new method of hypertree decompositions introduced here. It turned out that the method of bounded hypertree decomposition is strictly more general than the CSP decomposition methods so far developed in AI. In particular, we have compared the hypertree decomposition method to the structural CSP decomposition methods, including those based on *hinges* [25, 26], *biconnected components* [15], *cycle cutsets* [11], *tree clustering* [12], and *treewidth* [2]. We say that a method M is *applicable* to a class (possibly infinite set) of constraints (or queries) if every problem instance belonging to this class is classified as *tractable* according to method M. The main result in [21] is that the new method of bounded hypertree decomposition is applicable to a class of constraints (queries) whenever any of these methods is applicable. On the other hand, for each method M of the above list, there exist classes of (non binary) constraints having bounded hypertree-width, to which method M is not applicable.

Following [21], it is easy to see that, as for hypertree-width, the class of queries having bounded query-width encompasses all the above cited decomposition methods. By Theorem 3.4, deciding whether a query has bounded query-width is an NP-complete problem. Nevertheless, we next show that also this class is properly included in the class of queries of bounded hypertree-width. More precisely, we show that every k-width query-decomposition corresponds to an equivalent k-width hypertree-decomposition, but the converse is not true, in general. Recall that $hw(Q)$ and $qw(Q)$ denote the hypertree-width and the query-width of a conjunctive query Q.

THEOREM 6.1. (a) *For each conjunctive query Q it holds that $hw(Q) \leqslant qw(Q)$.*

(b) *There exist queries Q such that $hw(Q) < qw(Q)$.*

Proof. (a) Let Q be a conjunctive query and $\langle T, \lambda \rangle$ a query decomposition of Q. Without loss of generality, assume Q is pure (i.e., labels contain only atoms, see Section 3.1). Then, (T, χ, λ) is a hypertree decomposition of Q, where, for any vertex v of T, $\chi(v)$ consists of the set of variables $var(\lambda(v))$ occurring in the atoms $\lambda(v)$. Indeed, because the properties of query decompositions hold for $\langle T, \lambda \rangle$, $\langle T, \chi, \lambda \rangle$ verifies Condition 1 and 2 of Definition 4.1. Condition 3 and 4 follows immediately, as $\chi(p) = var(\lambda(p))$ by construction. Therefore, $hw(Q) \leqslant qw(Q)$.

(b) The query Q_5 of Example 4.3 has no query decompositions of width 2. However, Figure 5 shows a query decomposition of Q_5 having width 3, and thus $qw(Q_5) = 3$ holds. However, $hw(Q_5) = 2$, as witnessed by the hypertree decomposition shown in Figure 6. ∎

We assume the reader is familiar with the notion of *graph treewidth* (cf. [2, 34]). There are two possibilities to define the treewidth of a conjunctive query according to whether one considers the *primal graph* $G(Q)$ of a query (also called its Gaifman graph), or the *variable-atom incidence graph* $VAIG(Q)$ of Q (see also [29]). The primal graph $G(Q)$ has as vertices the variables of Q, and as edges all pairs $\{X, Y\}$ such that both X and Y occur together in some atom of Q. The variable-atom incidence graph $VAIG(Q)$ is the bipartite graph whose vertices are both the atoms of Q and the variables of Q, and whose edges connect an atom A with a variable X if and only if X occurs in A. From results in [21], it immediately follows that there are classes of queries having an unbounded treewidth with respect to their primal graph but having hypertree width 1. Let us here deal with the treewidth of a query Q defined as the treewidth of $VAIG(Q)$. Denote this treewidth by $tw(Q)$. Chekuri and Rajaraman [9] proved that, for any query Q, $tw(Q)/a \leqslant qw(Q) \leqslant tw(Q)+1$, where a is the maximum predicate arity in Q. In [9] the assumption of bounded query predicate arity a was made. Under this assumption, the above inequation $tw(Q)/a \leqslant qw(Q)$ implies that whenever the query width of a class of queries is bounded by a constant, then also the treewidth of the queries in this class is bounded by a constant. As shown below, it is easy to see that this is not valid in the general case.

THEOREM 6.2. *There is a class C of queries having query width 1, and thus hypertree width 1, but unbounded treewidth.*

Proof. Consider the set of variables $V_n = W_n \cup U_n$ where $W_n = \{X_1, \ldots, X_n\}$ and $U_n = \{Y_1, \ldots, Y_n\}$. Let Q_n be defined by

$$Q_n = ans \leftarrow q(X_1, \ldots, X_n, Y_1) \wedge q(X_1, \ldots, X_n, Y_2) \wedge q(X_1, \ldots, X_n, Y_n).$$

Let C be the class consisting of all Q_n for all positive integers n. Each Q_n has query width 1. In fact, a query decomposition of width 1 can be obtained by taking the first atom $q(X_1, \ldots, X_n, Y_1)$ as root and attaching every other atom of Q_n as child to this root. On the other hand, we have $tw(Q_n) = n$. In fact, the graph $VAIG(Q_n)$ contains as subgraph the complete bipartite graph $atoms(Q) \times W_n$ which has treewidth n, while the other edges relating each variable of U_n to a single atom of Q do not contribute to the treewidth. ∎

7. NP-COMPLETENESS OF BOUNDED QUERY-WIDTH (PROOF)

In this section, we provide a formal proof of Theorem 3.4. We need some preliminary results and definitions.

A *k-element-vertex* of a query decomposition (T, λ) is a vertex v of T such that $|\lambda(v)| = k$.

LEMMA 7.1. *Let Q be a query having variable set $var(Q) = \Gamma \cup Rest$, where*

$$\Gamma = \{V_{ij} \mid 1 \leqslant i < j \leqslant 8\},$$

and Rest is an arbitrary set of further variables. Assume the set atoms(Q) contains as subset a set $\Pi = \{P_1, \ldots, P_8\}$ *of 8 atoms, where, for* $1 \leqslant i \leqslant 8$, $var(P_i) \cap \Gamma = \{V_{1i}, V_{2i}, \ldots, V_{i-1\,i}, V_{i\,i+1}, \ldots, V_{i8}\}$, *i.e.,*

$$var(P_i) \cap \Gamma = \bigcup_{k<i} \{V_{ki}\} \cup \bigcup_{i<k} \{V_{ik}\}.$$

and $\forall A \in atoms(Q) - \Pi : var(A) \cap \Gamma = \varnothing$.

If Q admits a pure query decomposition (T, λ) *of width 4, then there exist two adjacent 4-element-vertices* p_1 *and* p_2 *of T such that* $\lambda(p_1) \cup \lambda(p_2) = \Pi$.

Proof. For each subgraph G of T, denote by $\lambda(G)$ the union of all $\lambda(v)$ such that v is a vertex of G. Moreover, for any branch B of T and vertices v_1, v_2 on B, $[v_1, v_2]$ denotes the segment of B whose extremal vertices are v_1 and v_2.

Root the tree $T = (V, E)$ in an arbitrary vertex r. Assume for each branch B of the rooted tree T, $\Pi \not\subseteq \lambda(B)$. Let Δ be a set of branches of T such that $\Pi \subseteq \lambda(\Delta)$ and such that Δ is minimal with respect to this property. We have $|\Delta| \geqslant 2$. Let B_i and B_j be two distinct branches of Δ. From the minimality of Δ it follows that $\lambda(B_i) \cap \Pi \not\subseteq \lambda(B_j) \cap \Pi$ and $\lambda(B_j) \cap \Pi \not\subseteq \lambda(B_i) \cap \Pi$. Therefore, there exist atoms $P_i, P_j \in \Pi$ such that $P_i \in \lambda(B_i)$, $P_i \notin \lambda(B_j)$ $P_j \in \lambda(B_j)$, and $P_j \notin \lambda(B_i)$. But this violates the connectedness condition. In fact, the atoms P_i and P_j have a common variable X ($X = V_{ij}$ or $X = V_{ji}$) that occurs in no other atoms than in P_i and P_j. Let v_c be the lowest common vertex belonging to both branches B_i and B_j. By the connectedness condition, $X \in var(\lambda(v_c))$, and thus either $P_i \in \lambda(v_c)$ or $P_j \in \lambda(v_c)$. But then one of P_i and P_j belongs to both branches B_i and B_j, which results in a contradiction. Therefore, there must exist a branch B^* such that $\Pi \subseteq \lambda(B^*)$. In the rest of the proof we will work with this branch B^*. The top vertex of B^* is the root r of T and its bottom vertex is some leaf s.

Let w be the lowest vertex v in B^* such that $|\lambda([v, s])| \geqslant 4$, and let w' be the parent of w. If $w = s$, then $|\lambda([w, s])| = |\lambda(w)| \leqslant 4$. Otherwise, let w'' be the child of w belonging to the branch B^*. By the choice of w, $|\lambda([w'', s])| \leqslant 3$. Moreover, $|\lambda(w)| \leqslant 4$, and therefore $|\lambda([w, s])| \leqslant 7$ holds. Since $|\Pi| = 8$, there exists an atom $P \in \Pi$ which occurs in $\lambda([r, w'])$ but not in $\lambda([w, s])$. Given that for each atom P' in $\lambda([w, s]) \cap \Pi$, P and P' share variables that occur in no other atoms, and given that P itself does not belong to $\lambda([w, s])$, it follows from the connectedness condition that all atoms of $\lambda([w, s])$ occur in $\lambda(w)$. Thus, $|\lambda(w) \cap \Pi| = 4$, i.e., $\lambda(w)$ contains exactly 4 atoms of Π. Moreover, $|\lambda([w, s]) \cap \Pi| = 4$. Let u be the first ancestor of w such that $\lambda(u) \neq \lambda(w)$, and let u' be its child. There is an atom $P_k \in \Pi$ contained in $\lambda(u')$ but not in $\lambda(u)$. Let Ω denote the 4 atoms in $\Pi - \lambda([u's])$. If an atom P_h of Ω did not belong to $\lambda(u)$, then the connectedness condition would be violated for the common variable of P_k and P_h which occurs exclusively in these two atoms. Therefore, u and u' are two adjacent 4-element-vertices of T such that $\lambda(u) \cup \lambda(u') = \Pi$. ∎

DEFINITION 7.2. Let S be a set of n elements. A *3-partition* $\{S_a, S_b, S_c\}$ of S consists of three nonempty subsets $S_a, S_b, S_c \subset S$ such that $S_a \cup S_b \cup S_c = S$, and $S_x \cap S_y = \varnothing$ for $x \neq y$ from $\{a, b, c\}$. The sets S_a, S_b, S_c are referred to as *classes*.

A 3-Partitioning-System (short *3PS*) Σ on a base set S is a set of 3-partitions of S,

$$\Sigma = \{ \{S_a^1, S_b^1, S_c^1\}, \{S_a^2, S_b^2, S_c^2\}, \ldots, \{S_a^m, S_b^m, S_c^m\}\},$$

where $\forall \sigma, \sigma' \in \Sigma$: $\sigma \neq \sigma' \Rightarrow \sigma \cap \sigma' = \varnothing$ (i.e., no class occurs in two or more elements of Σ).

We define $classes(\Sigma) := \bigcup_{\sigma \in \Sigma} \sigma$. The base set S of Σ is referred to as $base(\Sigma)$: $base(\Sigma) = \bigcup_{C \in classes(\Sigma)} C$.

A 3PS is *strict* if for all $S', S'', S''' \in classes(\Sigma)$ either $\{S', S'', S'''\} = \sigma$ for some $\sigma \in \Sigma$ or $S' \cup S'' \cup S''' \subset S$. In other words: the only way to obtain S as a union of three classes is via the specified 3-partitions of Σ; any other union of three classes results in a proper subset of S.

A 3PS Σ is referred to as an (m, k)-*3PS* if $|\Sigma| \geqslant m$ and $\forall C \in classes(\Sigma) : |C| \geqslant k$.

LEMMA 7.3. *For each $m > 0$ and $k > 0$, a strict (m, k)-3PS can be computed in $O(m^2 + km)$ time.*

Proof. Fix m and k. We will construct a set S such that there exists a strict (m, k)-3PS for S. Let $T = \{X_1, \ldots, X_{3k+m}\}$, $T' = \{X'_1, \ldots, X'_m\}$, and $T'' = \{X''_a, X''_b, X''_c\}$. Moreover, let $S = T \cup T' \cup T''$ and, for $1 \leqslant i \leqslant m$,

- $S_a^i = \{X_1, \ldots, X_{k+i-1}\} \cup \{X'_1, \ldots, X'_{m-i}\} \cup \{X''_a\}$
- $S_b^i = \{X_{k+i}, \ldots, X_{2k+i-1}\} \cup \{X''_b\}$
- $S_c^i = \{X_{2k+i}, \ldots, X_{3k+m}\} \cup \{X'_{m-i+1}, \ldots, X'_m\} \cup \{X''_c\}$

Then, $\Sigma = \{\{S_a^i, S_b^i, S_c^i\} \mid 1 \leqslant i \leqslant m\}$ is clearly an (m, k)-3PS for S. We next show that Σ is strict. Note that, because of the subset T'', the only way to cover S by three sets is by choosing three classes of the form $\{S_a^r, S_b^s, S_c^t\}$, for some triple of indexes $r, s, t \in \{1, \ldots, m\}$. Moreover, to cover the subset T', it must hold that $m - r \geqslant m - t$, and hence $r \leqslant t$. Assume by contradiction that r is strictly less than t. Then, with S_a^r and S_c^t, we cover at most the elements $\{X_1, \ldots, X_{k+t-2}\} \cup \{X_{2k+t}, \ldots, X_{3k+m}\}$ from the subset T. This means that we miss at least the $k+1$ elements $\{X_{k+t-1}, \ldots, X_{2k+t-1}\}$. However, by construction, for any s, S_b^s contains at most k elements from T. Thus, S_b^s cannot cover the missing elements from T, and we get a contradiction. It follows that $r = t$ holds, and it is easy to see that this entails $s = r = t$, too.

Note that each triple in Σ contains all the elements in S and that there are m triples. Thus, Σ can be clearly constructed in time $O(m(3k+m+m+3)) = O(m^2 + km)$. ∎

We can now prove Theorem 3.4, which states that deciding whether the query-width of a conjunctive query is at most 4 is NP-complete.

THEOREM 3.4. *Deciding whether the query-width of a conjunctive query is at most 4 is* NP*-complete.*

Proof. 1, "*Membership.*" Let Q be a query. It is easy to see that if there exists a query decomposition of width bounded by 4, then there also exists one of polynomial size (in fact, by a simple restructuring technique we can always remove identically labeled vertices from a decomposition tree, and thus for any conjunctive query Q only $O(|atoms(Q) \cup var(Q)|^4)$ need to be considered). Therefore, a query decomposition of width at most k can be found by a nondeterministic guess followed by a polynomial correctness check. The problem is thus in NP.

2, "*Hardness.*" We transform the well-known NP-complete problem EXACT COVER BY 3-SETS (XC3S) [16] to the problem of deciding whether, for a conjunctive query Q, $qw(Q) \leqslant 4$ holds. An instance of EXACT COVER BY 3-SETS consists of a pair $I = (R, \Delta)$ where R is a set of $r = 3s$ elements, and Δ is a collection of m 3-element subsets of R. The question is whether we can select s subsets out of Δ such that they form a partition of R.

Consider an instance $I = (R, \Delta)$ of XC3S. Let $\Delta = \{D_i | 1 \leqslant i \leqslant m\}$ and let $D_i = \{X^i_a, X^i_b, X^i_c\}$ for $1 \leqslant i \leqslant m$ (note that for $i \neq j$, some X^i_α and X^j_β may coincide).

For illustration, throughout this proof, we will use as running example the instance $I_e = (R_e, \Delta_e)$, where the set R_e to be partitioned is $\{X_1, X_2, X_3, X_4, X_5, X_6\}$, and Δ_e contains the following subsets: $D_1 = \{X_1, X_3, X_4\}$, $D_2 = \{X_1, X_2, X_4\}$, $D_3 = \{X_3, X_4, X_6\}$,and $D_4 = \{X_3, X_5, X_6\}$.

Generate a strict $(m+1, 2)$ 3PS $\Sigma = \{\sigma_0, \sigma_1, \ldots, \sigma_m\}$ on some base set $S = base(\Sigma)$. By Lemma 7.3, this can be done in $O((m+1)^2 + 2(m+1)) = O(m^2)$, and hence in polynomial time in the size of I. Let $\sigma_i = \{S^i_a, S^i_b, S^i_c\}$ for $0 \leqslant i \leqslant m$.

Identify each element of S with a separate variable and establish a fixed precedence order $\prec$ among the elements (variables) of S. If S' is a subset of S, and $S' = \{G_1, \ldots, G_l\}$, where $G_1 \prec G_2 \cdots \prec G_l$, then we will abbreviate the list of variables $G_1, \ldots, G_l$ by S' in query atoms. For example, instead of writing $p(a, G_1, \ldots, G_l, b)$, we write $p(a, S', b)$.

In order to transform the given instance $I = (R, \Delta)$ of XC3S to a conjunctive query Q, let us first define the following sets of variables Γ^ℓ and Π^ℓ_i, which are all taken to be disjoint from the variables in S.

For $0 \leqslant \ell \leqslant s$, let

$$\Gamma^\ell = \{V^\ell_{ij} \mid 1 \leqslant i < j \leqslant 8\},$$

and for $(1 \leqslant i \leqslant 8)$, let

$$\Pi^\ell_i = \{V^\ell_{1i}, V^\ell_{2i}, \ldots, V^\ell_{i-1\,i}, V^\ell_{i\,i+1}, \ldots, V^\ell_{i8}\}.$$

Let S'_a and S''_a be two nonempty sets which partition S^0_a. (Such a partition exists because S^0_a contains at least two elements.)

Define, for $0 \leqslant \ell \leqslant s$ the following sets of query atoms:

$$BLOCKA^\ell = \{q(\Pi^\ell_1, S'_a, Z_\ell), p_a(\Pi^\ell_2, S''_a), p_b(\Pi^\ell_3, S^0_b), p_c(\Pi^\ell_4, S^0_c)\}$$
$$BLOCKB^\ell = \{q(\Pi^\ell_5, S'_a, Y_\ell), p_a(\Pi^\ell_6, S''_a), p_b(\Pi^\ell_7, S^0_b), p_c(\Pi^\ell_8, S^0_c)\},$$

where the Y_ℓ and Z_ℓ variables are distinct fresh variables not occurring in any previously defined set. We further define

$$BLOCKSA = \bigcup_{0 \leq \ell \leq s} BLOCKA^\ell, \qquad BLOCKSB = \bigcup_{0 \leq \ell \leq s} BLOCKB^\ell,$$

$$\text{and } BLOCKS = BLOCKSA \cup BLOCKSB.$$

Define, for $1 \leqslant \ell \leqslant s$:

$$LINK_\ell = \{link(Y_{\ell-1}, Z_\ell)\} \text{ and } LINKS = \bigcup_{1 \leqslant \ell \leqslant s} LINK_\ell.$$

Finally, define for each set $D_i = \{X^i_a, X^i_b, X^i_c\}$ of Δ, $1 \leqslant i \leqslant m$, the set of atoms:

$$\Omega[D_i] = \{s(X^i_a, S^i_a), s(X^i_b, S^i_b), s(X^i_c, S^i_c)\}.$$

Let $\Omega = \bigcup_{1 \leqslant i \leqslant m} \Omega[D_i]$, and denote by $\Omega(D_i)$ the set of all atoms of Ω in which some variable of D_i occurs, i.e,

$$\Omega(D_i) = \{s(X, \alpha) \in \Omega \mid X \in D_i\}.$$

In our running example, we have the following four sets of atoms corresponding to the elements of Δ_e:

$$\Omega[D_1] = \{s(X_1, S^1_a), s(X_3, S^1_b), s(X_4, S^1_c)\},$$
$$\Omega[D_2] = \{s(X_1, S^2_a), s(X_2, S^2_b), s(X_4, S^2_c)\},$$
$$\Omega[D_3] = \{s(X_3, S^3_a), s(X_4, S^3_b), s(X_6, S^3_c)\},$$
$$\Omega[D_4] = \{s(X_3, S^4_a), s(X_5, S^4_b), s(X_6, S^4_c)\}.$$

Note that, according to the construction above, for each $1 \leqslant i \leqslant 4$, $\sigma_i = \{S^i_a, S^i_b, S^i_c\}$ is a member of a strict $(4+1, 2)$ 3PS Σ_e on some base set S_e. Each element in σ_i is thus a set of variables, and σ_i is a partition of the base set S_e.

Let Q be the query whose atom-set is $BLOCKS \cup LINKS \cup \Omega$.

We claim that Q has query-width 4 iff $I = (R, \Delta)$ is a positive instance of `EXACT COVER BY 3SETS`.

We call Q_e the query corresponding to instance I_e of our running example. Note that I_e is a positive instance of `EXACT COVER BY 3SETS`. Indeed, the sets D_2 and D_4 form a partition of R_e.

Let us first prove the *if* part. Assume that there exist s 3-sets $D^1, \ldots, D^s \in \Delta$ which exactly cover R, i.e., which form a partition of R. We describe a query-decomposition (T, λ) of Q.

The root v_{a0} of T is labeled by the set of atoms $BLOCKA^0$. The root has as unique child a vertex v_{b0} labeled by $BLOCKB^0$.

The decomposition tree is continued as follows. For each $1 \leqslant \ell \leqslant s$, do the following.

• Create a vertex $v_{c\ell}$ labeled by $LINK_\ell \cup \Omega[\mathrm{D}^\ell]$, and attach $v_{c\ell}$ as a child to $v_{b\ell-1}$.

• For each remaining atom $A \in \Omega(D^\ell) - \Omega[D^\ell]$, we create a new vertex, label it with $\{A\}$, and attach it as a leaf to $v_{c\ell}$. (Note that these remaining atoms, if any, stem from other elements of Δ, given that a variable may occur in several 3-sets.)

• Then, create a vertex $v_{a\ell}$ of T, label it by the set of atoms $BLOCKA^\ell$, and attach it as a child of $v_{c\ell}$. The vertex $v_{a\ell}$, in turn, has as only child a vertex $v_{b\ell}$ labeled by $BLOCKB^\ell$.

It is not hard to check that (T, λ) is indeed a valid query decomposition.

Since our example instance admits a solution, Q_e has a query decomposition of width 4. Figure 11 shows such a decomposition for Q_e. Note that the choice of sets D_2 and D_4 from Δ corresponds to the choice of two vertices of the query decomposition containing $\Omega[D_2]$ and $\Omega[D_4]$, respectively.

Let us now prove the *only-if* part. Assume (T, λ) is a width 4 query decomposition of the above defined query Q. By Proposition 3.3, we also assume, without loss of generality, that (T, λ) is a pure query decomposition. Since Q is connected, also T is connected.

We observe a number of relevant facts and make some assumptions.

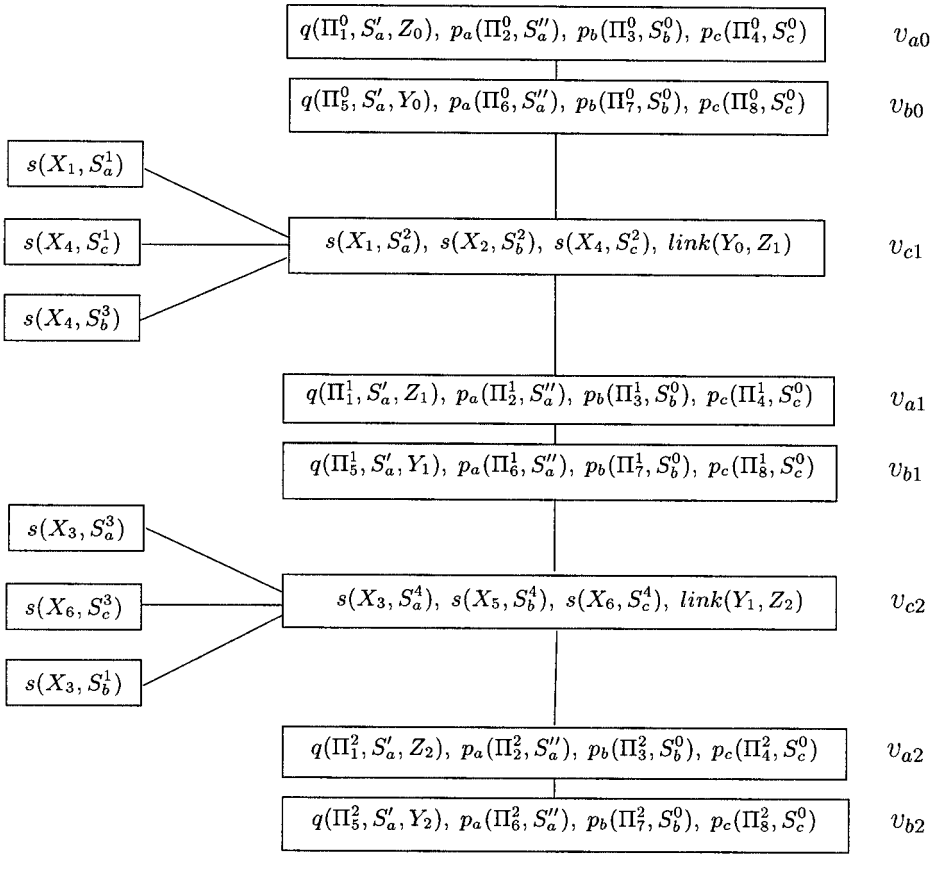

FIG. 11. A 4-width query decomposition of query Q_e.

Fact 1. By Lemma 7.1 for each $0 \leqslant \ell \leqslant s$, there must exist adjacent vertices $v_{a\ell}$ and $v_{b\ell}$ such that $\lambda(v_{a\ell}) \cup \lambda(v_{b\ell}) = BLOCKA^{\ell} \cup BLOCKB^{\ell}$.

Fact 2. It holds that $S \subseteq var(v_{a\ell})$ and $S \subseteq var(v_{b\ell})$. In fact, if this were not the case, then both vertices would miss variables from S, but since all variables of S occur together in other pairs of adjacent vertices, this would violate the *connectedness condition* and is thus impossible.

Fact 3. From the latter, and from the fact that the sets S'_a, S''_a, S^0_b, and S^0_c form a partition of S, it follows that each of the vertices $v_{a\ell}$ and $v_{b\ell}$ contains a q atom, a p_a atom, a p_b atom, and a p_c atom. Without loss of generality, we can thus make the following assumption.

Assumption. For $0 \leqslant \ell \leqslant s$ we have $Z_\ell \in var(v_{a\ell})$ and $Y_\ell \in var(v_{b\ell})$.

Fact 4. For $1 \leqslant \ell \leqslant s$, there exists a vertex $v_{c\ell}$ that lies on the unique path from $v_{b\ell-1}$ to $v_{a\ell}$ such that $\{Y_{\ell-1}, Z_\ell\} \subseteq var(v_{c\ell})$. This can be seen as follows. For any variable ϑ, a *ϑ-path* is a path π in T such that the variable ϑ occurs in the label $\lambda(v)$ of any vertex v of π. The atom $link(Y_{\ell-1}, Z_\ell)$ must belong to the set $\lambda(v'_{c\ell})$ of some vertex $v'_{c\ell}$ of T. Clearly, by the *connectedness* condition, $v'_{c\ell}$ is connected via a $Y_{\ell-1}$-path π_b to $v_{b\ell-1}$ and by a Z_ℓ-path π_a to $v_{a\ell}$. Let π denote the unique path from $v_{b\ell-1}$ to $v_{a\ell}$. Then π, π_a, and π_b intersect at exactly one vertex. This is the desired vertex $v_{c\ell}$.

Fact 5. For $1 \leqslant \ell \leqslant s$, $S \subseteq var(v_{c\ell})$. Trivial, because $v_{c\ell}$ lies on a path from $v_{b\ell-1}$ to $v_{a\ell}$ and $S \subseteq var(v_{b\ell-1})$ and $S \subseteq var(v_{a\ell})$. The fact follows by the *connectedness* condition.

Fact 6. For $1 \leqslant \ell \leqslant s$, $link(Y_{\ell-1}, Z_\ell)$ belongs to $\lambda(v_{c\ell})$ and there exists an i with $1 \leqslant i \leqslant m$ such that $\Omega[D_i] \subseteq \lambda(v_{c\ell})$; in summary, $\lambda(v_{c\ell}) = \{link(Y_{\ell-1}, Z_\ell)\} \cup \Omega[D_i]$. Let us prove this. By FACT 5 we know that all variables in S must be covered by $v_{c\ell}$. However, it also holds that $\{Y_{\ell-1}, Z_\ell\} \subseteq var(v_{c\ell})$ (see FACT 4). To cover the latter variables, there are two alternative choices:

1. both atoms $q(\Pi^{\ell-1}_5, S'_a, Y_{\ell-1})$ and $q(\Pi^{\ell}_1, S'_a, Z_\ell)$ belong to $\lambda(v_{c\ell})$; or
2. the atom $link(Y_{\ell-1}, Z_\ell)$ belongs to $\lambda(v_{c\ell})$.

Choice 1 is impossible: there exist no two other atoms $A, B \in atoms(Q)$ such that $var(A) \cup var(B) \cup S'_a = S$. We are thus left with Choice 2. Since the atom $link(Y_{\ell-1}, Z_\ell)$ does not contain any variable from S, there must be three other atoms in $\lambda(v_{c\ell})$ that together cover S. An inspection of the available atoms shows that the only possibility of covering S by three atoms is via some atom set $\Omega[D_i]$ for $1 \leqslant i \leqslant m$. The fact is proved.

Fact 7. For $1 \leqslant i < j \leqslant s$ it holds that v_{ai} lies on the unique path in T from v_{ci} to v_{cj}.

Consider the edge $\{v_{ai}, v_{bi}\}$. If we cut this edge from the tree T, then we obtain two disconnected trees T_a (containing v_{ai}) and T_b (containing v_{bi}). Since v_{ci} is connected via a Z_i-path to v_{ai}, but Z_i does not occur in $var(v_{bi})$, it holds that v_{ci} is contained in T_a. On the other hand, by "iterative" application of Fact 4 and of the *connectedness condition* it follows that there is a path π from v_{bi} to v_{cj} such that for

each vertex v of π it holds that $var(v) \cap Bigvars \neq \emptyset$, where $Bigvars = \{Y_h \mid i \leqslant h < j\} \cup \{Z_h \mid i < h < j\}$. Since $var(v_{ai}) \cap Bigvars = \emptyset$, π does not traverse v_{ai}. It follows that v_{ci} belongs to T_b. Therefore, the unique path linking v_{ci} to v_{cj} goes through the edge $\{v_{ai}, v_{bi}\}$, and thus contains the vertex v_{ai}.

Fact 8. For $0 \leqslant i < j \leqslant s$ it holds that $var(v_{ci}) \cap var(v_{cj}) = S$. By Fact 7, v_{ai} lies on the unique path from v_{ci} to v_{cj}. Therefore by the *connectedness condition* it holds that $var(v_{ci}) \cap var(v_{cj}) \subseteq var(v_{ai})$. Moreover, by Fact 6, no variable from $var(v_{ai}) - S$ is contained in both $var(v_{ci})$ and $var(v_{cj})$. Thus $var(v_{ci}) \cap var(v_{cj}) \subseteq S$. On the other hand, by Fact 5, $S \subseteq var(v_{ci})$ and $S \subseteq var(v_{cj})$, hence, $S \subseteq var(v_{ci}) \cap var(v_{cj})$. In summary, we obtain $var(v_{ci}) \cap var(v_{cj}) = S$.

For each $1 \leqslant \ell \leqslant s$, denote by D^ℓ the set D_i such that $\Omega[D_i] \subseteq \lambda(v_{c\ell})$ (see Fact 6). By FACT 8 it follows that the sets D^ℓ $(1 \leqslant \ell \leqslant s)$ are mutually disjoint. But then the union of these sets is of cardinality $3s = r$, and hence the union must coincide with R. Thus s subsets out of Δ cover R and (R, Δ) is a positive instance of EXACT COVER BY 3-SETS. ▌

APPENDIX A
HYPERTREE DECOMPOSITIONS OF HYPERGRAPHS

In Section 4 of this paper we consider hypertree decompositions of *conjunctive queries*. It is equally possible to define the concepts of hypertree decomposition and hypertree width in the slightly more abstract setting of *hypergraphs*. This was done, e.g., in [21]. Given that each Boolean conjunctive query has a corresponding hypergraph, the two settings are closely related. In particular, all definitions, results, and algorithms introduced in this paper in the context of conjunctive queries carry over to the more general context of hypergraphs. This is made explicit in this appendix.

Let $\mathscr{H} = (V, H)$ be a hypergraph. We will often use the term *variable* and the term *edge* to refer to the elements in V and H, respectively. Accordingly, we will denote the set V and H of $\mathscr{H}$ by $var(\mathscr{H})$ and $edges(\mathscr{H})$, respectively.

A *hypertree for a hypergraph* $\mathscr{H}$ is a triple $\langle T, \chi, \lambda \rangle$, where $T = (N, E)$ is a rooted tree, and χ and λ are labeling functions which associate to each vertex $p \in N$ two sets $\chi(p) \subseteq var(\mathscr{H})$ and $\lambda(p) \subseteq edges(\mathscr{H})$. The notation and definitions of Section 4 for hypertrees of conjunctive queries will be used also for hypertrees of hypergraphs.

DEFINITION A.1. A *hypertree decomposition* of a hypergraph $\mathscr{H}$ is a hypertree $HD = \langle T, \chi, \lambda \rangle$ for $\mathscr{H}$ which satisfies all the following conditions:

1. for each edge $h \in edges(\mathscr{H})$, there exists $p \in vertices(T)$ such that $h \subseteq \chi(p)$;
2. for each variable $Y \in var(\mathscr{H})$, the set $\{p \in vertices(T) \mid Y \in \chi(p)\}$ induces a (connected) subtree of T;
3. for each $p \in vertices(T)$, $\chi(p) \subseteq var(\lambda(p))$;
4. for each $p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

The *width* of a hypertree decomposition $\langle T, \chi, \lambda \rangle$ is $max_{p \in vertices(T)} |\lambda(p)|$. The *hypertree-width* $hw(\mathscr{H})$ of $\mathscr{H}$ is the minimum width over all its hypertree decompositions.

All other related notions, such as, e.g., *complete decomposition*, *normal form*, etc., directly carry over to hypertree decompositions of hypergraphs, and need not to be redefined here.

DEFINITION A.2. Let $\mathscr{H} = (V, H)$ be a hypergraph. Let $[v_1, \ldots, v_m]$ and $[h_1, \ldots, h_n]$ be lexicographic orderings of V and H, respectively. Moreover, let $DS = \{\mathrm{R}_1, \ldots, \mathrm{R}_n\}$ be a database schema such that, for each $1 \leqslant i \leqslant n$, the arity of relation R_i is equal to the cardinality of the edge h_i.

The *canonical query* $cq(\mathscr{H})$ of $\mathscr{H}$ is the following Boolean conjunctive query Q over DS.

$$Q\text{: } ans \leftarrow r_1(\mathbf{u_1}) \wedge \cdots \wedge r_n(\mathbf{u_n}),$$

where $r_1, \ldots, r_n$ are the relation names of $R_1, \ldots, R_n$ and, for each $1 \leqslant i \leqslant m$, $\mathbf{u_i}$ is the lexicographically-ordered list of the variables in h_i.

Since the set of edges of a hypergraph $\mathscr{H}$ is isomorphic to the set of atoms of its canonical query $cq(\mathscr{H})$, and $var(\mathscr{H}) = var(cq(\mathscr{H}))$, any hypertree of a hypergraph $\mathscr{H}$ can be seen as a hypertree of $cq(\mathscr{H})$ and vice versa. Under this view, the following statement follows immediately from Definition 4.1, Definition 1.1, and Definition 1.2.

THEOREM A.3. *Given a hypergraph $\mathscr{H}$, every hypertree decomposition of $\mathscr{H}$ is a hypertree decomposition of its canonical query and vice versa.*

COROLLARY A.4. *The hypertree-width of any hypergraph $\mathscr{H}$ is equal to the hypertree-width of its canonical query $cq(\mathscr{H})$.*

Since the canonical query of a hypergraph is evidently logspace computable, from Theorem 5.16 and the above result, we obtain that bounded hypertree-width hypergraphs are efficiently recognizable.

COROLLARY A.5. *Deciding whether a hypergraph $\mathscr{H}$ has k-bounded hypertree-width is in LOGCFL.*

From Theorem 1.3 and Theorem 5.18, we obtain that bounded-width hypertree decompositions of hypergraphs are efficiently computable.

COROLLARY A.6. *Computing a k-bounded hypertree decomposition (if any) of a hypergraph $\mathscr{H}$ is in $\mathrm{L}^{\mathrm{LOGCFL}}$, i.e., in functional LOGCFL.*

It is worthwhile noting that the hypertree-width of a query Q coincides with the hypertree-width of the hypergraph $H(Q)$ associated with Q (see Section 2.1).

THEOREM A.7. *The hypertree-width of a query Q is equal to the hypertree-width of the query hypergraph $H(Q)$ of Q.*

Proof. It suffices to show that, for each each hypertree decomposition of Q having width k, there is a hypertree decomposition of $H(Q)$ having width $\leqslant k$, and vice-versa.

Let $D = \langle T, \chi, \lambda \rangle$ be a hypertree decomposition of Q. For each $v \in vertices(T)$, let $\lambda'(v) := \{var(A) \mid A \in \lambda(V)\}$. Let $D' := \langle T, \chi, \lambda' \rangle$. Clearly, D' is a hypertree decomposition of $H(Q)$. Moreover, for each $v \in vertices(T)$, $|\lambda'(v)| \leqslant |\lambda(v)|$, hence the width of D' is not greater than the width of D.

For the other direction, let $D = \langle T, \chi, \lambda \rangle$ be a hypertree decomposition of $H(Q)$. Note that for an edge $e \in edges(H(Q))$ there may exist several atoms A in $atoms(Q)$ such that $var(A) = e$. Therefore, for each edge $e \in edges(H(Q))$ choose one particular atom A_e in $atoms(Q)$ such that $var(A_e) = e$. For each $v \in vertices(T)$, let $\lambda'(v) := \{A_e \mid e \in \lambda(V)\}$. Let $D' := \langle T, \chi, \lambda' \rangle$. Clearly, D' is a hypertree decomposition of Q. Moreover, D and D' are of the same width. (Note that D' is not necessarily a complete hypertree decomposition of Q, given that some atoms may not appear in any λ'-label; however, by Lemma 4.4, D' can be transformed into a complete hypertree decomposition of Q having the same width.) ∎

APPENDIX B
A DATALOG PROGRAM RECOGNIZING QUERIES OF k-BOUNDED HYPERTREE-WIDTH

In this short Appendix, we show a straightforward polynomial-time implementation of the LOGCFL algorithm $k-$`decomp`. In particular, we reduce (in polynomial time) the problem of deciding whether there exists a k-bounded hypertree-width decomposition of a given conjunctive query Q to the problem of evaluating a weakly stratified Datalog program [32, 40].

First, we associate an identifier (e.g., some constant number) to each k-vertex (non empty subset of Q consisting of k atoms at most) R of Q, and to each $[R]$-component C for any k-vertex R of Q. Moreover, we have a new identifier *root* which intuitively will be the root of any possible hypertree-decomposition, and a new identifier *varQ* which encodes the set of all the variables of the query and hence is seen as a component including any subset of $var(Q)$.

Then, we compute the following relations:[5]

- *k-vertex*$(\cdot)$: Contains a tuple $\langle R \rangle$ for each *k-vertex* R of Q.
- *component*$(\cdot, \cdot)$: Contains a tuple $\langle C_R, R \rangle$ for each $[R]$-component C_R of some k-vertex R.

Moreover, it contains the tuple $\langle varQ, root \rangle$.

- *meets-condition*$(\cdot, \cdot, \cdot)$: Contains any tuple $\langle S, R, C_R \rangle$ such that S and R are k-vertices, C_R is an $[R]$-component, and the following conditions hold: $var(S) \cap C_R \neq \varnothing$, and $\forall P \in atoms(C_R)\ var(P) \cap var(R) \subseteq var(S)$.

Moreover, it contains a tuple $\langle S, root, varQ \rangle$ for any k-vertex S.

- *subset*$(\cdot, \cdot)$: Contains any tuple $\langle C_S, C_R \rangle$ such that C_S is an $[S]$-*component* for some k-vertex S, C_R is an $[R]$-*component* for some k-vertex R, and $C_S \subset C_R$ holds.

[5] For the sake of clarity, we directly refer to objects by means of their associated identifiers (which we also use as logical terms in the program).

Let $\mathscr{P}$ be the following Datalog program:

1. k-decomposable$(R, C_R) \leftarrow k$-vertex(S), meets-conditions(S, R, C_R), $\neg$ undecomposable(S, C_R)
2. undecomposable$(S, C_R) \leftarrow$ component(C_S, S), subset(C_S, C_R), $\neg$ k-decomposable(S, C_S).

Note that each atom of the form *k-decomposable*(R, C_R) depends only on atoms *k-decomposable*(S, C_S) such that $C_S \subset C_R$, because of the base relation *subset*(C_S, C_R) in the body of the last rule of $\mathscr{P}$. Thus, program $\mathscr{P}$ is weakly stratified. As a consequence, $\mathscr{P}$ has a total well-founded model [42] M, and hence a unique stable model [17] which coincides with M. Recall that the well-founded model of any Datalog program can be computed in polynomial time [42].

It is easy to see that $hw(Q) \leqslant k$ if and only if model M contains the atom *k-decomposable*$(root, varQ)$.

In fact, if *k-decomposable*$(root, varQ)$ belongs to M, the atoms *k-decomposable* $(\cdot, \cdot)$ belonging to M, together with the base relations, encode all the NF hypertree decompositions of Q in normal form having width at most k. Thus, we can also compute from M a hypertree decomposition of Q, by using a simple top-down procedure in order to select one of these decompositions. For instance, we can choose as the root of such a hypertree decomposition any k-vertex S such that *meets-conditions*$(S, root, varQ) \in M$ and *undecomposable*$(S, varQ) \notin M$. Then, we can continue by choosing in a similar way one child of S for each $[S]$-*component* $C_S \subseteq var(Q)$, and so on. An extension of the above algorithm which *computes* a k-bounded hypertree decomposition is shown in [22]. See the hypertree decompositions' homepage [36] for downloading executables and examples.

ACKNOWLEDGMENTS

We thank Cristinel Mateis and Reinhard Pichler for their useful comments on earlier versions of this paper, and an anonymous referee who was very helpful with her (his) punctual comments and suggestions.

REFERENCES

1. S. Abiteboul, R. Hull, and V. Vianu, "Foundations of Databases," Addison–Wesley, Reading, MA, 1995.
2. S. Arnborg, Efficient algorithms for combinatorial problems on graphs with bounded decomposability—a survey, *BIT* **25** (1985), 2–23.
3. C. Beeri, R. Fagin, D. Maier, and M. Yannakakis, On the desiderability of acyclic database schemes, *J. Assoc. Comput. Mach.* **30** (1983), 479–513.
4. P. A. Bernstein and N. Goodman, The power of natural semijoins, *SIAM J. Comput.* **10** (1981), 751–771.
5. W. Bibel, Constraint satisfaction from a deductive viewpoint, *Artif. Intell.* **35** (1988), 401–413.
6. H. L. Bodlaender, A linear-time algorithm for finding tree-decompositions of small treewidth, *SIAM J. Comput.* **25** (1996), 1305–1317.
7. A. K. Chandra, D. C. Kozen, and L. J. Stockmeyer, Alternation, *J. Assoc. Comput. Mach.* **26** (1981), 114–133.

8. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational databases, *in* "ACM Symp. on Theory of Computing (STOC'77)," pp.77–90, 1977.
9. Ch. Chekuri and A. Rajaraman, Conjunctive query containment revisited, *in* "Proc. International Conference on Database Theory 1997 (ICDT'97), Delphi, Greece, 1997," Springer Lecture Notes on Computer Sciences, Vol. 1186, pp. 56–70, 1997. Full version: *Theoret. Comput. Sci.* **239** (2000), 211–229.
10. A. D'Atri and M. Moscarini, Recognition algorithms and design methodologies for acyclic database schemes, *Adv. Comput. Res.* **3** (1986), 43–68.
11. R. Dechter, Constraint networks, *in* "Encyclopedia of Artificial Intelligence," second ed., pp. 276–285, Wiley, New York, 1992.
12. R. Dechter and J. Pearl, Tree clustering for constraint networks, *Artif. Intell.* **38** (1989), 353–366.
13. R. Fagin, Degrees of acyclicity for hypergraphs and relational database schemes, *J. Assoc. Comput. Mach.* **30** (1983), 514–550.
14. R. Fagin, A. O. Mendelzon, and J. D. Ullman, A simplified universal relation assumption and its properties, *ACM Trans. Database Systems* **7** (1982), 343–360.
15. E. C. Freuder, A sufficient condition for backtrack-bounded search, *J. Assoc. Comput. Mach.* **32** (1985), 755–761.
16. M. R. Garey and D. S. Johnson, "Computers and Intractability. A Guide to the Theory of NP-Completeness, Freeman, New York, 1979.
17. M. Gelfond and V. Lifschitz, The stable model semantics for logic programming, *in* "Proc. of the Fifth Logic Programming Symposium," pp. 1070–1080, MIT Press, Cambridge, MA, 1988.
18. N. Goodman and O. Shmueli, Tree queries: A simple class of relational queries, *ACM Trans. Database Systems* **7** (1982), 653–677.
19. G. Gottlob, N. Leone, and F. Scarcello, The complexity of acyclic conjunctive queries, *J. Assoc. Comput. Mach.* **48** (2001), 431–498. Currently available at http//www.dbai.tuwein.ac.at/staff/gottlob/acyclic.ps. An extended abstract concerning part of this work has been published in "Proc. of the 39th IEEE Symposium on Foundations of Computer Science (FOCS'98)," pp. 706–715, Palo Alto, CA, 1998.
20. G. Gottlob, N. Leone, and F. Scarcello, Computing LOGCFL certificates, *Theoret. Comput. Sci.* **270** (2002), 761–777. A preliminary version appeared in "Proc. of the 26th International Colloquium on Automata, Languages and Programming (ICALP'99)," Lecture Notes on Computer Science, Vol. 1644, pp. 361–371, Springer, Prague, 1999.
21. G. Gottlob, N. Leone, and F. Scarcello, A comparison of structural CSP decomposition methods, *Artif. Intell.* **124** (2000), 243–282. A preliminary version appeared in "Proc. of the 16th Int. Joint Conference on Artificial Intelligence (IJCAI'99)," Vol. 1, pp. 394–399, 1999.
22. G. Gottlob, N. Leone, and F. Scarcello, On tractable queries and constraints, *in* "Proc. of Database and Expert Systems Applications (DEXA'99)," Lecture Notes on Computer Science, Vol. 1677, pp. 1–15, Springer, Florence, 1999.
23. G. Gottlob, N. Leone, and F. Scarcello, Robbers, marshals, and guards: Game theoretic and logical characterizations of hypertree width, Technical Report DBAI-TR-2000–42, available at http://www.dbai.tuwein.ac.at/staff/gottlob/robber.ps. Full version to appear in *J. Comput. System Sci.*
24. S. H. Greibach, The hardest context-free language, *SIAM J. Comput.* **2** (1973), 304–310.
25. M. Gyssens, P. G. Jeavons, and D. A. Cohen, Decomposing constraint satisfaction problems using database techniques, *Artif. Intell.* **66** (1994), 57–89.
26. M. Gyssens and J. Paredaens, A decomposition methodology for cyclic databases, *in* "Advances in Database Theory," Vol. 2, pp. 85–122, Plenum Press, New York, 1984.
27. P. Jeavons, D. Cohen, and M. Gyssens, Closure properties of constraints, *J. Assoc. Comput. Mach.* **44** (1997), 527–548.
28. D. S. Johnson, A catalog of complexity classes, *in* "Handbook of Theoretical Computer Science" (J. van Leeuwen, Ed.), Vol. A, Chap. 2, pp. 67–161, North-Holland, Amsterdam, 1990.

29. Ph. G. Kolaitis and M. Y. Vardi, Conjunctive-query containment and constraint satisfaction, *J. Comput. System Sci.* **61** (2000), 302–332. A preliminary version appeared in "Proc. of Symp. on Principles of Database Systems (PODS'98)," pp. 205–213, 1998.
30. D. Maier, "The Theory of Relational Databases," Computer Science Press, Rochville, MD, 1986.
31. C. H. Papadimitriou and M. Yannakakis, On the complexity of database queries, *in* "Proc. of Symp. on Principles of Database Systems (PODS'97)," Tucson, Arizona, pp. 12–19, 1997.
32. H. Przymusinska and T. Przymusinski, Weakly perfect model semantics for logic programs, *in* "Proc. Fifth Int. Conf. and Symp. on Logic Programming," pp. 1106–1120, 1988.
33. X. Qian. Query folding, *in* "Proc. of Int. Conf. on Data Engineering, New Orleans, Louisiana, Feb. 26–March 1, 1996 (ICDE'96)," pp. 48–55, 1996.
34. N. Robertson and P. D. Seymour, Graph minors II. Algorithmic aspects of tree-width, *J. Algorithms* **7** (1986), 309–322.
35. W. L. Ruzzo, Tree-size bounded alternation, *J. Comput. System Sci.* **21** (1980), 218–235.
36. F. Scarcello and A. Mazzitelli, The hypertree decompositions' homepage, http://www.info.deis.un.cal.it/~frank/Hypertrees/.
37. S. Skyum and L. G. Valiant, A complexity theory based on Boolean algebra, *J. Assoc. Comput. Mach.* **32** (1985), 484–502.
38. I. H. Sudborough, Time and tape bounded auxiliary pushdown automata, *in* "Mathematical Foundations of Computer Science (MFCS'77)," Lecture Notes on Computer Science, Vol. 53, pp. 493–503, Springer-Verlag, Berlin, 1977.
39. 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 J. Comput.* **13** (1984), 566–579.
40. J. D. Ullman, "Principles of Database and Knowledge Base Systems," Vol II, Computer Science Press, Rockville, MD, 1989.
41. J. D. Ullman, Information integration using logical views, *in* "Proc. of International Conference on Database Theory (ICDT'97)," Delphi, Greece, Jan. 1997, Lecture Notes on Computer Science, Vol. 1186, pp. 19–40, Springer-Verlag, Berlin, 1997.
42. A. Van Gelder, K. A. Ross, and J. S. Schlipf, The well-founded semantics for general logic programs, *J. Assoc. Comput. Mach.* **38** (1991), 620–650.
43. M. Vardi, Complexity of relational query languages, *in* "Proc. of the 14th ACM Symp. on Theory of Computing (STOC'82)," pp. 137–146, 1982.
44. M. Yannakakis, Algorithms for acyclic database schemes, *in* "Proc. of Int. Conf. on Very Large Data Bases (VLDB'81)," (C. Zaniolo and C. Delobel, Eds.), pp. 82–94, Cannes, France, 1981.
45. C. T. Yu and M. Z. Ozsoyoglu, On determining tree-query membership of a distributed query, *NFOR* **22** (1984), 261–282.