Information Systems 35 (2010) 278–298

Contents lists available at ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Tractable database design and datalog abduction through bounded treewidth ☆

Georg Gottlob [a], Reinhard Pichler [b], Fang Wei [c,*]

[a] Computing Laboratory, Oxford University, Oxford OX1 3QD, United Kingdom
[b] Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria
[c] Institut für Informatik, Albert-Ludwigs-Universität Freiburg, D-79110 Freiburg i. Br., Germany

ARTICLE INFO

Article history:
Received 10 October 2008
Received in revised form
8 September 2009
Accepted 30 September 2009
Recommended by: J. Van den Bussche

Keywords:
Normal forms
Database design
Tree decomposition
Bounded treewidth
Fixed-parameter tractability
Datalog abduction

ABSTRACT

Given that most elementary problems in database design are NP-hard, the currently used database design algorithms produce suboptimal results. For example, the current 3NF decomposition algorithms may continue further decomposing a relation even though it is already in 3NF. In this paper we study database design problems whose sets of functional dependencies have bounded treewidth. For such sets, we develop polynomial-time and highly parallelizable algorithms for a number of central database design problems such as:

- primality of an attribute;
- 3NF-test for a relational schema or subschema;
- BCNF-test for a subschema.

In order to define the treewidth of a relational schema, we shall associate a hypergraph with it. Note that there are two main possibilities of defining the treewidth of a hypergraph $\mathcal{H}$: One is via the primal graph of $\mathcal{H}$ and one is via the incidence graph of $\mathcal{H}$. Our algorithms apply to the case where the primal graph is considered. However, we also show that the tractability results still hold when the incidence graph is considered instead.

It turns out that our results have interesting applications to logic-based abduction. By the well-known relationship with the primality problem in database design and the relevance problem in propositional abduction, our new algorithms and tractability results can be easily carried over from the former field to the latter. Moreover, we show how these tractability results can be further extended from propositional abduction to abductive diagnosis based on non-ground datalog.

1. Introduction

One of the fundamental problems in relational database design is to test whether a schema satisfies the desired normal form. Unfortunately, most problems arising in this area are intractable. For instance, among the following six most important decision problems

- PRIMALITY (for a schema or for a subschema);
- 3NFTEST (for a schema or for a subschema);
- BCNFTEST (for a schema or for a subschema);

☆ This is an extended and enhanced version of results published in [31,32]. The work was supported by the Austrian Science Fund (FWF), project P20704-N18.

* Corresponding author. Tel.: +49 761 203 8125; fax: +49 761 203 8122.
E-mail addresses: georg.gottlob@comlab.ox.ac.uk (G. Gottlob), pichler@dbai.tuwien.ac.at (R. Pichler), fwei@informatik.uni-freiburg.de (F. Wei).

0306-4379/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.is.2009.09.003

only BCNFTEST for a schema is tractable [43]. Despite the intractability of the remaining problems listed above, there exist efficient algorithms for decomposing a relational schema into subschemas satisfying 3NF or BCNF (see e.g. [43,5,46]). However, without the ability of normal form checking, these decomposition algorithms may continue further decomposing a schema or subschema even though the desired normal form has already been achieved. The following example displays a simple relational schema, where the well-known 3NF decomposition algorithm *synthesis* from [5] keeps decomposing the schema even though it is already in 3NF.

Example 1.1. Consider the schema $ABCD$ with functional dependencies (FDs) $AB \rightarrow C$, $C \rightarrow D$, $D \rightarrow A$. This schema is already in 3NF since all its attributes are prime, i.e., they are part of a key. However, by using the synthesis algorithm from [5], we obtain a decomposition into the subschemas ABC, CD, DA.

The concept of treewidth and related notions have been successfully applied to many areas of computer science. Recently, several intractable problems in the database field and in AI (such as e.g., conjunctive query equivalence and CSP problems) have been shown to become solvable in polynomial time or even highly parallelizable if the underlying graph or hypergraph structure has bounded treewidth or hypertree-width (see [29]). Hence, the question naturally arises as to whether one can also identify such tractable fragments for the aforementioned decision problems in relational database design. In this work, we define the treewidth of a relational schema (R, F) (where F denotes the set of functional dependencies holding in R) by considering the following hypergraph $\mathcal{H}$: the attributes of R are the vertices of $\mathcal{H}$ and every hyperedge of $\mathcal{H}$ corresponds to the set of attributes occurring in a functional dependency $f \in F$. In order to define the treewidth of $\mathcal{H}$, one can either consider the *primal graph* $P(\mathcal{H})$ or the *incidence graph* $I(\mathcal{H})$. In this paper, we shall mainly deal with the primal graph (a formal definition of these concepts will be given in Section 2.2).

Example 1.2. Consider the relational schema from Example 1.1. The corresponding hypergraph $\mathcal{H}$ consists of the vertices $\{A, B, C, D\}$ and the hyperedges ABC, CD, AD. The hypergraph $\mathcal{H}$ plus the primal graph $P(\mathcal{H})$ and the incidence graph $I(\mathcal{H})$ are shown in Fig. 1.

An interesting approach to dealing with such intractable problems is parameterized complexity. In fact, hard problems can become tractable if some problem parameter is bounded by a fixed constant. Such problems are also called fixed-parameter tractable. One important parameter which has served to establish fixed-parameter tractability results in many areas of computer science is the *treewidth* [8], which measures the "tree-likeness" of a graph.

One possibility to prove the fixed-parameter tractability of the above-mentioned decision problems is to show that these problems can be expressed in terms of monadic second-order (MSO) formulae over the primal graph or over the incidence graph, respectively. The fixed-parameter tractability is thus an immediate consequence of Courcelle's Theorem [16] (see Section 5). A concrete algorithm can then be obtained by constructing a finite tree automaton (FTA) corresponding to the MSO formula and by checking whether a tree obtained from the tree decomposition is accepted by the FTA. However, algorithms resulting from such an MSO-to-FTA transformation often have two serious shortcomings: First, the intuition of the algorithm is usually lost when looking at the FTA. Second (and even more importantly), the FTA resulting from standard translation methods (see e.g. [21]) often suffers from a state explosion and tends to be excessively complicated. It was observed in [22,44] that this state explosion already occurs for comparatively simple MSO formulae. For these reasons, it is always preferable to have a dedicated algorithm rather than just an MSO-encoding—as was pointed out in [34].

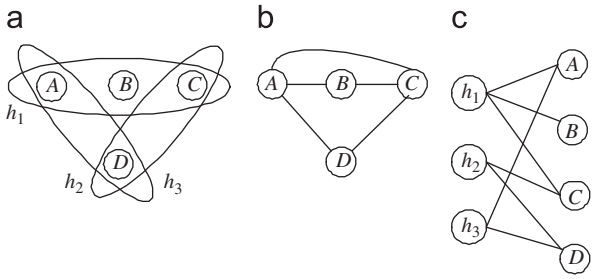

Fig. 1. (Hyper-)graphs from Example 1.2. (a) Hypergraph H; (b) primal graph $P(H)$; (c) incidence graph $I(H)$.

The main contributions of this paper are the following:

- For all of the above-mentioned database design problems (i.e., PRIMALITY, 3NFTEST, and BCNFTEST—both for a schema and for a subschema), we manage to identify tractable fragments via bounded treewidth. In case of bounded treewidth of the primal graph, we develop new, dedicated algorithms, from which the fixed-parameter tractability follows immediately. Actually, even if no tree decomposition is given, all of these problems are not only tractable but also highly parallelizable since they are in the class LogCFL for either notion of bounded treewidth.
- For the case of bounded treewidth of the incidence graph, we establish the fixed-parameter tractability by providing an MSO-encoding of these problems.
- Abduction is a very important technique for "reverse inference" in artificial intelligence. It is heavily used for diagnosis, e.g., in medical and technological applications [17,23]. Unfortunately, as shown in [19], the major decision and computation problems of abduction are NP-hard. In fact, the so-called relevance problem (i.e., deciding if a hypothesis is part of a possible explanation) is NP hard, even if the system description consists of propositional, definite Horn clauses only [23]. It turns out that our new methods developed in the area of database design can be naturally carried over to propositional abduction.

- Moreover, abduction is one of the AI processes where the datalog language has been fruitfully used for knowledge representation [2,39,12], and where database theory has a clear impact. In this paper, we present a significant extension of the above-mentioned tractability results. More precisely, we shall identify sufficient conditions under which abduction based on (non-ground!) datalog also becomes tractable.

The rest of the paper is organized as follows: In Section 2, we recall some basic terminology and results. New algorithms for the above-mentioned six decision problems in database design are presented in Sections 3 and 4. In Section 5, we prove the corresponding fixed-parameter tractability results also for the case of bounded treewidth of the incidence graph via appropriate MSO-encodings. The application of these results to normal form decompositions and to abductive diagnosis is dealt with in Sections 6 and 7, respectively. We conclude with Section 8.

2. Preliminaries

2.1. Database design

A relational schema is denoted as (R,F) where R is the set of attributes, and F the set of functional dependencies (FDs, for short) over R. If X and Y are sets of attributes and A is an attribute, then we may write XY, XA, $X-A$ to abbreviate the set notation $X\cup Y$, $X\cup\{A\}$, and $X\backslash\{A\}$, respectively.

Unless explicitly stated otherwise, we only consider FDs in *canonical form* here, i.e., FDs where there is only a single attribute on the right-hand side. The set of all FDs over the attributes in R that can be derived from F via *Armstrong's Axioms* (see [1]) is denoted as F^+. We write $F\models X\to A$ in order to denote that an FD $X\to A$ is in F^+. Below, we shall give a characterization of $F\models X\to A$ in terms of derivation sequences.

Given a relational schema (R,F) and a subset $X\subseteq R$, we write $clos_F(X)$ for the *closure* of X with respect to F, i.e., $A\in clos_F(X)$, iff $F\models X\to A$. If $F\models X\to A$, then we also say that "X determines (or decides) A". Note that we do not exclude the special case of FDs $X\to A$ with $X=\emptyset$. This simply means that the attribute A may only have a constant value in the relational schema (R,F).

Definition 2.1. If X determines all attributes $A\in R$, then X is called a *superkey*. If X is minimal with this property, then X is a *key*. The set of all keys in (R,F) is denoted as $K(R,F)$. An attribute A is called *prime* in (R,F), if it is contained in at least one key in $K(R,F)$.

In this paper, we often have to refer to the attributes occurring in an FD f. Let f be of the form $f=X\to A$. Then we write $lhs(f)$ and $rhs(f)$ to refer to X and A, respectively.

A "derivation sequence" of A from X in F is a sequence of the form $X\to X\cup\{A_1\}\to X\cup\{A_1,A_2\}\to\cdots\to X\cup\{A_1,\ldots,A_n\}$, s.t. $A_n=A$ and for every $i\in\{1,\ldots,n\}$, there exists an FD $f_i\in F$ with $lhs(f)\subseteq X\cup\{A_1,\ldots,A_{i-1}\}$ and $rhs(f)=A_i$. Of course, such a derivation sequence exists, iff $A\in clos_F(X)$. Moreover, $F\models X\to A$ holds iff $A\in clos_F(X)$.

Definition 2.2. A relational schema (R,F) is in BCNF, if for every FD $f=X\to A$ with $F\models f$ and $A\notin X$, the attribute set X is a superkey. A relational schema (R,F) is in 3NF, if for every FD $f=X\to A\in F$ with $F\models f$ and $A\notin X$, either X is a superkey, or A is prime.

Testing whether a relational schema (R,F) is in BCNF is an easy task. In fact, we just have to check that for all FDs $f\in F$, the left-hand side $lhs(f)$ is a superkey (see [48]). This is clearly feasible in polynomial time. In contrast, testing whether a relational schema is in 3NF is NP-complete, because the primality test is NP-complete [42].

Definition 2.3. Let (R,F) be a relational schema and $X\subseteq R$. The *schema projection* of F over X is defined as

$$F[X]=\{Y\to Z\mid F\models Y\to Z \text{ and } YZ\subseteq X\}$$

Let $R'\subseteq R$. Then $(R',F[R'])$ is referred to as a *subschema* of (R,F). Note that the size of $F[R']$ can be exponentially bigger than (R,F). The "classical" example for this phenomenon is as follows, see [20,43]:

$$R=\{A_1,\ldots,A_n,B_1,\ldots,B_n,C_1,\ldots,C_n,D\}$$
$$F=\{A_i\to C_i,B_i\to C_i\mid 1\le i\le n\}\cup\{C_1,\ldots,C_n\to D\}$$
$$R'=\{A_1,\ldots,A_n,B_1,\ldots,B_n,D\}$$

It is easy to verify that in $(R',F[R'])$ all 2^n dependencies of the form $Z_1,\ldots,Z_n\to D$ with $Z_i\in\{A_i,B_i\}$ hold. Moreover, $F[R']$ admits no representation of polynomial size.

For the complexity of an algorithm that deals with subschemas, it is therefore important to distinguish whether $F[R']$ is explicitly given or whether it is only implicitly given by (R,F) and R'. In all our algorithms in this paper, we assume that only (R,F) and R' are given and that $F[R']$ is thus specified *implicitly*. Our algorithms will of course avoid to compute an explicit representation $F[R']$ since, by the above considerations, such an explicit representation may have exponential size w.r.t. the size of R, F, and R'.

Let $R'\subseteq R$ and $(R',F[R'])$ be a subschema of (R,F). Then Definitions 2.1 and 2.2 straightforwardly carry over from schemas to subschemas by taking $(R',F[R'])$ rather than (R,F). For instance, an attribute A is called *prime* in $(R',F[R'])$, if it is contained in at least one key in $K(R',F[R'])$. Moreover, R' is in BCNF, if for every FD $f=X\to A\in F[R']$, X is a superkey of R'. Likewise, R' is in 3NF, if for every FD $f=X\to A\in F[R']$, either X is a superkey of R' or A is prime in R'.

Note that, although the BCNF test for a schema (R,F) is feasible in polynomial time, to check whether some subschema $R'\subseteq R$ is in BCNF is NP-complete [3]. The reason for this increase of the complexity is the following: For the BCNF-problem, we are given a relational schema (R,F). In other words, the FDs are part of the input. So we just have to examine each FD $f\in F$ and check if $lhs(f)$ is a superkey. For the Subschema-BCNF problem, we are given a relational schema (R,F) together with a subset $R'\subseteq R$. Hence, we have no explicit representation of the FDs $F[R']$

holding in R'. As mentioned above, $F[R']$ is only implicit in R', and (as illustrated by the above example) $F[R']$ might not admit a representation of polynomial size. Therefore, in the worst case, we have to go through an exponential number of FDs holding in R' and check for each such FD f if $lhs(f)$ is a superkey.

2.2. Tree decompositions and treewidth

A *hypergraph* is a pair $\mathcal{H} = \langle V, H \rangle$ consisting of a set V of vertices and a set H of hyperedges. A hyperedge $h \in H$ is a subset of V. The *primal graph* $P(\mathcal{H})$ (also called the Gaifman graph) has the same set of vertices as $\mathcal{H}$. Moreover, two vertices v_i, v_j are connected in $P(\mathcal{H})$ if they jointly occur in some hyperedge $h \in H$. On the other hand, the *incidence graph* $I(\mathcal{H})$ is a bipartite graph with vertices $V \cup H$ (i.e., the vertices of $I(\mathcal{H})$ are the vertices v_i of H plus the hyperedges h_j of H). Two vertices v_i and h_j are connected in $I(\mathcal{H})$ if in $\mathcal{H}$, the vertex v_i occurs in the hyperedge h_j.

A measure for the "tree-likeness" of a graph $\mathcal{G} = \langle V, E \rangle$ is the treewidth of $\mathcal{G}$, which we shall define below. A *tree decomposition* $\mathcal{T}$ of $\mathcal{G}$ is a pair $\langle T, \lambda \rangle$, where $T = \langle \mathcal{N}, \mathcal{E} \rangle$ is a tree (with nodes $\mathcal{N}$ and edges $\mathcal{E}$) and λ is a labeling function with $\lambda(N) \subseteq V$ for every node $N \in \mathcal{N}$, s.t. the following conditions hold:

1. $\forall v \in V$, there exists a node N in T, s.t. $v \in \lambda(N)$.
2. $\forall e \in E$, there exists a node N in T, s.t. $e \subseteq \lambda(N)$.
3. "Connectedness condition": $\forall v \in V$, the set of nodes $\{N | v \in \lambda(N)\}$ induces a connected subtree of T.

The sets $\lambda(N)$ for the nodes N in $\mathcal{T}$ are referred to as *bags*. The *width* of a tree decomposition $\langle T, \lambda \rangle$ is $\max(\{|\lambda(N)| - 1 : N \in \mathcal{N}\})$. Let $\mathcal{G} = \langle V, E \rangle$ be a graph. The *tree-width* $tw(\mathcal{G})$ is the minimum width over all its tree decompositions. Note that the maximum over $|\lambda(N)| - 1$ (rather than just over $|\lambda(N)|$) is taken in order to guarantee that trees (or, more generally, forests) are precisely the graphs with treewidth $= 1$. Cycles have treewidth $= 2$, and cliques of size n have treewidth $= n - 1$.

For given treewidth $w \geq 1$, it can be decided in linear time if a graph has treewidth $\leq w$. Moreover, in case of a positive answer, a tree decomposition of width w can be computed in linear time, see [9]. The practical usefulness of this linear time algorithm is limited due to a big multiplicative constant (which is doubly exponential in w) in the asymptotic time complexity. However, recently, a lot of progress has been made in developing heuristic based tree decomposition algorithms which can handle graphs with moderate size of several hundreds of vertices [40,10,52,11].

The treewidth $tw(\mathcal{H})$ of a hypergraph $\mathcal{H}$ can be either defined as the treewidth of the primal graph $P(\mathcal{H})$ or of the incidence graph $I(\mathcal{H})$. We thus set $tw_P(\mathcal{H}) = tw(P(\mathcal{H}))$ and $tw_I(\mathcal{H}) = tw(I(\mathcal{H}))$, respectively. In order to define the treewidth of a relational schema (R, F), we just have to indicate which hypergraph we are considering.

Definition 2.4. For a relational schema (R, F), we define the *hypergraph* of (R, F) as $\mathcal{H} = \langle V, H \rangle$ with $V = R$ and $H = \{\{A_1, \ldots, A_n, B\} | \ (A_1 \ldots A_n \rightarrow B) \in F\}$.

The primal graph and the incidence graph of (R, F) are simply defined as the corresponding graph of $\mathcal{H}$. Likewise, the *treewidth* of (R, F) is defined as $tw_P(\mathcal{H})$ or as $tw_I(\mathcal{H})$, respectively.

Let $\tau = \{P_1, \ldots, P_n\}$ be a set of predicate symbols. A *finite structure* $\mathfrak{A}$ over signature τ is given by a finite domain $A = dom(\mathfrak{A})$ and relations $P_i^{\mathfrak{A}} \subseteq A^\alpha$, where α denotes the arity of $P_i \in \tau$. A tree decomposition and the treewidth of a finite structure $\mathfrak{A}$ are naturally defined by defining the hypergraph corresponding to $\mathfrak{A}$.

Definition 2.5. Let $\mathfrak{A}$ be a finite structure with domain A. The *hypergraph* corresponding to $\mathfrak{A}$ is defined as $\mathcal{H} = \langle V, H \rangle$ with $V = A$ and $H = \{\{a_1, \ldots, a_n\} | \mathfrak{A}$ contains a relational ground atom $P(a_1 \ldots a_n)$ for some predicate symbol $P\}$.

The primal graph and the incidence graph of $\mathfrak{A}$ as well as the notions of *tree decomposition* and *treewidth* of $\mathfrak{A}$ are then defined in the obvious way via the hypergraph $\mathcal{H}$.

Example 2.6. Consider the relational schema from Example 1.1, whose hypergraph $\mathcal{H}$ plus primal graph $P(\mathcal{H})$ and incidence graph $I(\mathcal{H})$ were shown in Fig. 1. In Fig. 2, we show a possible tree decomposition of each of these graphs. Note that both decompositions have width 2. Since both graphs contain a cycle and only trees have treewidth $= 1$, these tree decompositions have minimal width. In other words, both graphs, $P(\mathcal{H})$ and $I(\mathcal{H})$, have treewidth $= 2$.

In this paper, we mainly consider the *treewidth via the primal graph*—the only exception being Section 5, which is devoted to the incidence graph. Hence, outside Section 5, we shall simply speak about the treewidth (or a tree decomposition, resp.) of a relational schema (R, F) or a structure $\mathfrak{A}$ in order to refer to the treewidth (or a tree decomposition, resp.) of the *primal graph* of the corresponding hypergraph $\mathcal{H}$.

Let $\mathcal{T}$ be a tree decomposition of the primal graph of $\mathcal{H}$, N a node in $\mathcal{T}$ and h a hyperedge in $\mathcal{H}$. We say that h is *covered* by the node N, if $h \subseteq \lambda(N)$. Analogously, we say that an FD f in a relational schema or a ground atom A in a finite structure is covered by N if the corresponding hyperedge in the corresponding hypergraph is. The following proposition is folklore (see e.g. [51]).

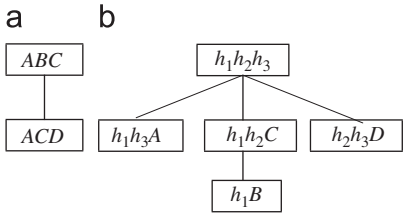

Fig. 2. Tree decompositions of Example 2.6. (a) Tree decomposition of the primal graph; (b) tree decomposition of the incidence graph.

Proposition 2.7. *Let $\mathcal{H}$ be a hypergraph and let $\mathcal{T}$ be a tree decomposition of the primal graph of $\mathcal{H}$. Then, for every hyperedge h in $\mathcal{H}$, there exists a node N in $\mathcal{T}$ with $h \subseteq \lambda(N)$.*

In other words, suppose that we have a relational schema (R,F) and a tree decomposition $\mathcal{T}$ of the primal graph of (R,F). Then, for every FD $f \in F$, there exists a node N in $\mathcal{T}$, s.t. f is covered by F. As an immediate consequence, we observe that an FD with many attributes on its left-hand side results in a high treewidth. Another source of high treewidth are big cliques in the primal graph. A clique in the primal graphs corresponds to a set X of attributes where any two attributes $A_i, A_j \in X$ jointly occur in some FD. For the treewidth via the incidence graph, FDs with many attributes on the left-hand side are harmless. On the other hand, big subgraphs forming a full bipartite graph lead to big treewidth. A full bipartite graph corresponds to a set X of attributes and a set G of FDs such that every A in X occurs in every FD f in G.

We conclude this section by discussing the intuition of tree decompositions and the consequences for constructing algorithms based on tree decompositions: Condition 1 in the definition of the labeling function λ just makes sure that no vertex is "forgotten". In a graph G without isolated vertices, condition 1 is trivially fulfilled whenever condition 2 is. Conditions 2 and 3 are the essence of this definition. Condition 2 guarantees that any two adjacent vertices v_i, v_j of the graph G jointly occur in the bag $\lambda(N)$ of some node N of the tree decomposition $\mathcal{T}$. Moreover, by Proposition 2.7, every FD f of a relational schema is covered by some bag $\lambda(N)$ in the tree decomposition of the corresponding primal graph. Condition 3 makes sure that in a (bottom-up or top-down) traversal of a tree decomposition $\mathcal{T}$, the following behavior is guaranteed: Suppose that along this traversal, we have encountered some vertex v (resp. some attribute A) in some bag $\lambda(N)$. Moreover, suppose that when continuing this traversal, we eventually reach some node N' such that v (resp. A) in no longer contained in the bag $\lambda(N')$. Then we can be sure that, along this traversal, we shall never encounter v (resp. A) again.

Algorithms exploiting low treewidth often follow a dynamic programming approach. Such an algorithm thus traverses a tree decomposition once in bottom-up direction and—in every node N of the tree decomposition $\mathcal{T}$—stores all the relevant information in some data structure which only depends on $\lambda(N)$ (but not on the entire input and not on the elements that appeared in the bag of some node N' below N but not in the bag $\lambda(N)$), see e.g. [50]. The fact that the relevant information does not have to take the elements in the bags $\lambda(N')$ of the nodes N' below N into account is due to conditions 2 and 3 imposed on the labeling function λ: By condition 3, when continuing the bottom-up traversal, the algorithm will never encounter again a bag with an element from $\lambda(N')\backslash\lambda(N)$. Hence, by condition 2, all edges containing one endpoint in $\lambda(N')\backslash\lambda(N)$ must be covered by the bag of some node N'' below N. These dynamic programming algorithms usually have to do some processing related to every edge of the graph. Hence, at every node N of the tree decomposition, we can be sure that all edges have already been processed that involve some endpoint in $\lambda(N')\backslash\lambda(N)$ for some node N' below N.

Our algorithms presented in the subsequent sections take a slightly different approach in that they are based on the idea of an alternating Turing machine using "guesses" to express non-determinism. The determinization of these algorithms would result in a top-down traversal of the tree decomposition implementing for-loops (rather than guesses) in each node N. The range of the counter variables for these for-loops only depends on the size of each bag $\lambda(N)$ (but not on the entire input and not on the elements that appeared in the bag of some node N' above N but not in the bag $\lambda(N)$). Again this is due to conditions 2 and 3 imposed on the labeling function λ.

In this paper, we shall prove that five otherwise intractable database design problems (i.e., PRIMALITY and 3NFTEST both for a schema and for a subschema as well as BCNFTEST for a subschema) become tractable when either the treewidth of the incidence graph or of the primal graph of the relational schemas under consideration is bounded by a constant. For bounded treewidth of the primal graph, we establish the tractability by presenting concrete algorithms in Sections 3 and 4. For bounded treewidth of the incidence graph, we prove the tractability by providing monadic-second order (MSO) encodings of these problems and by applying Courcelle's Theorem (see Section 5). It should be noted that these two kinds of tractability results (i.e., via a concrete algorithm for bounded treewidth of the primal graph resp. via Courcelle's Theorem for bounded treewidth of the incidence graph) are somehow *incomparable* in strength: On the one hand, a concrete, polynomial-time algorithm (where the multiplicative constant in the upper bound on the time complexity is singly exponential in the treewidth) is clearly preferable to a tractability result via Courcelle's Theorem. Indeed, as was noted in [34], Courcelle's Theorem is a valuable tool for detecting fixed-parameter tractability but algorithms directly derived from Courcelle's Theorem (e.g., by methods presented in [21]) are, in general, useless due to excessively big multiplicative constants. On the other hand, the following relationship between the treewidth of the incidence graph and of the primal graph is folklore. Applied to the problems considered here, it means that bounded treewidth of the incidence graph (rather than the primal graph) allows us to identify a strictly bigger class of relational schemas for which the above-mentioned problems become tractable.

Proposition 2.8. *Let $(\mathcal{H}_i)_{i\in I}$ be a family of hypergraphs whose treewidth of the primal graph is bounded by some constant. Then also the treewidth of the incidence graph of these hypergraphs is bounded by some constant. More precisely, for every hypergraph $\mathcal{H}$, we have $tw_I(\mathcal{H}) \leq tw_P(\mathcal{H}) + 1$.*

Proof. *Sketch*: It suffices to prove the second (stronger) assertion: Let $\mathcal{H}$ be a hypergraph and let $\mathcal{T}$ be a tree decomposition of the primal graph of $\mathcal{H}$ of width w. Then we can extend $\mathcal{T}$ to a tree decomposition $\mathcal{T}'$ of the incidence graph of $\mathcal{H}$ of width $\leq w+1$ by introducing the

following additional nodes: For every hyperedge h of $\mathcal{H}$, we select a node N in $\mathcal{T}$, s.t. h is covered by $\lambda(N)$. By Proposition 2.7, such an N exists. Then, in $\mathcal{T}'$, we create an additional child node N_h of N with label $\lambda(N_h) = \lambda(N) \cup \{h\}$. Clearly, $\lambda(N_h)$ covers all edges adjacent to h in the incidence graph of $\mathcal{H}$. Since we carry out this step for every hyperedge h in $\mathcal{H}$, $\mathcal{T}'$ fulfills condition 2. Moreover, conditions 1 and 3 are not violated by our construction. Hence, $\mathcal{T}'$ is indeed a tree decomposition of the incidence graph of $\mathcal{H}$. Moreover, its width is $\leq w+1$. □

It will turn out in Sections 3 and 4 that the tree decomposition of the primal graph makes the construction of an algorithm easier. The reason for this is that (by Proposition 2.7) we can be sure that in such a tree decomposition, every FD is covered by the bag $\lambda(N)$ of some node N in $\mathcal{T}$. Of course, this is not the case for tree decompositions of the incidence graph. On the other hand, our MSO-encodings in Section 5 will be based on predicates on $R \times F$ expressing properties like "attribute A occurs on the left-hand side (resp. on the right-hand side) of FD f". This kind of information is readily available in the incidence graph if we consider the edges in this graph as labeled (thus expressing if the attribute occurs on the left-hand side or on the right-hand side). In contrast, this kind of information is not available in the primal graph. It is, therefore, not obvious how to provide MSO-encodings also in this case. In summary, both notions of treewidth are worth considering since, in our case, both have advantages and disadvantages.

2.3. Complexity

LogCFL is the class of decision problems which are logspace reducible to a context-free language. The relationship between LogCFL and other well-known complexity classes is summarized as follows:

$$\mathsf{NC}^1 \subseteq \mathsf{L} \subseteq \mathsf{NL} \subseteq \mathsf{LogCFL} \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2 \subseteq \mathsf{P}$$

By L (resp. NL) we denote the class of problems that can be decided in deterministic (resp. non-deterministic) logspace. The class P contains the problems decidable in deterministic polynomial time. The classes AC^i and NC^i are logspace-uniform circuit-based parallel computation classes. For details on these classes, see e.g. [37]. The class 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 Greibachs hardest context-free language [33]. Recently, a number of interesting natural problems have been shown to be LogCFL-complete like the evaluation of acyclic Boolean conjunctive queries [27], (uniform) membership problems for languages given by tree automata [41], or the evaluation of CoreXPath queries (a fragment of the XML-query language XPath) [25].

Since $\mathsf{LogCFL} \subseteq \mathsf{AC}^1 \subseteq \mathsf{NC}^2$, the problems in LogCFL are *highly parallelizable*. In fact, they are solvable in logarithmic time by a concurrent-read, concurrent-write parallel random access machine (CRCW PRAM) with a polynomial number of processors, or in $\log^2-$time by an exclusive-read, exclusive-write (EREW) PRAM with a polynomial number of processors. Such PRAM algorithms can be obtained by methods from [38,49].

In this paper, we will use Ruzzo's characterization [49] of LogCFL by Alternating Turing machines (ATMs). In principle, an ATM [15] is defined like a non-deterministic Turing machine (NTM) with a finite set of states, a finite set of tape symbols, an initial state, and a transition relation. However, ATMs extend NTMs in that the states of an ATM are partitioned into existential and universal states (an NTM corresponds to the special case of existential states only). A computation tree of an ATM on an input string w is a tree whose nodes are labeled with configurations of M on w, such that the descendants of any internal node labeled by a universal (resp. 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 computation tree yields a certificate that the input is accepted. A complexity measure considered by Ruzzo [49] for the computation of an ATM is the *tree-size*, i.e. the minimal size of an accepting computation tree.

Definition 2.9 (*Ruzzo [49]*). A decision problem $\mathcal{P}$ 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 $\mathcal{P}$, there is at least one accepting computation tree T for M on w, s.t. the number of nodes in T is $\leq Z(n)$ and each node of T represents a configuration using $\leq S(n)$ space, where n is the size of w. Moreover, for any "no"-instance w of $\mathcal{P}$, there is no accepting computation tree for M.

Proposition 2.10 (*Ruzzo [49]*). LogCFL *coincides with those decision problems, which are recognized by ATMs operating simultaneously in tree-size* $O(n^{O(1)})$ *and space* $O(\log n)$.

3. Bounded treewidth and BCNF

Recall from Section 2.1 that it can be tested efficiently if a given relational schema (R,F) is in BCNF. Indeed, on just as to inspect all FDs $f \in F$ and check if the left-hand side $lhs(f)$ is a superkey (see [48]). The situation changes dramatically if, in addition to the schema (R,F), we are given a subschema $R' \subset R$ and we want to test if this subschema is in BCNF. As stated in Section 2.1, to check whether some subschema $R' \subset R$ is in BCNF is NP-complete. As explained in Section 2.1, the reason for the intractability is that the FDs $F[R']$, which hold in R', may only admit representations of exponential size. Hence, in the worst case, a Subschema-BCNF test has to inspect exponentially many FDs f from $F[R']$ and check if the left-hand side $lhs(f)$ is a superkey of R'.

In this section, we present an efficient algorithm (see Fig. 3) for the Subschema-BCNF problem in case that the set of FDs has bounded treewidth. Of course, this algorithm never explicitly computes a representation of $F[R']$. The algorithm in Fig. 3 is given in the style of an alternating Turing machine (ATM). The universal steps correspond to the for-each loops with the recursive calls of the sub-not-bcnf procedure (see step 7 resp. r7). For the

Algorithm Subschema-Not-BCNF
Input Relational schema (R, F), tree decomposition $\mathcal{T}$ of the primal graph of (R, F) with width k, subset $R' \subseteq R$.
Output "Accept", if R' is not in BCNF.

Global variables: A, B.

Procedure check-not-bcnf (Y_M: SetOfAttributes, Z_M,O_M: OrderedSetOfAttributes, M: Node in $\mathcal{T}$)
begin
Check 1: $A \notin Y_M$, $B \notin Y_M$, and $B \notin Z_M$.
Check 2: (* $Y_M \cup Z_M$ closed w.r.t. F *)
$\forall f \in F$: If f is covered by $\lambda(M)$ and $lhs(f) \subseteq Y_M \cup Z_M$ then $rhs(f) \in Y_M \cup Z_M$.
Check 3: $\forall C \in O_M$:
$\exists (D_1, \ldots, D_m \rightarrow C) \in F$ covered by $\lambda(M)$ s.t. $\forall i \leq m$: $(D_i \in Y_M) \vee (D_i \in Z_M \wedge D_i < C)$ holds.
if all Checks succeed **then Accept**
else HALT and **Reject**;
end;

Procedure sub-not-bcnf (Y_N: SetOfAttributes, Z_N, O_N: OrderedSetOfAttributes, N_i: Node in $\mathcal{T}$)
begin
1) **Guess** $Y_{N_i} \subseteq \lambda(N_i) \cap R'$ s.t.:
1.a) $\forall C \in \lambda(N) \cap \lambda(N_i)$: $C \in Y_N \Leftrightarrow C \in Y_{N_i}$
2) **Guess** an ordered set $Z_{N_i} \subseteq \lambda(N_i)$ s.t.:
2.a) If $A \in \lambda(N_i)$, then $A \in Z_{N_i}$,
2.b) $Y_{N_i} \cap Z_{N_i} = \emptyset$,
2.c) $\forall C \in \lambda(N) \cap \lambda(N_i)$: $C \in Z_N \Leftrightarrow C \in Z_{N_i}$
2.d) The orderings on Z_N and Z_{N_i} are consistent.
3) $O_{N_i} := O_N \cup (Z_{N_i} - Z_N)$;
3.a) **if not** $O_{N_i} \subseteq Z_{N_i}$ **then HALT** and **Reject**;
4) **Guess** an assignment $\alpha_{N_i} : O_{N_i} \rightarrow \{0, \ldots, n\}$, where n = number of child nodes of N_i.
5) check-not-bcnf (Y_{N_i}, Z_{N_i}, $\alpha_{N_i}^{-1}(0)$, N_i);
6) **if** the check 5) fails then **Reject**;
7) **if for each** $\ell \in \{1, \ldots, n\}$
sub-not-bcnf (Y_{N_i}, Z_{N_i}, $\alpha_{N_i}^{-1}(\ell)$, M_ℓ) = Accept, where M_ℓ is the ℓ-th child of N_i in $\mathcal{T}$
then Accept
else Reject;
end;

begin (* MAIN *)
r0) (* initializations *)
$N :=$ root of $\mathcal{T}$;
Guess $A, B \in R'$ with $A \neq B$;
r1) **Guess** $Y_N \subseteq \lambda(N) \cap R'$
r2) **Guess** an ordered set $Z_N \subseteq \lambda(N)$ s.t.:
r2.a) If $A \in \lambda(N)$, then $A \in Z_N$,
r2.b) $Y_N \cap Z_N = \emptyset$,
r3) $O_N := Z_N$;
r4) **Guess** an assignment $\alpha_N : O_N \rightarrow \{0, \ldots, n\}$, where n = number of child nodes of N.
r5) check-not-bcnf (Y_N, Z_N, $\alpha_N^{-1}(0)$, N);
r6) **if** the check 5) fails then **Reject**;
r7) **if for each** $i \in \{1, \ldots, n\}$ sub-not-bcnf (Y_N, Z_N, $\alpha_N^{-1}(i)$, N_i) = Accept, where N_i is the i-th child of N in $\mathcal{T}$
then Accept
else Reject;
end.

Fig. 3. Subschema-Not-BCNF test.

existential steps, we use the keyword "Guess" to indicate that at least one such value must exist which ultimately leads to an "Accept" result value. It is important to note that (apart from step r0, which is executed only once in the entire computation) the range of possible values from which we guess is bounded by a *constant* (depending on the treewidth but not on the overall size of the input). Hence, the guesses do not lead to non-determinism in the general sense. A corresponding deterministic algorithm (which replaces the guesses by nested for-loops over all possible values) would therefore work in polynomial time. We shall come back to this point at the end of this section when we discuss in more detail the determinization of the alternating algorithm.

The Subschema-Not-BCNF algorithm in Fig. 3 proceeds by a top-down traversal of the tree decomposition $\mathcal{T}$ rooted at any node. This tree traversal is realized via recursive calls of the procedure sub-not-bcnf. The goal of the algorithm is to find a BCNF-violation in R', i.e., attributes $A, B \in R'$ and a subset $\mathcal{Y} \subseteq R'$, s.t. $\mathcal{Y} \rightarrow_F A$, $A \notin \mathcal{Y}$, and $\mathcal{Y} \nrightarrow_F B$. In other words, $\mathcal{Y} \subset R'$ determines some attribute $A \in R' \setminus \mathcal{Y}$ even though $\mathcal{Y}$ is not a superkey in R' (since B is not determined by $\mathcal{Y}$). The attributes A and B are guessed in step r0. The attribute set $\mathcal{Y} \subseteq R'$ is guessed

during the traversal of the tree decomposition $\mathcal{T}$ as follows: At each node N_i in $\mathcal{T}$, we guess Y_{N_i} in step 1 with the intended meaning $Y_{N_i} = \lambda(N_i) \cap \mathcal{Y}$. Condition 1.a guarantees that the guess Y_{N_i} at N_i is consistent with the guess Y_N at the parent N of N_i.

Step 2 aims at guessing the closure $clos_F(\mathcal{Y})$ of $\mathcal{Y}$. The intended meaning of Z_{N_i} is $Z_{N_i} = \lambda(N_i) \cap (clos_F(\mathcal{Y}) \setminus \mathcal{Y})$. The conditions 2.a–2.d are straightforward. In particular, condition 2.a makes sure that A is indeed determined by $\mathcal{Y}$. The conditions 2.c and 2.d make sure that the guesses at any node N_i are consistent with the guesses at the parent N of N_i. The purpose of the ordering imposed on Z_{N_i} will become clear when we discuss the check-not-bcnf procedure. The ordering that we have in mind here is the following:

Definition 3.1. Let $clos_F(\mathcal{Y}) = \mathcal{Y} \cup \mathcal{Z}$ with $\mathcal{Y} \cap \mathcal{Z} = \emptyset$. We define the "derivation ordering" on $\mathcal{Z}$ by considering an arbitrary derivation sequence Δ of $\mathcal{Z}$ from $\mathcal{Y}$. Suppose that Δ has the form $\Delta \equiv \mathcal{Y} \to \mathcal{Y} \cup \{C_1\} \to \mathcal{Y} \cup \{C_1, C_2\} \to \cdots \to \mathcal{Y} \cup \{C_1, C_2, \ldots, C_z\}$, where $\mathcal{Z} = \{C_1, \ldots, C_z\}$ and $C_i \neq C_j$ if $i \neq j$. Then we set $C_i < C_j$ if $i < j$.

In other words, given an attribute set $\mathcal{Y}$, the derivation ordering $<$ is defined on the attributes which are in the closure of $\mathcal{Y}$ but not in $\mathcal{Y}$ itself. The ordering reflects the order in which these attributes are derived by a particular derivation sequence Δ. Of course, in general, there exist several derivation sequences and the derivation-ordering depends on the concrete choice of Δ. In the sequel, we assume Δ to be arbitrarily chosen but fixed (for any set $\mathcal{Y}$).

The set O_{N_i} computed at step 3 consists of all attributes in Z_{N_i} for which it has to be verified yet that they are indeed determined by $\mathcal{Y}$. The ordering on O_{N_i} can be taken over from Z_{N_i} since, by condition 3.a, $O_{N_i} \subseteq Z_{N_i}$ holds. This condition ensures for all attributes $C \in O_N$, that the property $C \in clos_F(\mathcal{Y})$ is verified as long as C is contained in $\lambda(N)$.

For $C \in O_{N_i}$, the place where the property $\mathcal{Y} \to C$ shall be verified is guessed in step 4, namely: For each $C \in O_{N_i}$, we either verify at node N_i that $\mathcal{Y} \to C$ holds (in this case, we guess $\alpha_{N_i}(C) = 0$) or the job to verify this property is passed on to the ℓ−th child of N_i (i.e., we guess $\alpha_{N_i}(C) = \ell$ with $\ell \geq 1$).

The check-not-bcnf procedure called at step 5 has the following tasks: Check 1 is clear. Check 2 ensures that $(\mathcal{Y} \cup \mathcal{Z})$ is closed w.r.t. the FDs F. Hence, it is ultimately guaranteed that $(\mathcal{Y} \cup \mathcal{Z}) \supseteq clos_F(\mathcal{Y})$ holds. Conversely, check 3 takes care of the property $(\mathcal{Y} \cup \mathcal{Z}) \subseteq clos_F(\mathcal{Y})$. More precisely, it is checked that all attributes in $\mathcal{Z}$ can be derived from $\mathcal{Y}$ together with *smaller* attributes from $\mathcal{Z}$. Hence, given that $D_i \in clos_F(\mathcal{Y})$ is checked elsewhere for all $i \in \{1, \ldots, m\}$ with $D_i \in Z_M$, we may assume at this place that $C \in clos_F(\mathcal{Y})$ holds. It is now also clear why we impose some ordering on $\mathcal{Z}$, namely: We have to make sure that there are no cycles in the derivation of the attributes in $\mathcal{Z}$.

In step 7, the procedure sub-not-bcnf is called recursively for all child nodes M_ℓ of N_i. The first two parameters Y_{N_i} and Z_{N_i} communicate the guesses at N_i to each M_ℓ. The third parameter passes precisely those attributes $C \in O_{N_i}$ to the ℓ−th child for which we have guessed $\alpha_{N_i}(C) = \ell$.

Note that the steps r1 through r7 at the root of $\mathcal{T}$ have exactly the same meaning as the steps 1 through 7 in procedure sub-not-bcnf. Of course, some tasks are slightly simpler since the root N has no parent. Step r0 carries out some initializations. In particular, the attributes A and B (which will witness the BCNF-violation) are guessed here and stored in global variables.

In Example 3.2 we illustrate the step by step execution of the algorithm from Fig. 3. As mentioned above, the Guess-statements can be realized by exhaustively listing all possible value combinations the size of which is bounded by a function that depends on the treewidth k (which is considered as constant) but not on the size of the input.

Example 3.2. Consider a relational schema $R = CDEF$, $R' = CEF$ with FDs $C \to D, D \to E, F \to D$. A tree decomposition of the primal graph is shown in Fig. 4. Its width is 1.

At step r0, we have to guess $A, B \in R'$. In the corresponding deterministic algorithm, this means, that we have to iterate in a loop over all $n(n-1)$ possible value combinations of A and B, where $n = |R'|$. For our further discussion, let us assume that $A = E$ and $B = F$. Of course, this value combination will eventually be reached by the loop over all possible values of A and B. Below, we shall continue the discussion of all "guesses" by assuming that a certain value is chosen. By this, we mean that each Guess-statement is implemented in the form of a loop over all possible values and that the assumed value will be eventually reached by the loop.

At step r1, we guess Y_N. There are 2^2 possible values for Y_N, and we assume that we take $Y_N = \{C\}$. At steps r2 and r3, we guess Z_N and O_N. Let us assume that we take $Z_N = \{D\}$ and, thus also $O_N = \{D\}$. At step r4, we have to guess an assignment for the only element D in O_N to either 0 or to one of the child nodes of N (corresponding to the values 1 and 2). Suppose that we choose $\alpha_N = D \to 0$, which means, the task of checking whether D is in the closure of Y_N is assigned to the root itself. The call of procedure check-not-bcnf at step r5 obviously returns Accept, thus we reach step r7, and call the procedure sub-not-bcnf for the two child nodes. Let us assume that the left child node is node 1 and the right child node is node 2.

Now we start to run the procedure sub-not-bcnf for node 1, with the input parameters as follows: $Y_N = \{C\}$, $Z_N = \{D\}$, $O_N = \emptyset$, and N is the root node. At step 1, we guess Y_{N_i}. Let us assume that we set $Y_{N_i} = \emptyset$. At step 2, we guess Z_{N_i}. We assume that $Z_{N_i} = \{D, E\}$. Thus at step 3, $O_{N_i} = \{E\}$. At step 4, we have to guess an assignment to the

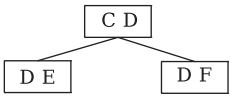

Fig. 4. Tree decomposition of Example 3.2.

only element E in O_{N_i}. Since node 1 is a leaf node, the only possible assignment is $\alpha_{N_i} = E \to 0$. Moreover, we can ignore step 7. Clearly, the check at step 5 is successful, thus the procedure returns Accept.

At node 2, the input parameters for the procedure sub-not-bcnf are identical to those for node 1. This time, assume that we guess Y_{N_i} to be $\emptyset$ and Z_{N_i} to be $\{D\}$. Thus, we get $O_{N_i} = \emptyset$. Again, the call of procedure check-not-bcnf is successful, thus the procedure returns Accept as well.

In total, the Subschema-Not-BCNF algorithm returns Accept, which means that $R' = CEF$ is not in BCNF. Indeed, the above discussion of one computation tree of the algorithm in Fig. 3 corresponds to the BCNF violation with $Y = \{C\}$, $A = E$ and $B = F$, such that $Y \subseteq R'$, $A, B \in R'$, $Y \to A$ and $Y \nrightarrow B$.

In Theorem 3.4, we shall give a formal proof of the correctness of the Subschema-Not-BCNF algorithm. More specifically, we shall show that there exists an accepting computation of the non-deterministic Subschema-Not-BCNF algorithm if and only if R' is not in BCNF. To this end, we first prove a lemma which is crucial for the "only if"-direction of the theorem.

Lemma 3.3. *Consider an accepting computation of the Subschema-Not-BCNF algorithm on input (R, F), $\mathcal{T}$, and R'. Moreover, let $\mathcal{Y}$ and $\mathcal{Z}$ be defined as $\mathcal{Y} := \bigcup_{N \in \mathcal{T}} Y_N$ and $\mathcal{Z} := \bigcup_{N \in \mathcal{T}} Z_N$. Then $clos_F(\mathcal{Y}) = \mathcal{Y} \cup \mathcal{Z}$ holds.*

Proof. First consider the inclusion $clos_F(\mathcal{Y}) \subseteq \mathcal{Y} \cup \mathcal{Z}$. It suffices to show that $\mathcal{Y} \cup \mathcal{Z}$ is closed w.r.t. F. But this property follows immediately from check 2 in the check-not-bcnf procedure.

Now consider the inclusion $clos_F(\mathcal{Y}) \supseteq \mathcal{Y} \cup \mathcal{Z}$. By the conditions 2.c and 2.d, the orderings that are guessed for two subsets $Z_N \subseteq \mathcal{Z}$ and $Z_{N'} \subseteq \mathcal{Z}$ are consistent for any two *neighboring nodes* $N, N' \in \mathcal{T}$. By the connectedness condition in the tree decomposition $\mathcal{T}$, this implies that the orderings guessed for any two subsets $Z_N \subseteq \mathcal{Z}$ and $Z_{N'} \subseteq \mathcal{Z}$ for *arbitrary nodes* $N, N' \in \mathcal{T}$ are consistent. Hence, it is possible to consistently extend the orderings guessed on all subsets $Z_{N_i} \subseteq \mathcal{Z}$ to a total ordering on all of $\mathcal{Z}$. The proof of the inclusion $clos_F(\mathcal{Y}) \supseteq \mathcal{Y} \cup \mathcal{Z}$ proceeds by induction on this ordering.[1]

Induction begin: Let C be minimal in $\mathcal{Z}$ and let N denote the first node (in top-down direction) of $\mathcal{T}$ with $C \in Z_N$. By step 3 of the algorithm, C is added to O_N. Moreover, by step 7, C remains in $O_{N'}$ in a sequence of descendants N' of N until finally $\alpha_M(C) = 0$ is guessed for $M = N$ or for some descendant M of N. In a successful computation, all checks in the check-not-bcnf procedure succeed. Hence, by check 3, there exists an FD $D_1, \ldots, D_m \to C$ in F s.t. $\forall i \in \{1, \ldots, m\}$, $(D_i \in Y_M) \vee (D_i \in Z_M \wedge D_i < C)$ holds. The second disjunct cannot be true when C is minimal in $\mathcal{Z}$. Hence, C is indeed determined by attributes in $\mathcal{Y}$.

Induction step: Now suppose that C is not minimal in $\mathcal{Z}$. By the same argumentation as above, we may conclude (for a successful computation), that eventually check 3 of the check-not-bcnf procedure will be successfully applied to C. Hence, by check 3, there exists an FD $D_1, \ldots, D_m \to C$ in F s.t. $\forall i \in \{1, \ldots, m\}$, $(D_i \in Y_M) \vee (D_i \in Z_M \wedge D_i < C)$ holds. Thus, each D_i is either in $\mathcal{Y} \subseteq clos_F(\mathcal{Y})$ or—by the induction hypothesis—in $clos_F(\mathcal{Y})$. In other words, C is determined by attributes in $clos_F(\mathcal{Y})$. Hence, C itself is in $clos_F(\mathcal{Y})$. □

Theorem 3.4. *Let (R, F) be a relational schema and let $R' \subseteq R$ be a subschema. Moreover, let $\mathcal{T}$ be a tree decomposition of (R, F). The Subschema-Not-BCNF algorithm in Fig. 3 accepts input $((R, F), \mathcal{T}, R')$ if and only if R' is not in BCNF.*

Proof. The following equivalence has to be shown: There exists an accepting computation of the non-deterministic Subschema-Not-BCNF algorithm $\Leftrightarrow$ R' is not in BCNF. □

"$\Rightarrow$" Let $\mathcal{Y} := \bigcup_{N \in \mathcal{T}} Y_N$ and let A, B be the attributes guessed in step r0. We claim that R' has the following BCNF-violation: $A \notin \mathcal{Y}$, $\mathcal{Y} \to_F A$, and $\mathcal{Y} \nrightarrow_F B$.

Let $\mathcal{Z} := \bigcup_{N \in \mathcal{T}} Z_N$. By the connectedness condition of $\mathcal{T}$ and condition 1.a of the algorithm, we may conclude that the sets Y_{N_i} have been consistently guessed for all nodes N_i. Hence, we have $Y_{N_i} = \lambda(N_i) \cap \mathcal{Y}$. Likewise, by the connectedness condition of $\mathcal{T}$ and the conditions 2.c and 2.d of the algorithm, we may conclude that the sets Z_{N_i} and the orderings on these sets have also been consistently guessed, i.e., $Z_{N_i} = \lambda(N_i) \cap \mathcal{Z}$ and the orderings on the subsets $Z_{N_i} \subseteq \mathcal{Z}$ can be consistently extended to a total ordering on $\mathcal{Z}$.

Of course, the following conditions are fulfilled: $\mathcal{Y} \subseteq R'$ (by step 1), $A \notin \mathcal{Y}$ (by check 1), $A \in \mathcal{Z}$ (by condition 2.a), $B \notin \mathcal{Y}$ (by check 1), and $B \notin \mathcal{Z}$ (also by check 1). Hence, the "$\Rightarrow$" – direction follows immediately from Lemma 3.3.

"$\Leftarrow$" Now suppose that there exists a BCNF-violation of R', i.e., $\exists \mathcal{Y} \subseteq R'$, $\exists A, B \in R'$, s.t. $A \notin \mathcal{Y}$, $\mathcal{Y} \to_F A$, and $\mathcal{Y} \nrightarrow_F B$. Let $\mathcal{Z} := clos_F(\mathcal{Y}) \setminus \mathcal{Y}$ and let Δ denote an arbitrary derivation sequence of $\mathcal{Z}$ from $\mathcal{Y}$.

We construct a computation tree τ of a successful computation of the Subschema-Not-BCNF algorithm as follows: The tree structure of the computation tree τ is identical to the tree structure of $\mathcal{T}$. At the root of $\mathcal{T}$, our algorithm guesses precisely the attributes $A, B \in R'$ with the above mentioned properties. Moreover, at each node N, the values $Y_N := \mathcal{Y} \cap \lambda(N)$ and $Z_N := \mathcal{Z} \cap \lambda(N)$ are guessed. The ordering that we have to guess for Z_N in step 2 is the derivation-ordering (for the arbitrarily chosen derivation sequence Δ) from Definition 3.1.

It remains to show that it is possible to guess an appropriate assignment α_N at each node N, s.t. the checks in the check-not-bcnf procedure always succeed. Actually, for the chosen values of A, B, Y_N, and Z_N, the checks 1 and 2 are clearly successful at any node N. In order to show that also check 3 always succeeds for appropriately chosen α_N, it suffices to prove the following Claim A.

[1] Note that we do not assume that the ordering thus constructed is indeed the derivation ordering defined in Definition 3.1. For the proof of Lemma 3.3 (and, thus, for the "$\Rightarrow$" – part of the proof of Theorem 3.4), any total ordering does the trick. The derivation ordering is only needed in the "$\Leftarrow$" – part of the proof of Theorem 3.4.

Claim A. *Every attribute $C \in \mathcal{Z}$ added to O_N at some node N by the Subschema-Not-BCNF algorithm will be eventually "eliminated" without producing an error, i.e., C is either eliminated at N (i.e., $\alpha_N(C) = 0$) or C is passed on to some descendant M of N where it is finally eliminated (i.e., $\alpha_M(C) = 0$).*

Proof of Claim A. Let $C \in \mathcal{Z}$ and let N denote the first node (in top-down direction) of the tree decomposition $\mathcal{T}$ with $C \in \lambda(N)$. Hence, C is guessed to be in Z_N in step 2 and C is added to O_N in step 3 of the algorithm. Now consider the derivation sequence Δ of $\mathcal{Z}$ from $\mathcal{Y}$. At some point, the attribute C is derived by Δ. Let $f \in F$ denote the FD which is used in Δ for deriving C. Then f is of the form $f = D_1, \ldots, D_m \to C$ and $\forall i \leq m$, D_i is either contained in $\mathcal{Y}$ or $D_i \in \mathcal{Z}$ and D_i has been derived prior to C. Hence, in the latter case, $D_i < C$ holds by the definition of our derivation-ordering.

Of course, there exists a node M in $\mathcal{T}$ s.t. $\lambda(M)$ covers f. By the connectedness condition of $\mathcal{T}$, we either have $N = M$ or M is a descendant of N and C occurs in $\lambda(N')$ for all nodes N' on the path from N to M. In the former case, we set $\alpha_N(C) = 0$. In the latter case, we choose the values of $\alpha_N(C)$ as well as of $\alpha_{N'}(C)$ for all these nodes N' in such a way that C is "passed on" to O_M. Here we finally set $\alpha_M(C) = 0$.

It remains to verify that check 3 in the check-not-bcnf procedure succeeds for the attribute C. The proof goes by induction on the derivation-ordering. Note that, in contrast to the "$\Rightarrow$" part of the proof of Theorem 3.4, we now need precisely the ordering defined in Definition 3.1.

If C is minimal in $\mathcal{Z}$, then C is derived directly from $\mathcal{Y}$, i.e., all D_i's are in $\mathcal{Y}$ and, therefore, also in Y_M. Hence, the first disjunct ($D_i \in Y_M$) in check 3 is true for each i. Now suppose that C is not minimal in $\mathcal{Z}$. For every $i \in \{1, \ldots, m\}$, either ($D_i \in \mathcal{Y}$) or ($D_i \in \mathcal{Z}$) and $D_i < C$. Hence, the disjunction $(D_i \in Y_M) \vee (D_i \in Z_M \wedge D_i < C)$ in check 3 is again true for every i.

This concludes the proof of Theorem 3.4. □

For the complexity of testing if a subschema is in BCNF, we have the following upper bound:

Theorem 3.5. *Let $k \geq 1$ be a fixed constant. Moreover, let (R, F) be a relational schema of treewidth $\leq k$ and let $R' \subseteq R$ be a subschema. Then it can be decided in polynomial time whether R' is in BCNF. Moreover, this decision problem is in* LogCFL.

Proof. Note that we do not require in the theorem that a tree decomposition $\mathcal{T}$ of (R, F) with width $k \geq 1$ has to be actually given. However, by [9], a tree decomposition $\mathcal{T}$ can be computed in linear time. Likewise, it has been shown in [28], that the computation of $\mathcal{T}$ is feasible in $\mathsf{L}^{\mathsf{LogCFL}}$ and that $\mathsf{L}^{\mathsf{LogCFL}}(\mathsf{LogCFL}) = \mathsf{LogCFL}$. Hence, in the remainder of the proof, we may assume that $\mathcal{T}$ is given.

We only show the LogCFL-membership. The polynomial time upper bound follows by the relationship $\mathsf{LogCFL} \subseteq \mathsf{P}$. It was shown in [13] that LogCFL is closed under complement. Hence, in order to establish the LogCFL-membership of testing whether some subschema R' is in BCNF, it suffices to show that testing whether R' is *not* in BCNF is in LogCFL.

Note that the algorithm Subschema-Not-BCNF in Fig. 3 can be regarded as a high-level description of an alternating Turing machine (ATM). The existential steps of this ATM are contained in the non-deterministic guesses of this algorithm. The universal steps are encoded by the for-each loops with the recursive calls of the sub-not-bcnf procedure.

Recall that ATMs can be characterized in terms of the tree-size (i.e, the number of nodes) of the computation tree τ (where each node corresponds to a configuration of M) and the space required by each configuration. By the characterization of LogCFL in [49], (which we recalled in Proposition 2.10) it suffices to show that the size of τ is polynomially bounded and the data manipulated at each node fit into log-space.

Let n denote the total size of the input and let τ denote the computation tree of a (successful) computation of the Subschema-Not-BCNF algorithm. On the one hand, the size of τ is polynomially bounded in n. On the other hand, the data structures needed at each node N_i (i.e., the values Y_N, Z_N, O_N guessed at the parent N of N_i plus the values of Y_{N_i}, Z_{N_i}, O_{N_i}, α_{N_i} at N_i can be represented by *constantly* many pointers (where this *constant* clearly depends on the treewidth k). Thus, the desired LogCFL-result follows from Ruzzo's characterization of LogCFL (recalled in Proposition 2.10). □

Remark. In Section 5, the analogous fixed-parameter tractability result is shown via an MSO-encoding and by applying Courcelle's Theorem—which makes use of the correspondence between MSO-formulae and *finite tree automata*. In fact, our Subschema-Not-BCNF algorithm in Fig. 3 is closely related to a (non-deterministic top-down) finite tree automaton whose states correspond to the (constantly many) possible values of the data structures Y_{N_i}, Z_{N_i}, α_{N_i}, etc. But of course, our dedicated algorithm essentially differs from an FTA that one would obtain via a standard MSO-to-FTA transformation (like the one in [21]).

We conclude this section by a brief discussion of a polynomial-time, deterministic algorithm corresponding to the alternating algorithm in Fig. 3: As has already been mentioned above, such a determinization essentially consists in replacing every "Guess" by a for-loop over all possible values. A sequence of several guesses (in steps r0–r4 resp. steps 1–4) corresponds to nested for-loops. Let k denote the width of a given tree decomposition $\mathcal{T}$ and let n denote the size of the input. Then the following number of value combinations have to be considered by the nested for-loops which are used to determinize the guesses. We only discuss the steps 1–4 as well as r0. The steps r1–r4 are analogous to steps 1–4. In step 1, the for-loop iterates over all possible subsets $Y_{N_i} \subseteq \lambda(N_i) \cap R'$. Since $|\lambda(N_i)| \leq k + 1$, there are at most 2^{k+1} possible values for Y_{N_i}. In step 2, we have to iterate over all possible ordered

sets $Z_{N_i} \subseteq \lambda(N_i)$ and check that conditions 2.a–2.d are fulfilled. We thus get the upper bound $(k+1)!$ on the total number of values for Z_{N_i}. In step 3, we have to iterate over all possible mappings from the elements in O_{N_i} to $\{0,1,\ldots,\ell\}$, where ℓ denotes the number of child nodes of N_i. Actually, w.l.o.g., we may assume that all nodes in a tree decomposition have at most 2 child nodes. Indeed, suppose that some node M has $m \geq 3$ child nodes $M'_1,\ldots,M'_m$. Then we can introduce $m-2$ copies $M_1,\ldots,M_{m-2}$ of M (with identical bag as M) and replace M with its m child nodes by the following tree fragment: M has 2 children M_1 and M'_1; for every $j \in \{1,\ldots,m-3\}$,M_j has 2 children M_{j+1} and $M_{j+1}{}'$; finally M_{m-2} has 2 children $M_{m-1}{}'$ and M'_m. This is a standard technique applied e.g. in [21]. The resulting tree decomposition (whose size is linearly bounded in the size of the original tree decomposition) has the desired property that every node M has at most 2 child nodes. But then we get the upper bound 3^{k+1} on the number of possible assignments α_{N_i} from the elements of each set O_{N_i} to the values $\{0,1,2\}$. In total, the nested for-loops corresponding to the guesses at steps 1, 2, and 4 thus have to consider $2^{k+1} * (k+1)! * 3^{k+1}$ possible value combinations. Recall that we are considering the treewidth k as a constant. Hence, the number of value combinations to be considered by these for-loops is also bounded by a constant (which is singly exponential in the treewidth). Of course, in step r0, the possible values to be processed do depend on the input; i.e., there are at most n^2 possible combinations of attributes A and B, where n denotes the size of the input. Finally, the actions carried out at each step of the program can either be done in time which depends only on the size $k+1$ of the bags or in linear time w.r.t. the input size (e.g., Check 2 in procedure check-not-bcnf which—in a naive implementation—has to iterate over all FDs in the input schema. Of course, there is plenty of room for improvement since this check is necessary only once for each FD and not every time a bag covers an FD). Moreover, the number of nodes in a tree decomposition is linearly bounded w.r.t. size n of the input. Hence, in total, even with a naive implementation, we end up with the upper bound $\mathcal{O}(f(k) * n^4)$ where n denotes the size of the input and $f(k)$ is a constant which only depends on the treewidth k (more precisely, it is singly exponential in k).

4. Bounded treewidth and 3NF

In this section we first present a new algorithm for checking the primality of an attribute in a subschema. By combining this primality test with the ideas from Section 3, we shall ultimately construct an efficient algorithm also for the 3NF-test of a subschema. Of course, as a special case, we thus also cover the primality problem and the 3NF-problem in a schema.

The Subschema-Primality-Test algorithm in Fig. 5 works as follows: Given a relation (R,F), a subschema $R' \subseteq R$, and an attribute $A \in R'$, the primality of A in the subschema (R',F') with $F' = F[R']$ can be tested via the following criterion: There exists a set $\mathcal{X} \subseteq R'$, s.t. $A \notin clos_F(\mathcal{X})$ and $R' \subseteq clos_F(\mathcal{X} \cup \{A\})$. This criterion is obviously a necessary and sufficient criterion for $A \in R'$ to be prime in R'. Moreover, it is clearly equivalent to the following criterion: There exist sets $\mathcal{X} \subseteq R', \mathcal{Y} \subseteq R$, and $\mathcal{Z} \subseteq R$, s.t. the following conditions hold:

1. $A \notin \mathcal{X}$, $A \notin \mathcal{Y}$, and $A \notin \mathcal{Z}$;
2. $\mathcal{X} \cap \mathcal{Y} = \emptyset$ and $clos_F(\mathcal{X}) = \mathcal{X} \cup \mathcal{Y}$;
3. $\mathcal{X} \cap \mathcal{Z} = \emptyset$ and $clos_F(\mathcal{X} \cup \{A\}) = \mathcal{X} \cup \{A\} \cup \mathcal{Z}$;
4. $R' \subseteq \mathcal{X} \cup \{A\} \cup \mathcal{Z}$.

The goal of the Subschema-Primality-Test algorithm in Fig. 5 is to guess these sets $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ similarly to the Subschema-Not-BCNF algorithm in Fig. 3.

Let $\mathcal{T}$ be a tree decomposition of (R,F), s.t. the width of $\mathcal{T}$ is bounded by some constant $k \geq 1$. The algorithm in Fig. 5 contains a procedure sub-prime-attribute, which is used to realize a top-down traversal of the tree decomposition $\mathcal{T}$, where the root can be any node N with $A \in \lambda(N)$. Along the top-down traversal of $\mathcal{T}$, the algorithm tries to successively guess all elements of $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$.

The computation at each node N_i of the tree decomposition $\mathcal{T}$ is very similar to the Subschema-Not-BCNF algorithm in Fig. 3. Again, the algorithm is given in the style of an ATM. As before, the algorithm can be easily determinized by replacing the guesses by nested for-loops.

In step 1, we guess X_{N_i} with the intended meaning $X_{N_i} = \lambda(N_i) \cap \mathcal{X}$. Condition 1.a guarantees that the guess X_{N_i} at N_i is consistent with the guess X_N at the parent N of N_i. Step 2 aims at guessing the closure $clos_F(\mathcal{X})$ of $\mathcal{X}$. The intended meaning of Y_{N_i} is $Y_{N_i} = \lambda(N_i) \cap (clos_F(\mathcal{X}) \backslash \mathcal{X})$. The conditions 2.a–2.c are straightforward. They play the analogous role to the conditions 2.b–2.d in the Subschema-Not-BCNF algorithm. In particular, the conditions 2.b and 2.c make sure that the guesses at any node N_i are consistent with the guesses at the parent N of N_i. As in the previous section, the purpose of the ordering imposed on Y_{N_i} is to prevent circular derivations (cf. check 3a in the check-sub-primality procedure in Fig. 6). The ordering that we have in mind is precisely the ordering from Definition 3.1, which we already used in the Subschema-Not-BCNF algorithm.

In step 3, we guess the closure of $\mathcal{X} \cup \{A\}$. The intended meaning of Z_{N_i} is $Z_{N_i} = \lambda(N_i) \cap [clos_F(\mathcal{X} \cup \{A\}) \backslash (\mathcal{X} \cup \{A\})]$. The conditions 3.a–3.c are clear by the analogy with step 2.

The set O_{N_i} computed at step 4 consists of all attributes in Y_{N_i} for which it has to be verified yet that they are indeed determined by $\mathcal{X}$. The ordering on O_{N_i} can be taken over from Y_{N_i} since, by condition 4.a, $O_{N_i} \subseteq Y_{N_i}$ holds. Likewise, the set P_{N_i} computed at step 5 consists of all attributes in Z_{N_i} for which it has to be verified yet that they are indeed determined by $\mathcal{X} \cup \{A\}$. The ordering on P_{N_i} can be taken over from P_{N_i} since, by condition 5.a, $P_{N_i} \subseteq Z_{N_i}$ holds.

The purpose of the assignment function α_{N_i} guessed in step 6 is to determine the place where the property $\mathcal{X} \to C$ has to be verified for each $C \in O_{N_i}$: as in the Subschema-Not-BCNF algorithm, 0 means the node N_i itself and $\ell \geq 1$ means the ℓ−th child of N_i. Likewise, the assignment

Algorithm Subschema-Primality-Test

Input Relational schema (R, F), tree decomposition $\mathcal{T}$ of the primal graph of (R, F) with width k, subset $R' \subseteq R$, attribute $A \in R'$.
Output "Accept", if A is prime in (R', F') with $F' = F[R']$.

Procedure sub-prime-attribute (X_N: SetOfAttributes, Y_N, Z_N, O_N, P_N: OrderedSetOfAttributes, N_i: Node in $\mathcal{T}$)
begin
1) **Guess** $X_{N_i} \subseteq \lambda(N_i) \cap R'$ s.t.:
 1.a) $\forall C \in \lambda(N) \cap \lambda(N_i)$: $C \in X_N \Leftrightarrow C \in X_{N_i}$
2) **Guess** an ordered set $Y_{N_i} \subseteq \lambda(N_i)$ s.t.:
 2.a) $X_{N_i} \cap Y_{N_i} = \emptyset$,
 2.b) $\forall C \in \lambda(N) \cap \lambda(N_i)$: $C \in Y_N \Leftrightarrow C \in Y_{N_i}$
 2.c) The orderings on Y_N and Y_{N_i} are consistent.
3) **Guess** an ordered set $Z_{N_i} \subseteq \lambda(N_i)$ s.t.:
 3.a) $X_{N_i} \cap Z_{N_i} = \emptyset$,
 3.b) $\forall C \in \lambda(N) \cap \lambda(N_i)$: $C \in Z_N \Leftrightarrow C \in Z_{N_i}$
 3.c) The orderings on Z_N and Z_{N_i} are consistent.
4) $O_{N_i} := O_N \cup (Y_{N_i} - Y_N)$;
 4.a) **if not** $O_{N_i} \subseteq Y_{N_i}$ **then HALT** and **Reject**;
5) $P_{N_i} := P_N \cup (Z_{N_i} - Z_N)$;
 5.a) **if not** $P_{N_i} \subseteq Z_{N_i}$ **then HALT** and **Reject**;
6) **Guess** an assignment $\alpha_{N_i} : O_{N_i} \to \{0, \ldots, n\}$, where n = number of child nodes of N_i.
7) **Guess** an assignment $\beta_{N_i} : P_{N_i} \to \{0, \ldots, n\}$, where n = number of child nodes of N_i.
8) check-sub-primality (X_N, Y_N, Z_N, $\alpha_N^{-1}(0)$, $\beta_N^{-1}(0)$, N_i);
9) **if** the check 8) fails then **Reject**;
10) **if for each** $\ell \in \{1, \ldots, n\}$
 sub-prime-attribute (X_{N_i}, Y_{N_i}, Z_{N_i}, $\alpha_{N_i}^{-1}(\ell)$, $\beta_{N_i}^{-1}(\ell)$, M_ℓ) = Accept, where M_ℓ is the ℓ-th child of N_i in $\mathcal{T}$
  then Accept
  else Reject;
end;

begin (* MAIN *)
 r0) (* initialization *)
  find some node N in $\mathcal{T}$ with $A \in \lambda(N)$;
  consider $\mathcal{T}$ as rooted at N;
 r1) **Guess** $X_N \subseteq \lambda(N) \cap R'$
 r2) **Guess** an ordered set $Y_N \subseteq \lambda(N)$ s.t.:
  r2.a) $X_N \cap Y_N = \emptyset$,
 r3) **Guess** an ordered set $Z_N \subseteq \lambda(N)$ s.t.:
  r3.a) $X_N \cap Z_N = \emptyset$,
 r4) $O_N := Y_N$;
 r5) $P_N := Z_N$;
 r6) **Guess** an assignment $\alpha_N : O_N \to \{0, \ldots, n\}$, where n = number of child nodes of N.
 r7) **Guess** an assignment $\beta_N : P_N \to \{0, \ldots, n\}$, where n = number of child nodes of N.
 r8) check-sub-primality (X_N, Y_N, Z_N, $\alpha_N^{-1}(0)$, $\beta_N^{-1}(0)$, N);
 r9) **if** the check 8) fails then **Reject**;
 r10) **if for each** $i \in \{1, \ldots, n\}$
  sub-prime-attribute (X_N, Y_N, Z_N, $\alpha_N^{-1}(i)$, $\beta_N^{-1}(i)$, N_i) = Accept, where N_i is the i-th child of N in $\mathcal{T}$
   then Accept
   else Reject;
end.

Fig. 5. Primality test for a Subschema.

function β_{N_i} guessed in step 7 determines the place where the property $(\mathcal{X} \cup \{A\}) \to C$ has to be verified for each $C \in P_{N_i}$.

The check-sub-primality procedure called at step 8 is displayed in the separate Fig. 6. The checks have the following tasks: Check 1a is clear. Check 1b makes sure that, ultimately, $R' \subseteq clos_F(\mathcal{X} \cup \{A\})$ holds. Check 2a guarantees that $(\mathcal{X} \cup \mathcal{Y}) \supseteq clos_F(\mathcal{X})$ holds, while check 2b ensures the property $(\mathcal{X} \cup \{A\} \cup \mathcal{Z}) \supseteq clos_F(\mathcal{X} \cup \{A\})$. Finally, check 3a ensures the property $(\mathcal{X} \cup \mathcal{Y}) \subseteq clos_F(\mathcal{X})$ while check 3b takes care of the property $(\mathcal{X} \cup \{A\} \cup \mathcal{Z}) \subseteq clos_F(\mathcal{X} \cup \{A\})$.

Step 9 is clear. Step 10 serves to control the recursive descent in the tree decomposition $\mathcal{T}$. The steps r1–r10 at the root N of $\mathcal{T}$ have exactly the same meaning as the steps 1–10 the procedure check-sub-primality. Of course, some tasks are slightly simpler since the root N has no parent. Step r0 makes sure that $A \in \lambda(N)$ holds.

In order to prove the correctness of the Subschema-Primality-Test algorithm, we have to show the following equivalence: There exists an accepting computation of the non-deterministic Subschema-Primality-Test algorithm if and only if A is prime in R'. Analogously to Lemma 3.3, the following lemma (which is crucial for the "only if"-direction of this equivalence) can be shown.

Lemma 4.1. *Consider an accepting computation of the Subschema-Primality-Test algorithm on input* (R, F), $\mathcal{T}$, R',

Procedure check-sub-primality (X_M: SetOfAttributes, Y_M, Z_M, O_M, P_M: OrderedSetOfAttributes, M: Node in $\mathcal{T}$)
begin
Check 1a: $A \notin X_M$, $A \notin Y_M$, $A \notin Z_M$.
Check 1b: $R' \cap \lambda(M) \subseteq X_M \cup \{A\} \cup Z_M$.
Check 2a: (* $X_M \cup Y_M$ closed w.r.t. F *)
$\forall f \in F$: If f is covered by $\lambda(M)$ and $lhs(f) \subseteq X_M \cup Y_M$ then $rhs(f) \in X_M \cup Y_M$.
Check 2b: (* $X_M \cup \{A\} \cup Z_M$ closed w.r.t. F *)
$\forall f \in F$: If f is covered by $\lambda(M)$ and $lhs(f) \subseteq X_M \cup \{A\} \cup Z_M$ then $rhs(f) \in X_M \cup \{A\} \cup Z_M$.
Check 3a: $\forall C \in O_M$:
$\exists(D_1, \ldots, D_m \to C) \in F$ covered by $\lambda(M)$ s.t. $\forall i \leq m$: $(D_i \in X_M) \vee (D_i \in Y_M \wedge D_i < C)$ holds.
Check 3b: $\forall C \in P_M$:
$\exists(D_1, \ldots, D_m \to C) \in F$ covered by $\lambda(M)$ s.t. $\forall i \leq m$: $(D_i \in X_M) \vee D_i = A \vee (D_i \in Z_M \wedge D_i < C)$ holds.
if all checks succeed **then Accept**
else HALT and **Reject**;
end;

Fig. 6. Checks for the primality test.

and A. Moreover, let $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ be defined as

$$\mathcal{X} := \bigcup_{N \in \mathcal{T}} X_N, \mathcal{Y} := \bigcup_{N \in \mathcal{T}} Y_N \quad \text{and} \quad \mathcal{Z} := \bigcup_{N \in \mathcal{T}} Z_N.$$

Then the following relations hold:

$$clos_F(\mathcal{X}) = \mathcal{X} \cup \mathcal{Y} \quad \text{and} \quad clos_F(\mathcal{X} \cup \{A\}) = \mathcal{X} \cup \{A\} \cup \mathcal{Z}.$$

The proof uses exactly the same arguments as the proof of Lemma 3.3 and is therefore omitted. Analogously to Theorem 3.4, we thus get the following result concerning the correctness of the Subschema-Primality-Test algorithm.

Theorem 4.2. *Let (R, F) be a relational schema and let $R' \subseteq R$ be a subschema, and $A \in R'$ an attribute. Moreover, let $\mathcal{T}$ be a tree decomposition of (R, F). The Subschema-Primality-Test algorithm in Fig. 5 accepts the input $((R, F), \mathcal{T}, R', A)$, if and only if A is a prime attribute in $(R', F[R'])$.*

For the complexity of the Subschema-Primality-Test algorithm, we get the following upper bound:

Theorem 4.3. *Let $k \geq 1$ be a fixed constant. Moreover, let (R, F) be a relational schema of treewidth $\leq k$, let $R' \subseteq R$ and $A \in R'$. Then it can be decided in linear time whether A is prime in $(R', F[R'])$. Moreover, this decision problem is in* LogCFL.

Proof. The LogCFL-membership can be shown analogously to Theorem 3.5. We only prove the linear time upper bound here, which can be seen as follows: First of all, the data structures guessed in the algorithm Subschema-Primality-Test depend only on the size k of the labels $\lambda(N)$ and $\lambda(N_i)$ rather than on the size of the entire input. In order to turn the non-deterministic algorithm into a deterministic one, we basically loop over all possible values of X_{N_i}, Y_{N_i}, Z_{N_i}, α_{N_i}, β_{N_i}, etc. However, in order to avoid doing the same computation repeatedly for some subtree, some care is required. For this purpose, we have to apply a dynamic programming (or tabling) approach, by which we store all computation results and re-use them whenever possible.

Hence, at each node of the tree decomposition $\mathcal{T}$ (whose size is of course linearly bounded), each collection of guesses of X_{N_i}, Y_{N_i}, Z_{N_i}, α_{N_i}, β_{N_i}, etc. has to be processed at most once. Moreover, almost all of the steps carried out by the main-program and by the two procedures depend on the labels $\lambda(N)$ and $\lambda(N_i)$ rather than on the size of the entire input. The only two places where one has to be careful are step 10 in the procedure sub-prime-attribute and the checks 2a and 2b in the procedure check-sub-primality, which seem to require linear time whenever they are executed. Note, however, that the overall complexity of step 10 (in all of the recursive calls of the procedure prime-attribute) corresponds to the total number of recursive calls of this procedure—which is in fact linear w.r.t. the size of the input. As far as the checks 2a and 2b in the procedure check-sub-primality are concerned, the following slight modification is required: The very purpose of the algorithm of [9] is to construct a tree decomposition $\mathcal{T}$ such that every hyperedge of the hypergraph of (R, F) (or, equivalently, every FD $f \in F$) is covered by at least one bag $\lambda(N)$. This algorithm can be easily extended such that every node N of $\mathcal{T}$ is annotated with the FDs $f \in F$ that N is meant to cover. Moreover, this annotation is done in such a way that each $f \in F$ is used in the annotation of exactly one node N of $\mathcal{T}$. Then the checks 2a and 2b of the procedure check-sub-primality can be modified in that they are only applied to those FDs f which occur in the annotation of the node M of $\mathcal{T}$. The overall complexity of all calls of the procedure check-sub-primality is thus linearly bounded. □

Finally, we sketch an algorithm "Subschema-Not-3NF", which tests whether some subschema $R' \subseteq R$ is *not* in 3NF. Suppose that we are given a relational schema (R, F), a subschema $R' \subseteq R$, and a tree decomposition $\mathcal{T}$ of (R, F), s.t. the width of $\mathcal{T}$ is bounded by some constant $k \geq 1$. Our Subschema-Not-3NF algorithm is obtained by the following modification of the Subschema-Not-BCNF algorithm from Fig. 3: When guessing the attribute A in step r0, we have to check that A is *not prime* in $(R', F[F'])$. By Theorem 4.3 and by the fact that LogCFL is closed under complement (see [13]), this check can be done by a LogCFL-algorithm. But then, since $\mathsf{L}^{\mathsf{LogCFL}}(\mathsf{LogCFL}) = \mathsf{LogCFL}$

(see [28]), the whole test that $R' \subseteq R$ is *not* in 3NF, is in LogCFL. Making once more use of the closure of LogCFL under complement, we thus get

Theorem 4.4. *Let $k \geq 1$ be a fixed constant. Moreover, let (R, F) be a relational schema of treewidth $\leq k$ and let $R' \subseteq R$ be a subschema. Then it can be decided in polynomial time whether R' is in 3NF. Moreover, this decision problem is in* LogCFL.

5. Incidence graph

In this section, we prove the fixed-parameter tractability for the six decision problems considered so far (i.e., PRIMALITY, 3NFTEST, and BCNFTEST—both for a schema and for a subschema) by considering the treewidth of the *incidence graph* of the relational schema as our problem parameter. For this purpose, we show that these six decision problems can be expressed by monadic second-order (MSO) sentences. The fixed-parameter tractability results are then an immediate consequence of Courcelle's Theorem.

Monadic second-order logic extends first-order (FO) logic by the use of *set variables* (usually denoted by upper case letters), which range over sets of domain elements. In contrast, the *individual variables* (which are usually denoted by lower case letters) range over single domain elements. Suppose that we are considering structures over some signature τ of predicate symbols. An FO-formula φ over τ has as atomic formulae either atoms with some predicate symbol from τ or equality atoms. An MSO-formula φ over τ may additionally have atoms whose predicate symbol is a monadic predicate variable. For the sake of readability, we denote such an atom usually as $a \in X$ rather than $X(a)$. Likewise, we use set operators $\subseteq$ and $\subset$ with the obvious meaning. Courcelle's Theorem can be stated as follows:

Theorem 5.1 (*Courcelle [16]*). *Let φ be a fixed MSO-sentence and $k \geq 1$ a fixed constant. Deciding whether φ holds for an input graph G (more generally, for an input structure $\mathfrak{A}$) can be done in linear time if the treewidth of the graphs (resp. of the structures) under consideration is bounded by k.*

Let (R, F) be a relational schema, and $\mathcal{H}$ be the hypergraph of (R, F) (see Definition 2.4). We denote the treewidth of the *incidence graph* of (R, F) as $tw_I(\mathcal{H})$. The schema (R, F) can be represented in a straightforward way by means of a finite structure $\mathfrak{A}(R, F)$ over the signature $\tau = \{att, fd, lh, rh\}$. The predicates in τ have the following intended meaning: $att(a)$ means that a is an attribute in R, and $fd(h)$ means that h is a functional dependency in F. Moreover, $rh(a, h)$ (resp. $lh(a, h)$) means that a occurs on the right-hand side (resp. on the left-hand side) of the FD h. Let $\mathcal{H}_{\mathfrak{A}}$ be the hypergraph of $\mathfrak{A}(R, F)$ (see Definition 2.5), It is obvious that $tw_I(\mathcal{H}) = tw_I(\mathcal{H}_{\mathfrak{A}})$.

Recall that we only consider FDs in *canonical form* here, i.e., FDs where there is only a single attribute on the right-hand side. Next we introduce the MSO formulae encoding superkey, key and some auxiliary predicates.

MSO encodings of superkey, key and some auxiliary predicates

$X \subseteq R \equiv (\forall a) a \in X \rightarrow att(a)$
$R \subseteq X \equiv (\forall a) att(a) \rightarrow a \in X$
$X \subset R \equiv X \subseteq R \wedge (\exists a)(att(a) \wedge a \notin X)$
$Closed(X) \equiv (\forall h)[fd(h) \rightarrow (\exists a)[(rh(a, h) \wedge a \in X) \vee (lh(a, h) \wedge a \notin X)]]$
$Closure(X, Y) \equiv X \subseteq Y \wedge Closed(Y) \wedge \neg(\exists Y')[Y' \subset Y \wedge X \subseteq Y' \wedge Closed(Y')]$
$SK(X, Y) \equiv X \subseteq Y \wedge (\exists Z)[Closure(X, Z) \wedge Y \subseteq Z]$
$K(X, Y) \equiv SK(X, Y) \wedge \neg(\exists X')[X' \subset X \wedge SK(X', Y)]$
$Lhs(X, h) \equiv (\forall a)[a \in X \leftrightarrow lh(a, h)]$

The predicates defined above have the following meaning:

$Closed(X)$: X is closed w.r.t. F;
$Closure(X, Y)$: Y is the closure of X w.r.t. F;
$SK(X, Y)$: X is a superkey of the schema $(Y, F[Y])$;
$K(X, Y)$: X is a key of the schema $(Y, F[Y])$;
$Lhs(X, h)$: X is the lhs of the FD h.

With the above auxiliary predicates, the following MSO encodings are straightforward.

MSO encodings of PRIMALITY, BCNF and 3NF

$Prime(a) \equiv (\exists X)[K(X, R) \wedge a \in X]$
$BCNF() \equiv (\forall h, X)[(fd(h) \wedge Lhs(X, h)) \rightarrow SK(X, R)]$
$3NF() \equiv (\forall h, X)[(fd(h) \wedge Lhs(X, h)) \rightarrow [SK(X, R) \vee (\exists b)[rh(b, h) \wedge Prime(b)]]]$

The predicates defined above have the obvious meaning:

$Prime(a)$: a is prime in the schema (R, F).
$BCNF()$: The schema (R, F) is in BCNF.
$3NF()$: The schema (R, F) is in 3NF.

Now we consider the problems PRIMALITY, BCNF, and 3NF for a subschema. Let (R, F) be the relational schema and $(R', F[R'])$ be the *subschema* of (R, F), where $R' \subseteq R$. We construct a finite structure $\mathfrak{A}'(R, F, R')$, which contains $\mathfrak{A}(R, F)$ and a new relation $att'(\cdot)$, where $att'(a)$ means that a is an attribute in R'. Let $\mathcal{H}_{\mathfrak{A}'}$ be the hypergraph of $\mathfrak{A}'(R, F, R')$. Since att' is a unary relation and so does not affect the treewidth of the incidence graph, we have $tw_I(\mathcal{H}_{\mathfrak{A}'}) = tw_I(\mathcal{H}_{\mathfrak{A}})$. Moreover, auxiliary predicates related to R' (rather than R), such as $X \subseteq R'$, can be easily constructed by replacing att with att' in the definition of the predicates related to R. Then the properties PRIMALITY, BCNF and 3NF of a subschema can be expressed via the following MSO encoding.

MSO encodings of PRIMALITY, BCNF and 3NF in a subschema

$a \in Y \cap R' \equiv a \in Y \wedge att'(a)$
$PrimeSub(a, R') \equiv (\exists X)[K(X, R') \wedge a \in X]$
$BCNFSub(R') \equiv (\forall X, Y)[[X \subseteq R' \wedge Closure(X, Y) \wedge (\exists a)(a \notin X \wedge a \in Y \cap R')]$
$\quad \rightarrow SK(X, R')]$
$3NFSub(R') \equiv (\forall X, Y, a)[[X \subseteq R' \wedge Closure(X, Y) \wedge a \notin X \wedge a \in Y \cap R']$
$\quad \rightarrow [SK(X, R') \vee PrimeSub(a, R')]]$

The predicates defined above have the obvious meaning:

$PrimeSub(a, R, F, R')$: the attribute a is prime in the subschema $(R', F[R'])$ of (R, F).
$BCNFSub(R, F, R')$: the subschema $(R', F[R'])$ of the relational schema (R, F) is in BCNF.
$3NFSub(R, F, R')$: the subschema $(R', F[R'])$ is in 3NF.

Algorithm 3NF Decomposition
Input Relational schema (R, F), tree-decomposition $\mathcal{T}$ of the primal graph of (R, F) with width k.
Output Set $\mathcal{S}_{3NF}$ of subschemas in 3NF.

begin
$\mathcal{S}_{3NF} := \emptyset$; $R' := R$; stop := false;
repeat
if R' contains a 3NF-violation $A_1, \ldots, A_m \rightarrow B$ **then**
choose $\{A_{i_1}, \ldots, A_{i_k}\} \subseteq \{A_1, \ldots, A_m\}$ minimal with this property;
$\mathcal{S}_{3NF} := \mathcal{S}_{3NF} \cup \{\{A_{i_1}, \ldots, A_{i_k}, B\}\}$;
$R' := R' \setminus \{B\}$;
else
$\mathcal{S}_{3NF} := \mathcal{S}_{3NF} \cup \{R'\}$;
stop := true;
fi;
until stop;
end;

Fig. 7. 3NF-Decomposition algorithm.

Since all the decision problems considered in this paper can be expressed by means of MSO sentences, the following fixed-parameter tractability result is an immediate consequence of Courcelle's Theorem (Theorem 5.1).

Theorem 5.2. *The decision problems PRIMALITY, 3NFTEST, and BCNFTEST (both for a schema and for a subschema) can be solved in linear time, if the treewidth of the incidence graph of the relational schemas (R, F) under consideration is bounded by a constant.*

6. NF-decomposition revisited

The intractability of the decision problems recalled in the introduction is a severe obstacle to satisfactory normal form decomposition algorithms for 3NF and BCNF. In particular, as was illustrated in Example 1.1, without the ability to recognize the normal form, one inevitably runs the risk of further decomposing a schema even though the desired normal form has already been reached. However, in many situations, the relational schema under investigation has low treewidth. In this case, our algorithms presented in the previous sections open the grounds for a completely new approach to normal form decomposition. Rather than starting from the FDs and defining subschemas bottom-up so to speak, one can start from the schema itself, check for normal form violations and define appropriate subschemas top-down. In this section we present a simple 3NF decomposition algorithm based on this new approach, see Fig. 7.

The 3NF-Decomposition algorithm obviously computes a lossless join decomposition into 3NF subschemas. Moreover, by Theorem 4.4, it works in polynomial time (note that our NF-algorithms from Sections 3 and 4 can be easily extended so as to output a concrete NF-violation rather than just answering "Accept" or "Reject"). Of course, there is ample space for improvements and extensions of this algorithm. We conclude this section by outlining just a few directions for further refining this algorithm and for extending it to a BCNF decomposition algorithm:

(1) One is normally interested in an *FD-preserving* decomposition. Hence, as a post-processing step, one should check for all FDs in F whether they are embedded in the resulting set of subschemas. For every FD $A_1, \ldots, A_m \rightarrow B$ not embedded in $\mathcal{S}_{3NF}$, we can simply add the subschema $\{A_1, \ldots, A_m, B\}$ to $\mathcal{S}_{3NF}$. All this can be done in polynomial time (see [4]).
(2) When a 3NF-violation $A_1, \ldots, A_m \rightarrow B$ has been detected, one should not only split off the single attribute B from R'. Instead, one should search for further attributes $B_1, \ldots, B_\ell$ which are also determined by $A_1, \ldots, A_m$ and split off all these attributes together. Actually, the resulting subschema $S = \{A_1, \ldots, A_m, B, B_1, \ldots, B_\ell\}$ is not necessarily in 3NF. But this is no problem. We just have to check whether the subschema S already is in 3NF and, otherwise, apply the 3NF decomposition recursively to this subschema. By Theorem 4.4, also the 3NF-test in a subschema S of (R, F) can be done efficiently in case of bounded treewidth.
(3) Similarly, when we use the idea of the 3NF-Decomposition algorithm for a BCNF decomposition, then the subschemas produced are not necessarily in BCNF. But again, this can be efficiently detected (see Theorem 3.5) and we just have to apply the BCNF decomposition recursively to the subschema. Unfortunately, there is no way to guarantee that we actually find an *FD-preserving* decomposition into BCNF. However, by results shown in [3] (a lossless join, FD-preserving BCNF decomposition does not necessarily exist and it is coNP-hard to decide whether one exists), this can hardly be helped.

7. Logic-based abduction

In this section, we show how our new algorithms and fixed-parameter tractability results can be carried over

from our database design problems to an important class of problems in artificial intelligence, namely the relevance problem of logic-based abduction.

Abductive diagnosis aims at an explanation of some observed symptoms in terms of minimal sets of hypotheses (like failing components) which may have led to these symptoms [17]. Unfortunately, most of the decision problems in logic-based abduction are intractable [19]. For instance, the *relevance* problem (i.e., deciding if a given hypothesis is part of a possible explanation) is NP hard, even if the system description consists of propositional, definite Horn clauses only [23]. Hence, it is an important task to search for sufficient conditions under which these practically important problems become efficiently solvable.

By the close relationship with the PRIMALITY problem, we make our new algorithms also applicable to the relevance problem of propositional abduction. In particular, we show that the relevance problem becomes tractable if the system description is given by a set of propositional, definite Horn clauses with bounded treewidth.

Moreover, in this paper, we also present an extension of these results to abductive diagnosis via datalog. It is thus possible to actually tackle many real-world problems in AI with these methods. Suppose that a system description (e.g., of an electronic circuit) is given in form of a finite structure (i.e., the "extensional database", EDB) of bounded treewidth and a guarded *non-ground* datalog program that describes the propagation of faulty behavior. We will show that, also in this case, it can be determined efficiently whether the misbehavior of some system component is a possible cause for the observed symptoms. It should be noted that this extension to the non-ground case is by no means obvious, notwithstanding the clear analogy between the functional dependencies in database design and *propositional* abduction. The difficulty in proving our new result consists in showing that the abductive datalog problem is equivalent to a propositional abduction problem *and* the bounded treewidth of the EDB indeed propagates to the propositional rules.

7.1. Propositional abduction

In this work, we study *propositional abduction problems* (PAPs) of the following form.

Definition 7.1. A *propositional abduction problem* (a PAP, for short) $\mathcal{P}$ consists of a tuple $\langle V, Hyp, Obs, SD\rangle$, where V is a finite set of *variables*, $Hyp \subseteq V$ is the set of *hypotheses*, $Obs \subseteq V$ is the set of *observed symptoms*, and SD is a *system description* in the form of a set of definite, propositional Horn clauses with $SD \cup Hyp \models Obs$. Moreover, we assume that $|Obs|$ is bounded by some fixed constant k.

A set $\Delta \subseteq Hyp$ is a *diagnosis* (also called *solution*) to $\mathcal{P}$ if Δ is minimal s.t. $SD \cup \Delta \models Obs$ holds. A hypothesis $h \in Hyp$ is called *relevant* if h is contained in at least one diagnosis Δ of $\mathcal{P}$.

Note that the restriction on the cardinality of Obs is not a severe one since $|Obs|$ is often even assumed to be 1 (e.g., an alarm bell ringing, a bulb lighting up, etc.). The condition $SD \cup Hyp \models Obs$ is a prerequisite to ensure that there exists at least one diagnosis. This condition can be checked in linear time, see [18,45].

As was already mentioned above, deciding the relevance of a hypothesis h in a PAP $\mathcal{P}$ is NP-complete even with the above restrictions on SD, Hyp, and Obs [23].

Example 7.2. Consider the following PAP describing problems of a football team [35].

$$SD = \{weak\_defense \vee weak\_attack \rightarrow match\_lost, match\_lost \rightarrow manager\_sad \wedge press\_angrystar\_injured \rightarrow manager\_sad \wedge press\_sad\}$$
$$Obs = \{manager\_sad, press\_angry\}$$
$$Hyp = \{weak\_defense, weak\_attack, star\_injured\}.$$

It is convenient to abbreviate the propositional variables weak_defense, weak_attack, star_injured, match_lost, manager_sad, press_angry, and press_sad in this order as A_1, A_2, A_3, A_4, A_5, A_6, A_7. Obviously, SD is equivalent to the following set of definite Horn clauses:

$$SD' = \{A_4 \leftarrow A_1, A_4 \leftarrow A_2, A_5 \leftarrow A_4, A_6 \leftarrow A_4, A_5 \leftarrow A_3, A_7 \leftarrow A_3\}.$$

This PAP has two diagnoses, $\Delta_1 = \{A_1\}$ and $\Delta_2 = \{A_2\}$ (i.e., weak_defense and weak_attack, respectively).

Concerning the treewidth of a PAP, we proceed analogously to the treewidth of a relational schema in Definition 2.4.

Definition 7.3. For a PAP $\mathcal{P} = \langle V, Hyp, Obs, SD\rangle$, we define the *hypergraph* of $\mathcal{P}$ as $\mathcal{H} = \langle V, H\rangle$ with

$$H = \{\{A_1, \ldots, A_n, B\} \mid (B \leftarrow A_1 \ldots A_n) \in SD\}.$$

The primal graph and the incidence graph of $\mathcal{P}$ are simply defined as the corresponding graph of $\mathcal{H}$. Likewise, the *treewidth* of $\mathcal{P}$ is defined as $tw_P(\mathcal{H})$ or as $tw_I(\mathcal{H})$, respectively.

Recall that outside Section 5, we only consider the treewidth of the primal graph unless explicitly stated otherwise. Hence, we shall simply speak about the treewidth (or a tree decomposition, resp.) below in order to refer to $tw_P(\mathcal{H})$ (or a tree decomposition of the primal graph, resp.).

We are now ready to show that one may indeed carry over the fixed-parameter linearity and the LogCFL-membership from PRIMALITY to propositional abduction.

Theorem 7.4. *Let $k \geq 1$ be a fixed constant. Moreover, let $\mathcal{P} = \langle V, Hyp, Obs, SD\rangle$ be a PAP where SD has treewidth $\leq k$ and let $h \in Hyp$. It can be decided in linear time whether h is relevant in $\mathcal{P}$. Moreover, this decision problem is in* LogCFL.

Proof. By Theorem 4.3, it suffices to show that there is a linear-time, log-space reduction from the relevance problem to the PRIMALITY problem in a subschema s.t. the increase of the treewidth is bounded by a constant. By slight abuse of notation, we identify any definite Horn rule $B \leftarrow A_1, \ldots, A_n$ with the FD $A_1, \ldots, A_n \rightarrow B$. Then we reduce an arbitrary PAP $\mathcal{P} = \langle V, Hyp, Obs, SD\rangle$ to the relational schema (R, F) with $R = V$ and $F = SD \cup \{Obs \rightarrow B \mid B \in Hyp\}$. Moreover, we set $R' = Hyp$.

This reduction is clearly feasible in log-space. Moreover, we can obtain a tree decomposition of F from a tree decomposition of SD by adding Obs to every bag. Since we are only considering PAPs with $|Obs| \leq k$ for some constant k, we have $tw(F) \leq tw(SD) + k$. It remains to prove the correctness of this reduction, i.e. $h \in Hyp$ is relevant in the PAP $\mathcal{P}$ iff h is prime in the subschema $(R', F[R'])$. It suffices to show that, for every $\Delta \subseteq Hyp$, the following equivalence holds:

Δ is a diagnosis of $\mathcal{P} \Leftrightarrow \Delta$ is a key of $(R', F[R'])$.

Before we prove the two directions of this equivalence, we comment on the notation used below. Recall that we are identifying the set of definite Horn rules SD with the set of functional dependencies SD and vice versa. Hence, we shall write $SD \cup \Delta \models A$ in order to denote that the propositional atom A is implied by the propositional logic program $SD \cup \Delta$. Alternatively, we shall write $SD \models \Delta \rightarrow A$ in order to denote that the FD $\Delta \rightarrow A$ can be derived from the FDs SD. It is easy to check that these two conditions are equivalent.

"$\Rightarrow$" Suppose that $\Delta \subseteq Hyp$ is a solution of the PAP $\mathcal{P}$. We show that then (i) Δ is a superkey in $(R', F[R'])$ and (ii) Δ is minimal with this property.

(i) Since Δ is a solution of $\mathcal{P}$, we have $SD \cup \Delta \models A$ for every atom $A \in Obs$. Hence, in (R, F), the FD $\Delta \rightarrow A$ can be derived for every attribute $A \in Obs$. Thus, by the additional FDs $Obs \rightarrow B$, we can derive in (R, F) also the FD $\Delta \rightarrow B$ for every $B \in Hyp$. Note that $\Delta \subseteq Hyp$ and $B \in Hyp$. Hence, we actually have $F[R'] \models \Delta \rightarrow B$ for all $B \in Hyp$. Thus, Δ is a superkey in $(R', F[R'])$.

(ii) We prove the minimality of Δ indirectly. Suppose to the contrary that there exists a strictly smaller key $\Delta' \subset \Delta$ of $(R', F[R'])$. Exactly as in part (i) of the "$\Leftarrow$" − direction treated below, one can show that then $SD \cup \Delta' \models Obs$ holds, contradicting the minimality of the solution Δ of $\mathcal{P}$.

"$\Leftarrow$" Suppose that $\Delta \subseteq Hyp$ is a key of $(R', F[R'])$. We show that then (i) $SD \cup \Delta \models Obs$ and (ii) Δ is minimal with this property.

(i) By assumption, $F[R'] \models \Delta \rightarrow B$ for all $B \in Hyp$. First suppose that for at least one B, the rule $Obs \rightarrow B$ is used in the derivation of B from Δ. This means that all FDs $\Delta \rightarrow A$ for all $A \in Obs$ can be derived from SD. In terms of the PAP $\mathcal{P}$, we thus have the desired implication $SD \cup \Delta \models Obs$. On the other hand, suppose that all of the attributes B can be derived from Δ by only using the FDs in SD, i.e., $SD \models \Delta \rightarrow B$ for every $B \in Hyp$ or, equivalently, $SD \cup \Delta \models Hyp$. Recall from Definition 7.1 that, in any PAP, the condition $SD \cup Hyp \models Obs$ holds. Hence, in total, we have the desired implication $SD \cup \Delta \models Obs$.

(ii) It remains to show the minimality of Δ. Suppose to the contrary that there exists a strictly smaller solution $\Delta' \subset \Delta$ of the PAP $\mathcal{P}$. Exactly as in part (i) of the "$\Rightarrow$" − direction, one can show that then Δ' is also a superkey of $(R', F[R'])$, which contradicts the minimality of the key Δ of $(R', F[R'])$. □

Note that the above proof can also be applied to the situation where the treewidth of the *incidence graphs* is bounded by a constant. Together with Theorem 5.2, we thus obtain the corresponding fixed-parameter linearity for propositional abduction. Alternatively, one can of course proceed analogously to Section 5 and provide directly an MSO-encoding of the relevance problem of propositional abduction (see [30] for related results).

Example 7.5. Let us revisit the PAP $\mathcal{P}$ from Example 7.2. By the proof of Theorem 7.4, we can reduce $\mathcal{P}$ to the relational schema (R, F) with $R = V = \{A_1, \ldots, A_7\}$ and

$$F = \{A_1 \rightarrow A_4, A_2 \rightarrow A_4, A_4 \rightarrow A_5, A_4 \rightarrow A_6, A_3 \rightarrow A_5, A_3 \rightarrow A_7, A_5A_6 \rightarrow A_1, A_5A_6 \rightarrow A_2, A_5A_6 \rightarrow A_3\}.$$

Moreover, let $R' = \{A_1, A_2, A_3\}$. The subschema $(R', F[F'])$ has two keys, namely $K_1 = \{A_1\}$ and $K_2 = \{A_2\}$, which correspond to the diagnoses of the PAP $\mathcal{P}$.

7.2. Datalog abduction

In recent years, the datalog language has been successfully applied as a knowledge representation mechanism in the area of abductive diagnosis [2,39,12]. In this paper, we will restrict our attention to the so-called guarded fragment of datalog:

Definition 7.6 (*Gottlob et al. [24]*). A *guard* of a datalog rule is an atom A whose predicate symbol occurs in the input structure (i.e., the "EDB") s.t. all variables of the rule occur in A. A datalog rule is *guarded* if its body contains a guard. A guarded datalog program is a datalog problem whose rules are guarded.

Then we define datalog abduction as follows.

Definition 7.7. A *datalog abduction problem* consists of a tuple $\mathcal{P} = \langle EDB, P, Hyp, Obs \rangle$, where EDB is an input structure (i.e., set of ground atoms), P is a set of guarded, definite Horn datalog rules, Hyp and Obs are sets of ground atoms with $EDB \cup P \cup Hyp \models Obs$. As in the propositional case, we assume $|Obs| \leq k$ for some constant k. Moreover, we consider the set of predicate symbols occurring in $\mathcal{P}$ as arbitrarily chosen but fixed.

A *diagnosis* is a minimal subset $\Delta \subseteq Hyp$ with $EDB \cup P \cup \Delta \models Obs$. An atom $h \in Hyp$ is called *relevant*, if there exists at least one diagnosis Δ with $h \in \Delta$.

Note that in a datalog abduction problem, the system description consists of an input structure (the EDB) and a (normally non-ground) datalog program P. A PAP as defined in the previous section corresponds to the special case where the EDB is omitted and P contains only ground rules.

Example 7.8. Consider the full-adder in Fig. 8, which has faulty output bits (indicated by *). We describe this diagnosis problem by a datalog abduction problem:

$$EDB = \{\mathsf{one}(a), \mathsf{zero}(b), \mathsf{one}(c), \mathsf{xor}(a, b, s, xor_1), \mathsf{xor}(s, c, sum, xor_2), \mathsf{and}(a, b, c_1, and_1), \mathsf{and}(s, c, c_2, and_2), \mathsf{or}(c_1, c_2, carry, or_1)\}.$$

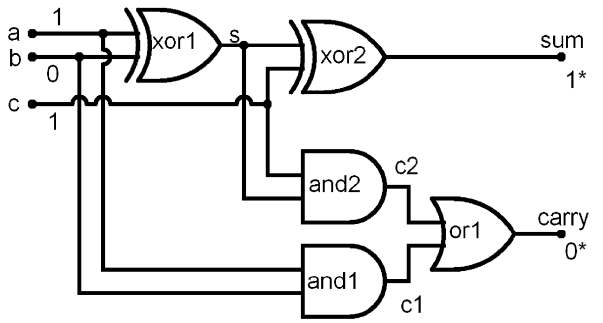

Fig. 8. Full-adder with incorrect output values.

The guarded datalog program P contains rules that model the *normal* and the *faulty* behavior for each gate type (i.e., and, or, and xor). We only show the datalog rules for the gate type xor. The other types are handled analogously. For the normal behavior, we have the folowing four rules:

$$\mathsf{zero}(O) \leftarrow \mathsf{xor}(I_1, I_2, O, G), \mathsf{one}(I_1), \mathsf{one}(I_2),$$
$$\mathsf{one}(O) \leftarrow \mathsf{xor}(I_1, I_2, O, G), \mathsf{one}(I_1), \mathsf{zero}(I_2),$$
$$\mathsf{one}(O) \leftarrow \mathsf{xor}(I_1, I_2, O, G), \mathsf{zero}(I_1), \mathsf{one}(I_2),$$
$$\mathsf{zero}(O) \leftarrow \mathsf{xor}(I_1, I_2, O, G), \mathsf{zero}(I_1), \mathsf{zero}(I_2),$$

where the atom $\mathsf{xor}(I_1, I_2, O, G)$ is the *guard* in all rules. The faulty behavior is modeled by another collection of four rules:

$$\mathsf{one}(O) \leftarrow \mathsf{xor}(I_1, I_2, O, G), \mathsf{one}(I_1), \mathsf{one}(I_2), \mathsf{faulty}(G),$$
$$\mathsf{zero}(O) \leftarrow \mathsf{xor}(I_1, I_2, O, G), \mathsf{one}(I_1), \mathsf{zero}(I_2), \mathsf{faulty}(G),$$
$$\mathsf{zero}(O) \leftarrow \mathsf{xor}(I_1, I_2, O, G), \mathsf{zero}(I_1), \mathsf{one}(I_2), \mathsf{faulty}(G)$$
$$\mathsf{one}(O) \leftarrow \mathsf{xor}(I_1, I_2, O, G), \mathsf{zero}(I_1), \mathsf{zero}(I_2), \mathsf{faulty}(G).$$

Finally, we define the sets *Obs* and *Hyp* as follows:

$Obs := \{\mathsf{one}(sum), \mathsf{zero}(carry)\}$ and
$Hyp := \{\mathsf{faulty}(xor_1), \mathsf{faulty}(xor_2), \mathsf{faulty}(and_1), \mathsf{faulty}(and_2), \mathsf{faulty}(or_1)\}$.

Of course, if one has already verified that some component c works properly, then one will remove the atom $\mathsf{faulty}(c)$ from *Hyp*.

This abduction problem has three diagnoses, namely $\Delta_1 = \{\mathsf{faulty}(xor_1)\}$, $\Delta_2 = \{\mathsf{faulty}(xor_2), \mathsf{faulty}(or_1)\}$, and $\Delta_3 = \{\mathsf{faulty}(xor_2), \mathsf{faulty}(and_2)\}$.

As in the propositional case, we concentrate on the relevance problem. Note that the knowledge as to whether the malfunction of a given component is part of a possible explanation of the observed symptoms (i.e., this component is *relevant*) is of big practical value. For instance, consider executing the diagnosis of an electronic circuit with thousands of gates. The distinction between relevant and irrelevant components can help to drastically reduce the effort for the diagnosis task.

We now extend the fixed-parameter tractability result of the relevance problem from propositional abduction to datalog abduction.

Theorem 7.9. *Let $k \geq 1$ be a fixed constant. Moreover, let $\mathcal{P} = \langle EDB, P, Hyp, Obs \rangle$ be a datalog abduction problem, s.t. the input structure EDB has treewidth (of the primal graph) $\leq k$. Then, for any $h \in Hyp$, it can be decided in polynomial time whether h is relevant in $\mathcal{P}$.*

Proof. By Theorem 7.4, it suffices to show the following two facts: First we have to show that the datalog program P is equivalent to a ground program of polynomial size, and second we have to show that the bounded treewidth propagates from the input structure to the ground program.

1. *Size of an equivalent ground program.* The grounding of a datalog program Π relatively to some structure $\mathfrak{A}$ is obtained by computing all possible instantiations of the variables occurring in Π by all constants in the active domain [14]. The resulting program $ground(\Pi)$ is equivalent to Π, i.e., for any atom A, we have $\Pi \cup \mathfrak{A} \models A \Leftrightarrow ground(\Pi) \cup \mathfrak{A} \models A$. In general, the program $ground(\Pi)$ is exponentially big. However, if Π is a guarded datalog program, then one can restrict $ground(\Pi)$ to an equivalent set $ground'(\Pi)$ which can be computed in quadratic time and whose size is also quadratically bounded, namely $O(|\mathfrak{A}| \cdot |\Pi|)$ [24]. Intuitively, this can be seen as follows: For each rule r in Π, there are at most $|\mathfrak{A}|$ instantiations for the guard of r which actually exist in $\mathfrak{A}$. On the other hand, all ground rules where the guard is instantiated to an atom outside $\mathfrak{A}$ can be simply deleted (because this body will never be true).
2. *Bounded treewidth of the ground program.* By assumption, there exists a tree decomposition $\mathcal{T}$ of *EDB* of width $< k$ for some constant k. Note that the bags $\lambda(N)$ in $\mathcal{T}$ are sets of domain elements. It is convenient to denote the domain elements in a ground atom A as $dom(A)$. Moreover, we refer to the ground atoms obtained by instantiating a guard atom as "ground guards". For every ground guard A, we may assume w.l.o.g., that $\mathcal{T}$ has a leaf node N s.t. $\lambda(N) = dom(A)$, since we may clearly append a new leaf node N with this property to a node N' with $\lambda(N) \supseteq dom(A)$. Then we construct a tree decomposition $\mathcal{T}'$ of the ground program $ground'(P)$ as follows.

$\mathcal{T}'$ has the same tree structure as $\mathcal{T}$. Note that now the bags $\lambda'(N)$ consist of ground atoms occurring in $ground'(P)$. Let N be a leaf node with $\lambda(N) = dom(A)$ for some ground guard A. Then we insert into $\lambda'(N)$ the atom A plus all ground atoms B that occur in at least one rule $r \in ground'(P)$, s.t. A is the ground guard of r. By construction, we have $\lambda(N) = dom(A)$ and $dom(B) \subseteq dom(A)$. Hence, $dom(B) \subseteq \lambda(N)$ holds as well.

Suppose that now a ground atom B occurs in two distinct bags $\lambda'(N_1)$ and $\lambda'(N_2)$. In order to preserve the connectedness condition, we thus also have to add B to any bag $\lambda'(M)$ on the path from N_1 to N_2. As we have argued above, we know that $dom(B) \subseteq \lambda(N_1)$ and $dom(B) \subseteq \lambda(N_2)$ holds. Thus, by the connectedness condition on the tree decomposition $\mathcal{T}$, we may conclude that also $dom(B) \subseteq \lambda(M)$ holds for all nodes M on the path from N_1 to N_2. In other words, the following implication holds for every ground atom B: If $B \in \lambda'(N)$ then $dom(B) \subseteq \lambda(N)$. Hence, as a (very rough) upper bound on the width of $\mathcal{T}'$,

we get $m * k^l$, where m is the number of predicate symbols in *EDB* and l is their maximum arity. □

We conclude this section with some heuristics which will normally lead to a much more efficient reduction from datalog abduction to propositional abduction (even though the worst-case complexity from the proof of Theorem 7.9 is not affected).

1. *Further simplification of ground′(P).* In the proof of Theorem 7.9 we were contented with deleting all ground rules where the ground guard is not contained in the EDB. Of course, this idea can be extended to any ground atoms with an EDB-predicate symbol. We thus apply the following simplification. Let r be a ground rule in *ground′(P)* and let A be an atom occurring in the body of r. Moreover suppose that A has a predicate symbol from the EDB. Then we may apply the following simplifications: If $A \in EDB$, then A may be deleted from the rule (since it is clearly true). If $A \notin EDB$ then r may be deleted from *ground′(P)* (since the body of r will never be true).
2. *Simplification of the tree decomposition $\mathcal{T}'$.* In the proof of Theorem 7.9 we started off with a tree decomposition $\mathcal{T}$ of *EDB* and constructed a tree decomposition $\mathcal{T}'$ of the ground logic program *ground′(P)*. A close inspection of the proof reveals that we could have started off with a tree decomposition of the subset $EDB' \subseteq EDB$ with

   $EDB' = \{A \in EDB | A \text{ is a ground guard}\}$

   Moreover, the tree decomposition will in general become much simpler if we first apply the above simplifications to *ground′(P)*.

Example 7.10. Consider again the PAP $\mathcal{P}$ from Example 7.8. If we first ground P and then apply the above simplifications, then we get the following ground program (which is equivalent to $EDB \cup P$):

one(s),	zero(s) ← faulty(xor_1),
zero(c_1),	one(c_1) ← faulty(and_1),
one(sum) ← zero(s),	one(sum) ← one(s), faulty(xor_2),
zero(sum) ← one(s),	zero(sum) ← zero(s), faulty(xor_2),
zero(c_2) ← zero(s),	zero(c_2) ← one(s), faulty(and_2),
one(c_2) ← one(s),	one(c_2) ← zero(s), faulty(and_2),
one($carry$) ← one(c_1), zero(c_2),	zero($carry$) ← one(c_1), zero(c_2), faulty(or_1),
one($carry$) ← zero(c_1), one(c_2),	zero($carry$) ← zero(c_1), one(c_2), faulty(or_1),
one($carry$) ← one(c_1), one(c_2),	zero($carry$) ← one(c_1), one(c_2), faulty(or_1),
zero($carry$) ← zero(c_1), zero(c_2),	one($carry$) ← zero(c_1), zero(c_2), faulty(or_1).

As we have seen in the proof of Theorem 7.9, a guarded datalog program Π is very close to a ground program. Hence, for a given extensional database *EDB*, one could right from the beginning specify the system description by a ground program, whose size is at most $|EDB| \cdot |\Pi|$. Nevertheless, we consider guarded datalog as a useful tool from a modeling point of view: For instance, in the diagnosis example of the Boolean circuit in Example 7.8, the system description in guarded datalog (with a fixed set of rules per gate type, e.g., eight rules in total for the gate type xor) seems much clearer than if we considered the corresponding ground program. In particular, we thus achieve a separation between the general behavior of a problem class (specified by the datalog rules which hold for any circuit and have to be set up "once and for all") and the concrete Boolean circuit (given by the extensional database). In this section, we have shown that the separation between general behavior and concrete problem instances can also be maintained for identifying fixed-parameter tractable fragments: Indeed, by Theorem 7.9, it suffices to inspect the treewidth of the extensional databases under consideration for identifying fixed-parameter tractability.

8. Conclusion

In this paper, we have presented new algorithms for six fundamental decision problems in database design, namely PRIMALITY, 3NFTEST, and BCNFTEST (both for a schema and for a subschema). These algorithms work in polynomial time in case that the input relational schema has bounded treewidth. To the best of our knowledge, these are the first results on tractable fragments of decision problems in database design via bounded treewidth and we are planning to further investigate the potential benefit of the concept of treewidth in this area.

The algorithms presented in this work are based on the definition of treewidth via the primal graph of a hypergraph $\mathcal{H}$. For the case that the treewidth is defined via the incidence graph, we have proved the corresponding fixed-parameter tractability results via appropriate MSO encodings. Dedicated algorithms also in the latter case are left for future work. Likewise, applying other notions of treewidth (like directed treewidth, see e.g. [7,6,47,36]) to the problems studied here is an interesting target for future research.

It would be tempting to try to identify tractable fragments of these problems directly via the hypergraph of relational schemas (rather than via the primal or incidence graph corresponding to the hypergraph). The first candidate for such investigations would be the notion of hypertree-width which has been successfully used to identify tractable fragments of conjunctive query evaluation and of constraint satisfaction problems [26,29]. However, the hypertree-width does not seem to be applicable in our context. The reason for this is that the hypertree-width of a hypergraph becomes low in the presence of big hyperedges. In the extreme case, any hypergraph $\mathcal{H}$ can be transformed into a hypergraph $\mathcal{H}'$ of hypertree-width 1 by adding a hyperedge which contains all vertices of $\mathcal{H}$. For the hypergraph corresponding to a relational schema (R, F), this effect can be easily achieved by adding the trivial FD $R \rightarrow A$ for some attribute A. Of course, this syntactic trick does not make any of the decision problems considered here any easier.

In this paper, we have also pointed out that these results can be carried over to propositional abduction and further extended to abductive datalog. Again, we restricted ourselves to the treewidth defined via the primal graph of a hypergraph $\mathcal{H}$. In a recent paper (see [30]) we

proved many more fixed-parameter tractability results in the area of non-monotonic reasoning and knowledge representation for bounded treewidth of the incidence graph. These results were shown—similarly to Section 5—via appropriate MSO encodings by making use of Courcelle's Theorem. As mentioned above, dedicated algorithm for these problems would be much more desirable than the MSO-encoding. The search for such dedicated algorithm is left for future work in this area.

Acknowledgements

We are grateful to the anonymous reviewers who helped substantially improve the paper.

References

[1] W. Armstrong, Dependency structures of data base relationships, in: IFIP Congress, 1974, pp. 580–583.
[2] J. Balsa, V. Dahl, J.P. Lopes, Datalog grammars for abductive syntactic error diagnosis and repair, in: Proceedings of the NLULP'95, International Workshop on Natural Language Understanding and Logic Programming, 1995.
[3] C. Beeri, P.A. Bernstein, Computational problems related to the design of normal form relational schemas, ACM Trans. Database Syst. 4 (1) (1979) 30–59.
[4] C. Beeri, P. Honeyman, Preserving functional dependencies, SIAM J. Comput. 10 (3) (1981) 647–656.
[5] P.A. Bernstein, Synthesizing third normal form relations from functional dependencies, ACM Trans. Database Syst. 1 (4) (1976) 277–298.
[6] D. Berwanger, A. Dawar, P. Hunter, S. Kreutzer, DAG-width and parity games, in: Proceedings of the STACS'06, Lecture Notes in Computer Science, vol. 3884, Springer, Berlin, 2006. pp. 524–536.
[7] D. Berwanger, E. Grädel, Entanglement—a measure for the complexity of directed graphs with applications to logic and games, in: Proceedings of the LPAR'04, Lecture Notes in Computer Science, vol. 3452, Springer, Berlin, 2005, pp. 209–223.
[8] H.L. Bodlaender, A tourist guide through treewidth, Acta Cybern. 11 (1–2) (1993) 1–22.
[9] H.L. Bodlaender, A linear-time algorithm for finding tree-decompositions of small treewidth, SIAM J. Comput. 25 (6) (1996) 1305–1317.
[10] H.L. Bodlaender, A.M.C.A. Koster, Safe separators for treewidth, Discrete Math. 306 (3) (2006) 337–350.
[11] H.L. Bodlaender, A.M.C.A. Koster, Combinatorial optimization on graphs of bounded treewidth, Comput. J. 51 (3) (2008) 255–269.
[12] P. Bonatti, P. Samarati, Regulating service access and information release on the web, in: Proceedings of the CCS '00: ACM conference on Computer and Communications Security, ACM Press, New York, 2000, pp. 134–143.
[13] A. Borodin, S.A. Cook, P.W. Dymond, W.L. Ruzzo, M. Tompa, Two applications of inductive counting for complementation problems, SIAM J. Comput. 18 (3) (1989) 559–578.
[14] S. Ceri, G. Gottlob, L. Tanca, Logic Programming and Databases, Springer, New York, USA, 1990.
[15] A.K. Chandra, D. Kozen, L.J. Stockmeyer, Alternation, J. ACM 28 (1) (1981) 114–133.
[16] B. Courcelle, Graph rewriting: an algebraic and logic approach, in: Handbook of Theoretical Computer Science, vol. B, Elsevier Science Publishers, 1990, pp. 193–242.
[17] J. de Kleer, A.K. Mackworth, R. Reiter, Characterizing diagnoses and systems, Artif. Intell. 56 (2–3) (1992) 197–222.
[18] W.F. Dowling, J.H. Gallier, Linear-time algorithms for testing the satisfiability of propositional Horn formulae, J. Log. Program. 1 (3) (1984) 267–284.
[19] T. Eiter, G. Gottlob, The complexity of logic-based abduction, J. ACM 42 (1) (1995) 3–42.
[20] P.C. Fischer, J.H. Jou, D.-M. Tsou, Succinctness in dependency systems, Theor. Comput. Sci. 24 (1983) 323–329.
[21] J. Flum, M. Frick, M. Grohe, Query evaluation via tree-decompositions, J. ACM 49 (6) (2002) 716–752.
[22] M. Frick, M. Grohe, The complexity of first-order and monadic second-order logic revisited, in: Proceedings of the LICS'02, IEEE Computer Society, 2002, pp. 215–224.
[23] G. Friedrich, G. Gottlob, W. Nejdl, Hypothesis classification, abductive diagnosis and therapy, in: Proceedings of the International Workshop on Expert Systems in Engineering: Principles and Applications, Lecture Notes in Computer Science, vol. 462, Springer, Berlin, 1990, pp. 69–78.
[24] G. Gottlob, E. Grädel, H. Veith, Datalog lite: a deductive query language with linear time model checking, ACM Trans. Comput. Logic 3 (1) (2002) 42–79.
[25] G. Gottlob, C. Koch, R. Pichler, L. Segoufin, The complexity of XPath query evaluation and XML typing, J. ACM 52 (2) (2005) 284–335.
[26] G. Gottlob, N. Leone, F. Scarcello, A comparison of structural CSP decomposition methods, Artif. Intell. 124 (2) (2000) 243–282.
[27] G. Gottlob, N. Leone, F. Scarcello, The complexity of acyclic conjunctive queries, J. ACM 48 (3) (2001) 431–498.
[28] G. Gottlob, N. Leone, F. Scarcello, Computing LOGCFL certificates, Theor. Comput. Sci. 270 (1–2) (2002) 761–777.
[29] G. Gottlob, N. Leone, F. Scarcello, Hypertree decompositions and tractable queries, J. Comput. Syst. Sci. 64 (3) (2002) 579–627.
[30] G. Gottlob, R. Pichler, F. Wei, Bounded treewidth as a key to tractability of knowledge representation and reasoning, in: Proceedings of the AAAI'06, AAAI Press, New York, 2006, pp. 250–256.
[31] G. Gottlob, R. Pichler, F. Wei, Tractable database design through bounded treewidth, in: Proceedings of the PODS'06, ACM Press, New York, 2006, pp. 124–133.
[32] G. Gottlob, R. Pichler, F. Wei, Efficient datalog abduction through bounded treewidth, in: Proceeings of the AAAI'07, AAAI Press, New York, 2007, pp. 1626–1631.
[33] S.A. Greibach, The hardest context-free language, SIAM J. Comput. 2 (4) (1973) 304–310.
[34] M. Grohe, Descriptive and parameterized complexity, in: Proceedigns of the CSL'99, Lecture Notes in Computer Science, vol. 1683, Springer, Berlin, 1999, pp. 14–31.
[35] M. Hermann, R. Pichler, Counting complexity of propositional abduction, in: Proceedings of the IJCAI-07, 2007, pp. 417–422.
[36] P. Hunter, S. Kreutzer, Digraph measures: Kelly decompositions, games, and ordering, Theor. Comput. Sci. 399 (2008) 206–219.
[37] D.S. Johnson, A catalog of complexity classes, in: Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A), Elsevier Science Publishers B.V., North-Holland, 1990, pp. 67–161.
[38] R.M. Karp, V. Ramachandran, Parallel algorithms for shared-memory machines, in: Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity (A), Elsevier, MIT Press, 1990, pp. 869–942.
[39] H. Koshutanski, F. Massacci, An access control framework for business processes for web services, in: Proceedings of the XMLSEC '03: ACM Workshop on XML Security, 2003, pp. 15–24.
[40] A.M.C.A. Koster, H.L. Bodlaender, S.P.M. van Hoesel, Treewidth: computational experiments, Electronic Notes Discrete Math. 8 (2001) 54–57.
[41] M. Lohrey, On the parallel complexity of tree automata, in: Proceedings of the RTA'01, Lecture Notes on Computer Science, vol. 2051, Springer, Berlin, 2001, pp. 201–215.
[42] C.L. Lucchesi, S.L. Osborn, Candidate keys for relations, J. Comput. Syst. Sci. 17 (2) (1978) 270–279.
[43] H. Mannila, K.-J. Räihä, The Design of Relational Databases, Addison-Wesley, Reading, MA, 1992.
[44] H. Maryns, On the implementation of tree automata: limitations of the naive approach, in: Proceedings of the TLT'06: International Treebanks and Linguistic Theories Conference, 2006, pp. 235–246.
[45] M. Minoux, LTUR: a simplified linear-time unit resolution algorithm for horn formulae and computer implementation, Inf. Process. Lett. 29 (1) (1988) 1–12.
[46] K.K. Nambiar, B. Gopinath, T. Nagaraj, Manjunath, Boyce–Codd normal form decomposition, J. Comput. Math. Appl. 33 (4) (1997).
[47] J. Obdržálek, DAG-width: connectivity measure for directed graphs, in: Proceedings of the SODA'06, ACM Press, New York, 2006, pp. 814–821.
[48] S.L. Osborn, Testing for existence of a covering Boyce–Codd normal form, Inf. Process. Lett. 8 (1) (1979) 11–14.

[49] W.L. Ruzzo, Tree-size bounded alternation, J. Comput. Syst. Sci. 21 (2) (1980) 218–235.
[50] M. Samer, S. Szeider, Algorithms for propositional model counting, in: Proceedings of the LPAR'07, Lecture Notes in Computer Science, vol. 4790, 2007, pp. 484–498.
[51] S. Szeider, On fixed-parameter tractable parameterizations of SAT, in: Proceedings of the SAT'03, Selected Revised Papers, Lecture Notes in Computer Science, vol. 2919, 2004, pp. 188–202.
[52] F. van den Eijkhof, H.L. Bodlaender, A.M.C.A. Koster, Safe reduction rules for weighted treewidth, Algorithmica 47 (2) (2007) 139–158.