The Complexity of Acyclic Conjunctive Queries

GEORG GOTTLOB

Technische Universität Wien, Vienna, Austria

AND

NICOLA LEONE AND FRANCESCO SCARCELLO

University of Calabria, Rende, Italy

Abstract. This paper deals with the evaluation of acyclic Boolean conjunctive queries in relational databases. By well-known results of Yannakakis [1981], this problem is solvable in polynomial time; its precise complexity, however, has not been pinpointed so far. We show that the problem of evaluating acyclic Boolean conjunctive queries is complete for LOGCFL, the class of decision problems that are logspace-reducible to a context-free language. Since LOGCFL is contained in AC^1 and NC^2, the evaluation problem of acyclic Boolean conjunctive queries is highly parallelizable. We present a parallel database algorithm solving this problem with a logarithmic number of parallel join operations. The algorithm is generalized to computing the output of relevant classes of non-Boolean queries. We also show that the acyclic versions of the following well-known database and AI problems are all LOGCFL-complete: The Query Output Tuple problem for conjunctive queries, Conjunctive Query Containment, Clause Subsumption, and Constraint Satisfaction. The LOGCFL-completeness result is extended to the class of queries of bounded treewidth and to other relevant query classes which are more general than the acyclic queries.

Categories and Subject Descriptors: F.2.2. [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures; complexity of proof-procedures*; H.2.4. [**Database Management**]: Systems—*query processing; parallel databases; rule-based databases*; I.2.3. [**Artificial Intelligence**]: Deduction and Theorem Proving—*answer/reason extraction, deduction, inference engines, logic programming, resolution*; I.2.8. [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*backtracking, graph and tree search strategies, plan execution, formation, and generation*

A preliminary version of this paper appeared in the *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS '98)* (Palo Alto, Calif., Nov. 8–11). IEEE Computer Society Press, Los Alamitos, Calif., 1998.

This research was supported by FWF (Austrian Science Funds) under the project Z29-INF.

Part of the work of F. Scarcello has been carried out while visiting the Technische Universität Wien.

Part of the work of N. Leone has been carried out while he was with the Technische Universität Wien.

Authors' addresses: G. Gottlob, Institut für Informationssysteme, Technische Universität Wien, A-1040 Vienna, Austria, e-mail: gottlob@dbai.tuwien.ac.at; N. Leone, Department of Mathematics, University of Calabria, I-87030 Rende, Italy, e-mail: leone@unical.it; F. Scarcello, D.E.I.S., University of Calabria, I-87030 Rende, Italy, e-mail: scarcello@deis.unical.it.

General Terms: Theory

Additional Key Words and Phrases: Acyclic hypergraph, algorithm, bounded treewidth, conjunctive query, constraint, constraint satisfaction problem, CSP, database theory, degree of cyclicity, hinge, join tree, LOGCFL, parallel algorithm, query containment, query-width, subsumption, tree query

1. *Introduction and Overview of Results*

1.1. MAIN PROBLEM STUDIED: THE COMPLEXITY OF ACYCLIC CONJUNCTIVE QUERIES. Conjunctive queries, equivalent in expressive power to Select–Project–Join queries, constitute the most basic, most important, and best-studied type of database queries. The problem of *evaluating* a conjunctive query over a relational database is a fundamental problem in the field of database systems. Since the early days of relational databases, the complexity of this problem and of its variants was studied, and various evaluation algorithms, either for general conjunctive queries or for syntactical restrictions, were developed. The complexity research actually started in 1977 with a landmark paper by Chandra and Merlin [1977], where, among other results, it was proven that the problem of evaluating a Boolean conjunctive query over a relational database is NP-complete. This problem, which also appears as Problem SR31 in Garey and Johnson's book [Garey and Johnson 1979], will be referred to as *Boolean Conjunctive Query* (*BCQ*) in the present paper. Since Chandra and Merlin's 1977 paper, complexity issues related to conjunctive queries have been of lasting interest as witnessed by a number of very recent contributions.[1]

In this paper, we adopt the logical representation of a relational database [Abiteboul and Hull 1995; Ullman 1989], where data tuples are identified with logical ground atoms, and conjunctive queries are represented as datalog rules. We will, in the first place, deal with *Boolean* conjunctive queries represented by rules whose heads are variable free, that is, propositional (see Example 1.1 below). From our results on Boolean queries, we are able to derive complexity results on important database problems concerning general (not necessarily Boolean) conjunctive queries, as discussed at the end of this section.

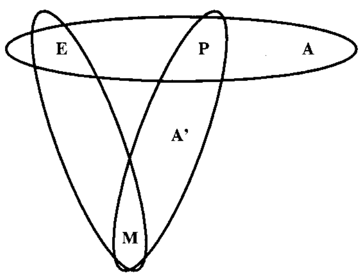
Example 1.1. Consider a relational database with the following relation schemas:

```
works(Emp#, Proj#, Assigned)

manages(Emp#, Proj#, Assigned)

relative(Emp1, Emp2)
```

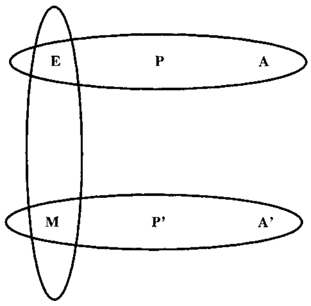
The Boolean conjunctive query Q_1 below checks whether some employee works in a project managed by his/her relative.

$$Q_1\colon ans \leftarrow \texttt{works}(E,P,A) \wedge \texttt{manages}(M,P,A') \wedge \texttt{relative}(E,M).$$

[1] See, for example, Chekuri and Rajaraman [2000], Gottlob et al. [2000b], Kolaitis and Vardi [2000], and Papadimitriou and Yannakakis [1997].



(a)



(b)

FIG. 1. (a) Hypergraph $H(Q_1)$, and (b) hypergraph $H(Q_2)$ of queries Q_1 and Q_2, in Example 1.1.

Here E, P, M, A, A' are variables, and the query is answered positively if there exists a substitution ϑ assigning data values (constants) to these variables such that the rule body evaluates to *true* over the given database.

The Boolean conjunctive query Q_2 below checks whether there is a manager who has a relative working in the company.

$$Q_2\text{: } ans \leftarrow \texttt{works}(E, P, A) \wedge \texttt{manages}(M, P', A') \wedge \texttt{relative}(E, M).$$

The *hypergraph* $H(Q)$ associated to a conjunctive query Q is defined as $H(Q) = (V, E)$, where the set V of vertices consists of all variables occurring in the body of Q, while the set E of hyperedges contains, for each atom A in the rule body, the set $var(A)$ of all variables occurring in A. The hypergraphs corresponding to queries Q_1 and Q_2 of the above example are depicted in Figure 1.

A query Q is *cyclic* (*acyclic*) if its associated hypergraph $H(Q)$ is cyclic (acyclic). We refer to the standard notion of cyclicity/acyclicity (also called α-acyclicity [Fagin 1983]) in hypergraphs used in database theory [Abiteboul and Hull 1995; Maier 1986; Ullman 1989] (a formal definition is given in Section 2.2).

Note that query Q_1 of Example 1.1 is cyclic, while Q_2 is acyclic. Acyclic queries arise very often in real database applications.

The main decision problem studied in this paper is the following:

ABCQ Given a database **db** and an acyclic Boolean conjunctive query Q, decide whether Q evaluates to true on **db**.

Our main goal is to determine the precise complexity of ABCQ. Note that, since both the database **db** and the query Q are part of an input-instance of ABCQ, what we are interested in is often referred to as the *combined complexity* [Vardi 1982].

Acyclic conjunctive queries were the object of a large number of investigations.[2] In particular, it was shown that the class of acyclic queries coincides with the class of *tree queries* [Beeri et al. 1983], see also Abiteboul et al. [1995], Maier [1986], and Ullman [1989]. The latter are queries that are representable by a *join*

[2] See, for example, Bernstein and Goodman [1981], Chekuri and Rajaraman [2000], D'Atri and Moscarini [1986], Fagin [1983], Fagin et al. [1982], Goodman and Shmueli [1982], Kolaitis and Vardi [2000], Malvestuto [1986], Papadimitriou and Yannakakis [1997], Saccà [1985], Sagiv and Shmueli [1993], Yannakakis [1981], and Yu and Özsoyoğlu [1984].

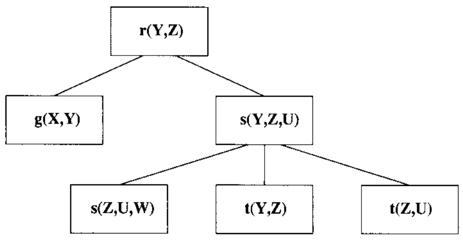


FIG. 2. A join tree for the acyclic query Q_3 in Example 1.2.

tree (or *join forest*). A *join forest* $F(Q)$ for an acyclic conjunctive query Q is a forest whose vertices are the atoms in the body of Q such that, whenever the same variable X occurs in two atoms A_1 and A_2, then A_1 and A_2 are connected in $F(Q)$, and X occurs in each atom on the unique path linking A_1 and A_2. A *join tree* is a join forest having a single connected component.

Example 1.2. Consider the following query:

$$Q_3\colon ans \leftarrow r(Y, Z) \wedge g(X, Y) \wedge s(Y, Z, U) \wedge s(Z, U, W) \wedge t(Y, Z) \wedge t(Z, U).$$

A join tree for Q_3 is shown in Figure 2.

By well-known results of Yannakakis [1981] acyclic Boolean conjunctive queries (and even non-Boolean acyclic conjunctive queries) are solvable in polynomial time. In the Boolean case, the algorithm roughly works as follows. First compute a join tree (or forest) T for the given query Q. Then process (each component of) T in a bottom-up manner (from the leaves towards the root) by performing a sequence of semi-joins, one for each tree edge, starting at the leaves. For each bottom-top directed edge (R, S) of the join-tree, the relation s corresponding to node S is reduced by semi-joining it with the relation r corresponding to R. The Boolean query evaluates to *true* if and only if at the end of this procedure the relation corresponding to the root is not empty. Note that, in a slightly restricted setting, a very similar semi-join based evaluation procedure was independently developed by Bernstein and Chiu [1981].

Yannakakis's algorithm is of sequential nature. In particular, if Q has a chain of length n as join tree, then the algorithm performs $n - 1$ joins in sequence. Notwithstanding the sequential character of Yannakakis's algorithm, ABCQ could not be shown to be complete for polynomial time. In an unpublished manuscript, Dahlhaus [1990] proved that acyclic conjunctive query non-emptiness, which is equivalent to ABCQ, is in NC^2. For restricted versions of ABCQ, NC^2 algorithms have been given in [Kasif and Delcher 1994; Zhang and Mackworth 1993] (see the discussion in Section 7.4). However, the precise complexity of ABCQ has never been pinpointed. We solve it in the present paper.

A problem closely related to ABCQ is the following problem JTREE:

JTREE: Given a database **db** and a join tree of an acyclic Boolean conjunctive query Q, decide whether Q evaluates to *true* on **db**.

JTREE is thus a version of ABCQ, where the join tree is already given and does not need to be computed. We are also interested in the precise complexity of JTREE.

1.2. RESULTS. In Section 3, we relate the problems ABCQ and JTREE by proving that the following tasks are in SL (Symmetric Logspace):

—Deciding whether a given conjunctive query Q is acyclic.

—Deciding whether a pair $\langle A_1, A_2 \rangle$ of query atoms of an acyclic query Q is an edge of a canonical join forest of Q.

Intuitively, this means that testing acyclicity and computing join trees are extremely easy tasks, and there is thus a very easy reduction from ABCQ to JTREE. The reduction can be carried out by a logspace machine with oracle in SL and is thus highly parallelizable.

Note that linear-time *sequential* methods for computing a join tree are well known [Tarjan and Yannakakis 1984] (see also Bernstein and Goodman [1981], Graham [1979], and Yu and Özsoyoğlu [1979] for earlier polynomial algorithms), while the previous best parallel complexity result was an NC^2 upper bound by Naor et al. [1989] (see also Dahlhaus and Karpinski [1996] and Klein [1996] for similar results on related problems).

In Section 4, we prove our main result:

ABCQ and JTREE are both complete for LOGCFL.

The complexity class LOGCFL consists of all decision problems that are logspace-reducible to a context-free language. As shown in Borodin et al. [1989], LOGCFL is closed under complementation. An example of a problem complete for LOGCFL is Greibach's hardest context-free language [Greibach 1973]. There are relatively few other natural problems known to be LOGCFL-complete (see, e.g., Bédard et al. [1993], Skyum and Valiant [1985], and Sudborough [1977]). The relationship between LOGCFL and the other relevant complexity classes used in this paper is summarized in the following chain of inclusions:

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

Here, L is logspace, AC^i and NC^i are *logspace-uniform* parallel complexity classes based on the corresponding types of Boolean circuits (see Section 2), SL is symmetric logspace, NL is nondeterministic logspace, P is polynomial time, and NP is nondeterministic polynomial time.

The membership part of our completeness proof uses an important characterization of LOGCFL by Sudborough [1977; 1978]. LOGCFL coincides with the class of languages accepted by nondeterministic auxiliary push-down automata in logarithmic space and polynomial time. We construct such an automaton for processing a join tree over a database. The key idea is that, unlike Yannakakis's algorithm, our automaton processes the tree *top down* by choosing nondeterministically a set of witness tuples.

The hardness part of our completeness proof relies on another important characterization of LOGCFL, found by Venkateswaran [1991]: LOGCFL coincides with the class SAC^1 of problems solvable by logspace-uniform families of semi-unbounded AC^1 Boolean circuits (SAC^1 circuits). We show that SAC^1 circuits can be translated into acyclic conjunctive queries. We prove that the problems ABCQ and JTREE remain hard for LOGCFL, even in case the acyclic hypergraph corresponding to the query is an acyclic *graph*.

Our main complexity result is somewhat surprising and provides perhaps a better understanding of the nature of LOGCFL. Intuitively, problems in LOGCFL contain two seemingly orthogonal dimensions (or sources) of nondeterminism: (i) the different choices of a suitable tree shape, and (ii) the different possible choices of filling the chosen tree structure with suitable data items such that some local consistency conditions along the edges of the tree are satisfied. To illustrate this, think of a context-free language $\mathcal{L}$ generated by a context-free grammar G. To show that a word w belongs to $\mathcal{L}$, we need to construct a derivation tree for w. There is, in general, a choice of several possible *shapes* for such trees (from short and bushy to long and thin). Moreover, each suitable tree skeleton can be filled (or labeled) possibly in various ways with actual derivation rules such that the root is labeled with a rule for start symbol S and each further rule in the tree matches the rule chosen for its parent node (this is the *local* consistency condition). All previously known problems complete for LOGCFL we are aware of carry these two (apparently orthogonal) dimensions of nondeterminism.

On the other hand, the problem JTREE carries only source (ii) of nondeterminism. Source (i) does not subsist in JTREE, given that with each instance of JTREE the join tree is provided, and the aim is just to fill this tree with appropriate data values (by finding an appropriate substitution for the variables). From the LOGCFL-completeness of JTREE, it thus follows that LOGCFL can be characterized as the class of problems logspace-reducible to fill-in problems, where a "mould" in form of an acyclic graph should be filled with values from a given set such that certain local consistency conditions are met. Such problems are a restricted version of the constraint satisfaction problem in AI, which will be discussed below.

The problem ABCQ of conjunctive query evaluation is logspace-equivalent to a number of well-known and important database and AI problems that are all NP-complete in their general version. From our main result, we easily obtain complexity results for the acyclic versions of those problems. In particular, we show that the following four problems are all complete for LOGCFL, and thus highly parallelizable:

—*The Acyclic Query Output Tuple Problem.* Given an acyclic conjunctive query Q, a database **db**, and a tuple t, determine whether t belongs to the answer $Q(\mathbf{db})$ of Q over **db**. Note that, if the number of variables in Q's head predicate is bounded by a constant, then *computing* the complete output of Q is actually in $\mathrm{L}^{\mathrm{LOGCFL}}$ and can thus be done by AC^1 circuits, as well as by NC^2 circuits.

—*Acyclic Conjunctive Query Containment.* Decide whether a conjunctive query Q_1 is contained in an acyclic conjunctive query Q_2. Query Q_1 is contained in query Q_2 if for each database instance **db**, the answer $Q_1(\mathbf{db})$ is a subset of $Q_2(\mathbf{db})$. Polynomial sequential methods for solving this restriction of the general (NP-complete) conjunctive query containment problem were described by Qian [1996], and independently by Chekuri and Rajaraman [2000].

—*Acyclic Clause Subsumption.* Check whether a (general) acyclic clause C subsumes a clause D, that is, whether there exists a substitution ϑ such that $C\vartheta \subseteq D$. A (general) clause is a disjunction of (positive or negative) literals, possibly containing function symbols. Note that subsumption is an extremely important technique used in clause-based theorem proving [Bachmair et al. 1996].

—*Acyclic Constraint Satisfaction.* Given a finite set *Var* of variables, a finite domain U of values, a set of constraints $\mathcal{C} = \{C_1, C_2, \ldots, C_q\}$, where each constraint C_i is a pair (S_i, r_i), and where S_i is a list of variables of length m_i and r_i is an m_i-ary relation over U, such that the hypergraph of the sets of variables corresponding to the sets S_i is acyclic, decide whether there is a substitution $\vartheta: Var \rightarrow U$, such that, for each $1 \leq i \leq q$, $S_i\vartheta \in r_i$. The connection between constraint satisfaction problems in the AI setting and conjunctive queries has been well acknowledged in the literature, and it is well known that acyclic constraint satisfaction problems are solvable in polynomial time.[3] Also note that NC algorithms for restricted versions of the acyclic case were previously known. In particular, Kasif and Delcher [1994] presented an NC^2 parallel algorithm for acyclic constraint networks having only binary constraints; hence, their problem can be represented by a tree, rather than by an acyclic hypergraph. A parallel algorithm for constraint-networks with nonbinary constraints is due to Zhang and Mackworth [1993]. Their algorithm takes as input a join tree T such that, for any relation in T, the number of attributes is bounded by a constant; moreover, they assume a fixed universe U.

From the proof of our main theorem, it follows that all these problems remain LOGCFL-complete even in case the associated acyclic hypergraph is restricted to be an acyclic *graph* (i.e., if only binary relations or constraints are used).

Graph or hypergraph acyclicity is a key-property responsible for the polynomial solvability of problems that are in general NP-hard, such as ABCQ and all other equivalent problems we just described. In graph theory, the notion of *constant treewidth* has been introduced as an approximation of the concept of acyclicity. The treewidth of a graph is a measure of its cyclicity. In particular, the acyclic graphs have treewisth 1, while the class of all graphs of treewidth bounded by a constant $k > 1$ is much larger, but still shares many good computational properties with the class of acyclic graphs [Arnborg et al. 1991; Wanke 1994].

Formally, the concept of treewidth is introduced via the notion of *tree-decomposition* (see Section 6 for details). A graph G has treewidth c if c is the smallest possible width of any tree-decomposition of G. The concept of bounded treewidth can be generalized to hypergraphs, and thus to conjunctive queries [Chekuri and Rajaraman 2000].

Chekuri and Rajaraman [2000] have recently shown that the conjunctive query containment problem, that is, checking whether Q_1 is contained in Q_2, is decidable in polynomial time for instances where the treewidth of Q_2 is bounded by a constant k. Since BCQ is efficiently reducible to the conjunctive query containment problem [Chandra and Merlin 1977], it follows that the evaluation problem for Boolean conjunctive queries of bounded treewidth is feasible in polynomial time. In Section 6, we determine the precise complexity of this problem by showing that BCQ restricted to queries of bounded treewidth is

[3] See, for example, Dechter and Pearl [1989], Gyssens et al. [1994], Kolaitis and Vardi [2000], and Pearson and Jeavons [1997], where the polynomiality result is extended to generalizations of the acyclic case.

complete for LOGCFL. Note that this result holds even if a tree-decomposition is not part of the problem instance.

The treewidth is not the only possible cyclicity measure for graphs, hypergraphs, and queries. Two related yet more general concepts, the *degree of cyclicity* [Gyssens et al. 1994], and the *query-width* [Chekuri and Rajaraman 2000] have been discussed in the literature. The first is based on the notion of *hinge-tree decomposition*, while the second is based on the concept of *query-decomposition* (details are given in Section 6). We show that solving Boolean conjunctive queries whose degree of cyclicity is bounded by a constant or whose query-width is bounded by a constant is LOGCFL-complete, provided a suitable decomposition is given together with the problem instance.

Since LOGCFL $\subseteq$ AC^1 $\subseteq$ NC^2, by our results, the problems ABCQ and JTREE as well as all other equivalent problems discussed in this paper, 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 algorithm can be derived easily by well-known methods [Karp and Ramachandran 1990; Ruzzo 1980].

However, note that the PRAM model does not really match the features of parallel relational database management systems (DBMS), where the relational operators (selection, projection, join, semi-join, etc.) are treated as primitives. In such database models, there is a distinction between *intra-operation parallelism*, that is, the parallel execution of a single primitive operation (e.g., join), and *inter-operation parallelism*, that is, processing different primitive operations in parallel [Wilschut et al. 1995]. We may assume that in a given parallel DBMS the issues of intra-operation parallelism are solved, that is, that a set of highly parallel procedures for performing single relational operations is given. The remaining key problem for query optimization is thus to exploit inter-operation parallelism as much as possible for each query Q. This means that a parallel query execution plan for Q should be found, where as many as possible operations are performed in parallel.

In Section 7, we present a method for solving acyclic Boolean conjunctive queries achieving a very high degree of inter-operation parallelism. We assume that the query is given in form of a join tree, thus, actually we solve the problem JTREE (but as said before, ABCQ is easily reduced to JTREE). Our parallel method for evaluating JTREE is based on a *tree contraction* technique that closely resembles the method of Karp and Ramachandran [1990] for evaluating arithmetical expressions. In analogy to Karp and Ramachandran [1990] we introduce a *shunt* operation for tree contraction. This shunt operation is entirely defined in terms of relational algebra and thus directly executable by a relational DBMS. We present a parallel algorithm, called DB-SHUNT, which is based on such an operation and evaluates a given JTREE instance by performing $O(\log n)$ parallel steps using $O(n)$ processors. In each step, every processor performs a constant number of relational operations, while keeping the size of any intermediate relation quadratic in the maximum relation size of the database.

As a slight modification of the DB-SHUNT algorithm, we present an algorithm ACQ that computes the output of a non-Boolean acyclic conjunctive query in a highly parallel fashion, performing a sequence of $O(\log n)$ parallel joins. Note that the size of the result of such a query may be exponential in the size of the database. But in many cases of practical relevance, the number of output attributes (i.e., the number of variables in the head-atom) of a query is rather small. It thus makes sense to assume that the number of output-attributes is bounded by a constant, in which case the size of the output is polynomial in the size of the input. Under this assumption, the ACQ algorithm behaves extremely well, using a polynomial total number of operations and requiring a rather limited intermediate storage space. In a follow-up paper [Gottlob et al. 2000], we describe a more sophisticated parallel algorithm whose intermediate storage space is polynomially bounded in the combined size of the input plus the output, even if the number of output-attributes is unbounded.

2. *Preliminaries and Basic Concepts*

2.1. DATABASES AND QUERIES. For a background on databases, conjunctive queries, etc., see Abiteboul et al. [1995], Maier [1986], and Ullman [1989]. We define only the most relevant concepts here.

A relation schema R consists of a name (name of the relation) r and a finite ordered list of attributes. To each attribute A of the schema, a countable domain $Dom(A)$ of atomic values is associated. A *relation instance* (or, simply, a *relation*) over schema $R = (A_1, \ldots, A_k)$ is a finite subset of the Cartesian product $Dom(A_1) \times \cdots \times Dom(A_k)$. The elements of relations are called *tuples*. A database schema DS consists of a finite set of relation schemas. A *database instance*, or simply *database*, **db** over database schema $DS = \{R_1, \ldots, R_m\}$ consists of relation instances $r_1, \ldots, r_m$ for the schemas $R_1, \ldots, R_m$, respectively, and a finite universe $U \subseteq \bigcup_{R_i(A^i_1, \ldots, A^i_{k_i}) \in DS}(Dom(A^i_1) \cup \cdots \cup Dom(A^i_{k_i}))$ such that all data values occurring in **db** are from U.

In this paper, we will adopt the standard convention [Abiteboul et al. 1995; Ullman 1989] of identifying a relational database instance with a logical theory consisting of ground facts. Thus, a tuple $\langle a_1, \ldots a_k \rangle$, belonging to relation r, will be identified with the ground atom $r(a_1, \ldots, a_k)$. The fact that a tuple $\langle a_1, \ldots, a_k \rangle$ belongs to relation r of a database instance **db** is thus simply denoted by $r(a_1, \ldots, a_k) \in \mathbf{db}$.

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

$$Q\colon ans(\mathbf{u}) \leftarrow r_1(\mathbf{u}_1) \wedge \cdots \wedge r_n(\mathbf{u}_n),$$

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

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

The conjunctive query Q is a *Boolean conjunctive query* if its head predicate *ans* has arity 0, thus the head is a purely propositional atom and does not contain variables.

Query Q evaluates to *true* if there exists a substitution ϑ such that, for $1 \le i \le n$, $r_i(\mathbf{u_i})\vartheta \in \mathbf{db}$; otherwise, the query evaluates to *false*.

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

Note that conjunctive queries as defined here are equivalent to conjunctive queries in the more classical setting of relational calculus, as well as to Select-Project-Join queries in the classical setting of relational algebra, or to simple SQL queries of the type

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

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

Two sample queries are shown in Example 1.1.

In order to study the complexity of evaluating Boolean conjunctive queries, we will refer to Turing-machine-based complexity classes. The algorithms appearing in our proofs mainly use high-level primitive operations which abstract from specific machine-level encodings of the used concepts. Notwithstanding, it is useful and instructive to describe how the low-level encoding of the main concepts may look like. We will refer to this encoding in Section 2.5 where we will briefly explain why certain primitive database operations are indeed feasible in logarithmic space.

Atomic data items are represented by (variable-length) bit-strings. A k-ary data-tuple is encoded as a list of k data items. Lists and list elements are suitably delimited by special tape symbols such as parentheses and commas. A relation is encoded as a list $(r, a, t_1, \ldots, t_n)$, where r is a bit-string encoding of the relation name, a is the (binary representation of the) arity of the relation, and $t_1, \ldots, t_n$ are the encodings of the tuples belonging to the relation. A database **db** consists of a list $(r_1, \ldots, r_m)$, where $(r_1, \ldots, r_m)$ are the encodings of the relations belonging to **db**. The encoding of a query atom consists of a relation name followed by a list $(a_1, \ldots, a_h)$ of term encodings. A variable is encoded as a sequence vs, where v is a special symbol and s is a bit-string encoding the variable name. A constant is an atomic data item. A query is encoded as a list of atoms. In the sequel of the paper, the encoding of any data structure X is denoted by $enc(X)$.

2.2. Acyclic Hypergraphs. The concept of hypergraph acyclicity plays a very important role in database theory [Abiteboul et al. 1995; Maier 1986; Ullman 1989].

A hypergraph $H = (V, E)$ consists of a set V of vertices (nodes) and a set $E \subseteq 2^V$ of hyperedges (short: edges).

Given a hypergraph $H = (V, E)$, the *GYO-reduct* $GYO(H)$ [Graham 1979; Yu and Özsoyoğlu 1979] is the hypergraph obtained from H by repeatedly applying the following rules as long as possible:

(1) Remove hyperedges that are empty or contained in other hyperedges;
(2) Remove vertices that appear in at most one hyperedge.

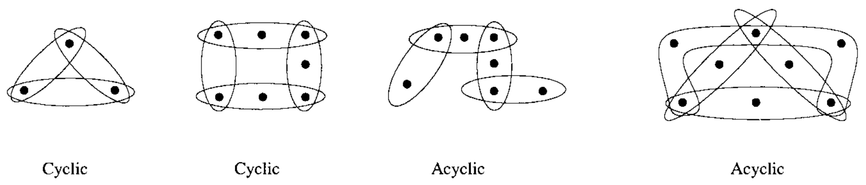


FIG. 3. Cyclic and acyclic hypergraphs.

Observe that, since E is a set of hyperedges, also the execution of step (2) may reduce the number of hyperedges, if two hyperedges become identical after the removal of some of their vertices.

It was noted [Beeri et al. 1983] that the order of rule applications does not matter and that the final hypergraph $GYO(H)$ is unique.

A hypergraph H is *acyclic* if $GYO(H)$ is the empty hypergraph, that is, $GYO(H) = (\emptyset, \emptyset)$.

There exist various equivalent characterizations of acyclic hypergraphs.[4]

In Figure 3, examples of cyclic and acyclic hypergraphs are exhibited.

Acyclicity as defined here is the usual concept of acyclicity in the context of databases. It is referred to as α-acyclicity in Fagin [1983]. This is the least restrictive concept of hypergraph acyclicity among all those defined in the literature. There are several more restrictive notions of acyclicity such as, for example, β-acyclicity, γ-acyclicity, and Berge-acyclicity [Berge 1976; Fagin 1983]. In particular, the following chain of implications holds for any hypergraph H: H is Berge-acyclic $\Rightarrow$ H is γ-acyclic $\Rightarrow$ H is β-acyclic $\Rightarrow$ H is acyclic (i.e., α-acyclic). However, in general, none of the reverse implications holds.

As we will see, our main result, the LOGCFL-completeness of acyclic Boolean conjunctive queries, does not only hold for α-acyclic hypergraphs, but does actually hold for any of the above-mentioned types of hypergraph acyclicity. The same holds for all LOGCFL-completeness results presented in Section 5.

2.3. ACYCLIC CONJUNCTIVE QUERIES AND JOIN TREES. If Q is a conjunctive query, we define the hypergraph $H(Q) = (V, E)$ associated to Q as follows: The set of vertices V consists of all variables occurring in Q. For each atom $r_i(\mathbf{u_i})$ in the body of Q, the set E contains a hyperedge consisting of all variables occurring in $\mathbf{u_i}$. Note that the cardinality of E can be smaller than the cardinality of $atoms(Q)$ because two query atoms having exactly the same set of variables in their arguments give rise to only one edge in E. For example, the three query atoms $r(X, Y)$, $r(Y, X)$, and $s(X, X, Y)$ all correspond to a unique hyperedge $\{X, Y\}$.

If $H(Q)$ is acyclic, then Q is referred to as an *acyclic conjunctive query*.

Example 2.1. Consider again query Q_1 and Q_2 of Example 1.1 (see Introduction). The hypergraphs $H(Q_1)$ and $H(Q_2)$ corresponding to queries Q_1 and Q_2, respectively, are depicted in Figure 1. It is easy to see that $H(Q_1)$ is a cyclic hypergraph, while $H(Q_2)$ is an acyclic hypergraph. As a consequence, Q_1 is a cyclic query and Q_2 is an acyclic query.

[4] See, for example, Beeri et al. [1981; 1983], Chase [1981], Goodman and Shmueli [1983], Hull [1983], and Maier [1986].

Acyclic conjunctive queries were the object of a large number of investigations.[5] In particular, it was shown that the class of acyclic queries coincides with the class of *tree queries* [Beeri et al. 1983] (see also Abiteboul et al. [1995], Maier [1986], and Ullman [1989]). The latter are queries which are representable by a join tree (or join forest).

A *join forest* for a query Q is a forest G whose set of vertices V_G is the set $atoms(Q)$ and such that, for each pair of atoms A_1 and A_2 in V_G having variables in common, the following conditions hold:

(1) A_1 and A_2 belong to the same connected component of G, and
(2) all variables common to A_1 and A_2 occur in every atom on the (unique) path in G from A_1 to A_2.

If G is a tree, then it is called a *join tree* for query Q.

Figure 2 shows a join tree for the query Q_3 of Example 1.2.

At the machine level, a join forest is encoded by a list containing the atom encodings of all atoms occurring in the query, followed by a list containing, for each edge $\{A, A'\}$ belonging to the join forest, a pair $(enc(A), enc(A'))$, where $enc(A)$ lexicographically precedes $enc(A')$.

2.4. LOGCFL AND OTHER RELEVANT COMPLEXITY CLASSES. We shall identify a formal language $\mathcal{L}$ over an alphabet A with the following decision problem:

Instance: A word w from A^*

Question: Does w belong to $\mathcal{L}$?

The complexity class LOGCFL consists of all those decision problems that are logspace-reducible to a context-free language. An obvious example of a problem complete for LOGCFL is Greibach's hardest context-free language [Greibach 1973]. There are very few other natural problems known to be LOGCFL-complete (see, e.g., Skyum and Valiant [1985] and Sudborough [1977]). *Symmetric Logspace*, denoted by SL, is the class of all those decision problems solvable by logspace symmetric nondeterministic Turing machines [Johnson 1990; Lewis and Papadimitriou 1982]. A symmetric Turing machine is a machine whose transition relation between configurations is symmetric.

The relationship between LOGCFL, SL, and the other relevant complexity classes used in this paper is summarized in the following chain of inclusions:

$$\mathrm{AC}^0 \subseteq \mathrm{NC}^1 \subseteq \mathrm{L} \subseteq \mathrm{SL} \subseteq \mathrm{NL} \subseteq \mathrm{LOGCFL} \subseteq \mathrm{AC}^1 \subseteq \mathrm{NC}^2 \subseteq \mathrm{P} \subseteq \mathrm{NP}.$$

Here, AC^i and NC^i are *logspace-uniform* circuit-based parallel computation classes defined below, L is logspace, SL is symmetric logspace, NL is nondeterministic logspace, P is polynomial time, and NP is nondeterministic polynomial time. For definitions of the latter classes, for complete problems, and for references concerning their mutual relationships, see Johnson [1990].

It is conjectured that problems complete for P are inherently sequential, and cannot take advantage from parallel computation. A problem is instead called *highly parallelizable* if it can be solved in polylogarithmic time using a polynomial

[5] See, for example, Bernstein and Goodman [1981], D'Atri and Moscarini [1986], Fagin [1983], Fagin et al. [1982], Goodman and Shmueli [1982], Malvestuto [1986], Papadimitriou and Yannakakis [1997], Saccà [1985], Sagiv and Shmueli [1993], and Yannakakis [1981].

number of processors working in parallel [Greenlaw et al. 1995]. There are several models of parallel computation. One of the simplest and most accepted models is the *Boolean circuit model*, which we describe below. There are other models of parallel computation such as, for instance, the model of *Parallel Random Access Machines* (*PRAM*) [Greenlaw et al. 1995; Karp and Ramachandran 1990]. Even though these models are very different in terms of primitive operations, they coincide in the notion of high parallelizability.

A *Boolean circuit* G_n with n inputs is a finite directed acyclic graph whose nodes are called gates and are labeled as follows. Gates of fan-in (indegree) zero are called *circuit input gates* and are labeled from the set $\{$*false*, *true*, $z_1, z_2, \ldots, z_n, \neg z_1, \neg z_2, \ldots, \neg z_n\}$. All other gates are labeled either AND, OR, or NOT. The fan-in of gates labeled NOT must be one. The unique node with fan-out (outdegree) zero is called output gate. The evaluation of G_n on input string w of length n is defined in the standard way. In particular, any input gate g labeled by z_i (respectively, $\neg z_i$) gets value *true* (respectively, *false*) if the ith bit of w is 1 (respectively, 0); otherwise, g gets value *false* (respectively, *true*).

A Boolean circuit is thus given as a triple (N, A, *label*), where N is the set of nodes (gates), A is the set of arcs, and *label* is the labeling of the nodes as described.

The *depth* of a Boolean circuit G is the length of a longest path in G from a circuit input gate to the output gate of G. The size $S(G)$ of G is the number of gates (including input-gates) in G.

A family $\mathcal{G}$ of Boolean circuits is a sequence $(G_0, G_1, G_2, \ldots)$, where the nth circuit G_n has n inputs. Such a family is *logspace-uniform* if there exists a logspace Turing machine which, on the input string containing n bits 1, outputs the circuit G_n [Ruzzo 1981]. Note that the size of the nth circuit G_n of a logspace-uniform family $\mathcal{G}$ is polynomial in n.

The language L accepted by a family $\mathcal{G}$ of circuits is defined as follows: $L = \bigcup_{n \geq 0} L_n$, where L_n is the set of input strings accepted by the nth member G_n of the family. An input string w of length n is accepted by the circuit G_n if G_n evaluates to *true* on input w.

A family $\mathcal{G}$ of Boolean circuits has *bounded fan-in* if there exists a constant c such that each gate of each member G_n of $\mathcal{G}$ has its fan-in bounded by c.

A family $\mathcal{G}$ of Boolean circuits is *semi-unbounded* if the following two conditions are met:

—All circuits of $\mathcal{G}$ involve as non-leaves only AND and OR gates, but no NOT gates (negation may thus only occur at the circuit input gates); and

—There is a constant c such that each AND gate of any member G_n of $\mathcal{G}$ has its fan-in bounded by c (the OR gates may have unbounded fan-in).

For $i \geq 1$, AC^i denotes the class of all languages recognized by logspace-uniform families of Boolean circuits of depth $O(\log^i n)$.

For $i \geq 1$, NC^i denotes the class of all languages recognized by logspace-uniform families of Boolean circuits of depth $O(\log^i n)$ having bounded fan-in.

For $i \geq 1$, SAC^i denotes the class of all languages recognized by semi-unbounded logspace-uniform families of Boolean circuits of depth $O(\log^i n)$.

An important characterization of LOGCFL was found by Venkateswaran [1991]. He showed that LOGCFL coincides with the class SAC^1.

PROPOSITION 2.2 [VENKATESWARAN 1991]. *LOGCFL = SAC^1.*

Since LOGCFL = SAC^1 $\subseteq$ AC^1 $\subseteq$ NC^2, the problems in LOGCFL are all highly parallelizable. In fact, each problem in LOGCFL is solvable in logarithmic time by a concurrent-read concurrent-write parallel random access machine (CRCW PRAM) with a polynomial number of processors, or in log^2-time by an exclusive-read exclusive-write PRAM (EREW PRAM) with a polynomial number of processors [Ruzzo 1980].

Another interesting characterization of LOGCFL in terms of alternating Turing machines can be found in Ruzzo [1980].

Using the characterization of LOGCFL in Proposition 2.2, Borodin et al. [1989] have shown that LOGCFL is closed under complementation:

PROPOSITION 2.3 [BORODIN ET AL. 1989]. *LOGCFL is closed under complementation.*

Let C be a complexity class. Turing machines that work in logspace but can use an oracle in C are called L^C machines. (This terminology extends to both, acceptors and transducers.) We here adopt the standard model by Ladner and Lynch [1976] where an oracle machine has a unique write-only oracle tape which is not subject to the logarithmic space bound and which is automatically erased after an oracle-query is made. Since the oracle space is not counted, an L^C machine may ask polynomially long queries to its oracle. For a more precise description of oracle Turing machines, see Ladner and Lynch [1976] or Garey and Johnson [1979].

For each class C, the complexity class L^C consists of all those decision problems solvable by an L^C acceptor. In particular, L^{LOGCFL} is the class of all problems solvable in deterministic logspace with an oracle in LOGCFL, and L^{SL} is the class of all problems solvable in deterministic logspace with an oracle in SL.

We denote by L^{LOGCFL}(LOGCFL) the closure of LOGCFL under L^{LOGCFL} reductions, that is, the class of all problems that are many-one-reducible via a deterministic logspace oracle Turing machine with oracle in LOGCFL to a problem in LOGCFL. In other words, a problem belongs to L^{LOGCFL}(LOGCFL) if it can be transformed by an L^{LOGCFL} transducer into an equivalent problem in LOGCFL.

Sudborough [1977] remarked (without proof) that Proposition 2.3 implies that L^{LOGCFL} = LOGCFL. A slightly stronger result has been recently shown in Gottlob et al. [2000a].

LEMMA 2.4 [GOTTLOB ET AL. 2000A]. L^{LOGCFL}(LOGCFL) = L^{LOGCFL} = LOGCFL.

LOGCFL-completeness is considered under logspace reductions. Thus, a problem A is *complete* for LOGCFL if A belongs to LOGCFL and if each other problem in LOGCFL can be transformed to A by a logspace reduction.

2.5. ON LOGSPACE COMPUTATIONS. In this section, we briefly recall some well-known results about (deterministic) logspace computations that will be used in many of our proofs.

A logspace machine has an unbounded read-only input tape and a worktape subject to the logarithmic space bound.[6]

Some key capabilities of logspace machines are informally summarized as follows:

—they can maintain (a constant number of) counters on the worktape and increment or decrement such counters;

—they can locate particular items of a well-structured input, for example, by counting parentheses and other separators. For example, given a list and an index i, they can locate the first bit of the ith list element. Moreover, they can represent input items (e.g., integers, lists, list elements, etc.) via pointers consisting of input-tape addresses (or other suitable indices);

—they can access, process and compare input items in a bit-by-bit fashion, without ever loading an entire input item into the workspace;

—they can iterate any process a polynomial number of times, perform WHILE loops, and logspace subroutines. In particular, they can cycle over (tuples of) elements of input lists, identify common elements, and process them one by one.

From these properties, it can be seen that many commonly used functions and operations, such as addition, subtraction, and multiplication of binary integers, comparing data items or lists, copying data items, searching, sorting, merging and reversing lists, and so on, are feasible in logspace (cf., e.g., Jones [1997] and Papadimitriou [1994]). In the database context, the following important result is well known.

PROPOSITION 2.5 [VARDI 1982]. (For a proof, see also Abiteboul et al. [1995].) *Let Q be any fixed query expressible in first-order logic or, equivalently, in relational algebra over a fixed database schema. The result of Q can be computed in logspace. In particular, for every database* **db**, *$Q(\mathbf{db})$ can be computed in space $O(log\|\mathbf{db}\|)$.*

For illustration, let us sketch a logspace transducer T that performs the join of two relations r and s on attribute A, where r and s are encoded as described in Section 2.1. T maintains two tuple pointers τ_r and τ_s for the tuples of the relations r and s, respectively. In an outer loop, τ_r is advanced to point successively to each tuple of r. For each such tuple t_r, an inner loop makes τ_s successively point to each tuple t_s of s. For each combination of tuples t_r, t_s, a subroutine identifies the locations of the respective fields corresponding to attribute A and checks, bit after bit, whether the A-values of t_r and t_s are equal. If so, the relevant parts of t_r and t_s are (bitwise) copied to the output tape, so to form a tuple of the result. Note that the result of a join may be quadratic in the sizes of the operand relations.

Actually, we will need the following slightly stronger result on joins, where the relational schemas are not fixed:

[6] Without loss of generality, we limit our attention to machines having a single worktape. In fact, each logspace Turing machine with several worktapes can be simulated by an equivalent logspace machine having only one worktape (cf. Hopcroft and Ullman [1979]).

PROPOSITION 2.6. *The following problem is feasible in logspace. Given in input two relations r and s and their respective schemas R and S (represented as attribute lists), compute the result of the natural join* $r \bowtie s$.

PROOF. A logspace transducer solving this task behaves similarly to the above described transducer T, except that it must identify the join attributes and check the equality of two tuples with respect to several attributes (the common attributes of the schemas). For identifying the join attributes, additional pointers that scan the two attribute lists can be used. The bit-by-bit comparison of the respective data fields of two tuples is still feasible in logspace. □

There are some graph search problems whose feasibility in logspace is not easy to show. For example, it is proven in [Cook and McKenzie 1987] that a logspace machine can traverse a rooted tree in depth-first order:

PROPOSITION 2.7 ([COOK AND MCKENZIE 1987]). *The following problem can be solved by a logspace transducer. Given a rooted tree, compute a depth-first traversal of the tree starting at the root (i.e., list the nodes of the tree in any order consistent with the recursive paradigm: "visit a node and then visit each of its subtrees").*

This result is valid for any of the standard encodings of graphs, in particular, for the encodings by adjacency lists as used later on in this paper.

An important property of logspace reductions is that they are transitive. Indeed, the composition of a constant number of logspace transducers can be performed by a single logspace transducer, as well.

PROPOSITION 2.8. *Logspace reductions are transitive. That is, if a problem A is logspace reducible to a problem B, and if B in turn is logspace reducible to a problem C, then A is logspace reducible to C.*

For a proof see, for example, Papadimitriou [1994, Proposition 8.2]. The proof is simple but not trivial, given that the intermediate result of the composition of two logspace computations may be of polynomial size and may not fit into logarithmic space.

Let $\mathcal{C}$ be a complexity class among L, NL, SL, and LOGCFL. For proving membership of a problem A in $\mathcal{C}$, we will often proceed as follows. We first note that any instance I of A can be converted in logspace to an instance I' of another problem A', then we actually prove that A' is in $\mathcal{C}$. This approach is justified by the following well-known proposition.

PROPOSITION 2.9. *The classes L, SL, NL, and LOGCFL are all closed under logspace reductions, that is, if a problem A is logspace reducible to a problem A′ in any of these classes, then A lies in the same class.*

Proofs for L and NL can be found in textbooks such as, for example, Balcázar et al. [1988]. For LOGCFL, the assertion follows immediately from the definition of LOGCFL and Proposition 2.8. For SL, the assertion was shown in Lewis and Papadimitriou [1982].

By iterated application of Proposition 2.8, we immediately get the following generalization of Proposition 2.6:

PROPOSITION 2.10. *Let k be a constant. The following problem can be solved in logspace. Given in input a list of k relations $r_1, \dots, r_k$ and their respective schemas $R_1, \dots, R_k$ (represented as attribute lists), compute the natural join $r_1 \bowtie r_2 \bowtie \cdots \bowtie r_k$.*

2.6. DEFINITION OF MAIN DECISION PROBLEMS. We define below three decision problems: BCQ, ABCQ, and JTREE.

BCQ: Given a database **db** and a Boolean conjunctive query Q, decide whether Q evaluates to *true* on **db**.

ABCQ: Given a database **db** and an acyclic Boolean conjunctive query Q, decide whether Q evaluates to *true* on **db**.

JTREE: Given a database **db** and a join tree of an acyclic Boolean conjunctive query Q, decide whether Q evaluates to *true* on **db**.

Note that BCQ is a very well-known problem. It appears as Problem number SR31 in Garey and Johnson [1979]. It was shown to be NP-complete by Chandra and Merlin [1977].

ABCQ is just the acyclic version of BCQ. We will show as the main result of this paper that ABCQ is complete for LOGCFL.

The formulation of ABCQ assumes that it is guaranteed that the hypergraph of the given query is acyclic. This condition on the input is not known to be verifiable in logspace. Thus, ABCQ, as it stands, is actually a *promise* problem. However, this is not really relevant. We will show in the next section that testing whether a query has an acyclic hypergraph is in SL, and thus also in LOGCFL. Thus, this test comes—in a precise sense—for free.

A nonpromise version of ABCQ having—by our results—the same complexity as ABCQ is the following:

ABCQ′: Given a database **db** and a Boolean conjunctive query Q, decide whether Q is acyclic and evaluates to *true*.

3. *Determining Acyclicity and Computing Join Forests*

In this section, we show that determining whether a conjunctive query Q is acyclic is in SL (Symmetric Logspace). Moreover, we prove that a join forest for Q can be "computed" in SL. Both results are not difficult to show as they are consequences of results by other authors.

Since SL is a class of *decision* problems, by "computing" we actually mean determining implicitly a particular join forest T for Q by providing a method for deciding whether a given pair $\{A_1, A_2\}$ of atoms of Q is an edge of T.

For two atoms $A, A' \in atoms(Q)$ of a query Q, we define the weight $w(A, A')$ as the number of variables that A and A' have in common.

We now define the *Weighted Query Graph* $WG(Q)$ of a Boolean conjunctive query Q. This graph has as vertices the set $atoms(Q)$. The edges of $WG(Q)$ are all (unordered) pairs of query atoms $\{A, A'\}$ such that $A \neq A'$ and $w(A, A') \neq 0$. Moreover, edges of $WG(Q)$ are weighted according to w, that is, the weight of $\{A, A'\}$ is $w(A, A')$.

At the machine level, the graph $WG(Q)$ is encoded by a list containing the atom encodings of all atoms occurring in the query, followed by a list containing a triple $(enc(A), enc(A'), w(A, A'))$ for each edge $\{A, A'\}$ belonging to

$WG(Q)$, where $w(A, A')$ is represented in binary, and $enc(A)$ lexicographically precedes $enc(A')$.

We define a total ordering $<$ on the edges of $WG(Q)$ as follows: for any pair of edges $\{A, A'\}$ and $\{B, B'\}$ of $WG(Q)$, $\{A, A'\} < \{B, B'\}$ if either $w(A, A') > w(B, B')$ or $w(A, A') = w(B, B')$ and the encoding of the pair $\{A, A'\}$ is, as bit string, lexicographically before the encoding of the pair $\{B, B'\}$. Note that $WG(Q)$ is logspace-computable from Q. This can be easily seen in the light of the discussion in Section 2.5. Indeed, a logspace machine may cycle over all pairs A, $A' \in atoms(Q)$ and count the number of common variables. In a similar way, a logspace machine can compute the ordering $<$, that is, the list of all ordered pairs (e, e') of edges of $WG(Q)$ such that $e < e'$.

Given two spanning forests F and F' of $WG(Q)$, let $(e_1, \ldots, e_n)$ and $(e'_1, \ldots, e'_n)$ be, respectively, the lists of the edges of F and of F' ordered according to $<$. We say that F lexicographically precedes F' with respect to the basic ordering $<$, if there exists an index i such that (i) $e_i < e'_i$, and (ii) $e_k = e'_k$ for each $1 \leq k < i$. We denote by $LFF_<(Q)$ the lexicographically first spanning forest of $WG(Q)$ with respect to the basic ordering $<$.

LEMMA 3.1. *$LFF_<(Q)$ is a maximal-weight spanning forest of $WG(Q)$.*

PROOF. Apply Kruskal's [1956] well-known greedy algorithm (see also, e.g., Papadimitriou and Steiglitz [1982]) for computing a maximal-weight spanning forest to $WG(Q)$. Kruskal's algorithm requires to consider the edges of $WG(Q)$ by decreasing weight. Thus the order $<$ is a correct order for Kruskal's algorithm, and, when adopting this order, the algorithm outputs a maximal-weight spanning forest. Moreover, by the chosen edge-ordering $<$, Kruskal's algorithm obviously outputs the lexicographically first spanning forest according to $<$, that is, $LFF_<(Q)$. □

Based on earlier ideas of Reif [1984] and Cook [1985], the following complexity result was recently shown by Nisan and Ta-Shma [1995]:

PROPOSITION 3.2 [NISAN AND TA-SHMA 1995]. *Let $<$ be a given total ordering on the edges of a graph G. Checking whether an edge of G belongs to the lexicographically first spanning forest according to the basic ordering $<$ is in SL.*

Using this result, they could show the following:

PROPOSITION 3.3 [NISAN AND TA-SHMA 1995]

(1) *SL is closed under complementation.*
(2) $L^{SL} = SL$.

From Propositions 3.2 and 3.3, and from the already noted fact that the ordering $<$ is logspace-computable from any given query, we get the following lemma:

LEMMA 3.4. *For any given query Q and pair $\{A_1, A_2\}$ of query atoms, the following tasks are both in SL:*

—*checking whether $\{A_1, A_2\}$ is an edge of $LFF_<(Q)$;*
—*checking whether $\{A_1, A_2\}$ is not an edge of $LFF_<(Q)$.*

Let Q be a query. For each variable $X \in var(Q)$, let $Class(X) = \{A \in atoms(Q) | X$ occurs in $A\}$. The *weight* $w(Q)$ *of query* Q is defined by

$$w(Q) = \sum_{X \in var(Q)} (|Class(X)| - 1).$$

The following results on acyclic queries and join forests are immediate consequences of results by Bernstein and Goodman [1981] (and the fact that their tree-queries are equivalent to the acyclic queries [Beeri et al. 1983]).

PROPOSITION 3.5 [BERNSTEIN AND GOODMAN 1981]. *Let Q be a conjunctive query*:

(1) *The query Q is acyclic if and only if the weight $w(Q)$ is equal to the weight of a maximal-weight spanning forest of $WG(Q)$. Thus, in particular, Q is acyclic if and only if $w(Q) = \Sigma_{\{A,A'\} \in LFF_{<}(Q)} w(A, A')$.*
(2) *If Q is acyclic, then any maximal-weight spanning forest of $WG(Q)$ is a join forest for Q. Thus, in particular, $LFF_{<}(Q)$ is a join forest for Q.*

We now state the main result of this section:

THEOREM 3.6. *The following tasks are in SL*:

(1) *deciding whether a given conjunctive query is acyclic*;
(2) *deciding whether a pair of query atoms $\{A_1, A_2\}$ of an acyclic conjunctive query Q is an edge of its join forest $LFF_{<}(Q)$*;
(3) *deciding whether a pair of query atoms $\{A_1, A_2\}$ of an acyclic conjunctive query Q is not an edge of its join forest $LFF_{<}(Q)$.*

PROOF

Part (1). Let Q be a conjunctive query. By Proposition 3.5(1), it suffices to check whether $w(Q)$ is equal to the weight of $LFF_{<}(Q)$.

This can be done by an L^{SL} machine T as follows: First, T computes $w(Q)$. This is just a logspace task. A counter c is initialized to 0. In an outer loop, T cycles over all the variables of Q. Each variable X is successively addressed by a pointer VP that points to the leftmost occurrence of $enc(X)$ in $enc(Q)$. An inner loop counts the number of atoms of Q where the variable X designated by VP occurs, using another counter c', and increments c by $c' - 1$. Next, T computes the total weight $\Sigma_{\{A,A'\} \in LFF_{<}(Q)} w(A, A')$ of $LFF_{<}(Q)$. A counter cw is initialized to 0. Then, T cycles through all pairs of query atoms $\{A, A'\}$ and decides whether $\{A, A'\}$ belongs to $LFF_{<}(Q)$; if so, cw is incremented by $w(A, A')$. Since computing $w(A, A')$ is a simple counting task in L, the overall process is, by Lemma 3.4, in L^{SL}, and thus, by Proposition 3.3, in SL. Finally, T checks whether $c = cw$ and accepts if so.

Parts (2) and (3) of the theorem follow immediately from Lemma 3.4. □

Since we can trivially associate a conjunctive query to every hypergraph, the above theorem entails the following complexity result on hypergraphs.

COROLLARY 3.7. *Deciding whether a given hypergraph is acyclic is in SL.*

SL is currently the best known upper-bound in terms of *parallel* complexity classes for hypergraph acyclicity testing. An NC^2 parallel algorithm for this task has been developed by Naor et al. [1989]. Note that very efficient *sequential* algorithms were proposed much earlier. Tarjan and Yannakakis [1984] showed that checking whether a hypergraph is acyclic as well as computing a join forest for an acyclic conjunctive query can be done in linear time.

4. *The Complexity of ABCQ and JTREE*

The problem BCQ was shown to be NP-complete by Chandra and Merlin [1977]. Its acyclic version ABCQ (as well as JTREE) was shown to be solvable in polynomial time by Yannakakis [1981]. The precise complexity of ABCQ and JTREE, however, remained unsolved since then. The algorithm for computing the answer to acyclic conjunctive queries proposed by Yannakakis [1981] is inherently sequential. It processes the join tree T of a query Q in a bottom-up manner by performing a sequence of joins (or semi-joins in the case of Boolean queries), one join (or semi-join) for each edge of the tree, starting at the leaves. In particular, if Q has a chain of length n as join tree, then the answer to Q is computed by processing this chain by performing $n - 1$ joins in sequence. Notwithstanding the sequential character of Yannakakis's algorithm, ABCQ could not be shown to be complete for P.

Our main result—proved in the present section—states that ABCQ and JTREE are complete for LOGCFL. Consequently, these problems are highly parallelizable.

THEOREM 4.1. *JTREE and ABCQ are LOGCFL-complete.*

Given that the proof is rather involved, we deal with membership and hardness in two separate subsections.

4.1. JTREE AND ABCQ ARE IN LOGCFL. To prove membership of JTREE and ABCQ in LOGCFL, we use an important characterization of LOGCFL by Sudborough [1977; 1978].

A *nondeterministic auxiliary pushdown automaton* (*NAuxPDA*) [Cook 1971] consists of a nondeterministic Turing machine having a two-way end-marked read-only input tape, one or more read/write worktapes and, in addition, a pushdown store (stack). The space used on the pushdown store is not subject to a space bound on a NAuxPDA.

PROPOSITION 4.2 (SUDBOROUGH [1977; 1978]). *LOGCFL coincides with the class of languages accepted by NAuxPDAs in logarithmic space and polynomial time.*

THEOREM 4.3. *JTREE and ABCQ are in LOGCFL.*

PROOF. We first show that JTREE is in LOGCFL. Let T be a join tree of an acyclic Boolean conjunctive query Q on a database schema $DS = \{R_1, \ldots, R_k\}$. Let $atoms(Q) = \{A_1, \ldots, A_n\}$. At the machine level, T is encoded as described in Section 2.3.

Let $\mathbf{db} = \{r_1, \ldots, r_k\}$ be a database over schema DS and let U be the universe of $\mathbf{db}$. For $1 \leq i \leq n$, we denote by $rel(A_i)$ the relation referred to by query atom A_i, that is, if $A_i = r_j(t_1, \ldots, t_k)$, then $rel(A_i) = r_j$. Moreover, for $1 \leq i \leq n$, $\rho(i)$ denotes the index of the relation $rel(A_i)$, that is, if $rel(A_i) = r_j$, then $\rho(i) = j$.

Recall from Section 2.1 that the tuples (i.e., ground atoms) of each relation r_i of **db** are stored as a list, and refer to the jth tuple of r_i as t_i^j. Each tuple t_i^j is thus unambiguously identified by the pair of indices (i, j) whose storage requires logarithmic space only.

As a preprocessing step, root the tree T in any arbitrary node, say, in A_1. For $1 < i \leq n$, let *parent*(A_i) be the parent node of A_i and let $\pi(i)$ be the index such that $A_{\pi(i)} = \mathit{parent}(A_i)$. Rooting the tree is obviously feasible in logspace. From Proposition 2.7, it follows that the computation of $\pi(i)$ from i is feasible in logspace, too.

Recall that Q evaluates to *true* over **db** if there exists a substitution ϑ: $\mathit{var}(Q) \rightarrow U$ such that, for each atom A_i, $1 \leq i \leq n$, $A_i\vartheta \in \mathbf{db}$.

For an atom $A \in \{A_1, \ldots, A_n\}$, denote by *var*$(A)$ the set of variables occurring in A.

A local substitution for $A \in \{A_1, \ldots, A_n\}$ is a mapping λ: $\mathit{var}(A) \rightarrow U$. For such a substitution λ, let $\mathit{dom}(\lambda) := \mathit{var}(A)$.

Let λ_i and λ_j be two local substitutions for the atoms A_i and A_j, respectively. We say that λ_i agrees with λ_j if for each variable $X \in \mathit{var}(Q)$ occurring in both A_i and A_j, $\lambda_i(X) = \lambda_j(X)$. The following claim is proved in Appendix A.

CLAIM A. *Q evaluates to true over* **db** *if and only if there exist n local substitutions $\lambda_1, \ldots, \lambda_n$, for the atoms $A_1, \ldots, A_n$, respectively, such that,* (*i*) *for $1 \leq i \leq n$, $A_i\lambda_i \in$* **db**, *and* (*ii*) *for $1 < i \leq n$, λ_i agrees with $\lambda_{\pi(i)}$.*

Let A_i be an atom of Q and t a tuple (i.e., ground atom) belonging to **db**. We say that t is a *matching tuple* for A_i if there exists a local substitution λ: $\mathit{var}(A_i) \rightarrow U$ such that $A_i\lambda = t$. Since t is a ground atom, this substitution is uniquely determined by t, and will be denoted by *subst*(A_i, t). If the atom A_i is given by its index i, and if the matching tuple t is given by a pair $\tau = (\rho(i), j)$, where j is an index, then, by abuse of notation, let $\mathit{subst}(i, \tau) := \mathit{subst}(A_i, t)$.

We now describe a polynomial-time logspace NAuxPDA M that accepts input $\langle \mathbf{db}, T \rangle$ if and only if the query Q corresponding to T evaluates to *true* over **db**.

The automaton M has a single worktape on which it maintains four data structures referred to as *registers*:

—*AA* (for *actual atom*), which will contain the index i of the query atom A_i of the currently visited node N of T;

—*PA* (for *parent atom*), which will contain the index p of the atom A_p of the parent node of N;

—*AT* (for *actual tuple*), which will contain a tuple identifier (k, j), where k is a relation index and j a tuple index;

—*PT* (for *parent tuple*), which will also contain a tuple identifier.

The automaton M traverses the tree T in a depth-first left-to-right manner starting from the root by moving downward and upward along the edges of T. The pushdown stack of M always contains the path from the root to the actual node and information about the last child of the actual node which was already visited, allowing the machine to backtrack and thus to traverse the tree in the described manner. We will not describe the (rather trivial) maintenance of this path information on the stack.[7] The stack will also contain further information as described below.

The registers AA, PA, AT, and PT are initially empty.

When the root node A_1 is initially visited, AA is set to 1 and a tuple $tuple(A_1) = t^j_{\rho(1)}$ from relation $r_{\rho(1)} = rel(A_1)$ is nondeterministically chosen whose indices $(\rho(1), j)$ are stored in register AT.

When a node N_i of T labeled with atom A_i is entered by a downward move (coming from its parent node N_p), then the values PA and PT are pushed onto the stack, the assignments $PA := AA$, $PT := AT$, and $AA := i$ are performed, and M chooses nondeterministically a tuple $tuple(A_i) = t^j_{\rho(i)}$ from relation $r_{\rho(i)} = rel(A_i)$ whose indices $(\rho(i), j)$ are stored in register AT. The machine then performs the following subroutines *MATCH*(AA, AT) and *COMPARE* (AA, AT, PA, PT):

—*MATCH*(AA, AT). Check whether the atom identified by AA matches the tuple (i.e., ground atom) identified by AT. If so, RETURN, else HALT and REJECT;

—*COMPARE*(AA, AT, PA, PT). Check whether the local substitutions *subst*(PA, PT) and *subst*(AA, AT) agree. If so, RETURN, else HALT and REJECT.

After successful termination of these subroutines, machine M proceeds as follows:

—If N_i is the rightmost leaf, M stops in the ACCEPT state.

—If N_i is a leaf, but not the rightmost one, then the machine M proceeds by making an upward move in the tree T and revisiting the parent node N_p of N_i.

—If N_i is not a leaf, then M proceeds by visiting the first child of N_i.

If a node N_i is entered by an upward move (i.e., coming from one of its children), then the correct (previously computed) values of AA and AT for N_i are obtained by performing the assignments $AA := PA$, and $AT := PT$, and the correct values of PA and PT are obtained by popping these values from the stack and storing them into the respective registers.

If all children of N_i have been visited, then M moves up to N_i's parent node. Otherwise, M visits the next child of N_i. This completes the description of M.

It is not hard to see that M stops in the ACCEPT state if and only if Q evaluates to *true* over **db**. In fact, if Q evaluates to *true* over **db**, then there exist partial substitutions $\lambda_1, \ldots, \lambda_n$ as required by Claim A. It is clear that by choosing $tuple(A_i) := A_i\lambda_i$ for $1 \leq i \leq n$, machine M will accept. On the other hand, if M accepts, then let $\lambda_i := subst(A_i, tuple(A_i))$ for $1 \leq i \leq n$. Clearly, $A_i\lambda_i \in$ **db**, and thus condition (1) of Claim A is met. Moreover, for $1 < i \leq n$, the positive exit of subroutine *COMPARE*(AA, AT, PA, PT) while visiting the node corresponding to A_i guarantees that λ_i agrees with $\lambda_{\pi(i)}$, and hence also condition (2) of the claim is met. Thus, by Claim A, Q evaluates to *true* over **db**.

Since the four registers AA, AT, PA, and PT contain *indices* rather than atoms or tuples, they only require logarithmic space. Moreover, the subroutines

[7] By using more sophisticated techniques, such as those described in Cook and McKenzie [1987], it is possible to traverse the tree in logspace without using a pushdown store (see Proposition 2.7 of the present paper). Thus, the pushdown store is not really necessary for controlling the tree traversal.

MATCH and *COMPARE* require only deterministic logarithmic space. Thus, M is a logspace NAuxPDA. Since the size of tree T visited by M is polynomial in the size of the input instance (as T is part of the input), and M visits each node of T only a polynomial number of times (once by a downward move, and at most m times by an upward move, where m is the number of children), and since only polynomial time is spent at each visit, M works in polynomial time. By Proposition 4.2, JTREE is thus in LOGCFL.

Denote by JFOREST the decision problem defined in a similar way as JTREE, except that a join forest is given instead of a join tree. It is easy to see that JFOREST $\in$ $\mathrm{L}^{\mathrm{JTREE}}$. In fact, an instance of JFOREST is answered positive if and only if all trees of the forest are yes-instances of JTREE. A logspace oracle machine M^* with JTREE oracle may act as follows to check a JFOREST instance $(\mathbf{db}, F)$, where $\mathbf{db}$ is a database and F a join forest. For each node N of F, M^* first copies $\mathbf{db}$ to the oracle tape and then writes the connected component of N to the oracle tape. (Doing the latter is feasible in logspace, since connectivity in acyclic graphs can be checked in logspace [Cook and McKenzie 1987].) Then M^* makes the oracle query and rejects in case the query is answered negatively. The machine M^* accepts when all nodes N of F have given positive oracle answers. It thus holds that JFOREST is in LOGCFL, by Lemma 2.4.

By Theorem 3.6, the join forest $LFF_{<}(Q)$ of an acyclic query Q is computable via an $\mathrm{L}^{\mathrm{LOGCFL}}$ procedure from Q. Thus, ABCQ is $\mathrm{L}^{\mathrm{LOGCFL}}$-reducible to JFOREST. Since JFOREST is in LOGCFL, ABCQ is in $\mathrm{L}^{\mathrm{LOGCFL}}$(LOGCFL) and thus, by Lemma 2.4, in LOGCFL. $\square$

4.2. LOGCFL-Hardness. For proving the LOGCFL-hardness of JTREE and of ABCQ, we use Venkateswaran's characterization of LOGCFL as the class SAC^1 of problems solvable by logspace-uniform families of semi-unbounded AC^1 Boolean circuits (SAC^1 circuits) as described in Section 2.4.

A *proof tree* T for a semi-unbounded Boolean circuit G on input word w is a rooted tree such that each node N of T is labeled with a gate $gate(N)$ from G and such that the following *labeling conditions* are satisfied:

(1) The root of T is labeled g_{out}, where g_{out} is the output gate of G.
(2) If $gate(N) = g$ for some node N of T, and g is an AND gate of G having fan-in k, then N has exactly k children $N_1, \ldots, N_k$ in T that are labeled by the k predecessor gates of gate g in G, respectively.
(3) If $gate(N) = g$ for some node N of T, and g is an OR gate of G, then N has a unique child in T, and this child is labeled by a gate g' of G that is a predecessor of gate g in G.
(4) Each leaf N of T is labeled by an input gate g of G that corresponds to an input bit 1 in w, or to the constant *true*. More precisely, g satisfies one of the following conditions: g is labeled by z_i in G and the ith bit of w is 1, or g is labeled $\neg z_i$ in G (i.e., g is a NOT gate connected to the ith input bit) and the ith bit of w is 0, or g is labeled *true* in G.

Note that the concept of proof tree as defined above is just a syntactic variant of the concept of *accepting subtree* defined in Venkateswaran [1991]. The following proposition is well known (see, e.g., Fact 1 in Venkateswaran [1991]) and easy to verify.

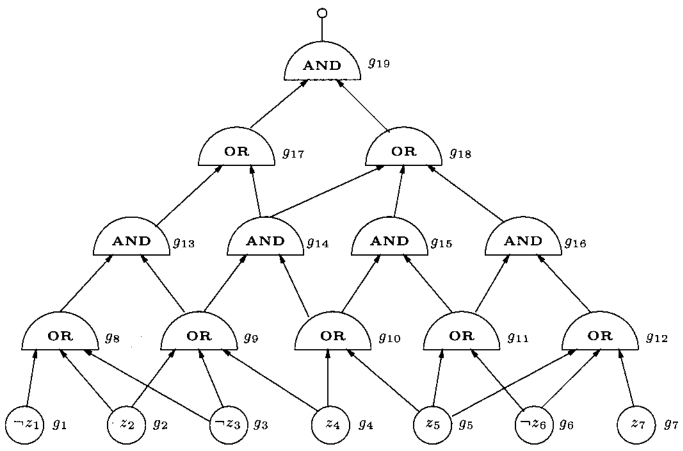


FIG. 4. A circuit G_7 of depth 4.

PROPOSITION 4.4. *A semi-unbounded Boolean circuit G accepts w if and only if there exists a proof tree for G on input w.*

A SAC^1 family $\mathcal{G}$ of Boolean circuits is in *normal form* (NF) if the following conditions are satisfied:

(a) The fan-in of all AND gates in all members of $\mathcal{G}$ is 2.
(b) The nodes of each circuit can be assigned to levels so that circuit-inputs are at level 0, and any gate of level i receives all its inputs from nodes of level $i - 1$.
(c) The circuit is *strictly alternating*, that is, all odd-level gates are OR gates and all even-level (non-input) gates are AND gates.
(d) Each circuit has an odd number of levels, and thus the output gate of each circuit is an AND gate.

Example 4.5. Let $\mathcal{G}$ be a SAC^1 family of Boolean circuits in *normal form*, whose 7th circuit G_7 is the one depicted in Figure 4. It can be verified easily that G_7 (and hence $\mathcal{G}$) accepts the following input w: $w_1 = 0$, $w_2 = 0$, $w_3 = 0$, $w_4 = 0$, $w_5 = 1$, $w_6 = 0$, $w_7 = 1$. Figure 6 shows two proof trees demonstrating that G_7 accepts w.

The following Lemma summarizes well-known normal-form results for SAC^1 circuits that are either trivial or appear in Borodin et al. [1989].

LEMMA 4.6. *For every logspace-uniform SAC^1 family $\mathcal{G}$ of Boolean circuits, there exists a logspace-uniform SAC^1 family $\mathcal{H}$ in NF such that $\mathcal{G}$ and $\mathcal{H}$ accept exactly the same language.*

PROOF. Normal Form property (a) can be achieved by a simple logspace computation that replaces AND gates with fan-in $c > 2$ by a suitable "pyramid" (of fixed shape) of AND gates. For properties (b) and (c), see the proof of Theorem 12 in Borodin et al. [1989]. Property (d) can be achieved by adding—if necessary—a new level with an AND gate as new output gate on top of the original output gate g (getting both its inputs from g). □

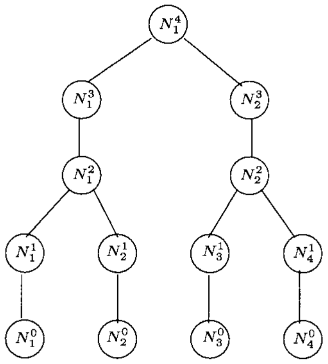


FIG. 5. The "Skeleton" tree $SKEL_4$.

Observe that if G is a circuit in NF of depth d accepting w, then all proof trees for G on w are of the same shape and are labeled with the same type of gates (AND, OR, input) at the corresponding positions. In particular, each proof tree for G on w is of even depth d with an odd number $d + 1$ of levels. The bottom level 0 contains the leaves (labeled by circuit input gates); each node at an even level $i > 0$ is labeled by an AND gate and has two children that are both located at level $i - 1$; each node at an odd level i is labeled OR and has exactly one child, which is located at level $i - 1$. Note that, if i is even and $0 \leq i \leq d$, then each proof tree for G on input w has exactly $2^{(d-i)/2}$ nodes at level i, as well as at level $i + 1$. The number $\rho_d(\ell)$ of nodes at level ℓ ($0 \leq \ell \leq d + 1$) of any proof tree is thus $\rho_d(\ell) = 2^{(d-2\lfloor \ell/2 \rfloor)/2} = 2^{d/2-\lfloor \ell/2 \rfloor}$. It follows that each proof tree for G on w is (as graph) isomorphic to the "skeleton tree" $SKEL_d = (V_d, E_d)$ defined by

$$V_d = \{N_j^i | 0 \leq i \leq d,\ 1 \leq j \leq \rho_d(i)\},$$

and where the set of edges E_d is defined as follows: If $i > 0$ is even, then each node N_j^i, for $1 \leq j \leq \rho_d(i)$, has as predecessors the nodes N_{2j-1}^{i-1} and N_{2j}^{i-1}. If i is odd, then each node N_j^i, for $1 \leq j \leq \rho_d(i)$, has as only predecessor the node N_j^{i-1}. Intuitively, N_j^i represents the jth node, from left to right, of level i of $SKEL_d$ (see Figure 5). The nodes at even nonzero levels of $SKEL_d$ are referred to as AND nodes, while the nodes at odd levels are referred to as OR nodes.

As explained, the skeleton tree $SKEL_d$ represents the common topological structure of all accepting proof trees for G on w *in case* G accepts w. This means that if we drop the label $gate(v)$ from each node v of a proof tree for G on w, then we obtain a tree isomorphic to $SKEL_d$. On the other hand, if we attach to $SKEL_d$ a labeling function *gate* obeying the *labeling rules* 1–4 as described above, then we get a proof tree (see Figure 6). The following lemma summarizes these facts. Its validity follows from the validity of Proposition 4.4 and from the definition of $SKEL_d$.

LEMMA 4.7. *Let $G = (V, E, label)$ be a Boolean circuit of depth d in NF, and let w be an input string for G. Then G accepts w if and only if there exists a function gate: $V_d \to V$, assigning to each node N_j^i of $SKEL_d$ a gate gate (N_j^i) of G, such that the following conditions are satisfied*:

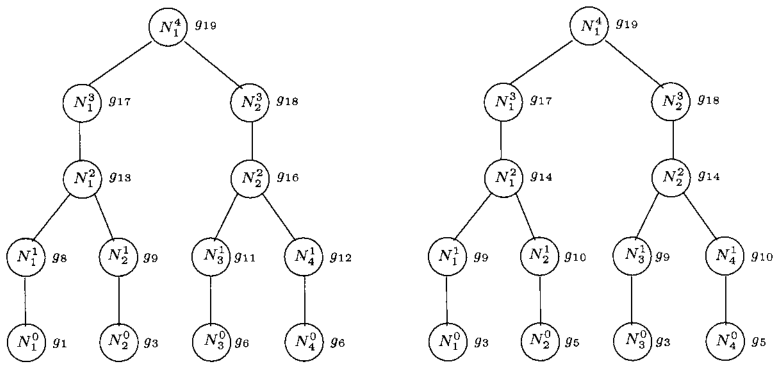


FIG. 6. Labelings of $SKEL_4$ corresponding to proof trees for circuit G_7 on string w in Example 4.5.

(1) *gate*$(N_1^d) = g_{out}$, *the output gate of* G.

(2) *If gate* $(N_j^i) = g$ *and* N_j^i *is an AND node of* $SKEL_d$ (*i.e.*, i *is even*), *then the children* N_{2j-1}^{i-1} *and* N_{2j}^{i-1} *are assigned by the labeling function gate the predecessor gates* g' *and* g'' *of gate* g *in* G, *respectively*.

(3) *If gate*$(N_j^i) = g$ *and* N_j^i *is an OR node of* $SKEL_d$ (*i.e.*, i *is odd*), *then the unique child* N_j^{i-1} *of* N_j^i *is assigned a gate* $g' =$ *gate*(N_j^{i-1}) *of* G *that is a predecessor of gate* g *in* G.

(4) *For each leaf* N_j^0 *of* $SKEL_d$, *gate*$(N_j^0) = g$, *where* g *is an input gate with label* l *in* G *satisfying one of the following conditions*: l *is the constant true*, *or* $l = z_i$ *and the ith bit of* w *is* 1, *or* $l = \neg z_i$ *and the ith bit of* w *is* 0.

We have now built all the necessary machinery for proving our hardness result.

THEOREM 4.8. *Both problems ABCQ and JTREE are hard for LOGCFL under logspace reductions. These problems remain LOGCFL-hard even in case the query involves only binary relations, that is, in case the associated acyclic hypergraph is an acyclic graph.*

PROOF. By Proposition 2.2 and Lemma 4.6, it is sufficient to show that the problem of deciding whether an input string w of length n is accepted by the nth circuit G_n of a logspace-uniform family $\mathcal{G}$ of SAC^1 circuits in NF, can be reduced in logspace to instances of ABCQ and JTREE.

Let $\mathcal{G}$ be a logspace-uniform family of SAC^1 circuits in NF and let w be a binary string of length n. Denote by d the depth of the nth circuit G_n of $\mathcal{G}$. (Note that the depth of a circuit G_n in normal form can be easily computed in logspace from G_n, and thus from 1^n; in fact, the depth of G_n is equal to the length of any path from a leaf of G_n to the root, i.e., it is the number of levels of G_n.)

We now construct an acyclic Boolean query Q and a database instance **db** such that Q evaluates to *true* over **db** if and only if G_n accepts w. The hypergraph $H(Q)$ associated to Q is actually a tree isomorphic to the skeleton $SKEL_d$ of G_n (see Figures 5 and 8). The variables of Q correspond exactly to the nodes of $SKEL_d$. If Q is satisfied over **db**, then each satisfying assignment of data values to the variables of Q corresponds to a correct labeling of the nodes of the skeleton $SKEL_d$ by gate names such that the resulting labeled tree is a proof tree for G_n on w.

leftand:	**rightand:**	**or:**
$leftand(g_{19}, g_{17})$	$rightand(g_{19}, g_{18})$	$or(g_{17}, g_{13})$
$leftand(g_{13}, g_{8})$	$rightand(g_{13}, g_{9})$	$or(g_{17}, g_{14})$
$leftand(g_{14}, g_{9})$	$rightand(g_{14}, g_{10})$	$or(g_{18}, g_{14})$
$leftand(g_{15}, g_{10})$	$rightand(g_{15}, g_{11})$	$\vdots$
$leftand(g_{16}, g_{11})$	$rightand(g_{16}, g_{12})$	$or(g_{12}, g_{7})$

FIG. 7. The database instance **db** = {*leftand*, *rightand*, *or*} for Example 4.5.

We first describe the construction of the database **db**. The universe of data values U of **db** is the set of all gates of G_n (i.e., the set of nodes of the graph G_n), say $U = \{g_1, \ldots, g_m\}$. The database **db** contains three binary relations *leftand*, *rightand*, and *or*. Intuitively, for an AND gate g of G_n, relation *leftand* encodes the connection between g and its first (left) predecessor and relation *rightand* the connection between g and its second (right) predecessor. The relation *or* encodes the OR branchings of the circuit G_n. Formally, for each AND gate g_a of G_n with children g_b and g_c, *leftand* contains the tuple $\langle g_a, g_b \rangle$ and *rightand* contains the tuple $\langle g_a, g_c \rangle$. The relations *leftand* and *rightand* contain no further tuples. For each OR gate g_a of G_n at some level $i > 1$, and for each child g_b of g_a, *or* contains the tuple $\langle g_a, g_b \rangle$. Further, each OR gate g_a at level 1 of G_n, and each child g_b of g_a give rise to a tuple $\langle g_a, g_b \rangle$ of the relation *or* if and only if the circuit-input gate g_b is labeled by the constant *true* or gets value *true* on w, that is, if and only if one of the following conditions applies:

—g_b is labeled z_i and the ith bit of string w is 1;
—g_b is labeled $\neg z_i$ and the ith bit of w is 0;
—g_b is labeled *true*.

There are no further tuples in the relation *or*. We have completely described **db**. For an example, see Figure 7.

Clearly, **db** can be computed from G_n and w by a logspace procedure. Moreover, the family $\mathcal{G}$ is logspace-uniform. Thus, by definition of logspace uniformity (see Section 2.4), G_n can be computed in logspace from w. Summarizing, by Proposition 2.8, **db** can be generated in logspace from w.

Let us define the query Q, which is obtained in an extremely simple way from $SKEL_d$. Recall that V_d denotes the set of nodes of $SKEL_d$. For each node $N_j^i \in V_d$, define a variable $var(N_j^i) = X_j^i$. Let $var(SKEL_d) = \{var(N) | N \in V_d\}$.

Then

$$Q = \bigwedge_{N_j^i \in V_d, i > 0} form(N_j^i),$$

where $form(N_j^i)$ is a conjunction of atoms defined as follows:

—If N_j^i is labeled AND (i.e., i is even), then $form(N_j^i)$ is the conjunction $leftand(var(N_j^i), var(M_1)) \wedge rightand(var(N_j^i), var(M_2))$, where M_1 and M_2 are the children of N_j^i in $SKEL_d$. Since we can name these children precisely as

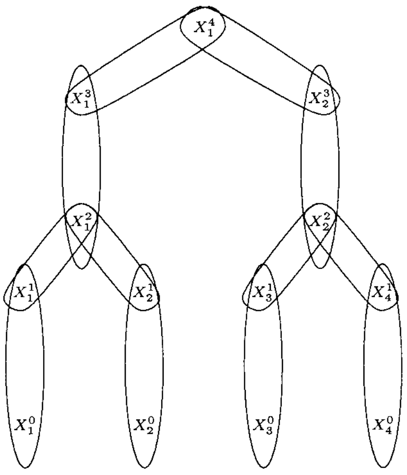


FIG. 8. Hypergraph corresponding to query Q_4.

$M_1 = N_{2j-1}^{i-1}$ and $M_2 = N_{2j}^{i-1}$, we have: $form(N_j^i) = leftand(X_j^i, X_{2j-1}^{i-1}) \wedge rightand(X_j^i, X_{2j}^{i-1})$.

—If N_j^i is labeled OR, then $form(N_j^i) = or(var(N_j^i), var(M))$, where M is the only child of N_j^i in $SKEL_d$. Since we can name this child precisely as $M = N_j^{i-1}$, we have: $form(N_j^i) = or(X_j^i, X_j^{i-1})$.

For example, from $SKEL_4$, we get the following acyclic Boolean conjunctive query Q_4:

$$Q_4 = \begin{cases} leftand(X_1^4, X_1^3) \wedge rightand(X_1^4, X_2^3) \wedge \\ or(X_1^3, X_1^2) \wedge or(X_2^3, X_2^2) \wedge \\ leftand(X_1^2, X_1^1) \wedge rightand(X_1^2, X_2^1) \wedge leftand(X_2^2, X_3^1) \wedge rightand(X_2^2, X_4^1) \wedge \\ or(X_1^1, X_1^0) \wedge or(X_2^1, X_2^0) \wedge or(X_3^1, X_3^0) \wedge or(X_4^1, X_4^0). \end{cases}$$

We denote by $var(Q)$ the variables occurring in Q. It holds: $var(Q) = var(SKEL_d)$.

Note that the hypergraph $H(Q)$ is a tree, hence the query Q is clearly acyclic. See Figure 8 for a picture.

The list of atoms of the query Q can be constructed by the following very simple logspace computation: for each $0 \leq i \leq d$ and $1 \leq j \leq \rho_d(i)$, do

—if i is even, then output the atoms $leftand(X_j^i, X_{2j-1}^{i-1})$ and $rightand(X_j^i, X_{2j}^{i-1})$;
—if i is odd, then output the atom $or(X_j^i, X_j^{i-1})$.

We claim that Q evaluates to *true* over **db** if and only if G_n accepts w.

We first prove the *if* part of this claim. If G_n accepts w, then, by Lemma 4.7, there exists a labeling function $gate\colon V_d \rightarrow U$ fulfilling the four properties of the Lemma. Let ϑ be the following substitution assigning to each variable in $var(Q)$ a data value from U: for each $X_j^i \in var(Q) = var(SKEL_d)$, $\vartheta(X_j^i) = gate(N_j^i)$. Observe that, by definition of **db**, for each atom A of Q, the substitution instance $A\vartheta$ occurs in any relation *leftand*, *rightand* or *or* of **db**. Thus, all ground atoms of $Q\vartheta$ are true in the database and, consequently, Q evaluates to *true* over **db**.

We now prove the *only-if* part of the claim.

Assume that Q evaluates to *true* over **db**. Then there exists a substitution ϑ: $var(Q) \to U$ such that for each atom A of Q, $A\vartheta \in$ **db**.

Let *gate*: $V_d \to U$ be the labeling function attaching the gate name $gate(N^i_j) := \vartheta(X^i_j)$ to each node N^i_j of $SKEL_d$. It suffices to prove that the labeling function *gate* fulfills the properties (1)–(4) of Lemma 4.7.

Property 1. $gate(N^d_1) = g_{out}$, the output gate of G.

Before proving this property, let us make a simple observation deriving directly from the definition of **db**. Call a sequence $g^1, g^2, \ldots, g^k$ of distinct gate names (i.e., of elements of U) an *alternating chain* of length $k - 1$ of **db** rooted in g^1, if **db** contains the ground atoms $leftand(g^1, g^2)$, $or(g^2, g^3)$, $leftand(g^3, g^4)$, and so on. We observe that, by construction of **db**, for all alternating chains $g^1, g^2, \ldots, g^{d+1}$ of length d rooted in g^1, it holds that g^1 is identical to g_{out}. In fact, it is easily verified that any alternating chain of **db** rooted in g^1 of length ℓ coincides with the gates encountered on a downward path of length ℓ starting at some AND gate g^1 in G_n. Note, however, that the downward paths of length d of G_n all start at the output gate g_{out}.

To prove Property 1, consider the leftmost branch of $SKEL_d$. This branch is made of the nodes

$$N^d_1, N^{d-1}_1, N^{d-2}_1, \ldots, N^1_1, N^0_1.$$

Let $g^i := \vartheta(X^i_1) = gate(N^i_1)$ for $0 \le i \le d$. By construction of the query Q, the following atoms are all in **db**: $leftand(g^d, g^{d-1})$, $or(g^{d-1}, g^{d-2})$, $leftand(g^{d-2}, g^{d-3})$, $\ldots$, $leftand(g^{2j}, g^{2j-1})$, $or(g^{2j-1}, g^{2j-2})$, $\ldots$, $leftand(g^2, g^1)$, $or(g^1, g^0)$. Thus, the sequence $g^d, g^{d-1}, \ldots, g^0$ is an alternating chain of length d of **db** rooted in g^d. It follows that the label g^d (i.e., $gate(N^d_1)$) is identical to g_{out}, that is, to G_n's output gate.

Property 2. If gate$(N^i_j) = g$ and N^i_j is an AND node of $SKEL_d$ (i.e., i is even), then the children N^{i-1}_{2j-1} and N^{i-1}_{2j} are assigned by the labeling function gate the predecessor gates g' and g'' of gate g in G, respectively.

Let N^i_j be an AND node of $SKEL_d$, and let $g = gate(N^i_j)$, that is, $\vartheta(X^i_j) = g$. Note that, by construction of **db**, g occurs only in two atoms of **db**. These atoms are of the form $leftand(g, g_1)$ and $rightand(g, g_2)$, where g_1 and g_2 are the gate names of the first and of the second predecessor of gate g in G_n, respectively. Since $\vartheta(X^i_j) = g$, there is only one choice for transforming the query atoms of the conjunction $form(N^i_j) = leftand(X^i_j, X^{i-1}_{2j-1}) \wedge rightand(X^i_j, X^{i-1}_{2j})$ via substitution ϑ into ground atoms contained in the database, and it must hold that $\vartheta(X^{i-1}_{2j-1}) = g_1$ and $\vartheta(X^{i-1}_{2j}) = g_2$. It follows that $gate(N^i_{2j-1}) = g_1$ and $gate(N^i_{2j}) = g_2$, which proves Property 2.

Property 3. *If gate*$(N^i_j) = g$ *and* N^i_j *is an OR node of* $SKEL_d$ (*i.e., i is odd*), *then the unique child* N^{i-1}_j *of* N^i_j *is assigned a gate* $g' = gate(N^{i-1}_j)$ *of* G *that is a predecessor of gate* g *in* G.

Let N^i_j, $i > 0$, be an OR node of T such that $gate(N^i_j) = g$. By definition of $SKEL_d$, N^i_j has as unique child in $SKEL_d$ the node N^{i-1}_j. It thus suffices to show that N^{i-1}_j is labeled with a predecessor of gate g in G_n. Note that $\vartheta(X^i_j) = g$. The only *or*-atoms of **db** in which g occurs are the atoms $or(g, g_1)$, $\ldots$, *or*(g, g_k), where $g_1, \ldots, g_k$ are the gate names of the children of gate g in G_n. Query Q contains an atom *form*(N_j^i) = *or*(X_j^i, X_j^{i-1}). Since $\vartheta(X_j^i) = g$, and *or*$(X_j^i, X_j^{i-1})\vartheta \in$ **db**, it holds that $\vartheta(X_j^{i-1}) \in \{g_1, \ldots, g_k\}$. Therefore, *gate*$(N_j^{i-1}) \in \{g_1, \ldots, g_k\}$, and property (3) is proved.

Property 4. *For each leaf N_i^0 of SKEL$_d$, gate(N_i^0) is an input gate of G, which either is labeled by the constant true or gets value true on w.*

Let N_i^0 be an input node (i.e., leaf) of T. Then N_i^0 is the unique child of the OR node N_i^1. Let *gate*$(N_i^1) = \vartheta(X_i^1) = g_a$, and let *gate*$(N_i^0) = \vartheta(X_i^0) = g_b$. Since Q contains the atom *form*(N_i^1) = *or*(X_i^1, X_i^0), and *form*$(N_i^1)\vartheta \in$ **db**, it holds that *or*$(g_a, g_b) \in$ **db**. By construction of relation *or* of **db**, this holds if and only if g_b is labeled by the constant *true* or gets value *true* on w.

This concludes the proof of the *only-if* part of our claim. We have thus shown that ABCQ is hard for LOGCFL under logspace reductions.

To see that the same holds for JTREE, it is sufficient to observe that a join tree $JT(Q) = (V, E)$ for the query Q defined above is the tree having $V = \{atoms(Q)\}$ as its set of vertices, and whose set of edges is

$$E = \{\{A_1, A_2\} \subseteq V | var(A_1) \cap var(A_2) \neq \emptyset\}.$$

The join tree $JT(Q)$ can be computed in logspace from Q as follows: First, copy the atoms of Q to the output tape. Then, let two pointers cycle over all pairs A_1, $A_2 \in atoms(Q)$, where $enc(A_1)$ lexicographically precedes $enc(A_2)$. For each such pair A_1, A_2, check whether $var(A_1) \cap var(A_2) \neq \emptyset$ (this check is feasible in logspace), and, if so, output the edge $\{A_1, A_2\}$. □

Observe that the join tree $JT(Q)$ associated with the conjunctive query Q in the above proof has depth logarithmic in the size of Q. Thus, LOGCFL-hardness holds even for instances of JTREE, where the join tree has logarithmic depth (compared to its size).

Note that we have also found a translation that directly translates the membership problem for any fixed CFG in Chomsky Normal Form into an ABCQ instance. This translation is somewhat more complicated and will be presented elsewhere.

From Theorems 4.3 and 4.8, we thus get the following:

COROLLARY 4.9. *ABCQ and JTREE are both complete for LOGCFL under logspace reductions.*

Recall from Section 2.2 that the notion of hypergraph acyclicity adopted in this paper is the standard notion used in the database literature and corresponds to Fagin's [1983] α-acyclicity. However, several other notions of hypergraph acyclicity exist, that all generalize the concept of graph-acyclicity [Fagin 1983]. Examples are β-acyclicity, γ-acyclicity, and Berge-acyclicity [Berge 1976; Fagin 1983].

It can be seen that the LOGCFL-completeness of acyclic Boolean conjunctive queries does not only hold for α-acyclic hypergraphs, but does actually hold for any of the above-mentioned types of hypergraph acyclicity. Membership in LOGCFL follows from the fact that, as noted in Section 2.2, the class of α-acyclic hypergraphs is the largest class, comprising all other acyclic classes. Hardness for LOGCFL follows from the fact that our hardness result (Theorem 4.8) holds even for queries whose associated hypergraph is actually a graph. On graphs, all notions of hypergraph acyclicity coincide.

5. *Equivalent Database and AI Problems*

In this section, we describe several important decision problems in the fields of databases and AI that involve hypergraphs. All these problems are logspace-equivalent to BCQ and are thus NP-complete. The NP-completeness of these problems is well-known. Moreover, one can find simple reductions between each of these problems and BCQ and vice versa that preserve hypergraph acyclicity, whence the acyclic version of all these problems is complete for LOGCFL and thus highly parallelizable. For each problem $\mathcal{P}$, we denote by A$\mathcal{P}$ its acyclic version.

The first problem CQOT deals with general (and not necessarily Boolean) conjunctive queries. It is a generalization of ABCQ.

Name: CQOT (conjunctive query output tuple).

Instance: Conjunctive query Q, tuple (i.e., ground atom) t, database instance **db**.

Question: Is t in the answer of Q on **db**?

Associated Hypergraph: The hypergraph $H(Q)$.

THEOREM 5.1. *The acyclic version ACQOT of CQOT is complete for LOGCFL.*

PROOF. To see that ACQOT is in LOGCFL, transform each instance I of ACQOT in logspace into an instance I' of ABCQ as follows. Denote by *head*(Q) and *body*(Q) the head and body of Q, respectively. Check in logspace whether t matches *head*(Q). If t does not match *head*(Q), then let I' be any trivial no-instance of ABCQ (e.g., an instance of ABCQ with an empty database). Otherwise, let ϑ be the substitution defined on the variables occurring in *head*(Q), such that *head*$(Q)\vartheta = t$. Let $I' = (\mathbf{db}, \mathit{body}(Q)\vartheta)$. Clearly, I is a yes-instance of ACQOT if and only if I' is a yes-instance of ABCQ. Moreover, the transformation is logspace-computable. LOGCFL-hardness follows from the fact that ABCQ is a special case of ACQOT. □

What about *computing* the complete answer $Q(\mathbf{db})$ to an acyclic conjunctive query Q over a database? This search problem requires exponential time in the worst case, since $Q(\mathbf{db})$ may contain an exponential number of tuples. The problem is, however, *output-polynomial*: Yannakakis's [1981] algorithm computes $Q(\mathbf{db})$ in time polynomial in the size of $Q(\mathbf{db})$ plus the size of the input instance. This computation problem becomes tractable—and highly parallelizable—if the number of variables that may occur in the head *head*(Q) of Q is bounded by a constant (a parallel database algorithm solving this problem with a logarithmic number of parallel join operations is presented in Section 7.3).

Our next decision problem is *Conjunctive Query Containment*, a very well-known problem since the early days of database theory (for references, see, e.g., Ullman [1989]).

Let Q_1 and Q_2 be two conjunctive queries. Q_1 is *contained* in Q_2, denoted $Q_1 \subseteq Q_2$, if and only if for every database instance **db**, $Q_1(\mathbf{db}) \subseteq Q_2(\mathbf{db})$.

Name: CQ-CONTAINMENT (conjunctive query containment).

Instance: Conjunctive queries Q_1 and Q_2.

Question: Is Q_1 contained in Q_2?

Associated Hypergraph: The hypergraph $H(Q_2)$.

As pointed out in Kolaitis and Vardi [2000], conjunctive-query containment is a topic of high relevance to current database research issues such as the problem of answering queries using materialized views [Levy et al. 1995; Rajamaran et al. 1995] and the related problem of integrating information from heterogeneous sources (see Ullman [1997] for a survey).

NP-completeness of CQ-CONTAINMENT was proven in Chandra and Merlin [1977]. The acyclic version ACQ-CONTAINMENT of this problem asks whether an arbitrary query Q_1 is contained in an acyclic query Q_2.

A *query folding* for an instance (Q_1, Q_2) of CQ-CONTAINMENT is a substitution ϑ replacing each variable $X \in var(Q_2)$ by suitable terms such that, for each atom A occurring in the body of Q_2, there is an atom B in the body of Q_1 with $A\vartheta = B$ and $head(Q_2)\vartheta = head(Q_1)$. The following proposition is well-known (see, e.g., Theorem 14.1 in Ullman [1989] or Theorem 11.10 in Maier [1986]):

PROPOSITION 5.2. *$Q_1 \subseteq Q_2$ if and only if there exists a query folding for (Q_1, Q_2).*

The next proposition (also well known, see, e.g., Chandra and Merlin [1977] for an equivalent formulation) follows as a simple corollary from Proposition 5.2. It establishes a very tight relationship between CQ-CONTAINMENT and CQOT, and thus also between CQ-CONTAINMENT and BCQ.

PROPOSITION 5.3. *Let $Q_1 = A_0 \leftarrow A_1 \wedge \cdots \wedge \cdots \wedge A_n$ and Q_2 be conjunctive queries. Then $Q_1 \subseteq Q_2$ if and only if $A_0\vartheta$ belongs to $Q_2(\mathbf{db^*})$, where ϑ is a ground substitution assigning to each variable X of $var(Q_1)$ a fresh constant c_X, and $\mathbf{db^*} = \{A_1\vartheta, A_2\vartheta, \ldots, A_k\vartheta\}$. The instance (Q_1, Q_2) of CQ-CONTAINMENT is thus equivalent to the instance $(Q_2, A_0\vartheta, \mathbf{db^*})$ of CQOT.*

It follows that, if Q_2 has an acyclic query hypergraph, then the test $Q_1 \subseteq Q_2$ is feasible in polynomial time by applying Yannakakis's algorithm. Thus, ACQ-CONTAINMENT is polynomial. This polynomiality result was also observed by Qian [1996], and independently by Chekuri and Rajaraman [2000], who developed a (sequential) special containment-checking algorithm for the acyclic case, which is then extended to the more general setting where Q_2 has bounded treewidth.

The following theorem settles the exact complexity of ACQ-CONTAINMENT.

THEOREM 5.4. *ACQ-CONTAINMENT is complete for LOGCFL.*

PROOF. The reduction described in Proposition 5.3 from CQ-CONTAINMENT to CQOT is clearly feasible by a logspace transducer. Moreover, it trivially preserves acyclicity. Thus ACQ-CONTAINMENT is logspace-reducible to the LOGCFL-complete problem ACQOT and is thus in LOGCFL.

To prove LOGCFL-hardness, let us reduce ABCQ to ACQ-CONTAINMENT. This turns out to be very easy. Take an instance $(\mathbf{db}, Q)$ of ABCQ, where Q is, without loss of generality, a conjunction of literals. Let $A_1, \ldots, A_n$ be the ground atoms (tuples) of **db**. Let q be a new propositional atom. Form the two queries Q_1 and Q_2 as follows:

$$Q_1 = q \leftarrow \bigwedge_{1 \le i \le n} A_i, \text{ and}$$

$$Q_2 = q \leftarrow Q.$$

Clearly, by Proposition 5.2, $Q_1 \subseteq Q_2$ if and only if Q evaluates to *true* over **db**. The transformation is obviously feasible in logspace (it is a mere rewriting). Thus, ACQ-CONTAINMENT is hard for LOGCFL. □

We now describe the clause subsumption problem SUBS, which is of central importance in the field of Automated Theorem Proving [Chang and Lee 1973]. Indeed, the performance of automated theorem provers can be drastically improved in reducing the search space by eliminating redundant formulas [Bachmair et al. 1996], and a key deletion strategy is based on subsumption tests [Wos et al. 1991].

Formally, a (general) *clause* is a set of literals, that is, of positive or negated atoms. A clause represents the logical disjunction of its literals. Note that the vocabulary here may contain function symbols too. A clause $\{A_1 \ldots, A_k, \neg B_1, \ldots, \neg B_r\}$ can also be written in implicational form, $A_1 \vee \cdots \vee A_k \leftarrow B_1 \wedge \cdots \wedge B_r$. Each conjunctive query is thus a clause satisfying the following restrictions: it is Horn (i.e., has at most one literal in its head) and it does not contain any function symbol.

A clause C subsumes a clause D, denoted by $C \geq D$, if there exists a substitution ϑ assigning a term to each variable of C such that $C\vartheta \subseteq D$. The problem SUBS is formally stated as follows:

Name: SUBS (clause subsumption).

Instance: Clauses C and D.

Question: Does C subsume D?

Associated Hypergraph: $H = (V, E)$, where $V = var(C)$ is the set of variables occurring in C and $E = \{var(L) | L \in C\}$, where $var(L)$ denotes the set of variables occurring in literal L.

By Proposition 5.2, SUBS is a generalization of CQ-CONTAINMENT. In particular, any instance (C_1, C_2) of CQ-CONTAINMENT is equivalent to the instance (C_2, C_1) of SUBS. The acyclic variant ASUBS of SUBS appears to be of high practical value, since many of the clauses generated by a theorem prover are acyclic (in fact most initial theories contain mainly acyclic clauses and acyclicity is often preserved by resolution).

THEOREM 5.5. *ASUBS is complete for LOGCFL.*

PROOF. Hardness follows trivially from the LOGCFL-completeness of ACQ-CONTAINMENT and the fact that ACQ-CONTAINMENT is a special case of ASUBS. Let us prove membership. We denote by ABCQ^f and ACQOT^f the extensions of the respective problems ABCQ and ACQOT, where function symbols are permitted both in database ground atoms and in the queries. For any clause K, let K^+ be the disjunction $\bigvee_{L \in K} L^+$, where $L^+ = L$, if L is a positive literal and $L^+ = p'(t_1, \ldots, t_m)$, if $L = \neg p(t_1, \ldots, t_m)$, p' being a new predicate symbol uniquely associated with predicate p.

Consider an instance (C, D) of ASUBS. It is clear that $C \geq D$ if and only if $C^+ \geq D^+$. Note that Proposition 5.2 was actually shown to hold for queries with function symbols [Ullman 1989]. Thus, by Proposition 5.3, the instance (C, D) of ASUBS is (logspace-)equivalent to an instance of ACQOT^f, and therefore to an instance of ABCQ^f. It thus remains to see that ABCQ^f is in LOGCFL. This is seen by showing that the proof of Theorem 4.3 holds in the presence of function symbols. An inspection of that proof reveals that a sufficient condition for its "survival" in the presence of function symbols is that suitable generalizations of the subroutines *MATCH* and *COMPARE* to the setting with function symbols can be given, which are feasible in nondeterministic logspace. The latter follows in turn from the well-known fact that *term matching* is in NL even in the presence of function symbols, and that the matching substitutions can be computed in nondeterministic logspace [Dwork et al. 1984]. (Note that this holds even if terms are represented compactly as directed acyclic graphs.) $\square$

Finally, let us switch to the problem of *constraint satisfaction*, another fundamental problem in Artificial Intelligence.

There are several equivalent definitions of constraint satisfaction. The following is taken almost verbatim from Jeavons et al. [1997]. An instance of a *constraint satisfaction problem* (*CSP*) (also *constraint network*) consists of

—a finite set *Var* of variables;

—a finite domain U of values;

—a set of constraints $\mathscr{C} = \{C_1, C_2, \ldots, C_q\}$.

Each constraint C_i is a pair (S_i, r_i), where S_i is a list of variables of length m_i called the *constraint scope*, and r_i is an m_i-ary relation over U, called the *constraint relation*. (The tuples of r_i indicate the allowed combinations of simultaneous values for the variables S_i).

A *solution* to a CSP instance is a substitution ϑ: $Var \rightarrow U$ such that, for each $1 \leq i \leq q$, $S_i\vartheta \in r_i$. The decision problem *constraint satisfiability* (CS) is then defined as follows:

Name: CS (constraint satisfiability).

Instance: CSP instance $I = (Var, U, \mathscr{C})$ as described.

Question: Does I have a solution?

Associated Hypergraph: $H = (V, E)$, where $V = Var$, and $E = \{var(S) | C = (S, r) \in \mathscr{C}\}$, where $var(S)$ denotes the set of variables in list S of constraint C.

From this definition, it is obvious that the problem CS is just a syntactic variant of BCQ. In particular, the acyclic version ACS of CS is polynomially solvable. This is well known, and, more generally, the connection between CSPs in the AI setting and database theory has been well acknowledged in the literature (see, e.g., Gyssens et al. [1994], Kolaitis and Vardi [2000], and Pearson and Jeavons [1997], where the polynomiality result is extended to generalizations of the acyclic case.). For the complexity of ACS, we (trivially) have the following:

THEOREM 5.6. *ACS is complete for LOGCFL.*

We have thus shown in this section that four interesting decision problems are complete for LOGCFL. This means that all these problems are highly parallelizable as they are contained in AC^1 and in NC^2. It is thus easy to design logtime CRCW PRAM algorithms using a polynomial number of processors, or $\log^2$-time EREW PRAM algorithms with a polynomial number of processors solving these problems.

To the best of our knowledge, nobody had previously determined the exact complexity of any of these four problems. Actually, for ACQOT, ACQ-CONTAINMENT, and ASUBS, even membership in NC was unknown. NC algorithms for restricted versions of CS have been presented by Kasif and Delcher [1994] and by Zhang and Mackworth [1993], and will be discussed in Section 7.

6. *Tractable Cyclic Queries*

As explained in Section 1.2, many relevant cyclic queries are—in a precise sense—close to acyclic queries because they can be decomposed via low-bandwidth decompositions into acyclic queries.

The main classes of bounded-width queries considered in database theory and in artificial intelligence are the following:

—*Queries of bounded treewidth*. Treewidth is the best-known graph-theoretic measure of tree-similarity. The concept of treewidth is based on the notion of *tree-decomposition* of a graph defined below in this section. The concept of treewidth is easily generalized to hypergraphs and thus to conjunctive queries. Conjunctive queries of bounded treewidth can be answered in polynomial time [Chekuri and Rajaraman 2000]. For each fixed k, deciding whether a query has treewidth k is in LOGCFL [Wanke 1994].

—*Queries of bounded degree of cyclicity*. This concept was introduced by Gyssens et al. [1994] and Gyssens and Paredaens [1984] and is based on the notion of *hinge-tree decomposition*. The smaller the degree of cyclicity of a hypergraph, the more the hypergraph resembles an acyclic hypergraph. Queries of bounded degree of cyclicity can be recognized and processed in polynomial time [Gyssens et al. 1994].

—*Queries of bounded query-width*. This notion is based on the concept of *query decomposition* [Chekuri and Rajaraman 2000]. Any tree-decomposition of width k, as well as any hinge-tree decomposition of width k, is also a query-decomposition of width k, but not vice-versa. Thus, query decompositions are the most general (i.e., most liberal) decompositions. It follows from results in Chekuri and Rajaraman [2000] that queries of bounded query-width can be answered in polynomial time, once a query-decomposition is given.

In this section, we extend our LOGCFL-completeness result for JTREE to the above bounded-width classes. We thus show that answering queries that are given together with a tree-decomposition, or with a hinge-tree decomposition or with a query-decomposition of bounded width, is LOGCFL-complete.

For queries of bounded treewidth, we are able to state an even stronger result (generalizing ABCQ rather than JTREE): answering queries of bounded treewidth is LOGCFL-complete, even if a tree-decomposition is not given together with the query. This stronger result relies on the recent results that, for fixed k, recognizing graphs of bounded treewidth is in LOGCFL [Wanke 1994], and

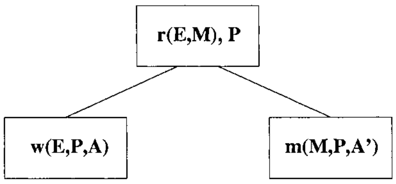


FIG. 9. A 2-width decomposition of Q_1.

computing a tree-decomposition of width k is feasible by a logspace procedure with an oracle in LOGCFL [Gottlob et al. 2000a].

All complexity results in this section are formulated for the evaluation problem of Boolean conjunctive queries. However, all these results clearly carry over to the corresponding versions of all problems described in Section 5, given that those problems are all logspace-equivalent to BCQ under preservation of the associated hypergraph.

For technical reasons, we will first consider bounded query-width (Section 6.1), then bounded treewidth (Section 6.2), and finally bounded degree of cyclicity (Section 6.3).

6.1. BOUNDED QUERY-WIDTH. The notion of *query-width* has been proposed in Chekuri and Rajaraman [2000] to single out large classes of tractable instances of the conjunctive query containment problem. As CQ-CONTAINMENT containment is strictly related to BCQ (see Section 5), the notion of query-width turns out to be a useful tool also to single out tractable classes of conjunctive queries.

The following definition is a slight modification of the original definition given in Chekuri and Rajaraman [2000].

Definition 6.1. A *query decomposition* of a conjunctive query Q is a pair $\langle T, \lambda \rangle$, where $T = (N, E)$ is a tree, and λ is a labeling function which associates to each vertex $p \in N$ a set $\lambda(p) \subseteq (atoms(Q) \cup var(Q))$, such that the following conditions are satisfied:

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

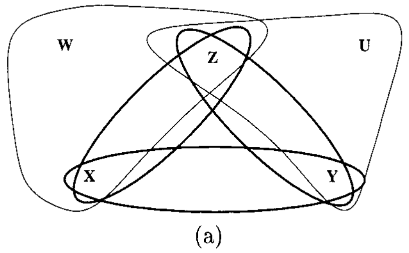
$$\{p \in N | Y \in \lambda(p)\} \cup \{p \in N | Y \text{ occurs in some atom } A \in \lambda(p)\}$$

induces a (connected) subtree of T.

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

Example 6.2. Figure 9 shows a 2-width query decomposition for the cyclic query of Example 1.1. Since the query-width of any cyclic query is strictly greater than one, it follows that the query-width of Q_1 is two. Note that this query decomposition is not pure, because the label of the topmost vertex is $\{r(E, M), P\}$ and contains also the variable P.

Consider the following query Q_4:



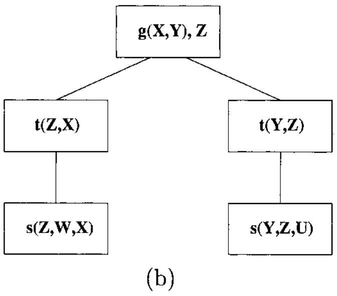


FIG. 10. (a) Hypergraph $H(Q_4)$, and (b) a 2-width decomposition of Q_4.

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

The query Q_4 is cyclic, since its hypergraph $H(Q_4)$ (Figure 10(a)) is cyclic. Thus, $qw(Q_4) > 1$ holds. Moreover, a 2-width decomposition of Q_4 is shown in Figure 10(b), whence the query-width of Q_4 is two. Note that the decomposition of Figure 10(b) is not pure; a pure 2-width decomposition of Q_4 is shown in Figure 11.

The notion of bounded query-width generalizes the notion of acyclicity [Chekuri and Rajaraman 2000]. Indeed, acyclic queries are exactly the conjunctive queries of query-width 1, because any join tree is a (pure) query decomposition of width 1.

The next lemma shows that we can focus our attention on pure query decompositions.

LEMMA 6.3. *Let Q be a conjunctive query and $\langle T, \lambda\rangle$ a c-width query decomposition of Q. Then,*

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

PROOF

(1) Let Q be a conjunctive query and $\langle T, \lambda\rangle$, with $T = (N, E)$, a query decomposition of Q of width c. If $\langle T, \lambda\rangle$ is a pure query decomposition of Q, then we are done. Otherwise, there exists a variable $Z \in var(Q)$ and a vertex $\bar{p} \in N$ such that $Z \in \lambda(\bar{p})$. We next show how λ can be modified to eliminate one occurrence of Z from the labels. By iterating this process, all occurrences of Z can be eliminated, and the occurrences of other variables can be eliminated similarly, obtaining a pure query decomposition. Since the variable Z belongs to $var(Q)$, it occurs in some atom $C \in atoms(Q)$. Moreover, by definition of query decomposition, $C \in \lambda(\hat{p})$ for some $\hat{p} \in N$, and the set

$$\{p \in N | Z \in \lambda(p)\} \ \cup \ \{p \in N | Z \text{ is an argument of some atom } A \in \lambda(p)\}$$

induces a (connected) subtree of T. Since both sets above are nonempty, it follows that there exists a pair of adjacent vertices p' and p'' of T such that Z occurs in some atom, say B, belonging to $\lambda(p')$ and $Z \in \lambda(p'')$. Define a new labeling function λ' for T as follows:

—$\lambda'(p) = \lambda(p)$ for any vertex $p \neq p''$;

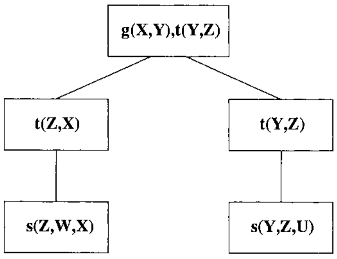


FIG. 11. A pure 2-width decomposition of Q_4.

—$\lambda'(p'') = (\lambda(p'') - \{Z\}) \cup \{B\}$.

We claim that $\langle T, \lambda' \rangle$ is a c-width query decomposition of Q. First we prove that $\langle T, \lambda' \rangle$ satisfies the three conditions of Definition 6.1. Condition 1 is satisfied because, for each vertex p, the set of atoms in $\lambda(p)$ is a subset of $\lambda'(p)$. Condition 2 is satisfied because the atom B, added to the label of p'', occurs in the label of vertex p' which is adjacent to p'', and p'' is (possibly transitively) connected to all other vertices whose labels contain B. A similar argument shows that the addition of B to the label of p'' does not violate Condition 3. Indeed, the variable Z occurs also in B (thus, nothing changes with respect to Z), and all other variables of B (that might not appear in $\lambda(p'')$) appear in the label of a vertex adjacent to p'', namely, they are arguments of atom $B \in \lambda'(p')$. Finally, observe that $\lambda'(p'')$ has the same cardinality as $\lambda(p'')$. Thus, $\langle T, \lambda' \rangle$ is a c-width query decomposition of Q. Consequently, by suitably iterating this process, we eventually obtain a pure c-width query decomposition of Q.

(2) It is easy to see that the procedure described above can be modified to be feasible on a logspace machine. Roughly, for each vertex p of T, we replace each variable $Z \in \lambda(p)$ by the atom $B_{p,z}$ defined as follows. Let p' be the lexicographically first vertex of T such that: (a) Z occurs in some atom $A \in \lambda(p')$, and (b) for any vertex p'' in the (unique) path from p to p' in T, $\lambda(p'')$ does not contain any atom having Z as an argument. Then, $B_{p,z}$ is the lexicographically first atom of $\lambda(p')$ containing Z. Since T is a tree, and the visit of a tree is feasible in logspace [Cook and McKenzie 1987], it is easy to see that, for any vertex p and variable $Z \in \lambda(p)$, $B_{p,z}$ is logspace-computable. Thus, the overall construction of the pure query decomposition of Q is feasible in logspace. □

For the sake of illustration, observe that the pure 2-width decomposition of Q_4 shown in Figure 11 can be obtained from the 2-width decomposition in Figure 10(b) by applying the elimination step in the algorithm in the proof of Lemma 6.3 once, to the root and its right child.

Intuitively, a query having a small query-width, possibly bounded by some fixed constant, can be answered efficiently. Let us define the *bounded query-width decomposition* problem.

BCQ$_k^{qd}$. Given a database **db**, a Boolean conjunctive query Q, and a query decomposition of Q of width bounded by a fixed constant $k > 0$, decide whether Q evaluates to *true* on **db**.

THEOREM 6.4. *BCQ$_k^{qd}$ is LOGCFL-complete, for any $k > 0$.*

PROOF

Hardness. JTREE trivially reduces to BCQ$_k^{qd}$, because any join tree is a 1-width query decomposition. Thus, hardness of BCQ$_k^{qd}$ for LOGCFL follows from Theorem 4.8.

Membership. Let Q be a Boolean conjunctive query over a database **db**, and QD a c-width query decomposition of Q, where $c \leq k$. Using QD, we compute a pure c-width query decomposition $\langle T, \lambda \rangle$ of Q. By virtue of Lemma 6.3, this task is feasible in logspace. Without loss of generality, we assume that any atom of Q is constant-free and does not contain any pair of duplicate variables (it is easy to see that any instance of BCQ$_k^{qd}$ can be logspace-transformed into an equivalent instance ($\bar{Q}$, **db**, $\langle \bar{T}, \bar{\lambda} \rangle$) where $\bar{Q}$ fulfills these properties).

We reduce (Q, **db**, $\langle T, \lambda \rangle$) to an instance ($Q'$, JT, **db**$'$) of JTREE such that $Q(\mathbf{db}) \equiv Q'(\mathbf{db}')$. For each vertex V of T, there is exactly one vertex V' in JT, and one relation v' in **db**$'$. V' is an atom with a new predicate symbol, such that the variables appearing as arguments in V' are exactly all variables appearing in the atoms of $\lambda(V)$. The corresponding relation v' is the result of the natural join of the relations corresponding to the atoms in $\lambda(V)$. There is an edge from V' to R' in JT if and only if there is an edge between the corresponding vertices V and R in T. The query Q' is the conjunction of all vertices (atoms) of JT.

From the properties of query decompositions, it follows that JT is a join tree of Q. Moreover, it is easy to see that $Q(\mathbf{db}) \equiv Q'(\mathbf{db}')$. Indeed, on the one hand, $Q'(\mathbf{db}')$ is true if and only if the join of all relations associated to the vertices of JT is not empty. On the other hand, $Q(\mathbf{db})$ is true if and only if the join of all relations associated to the labels of the vertices of T is not empty. Further, from the properties of the join operators, the join of the relations associated to the vertices of JT is equal to the join of the relations associated to the vertices of T. Thus, $Q(\mathbf{db}) \equiv Q'(\mathbf{db}')$.

The hardest task involved in the transformation from (Q, **db**, $\langle T, \lambda \rangle$) into ($Q'$, JT, **db**$'$) is the natural join evaluation, which is needed to generate the relations of the instance database **db**$'$. In particular, for each vertex V of T we perform the natural join of all relations associated to the atoms in $\lambda(V)$. However, since the number $|\lambda(V)|$ of relations at node V is bounded by the constant k, this join is logspace-computable by Proposition 2.10.

Thus, BCQ$_k^{qd}$ is reducible to JTREE, via the composition of a (constant) number of logspace transformations. From Proposition 2.8, it follows that BCQ$_k^{qd}$ is logspace-reducible to JTREE, and its LOGCFL membership stems from Theorem 4.3 and Proposition 2.9. □

Thus, if a query decomposition of width k is given for a query Q, the complexity of answering Q is the same as for the acyclic case.

The problem if it can be decided in polynomial time whether a given query has query-width $\leq k$ (for a fixed constant k) was left open in Chekuri and Rajaraman [2000]. A negative answer to this question was very recently given in Gottlob et al. [2000b], where it is shown that, even for $k = 4$, the problem is NP-complete. However, in Gottlob et al. [2000b], a more general notion of decomposition, called *hypertree decomposition* (and the corresponding notion of *hypertree width*), is introduced. Using the results of the present paper, it is shown in Gottlob et al. [2000b] that (a) for each k, the class of queries with query-width bounded by k is

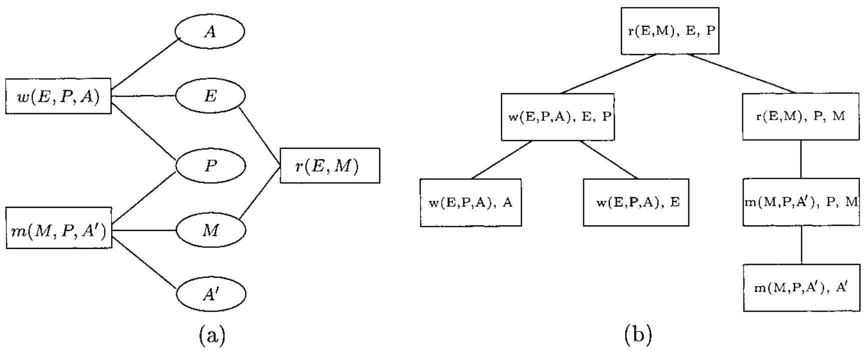


FIG. 12. (a) Incidence graph G_{Q_1} of the query Q_1, and (b) a tree decomposition of width 2 of G_{Q_1}.

properly contained in the class of queries whose hypertree-width is bounded by k; (b) unlike query-width, bounded hypertree-width is recognizable in LOGCFL; and (c) evaluating Boolean queries of bounded hypertree-width is LOGCFL-complete.

6.2. BOUNDED TREEWIDTH. The *incidence graph* [Chekuri and Rajaraman 2000] $G_Q = (V, F)$ of query Q has a vertex for each variable and for each atom of Q. There is an edge between a variable Y and an atom A whenever Y occurs in A.

Definition 6.5 [*Chekuri and Rajaraman* 2000; *Robertson and Seymour* 1986*a*]. A *tree decomposition* of a graph $G = (V, F)$ is a pair $\langle T, \lambda \rangle$, where $T = (N, E)$ is a tree, and λ is a labeling function associating to each vertex $p \in N$ a set of vertices $\lambda(p) \subseteq V$, such that the following conditions are satisfied:

(1) for each vertex b of G, there exists $p \in N$ such that $b \in \lambda(p)$;
(2) for each edge $\{b, d\} \in F$, there exists $p \in N$ such that $\{b, d\} \subseteq \lambda(p)$;
(3) for each vertex b of G, the set $\{p \in N | b \in \lambda(p)\}$ induces a (connected) subtree of T.

The *width* of the tree decomposition is $max_{p \in N}|\lambda(p)| - 1$. The *treewidth* of G is the minimum width over all its tree decompositions. The *treewidth* $tw(Q)$ of a conjunctive query Q is the treewidth of its incidence graph.

From the following well-known fact, it follows that bounded treewidth is a generalization of graph acyclicity.

PROPOSITION 6.6 [ROBERTSON AND SEYMOUR 1986B]. *A graph is acyclic iff its treewidth is* 1.

Example 6.7. The incidence graph of query Q_1 of Example 1.1 is shown in Figure 12(a). The treewidth of Q_1 is two, and a tree decomposition of width two of the incidence graph of Q_1 is shown in Figure 12(b).

LEMMA 6.8 [CHEKURI AND RAJARAMAN 2000]. *Every tree decomposition of width k of the incidence graph G_Q of a query Q is also a query decomposition of width $k + 1$ of Q.*

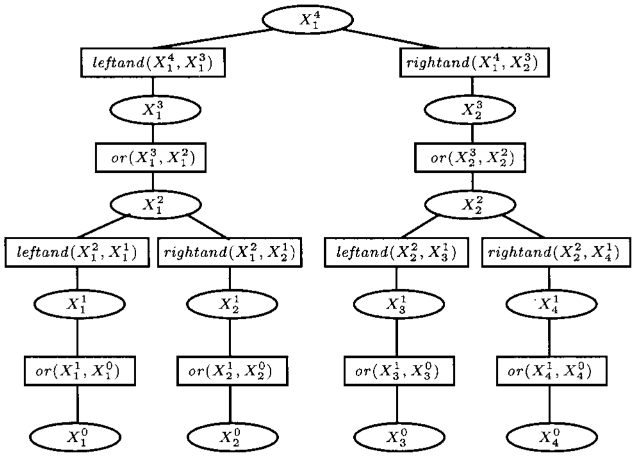


FIG. 13. Incidence graph of query Q_4.

In fact, for a conjunctive query Q, Chekuri and Rajaraman [2000] proved the following relationship between the treewidth of Q and the query-width of Q.

PROPOSITION 6.9 [CHEKURI AND RAJARAMAN 2000]. *For any conjunctive query Q, $tw(Q)/a \leq qw(Q) \leq tw(Q) + 1$, where a is the maximum predicate arity in Q.*

In analogy to BCQ$_k^{qd}$, we define the problem BCQ$_k^{td}$ as follows:

BCQ$_k^{td}$. Given a database **db**, a Boolean conjunctive query Q, and a tree decomposition of G_Q of width $\leq k$, decide whether Q evaluates to *true* on **db**.

It is not surprising that BCQ$_k^{td}$ has exactly the same complexity as BCQ$_k^{qd}$.

THEOREM 6.10. *For each $k > 0$, BCQ$_k^{td}$ is LOGCFL-complete.*

PROOF. By Theorem 6.4 and Lemma 6.8, it immediately follows that BCQ$_k^{td}$ is in LOGCFL.

To see LOGCFL-hardness, reconsider the ABCQ instances $\langle Q, \mathbf{db}\rangle$ constructed in the proof of Theorem 4.8. The incidence graph G_Q of each such instance $\langle Q, \mathbf{db}\rangle$ is a tree and thus acyclic. This is illustrated in Figure 13, which depicts the incidence graph of query Q_4 whose corresponding hypergraph was shown in Figure 8. Since G_Q is acyclic, by Proposition 6.6, Q has treewidth 1. Note that G_Q is logspace constructible from Q. In fact, the list of vertices of G_Q is obtained by copying all atoms and all variables of Q to the output; the list of edges of G_Q is obtained by cycling over all atoms A of Q and outputting an edge $\{A, X\}$ for each variable X occurring in A. As discussed in Section 2.5, this can be done in logspace. A simple width 1 tree decomposition $\langle T, \lambda\rangle$ of Q can be generated in logspace from the acyclic graph G_Q as follows:

(1) For each vertex a of G_Q create a vertex v_a of T and let $\lambda(v_a) := \{a\}$.
(2) For each edge $\{a, b\}$ of G_Q, create a vertex $v_{\{a,b\}}$, let $\lambda(v_{\{a,b\}}) := \{a, b\}$, and connect $v_{\{a,b\}}$ by an edge to v_a and v_b in T.

Given that G_Q is a tree, also T is a tree (T was obtained from G_Q by renaming vertices and by subdividing each edge with a new vertex). By construction, $\langle T, \lambda\rangle$ satisfies conditions (1) and (2) of Definition 6.5. Moreover, by construction of T, every vertex a of G_Q appears in $\langle T, \lambda\rangle$ just in the label $\lambda(v_a)$ of v_a and in the labels of vertices $v_{\{a,b\}}$ that are connected to v_a. Hence, condition (3) of Definition 6.5 (i.e., the connectedness condition) holds, too. Given that each label of a vertex of T is a set of at most two elements, $\langle T, \lambda\rangle$ is a width 1 tree decomposition of G_Q.

In summary, in the proof of Theorem 4.8, we have transformed (in logspace) instances of the LOGCFL-complete uniform SAC[1] circuit evaluation problem into specific instances (Q, **db**) of the ABCQ problem. Here we have shown that the latter can be in turn transformed in logspace into instances (Q, **db**, $\langle T, \lambda\rangle$) of BCQ_1^{td}. By Proposition 2.8, it follows that a LOGCFL-complete problem can be transformed in logspace into the problem BCQ_1^{td}. Hence BCQ_1^{td} is LOGCFL-hard. By the definition of BCQ_k^{td}, every instance of BCQ_1^{td} is also an instance of BCQ_k^{td} for $k > 1$. Therefore, for each $k \geq 1$, BCQ_k^{td} is LOGCFL-hard. □

Wanke [1994] has shown that, for a fixed constant k, checking whether a graph has treewidth k is in LOGCFL. By proving some general complexity-theoretic results and by using Wanke's result, the following was shown in Gottlob et al. [2000a]:

PROPOSITION 6.11 [GOTTLOB ET AL. 2000A]. *For each constant k, there exists an L^{LOGCFL} transducer T_k that behaves as follows on input G. If G is a graph of treewidth $\leq k$, then T_k outputs a tree-decomposition of width $\leq k$ of G. Otherwise, T_k halts with empty output.*

From this result, it follows that Boolean conjunctive queries of constant treewidth can be answered in LOGCFL even in case a tree-decomposition is not given. Before stating this as a theorem, let us define the relevant decision problem BCQ_k^{tw} precisely.

BCQ_k^{tw}. Given a database **db** and a Boolean conjunctive query Q with $tw(Q) \leq k$, decide whether Q evaluates to *true* over **db**.

THEOREM 6.12. *BCQ_k^{tw} is complete for LOGCFL, for any $k > 0$.*

PROOF. Hardness follows from the fact that BCQ_k^{td} is already LOGCFL-hard. We show membership. By Proposition 6.11, there exists an $\mathrm{L}^{\mathrm{LOGCFL}}$ transducer T transforming each BCQ_k^{tw} instance Q into an instance $\langle Q, \Delta\rangle$ of BCQ_k^{td}, where Δ is a tree-decomposition of Q of width at most k. Thus, BCQ_k^{tw} is $\mathrm{L}^{\mathrm{LOGCFL}}$-reducible to BCQ_k^{td}, which is in LOGCFL, that is, $\mathrm{BCQ}_k^{tw} \in \mathrm{L}^{\mathrm{LOGCFL}}(\mathrm{LOGCFL})$. From Lemma 2.4, it follows that BCQ_k^{tw} is in LOGCFL. □

6.3. BOUNDED DEGREE OF CYCLICITY. Closely related to the concepts of treewidth and query-width is the notion of *degree of cyclicity* of hypergraphs [Gyssens 1986; Gyssens et al. 1994; Gyssens and Paredaens 1984].

Let (V, E) be a hypergraph, let $H \subseteq E$, and let $F \subseteq E - H$. F is called *connected with respect to H* if, for any two edges $e, f \in F$, there exists a sequence $e_1, \ldots, e_n$ of edges in F such that (i) $e_1 = e$; (ii) for $i = 1, \ldots, n-1$, $e_i \cap e_{i+1}$ is not contained in $\bigcup_{h \in H} h$; and (iii) $e_n = f$ [Gyssens et al. 1994]. The maximal connected subsets of $E - H$ with respect to H are called the *connected components of $E - H$ with respect to H*. It is easy to see that the connected components of $E - H$ with respect to H form a partition of $E - H$. Moreover,

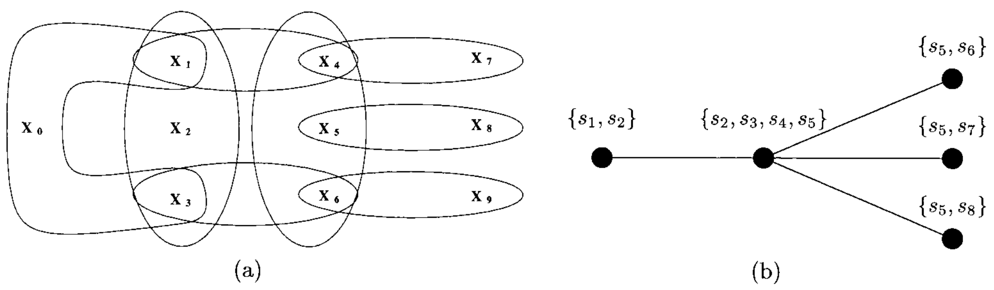


FIG. 14. (a) Hypergraph $H(Q_5)$, and (b) a hinge-tree decomposition of Q_5.

a hypergraph is called *reduced* if none of its edges is properly contained in any other.

Definition 6.13 [*Gyssens et al.* 1994; *Gyssens and Paredaens* 1984]. Let (V, E) be a reduced and connected hypergraph and let H be either E or a proper subset of E containing at least two edges. Let $H_1, \ldots, H_m$ be the connected components of $E - H$ with respect to H. Then, H is called a hinge if, for $i = 1, \ldots, m$, there exists an edge $h_i \in H$ such that

$$\left(\bigcup_{h \in H_i} h \right) \cap \left(\bigcup_{h \in H} h \right) \subseteq h_i.$$

A hinge is minimal if it does not contain any other hinge.

Example 6.14. Consider the following query Q_5 taken from Gyssens et al. [1994].

$$\begin{aligned} ans \leftarrow \quad & s_1(X_0, X_1, X_3) \wedge s_2(X_1, X_2, X_3) \wedge s_3(X_1, X_4) \wedge s_4(X_3, X_6) \wedge s_5(X_4, X_5, X_6) \\ & \wedge s_6(X_4, X_7) \wedge s_7(X_5, X_8) \wedge s_8(X_6, X_9). \end{aligned}$$

The hypergraph $H(Q_5)$ is cyclic as shown in Figure 14. The minimal hinges of $H(Q_5)$ are $\{s_1, s_2\}$, $\{s_1, s_3, s_4, s_5\}$, $\{s_2, s_3, s_4, s_5\}$, $\{s_5, s_6\}$, $\{s_5, s_7\}$, $\{s_5, s_8\}$, $\{s_3, s_6\}$, and $\{s_4, s_8\}$. (Since the query atoms have distinct predicate symbols, the hyperedge corresponding to an atom is denoted by its predicate symbol.)

The hypergraph $H(Q_1)$ of the query Q_1 of Example 1.1 has a unique minimal hinge $\{r, w, m\}$.

Definition 6.15 [*Gyssens et al.* 1994]. Let (V, E) be a hypergraph. A *hinge-tree* of (V, E) is a tree T such that

(1) the vertices of T are minimal hinges of (V, E);
(2) each edge in E is contained in at least one vertex of T;
(3) two adjacent vertices A and B of T share precisely one edge ℓ of E; moreover, ℓ consists exactly of the elements shared by A and B (i.e., $\ell = (\bigcup_{h \in A} h) \cap (\bigcup_{h \in B} h)$);
(4) for each pair of vertices A and B of T, $A \cap B$ is contained within each vertex on the path connecting A and B in T.

The size (i.e., the cardinality) of the largest vertex of T is called the degree of cyclicity of (V, E).

Observe that the notion of degree of cyclicity is well defined. Indeed, as shown in Gyssens [1986], each hypergraph (V, E) has at least one hinge-tree, and the size of the largest vertex in any hinge-tree of (V, E) is a fixed parameter of the hypergraph (V, E), that is, it is independent of the particular hinge-tree chosen. As bounded query-width, also the notion of bounded degree of cyclicity generalizes acyclicity. Indeed, the acyclic hypergraphs are precisely the hypergraphs with degree of cyclicity at most 2 [Gyssens 1986].

A hinge-tree of the hypergraph $H(Q)$ of a conjunctive query Q is called a *hinge-tree decomposition* of Q. The degree of cyclicity of Q, as well as a hinge-tree decomposition of Q, can be computed in polynomial time [Gyssens 1986].

Example 6.16. The hypergraph $H(Q_1)$ of the query Q_1 of Example 1.1 has only one hinge-tree consisting of the single vertex $\{r, w, m\}$; the degree of cyclicity of $H(Q_1)$ is three.

Figure 14(b) shows a hinge-tree T_5 of the hypergraph $H(Q_5)$ of the query Q_5 in Example 6.14. The degree of cyclicity of $H(Q_5)$ is four, because the largest vertex $\{s_2, s_3, s_4, s_5\}$ of T_5 has cardinality four.

We next define the *bounded hinge-tree decomposition problem*. Without loss of generality, in the sequel of the section, we assume that any conjunctive query Q does not contain any pair of atoms A and B such that $var(A) \subseteq var(B)$. This assumption implies also that the hypergraph $H(Q)$ of Q is reduced.

BCQ_k^{hd}. Given a database **db**, a Boolean conjunctive query Q, and a hinge-tree decomposition of Q having degree of cyclicity no more than a fixed constant $k \geq 1$, decide whether Q evaluates to *true* on **db**.

LEMMA 6.17. *Let Q be a conjunctive query such that $H(Q)$ has degree of cyclicity c, and let HD be a hinge-tree decomposition of Q. Then,*

(1) *there exists a c-width query decomposition QD of Q;*
(2) *QD is logspace-computable from Q and HD.*

PROOF. Let HD be a hinge-tree decomposition of the conjunctive query Q, and let c be the cardinality of the largest hinge of HD. Define the query decomposition $QD = \langle T, \lambda \rangle$ as follows. For each vertex H of HD, there is exactly one vertex p_H in the tree T. There is an edge from $p_{H'}$ to $p_{H''}$ in T if and only if there is an edge between the corresponding vertices H' and H'' in HD. The label of the vertex p_H of QD corresponding to the vertex H of HD is

$$\lambda(p_H) = \{A \in atoms(Q) | var(A) \in H\}.$$

It is easy to see that QD satisfies Conditions (1)–(3) of Definition 6.1. Indeed, Condition (2) and Condition (4) of Definition 6.15 of hinge-tree entail Condition (1) and Condition (3) of Definition 6.1, respectively. Condition (2) of Definition 6.1 follows from Condition (3) and Condition (4) of Definition 6.15 (recall that Q does not contain any pair of atoms A and B such that $var(A) \subseteq var(B)$, by assumption). Since the cardinality of the label $\lambda(p_H)$ of a vertex p_H of QD is equal to the cardinality of the corresponding hinge H of HD, we have that QD is a c-width query decomposition of Q. Moreover, QD is evidently logspace-computable from Q and HD. □

Note that, by virtue of Lemma 6.17, the query-width of a conjunctive query Q is less than or equal to the degree of cyclicity of $H(Q)$. In fact, for both queries considered in Example 6.16, the degree of cyclicity is strictly greater than the query-width.

THEOREM 6.18. *For each $k > 1$, BCQ_k^{hd} is LOGCFL-complete.*

PROOF. Membership follows from Lemma 6.17 and Theorem 6.4.

We next reduce JTREE to BCQ_k^{hd}, with $k = 2$. Let $(Q, JT, \mathbf{db})$ be an instance of JTREE. Without loss of generality, we assume that $|atoms(Q)| \geq 2$. We compute a hinge-tree decomposition $T = (V, E)$ of the hypergraph $H(Q)$ as follows:

—Initially, for each vertex A of JT, V contains a vertex $v_A = \{var(A)\}$; moreover, for each pair of adjacent vertices A, A' of JT, the corresponding vertices $v_A, v_{A'} \in V$ are adjacent in T, that is, $\{v_A, v_{A'}\} \in E$.

—Then, T is rooted at any arbitrary vertex r. For each nonroot vertex $v \in V$, v is replaced by $v \cup parent(v)$, where $parent(v)$ denotes the parent of v in (the rooted tree) T.

—Finally, let $c_1, \ldots, c_n$ $(n \geq 1)$ be the children of the root r, and let $l_1, \ldots, l_m$ $(m \geq 0)$ be the children of c_1 in T. The root r is replaced by c_1, and the children of c_1 become $c_2, \ldots, c_n, l_1, \ldots, l_m$.

Each of the three steps above can be performed by a logspace machine, and thus the entire construction is in logspace, by Proposition 2.8. Moreover, it is easy to see that the resulting tree T is a hinge-tree decomposition of $H(Q)$, and that each vertex of T has cardinality 2. (In fact, the degree of cyclicity of $H(Q)$ is 2, as it is an acyclic hypergraph with more than one hyperedge.) Thus, JTREE is logspace-reducible to BCQ_2^{hd}. From Theorem 4.8, it follows that BCQ_2^{hd} is LOGCFL-hard. □

7. *Parallel Database Algorithms*

Database applications involving huge amounts of data and complex queries that may contain large numbers of join operations between relations strongly motivate research on parallel algorithms for query evaluation.[8]

Theorem 4.3, demonstrating membership of ABCQ in LOGCFL, implicitly proves that the evaluation of acyclic queries can be profitably parallelized, that is, an exponential speed-up can be achieved on a parallel machine (with a polynomial number of processors). Indeed, by the result of Ruzzo [1980], LOGCFL is included in AC^1—the set of languages recognizable in logarithmic time on a CRCW parallel machine using a polynomial number of processors. Thus, our LOGCFL algorithm of Theorem 4.3, coupled with Ruzzo's parallel simulation of LOGCFL, yields an AC^1 method to solve ABCQ (and JTREE). However, this method would prove unsuitable in database contexts, where relational operations such as selection, projection, and join are used as atomic operations.

[8] See, for example, DeWitt and Gray [1992], Hasan et al. [1996], Hong and Stonebreaker [1993], Lakshmi and Yu [1990], and Wilschut et al. [1995].

In this section, we present two parallel database algorithms directly designed for a parallel database machine model, where join is considered a machine primitive. Our algorithms exploit *inter-operation* parallelism, that is, processing different primitive operations in parallel, minimizing the length of the sequence of parallel joins (while keeping the size of intermediate relations small). Note that the exploitation of inter-operation parallelism has been recognized as a relevant feature for parallel databases, and its importance has been confirmed also by concrete experiments [Wilschut et al. 1995]. *Intra-operation parallelism*, that is, the parallel execution of a single primitive operation (e.g., join), is orthogonal and can be added easily to our model by parallelizing single join executions.

In the following, we first provide a parallel algorithm for deciding JTREE. Then, we show how to extend such an algorithm to compute the answer to a non-Boolean conjunctive acyclic query. The latter algorithm works particularly well for *bounded-projection* queries, that is, conjunctive queries where the arity of the output atom *ans* is bounded by some fixed constant k.

The complexity of the proposed algorithms is analyzed on a parallel database machine model well-suited for the evaluation of inter-query parallelism. We consider a shared-memory parallel EREW (exclusive read exclusive write) DB-machine, where relational algebra operations are machine primitives. In one parallel step of the DB-machine, each machine's processor can perform a constant number of relational algebra operations on the database relations at hand. Transformations of query atoms, join trees, and schemas are considered cost-free provided that they are feasible in polynomial time.[9] The latter assumption is made to focus on database operations (polynomial time schema transformations are considered negligible compared to database operations).

The efficiency of a computation of the DB-machine will be measured according to the following cost parameters:

(a) the number of parallel steps;
(b) the number of relational processors employed in each step, that is, processors able to perform operations of relational algebra and related operations such as relational assignment statements (e.g., $r := s$, where r and s are relation variables);
(c) the size of the working database.

On the parallel database machine, the algorithms we provide here run in logarithmic time, employ a linear number of relational processors, and keep small the size of the working database during the entire computation (in particular, the DB-SHUNT algorithm runs always in polynomial space, while the ACQ algorithm guarantees polynomial space consumption if the number of output attributes is bounded by a constant).

Note that besides the acyclic query Q and the database instance **db**, our algorithms take as an additional input a join tree for Q. This is motivated by the following main reasons.

[9] Note that all nondatabase transformations performed in the algorithms proposed in this section are actually in logspace.

—The structure of the join-tree is exploited by the tree-contraction mechanisms of our algorithms.

—A join tree of an acyclic query can be computed very efficiently (see Theorem 3.6), and its computational cost is negligible in a database context (it is independent of the database size).

—The algorithms can be generalized easily to solve also cyclic conjunctive queries having a bounded query-width, once a suitable decomposition of the query is provided. (We briefly discuss this issue at the end of Section 7.2.)

—Suitable optimization techniques, taking into account different critical factors (like, e.g., the overall cost of computing joins), can be utilized to select the most appropriate join tree.

At the end of the section, we will compare our algorithms to other parallel algorithms (in the constraint-satisfaction area) that solve similar problems.

7.1. NF QUERY INSTANCES.

Recall that a conjunctive query Q on a database schema $DS = \{R_1, \ldots, R_m\}$ consists of a rule of the form

$$Q\colon ans(\mathbf{u}) \leftarrow r_1(\mathbf{u}_1) \wedge \cdots \wedge r_n(\mathbf{u}_n),$$

where $n \geq 0$, $r_1, \ldots, r_n$ are relation names (not necessarily distinct) of DS; *ans* is a relation name not in DS; and $\mathbf{u}, \mathbf{u_1}, \ldots, \mathbf{u_n}$ are lists of terms (i.e., variables or constants) of appropriate length. Throughout this section, we assume without loss of generality that **u** is a list of distinct variables and Q is a connected query, that is, its hypergraph $H(Q)$ is connected.

The algorithms proposed in this section take a triple (Q, JT, **db**) as the input, where JT is a join tree for the (acyclic) query Q on the database **db**. (We will omit the explicit specification of the database schema DS, which will be implicitly understood.) We next define a normal form for these (acyclic) query instances, and we show that it can be achieved efficiently.

Let $\mathscr{C} = (Q, JT, \mathbf{db})$ be an acyclic query instance (i.e., JT is a join-tree of an acyclic conjunctive query Q on **db**). We say that $\mathscr{C}$ is in *normal form* (NF) if the following conditions are satisfied:

(a) the join tree JT of Q is strictly binary, that is, every non-leaf vertex has exactly two children;

(b) every atom occurring in Q is constant-free and does not contain any pair of variables with the same name;

(c) any pair of atoms in Q has distinct predicates.

LEMMA 7.1. *For every acyclic query instance* $\mathscr{C} = (Q, JT, \mathbf{db})$, *there exists an instance* $\mathscr{C}' = (Q', JT', \mathbf{db}')$ *in normal form such that*

(1) $\mathscr{C}$ *and* $\mathscr{C}'$ *are equivalent, and*

(2) $\mathscr{C}'$ *is logspace-computable from* $\mathscr{C}$.

PROOF. Let $\mathscr{C} = (Q, JT, \mathbf{db})$ be an acyclic query instance. Possible constants or duplicate variables appearing in the same atom $P = p(\mathbf{u})$ of Q are eliminated by performing suitable selections and projections on the corresponding relation p; the resulting relation takes a new name (and the predicate of P also), if another atom with predicate p appears in the query.

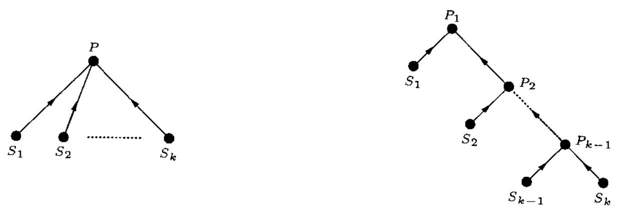


FIG. 15. "Binarization" of a k-ary vertex.

The main point in computing an equivalent NF version $\mathscr{C}'$ of $\mathscr{C}$ is the "binarization" of JT (property (a)). To this end, we replace each vertex P having $k > 2$ children $\{S_1, \ldots, S_k\}$ in JT by $k - 1$ copies $\{P_1, \ldots, P_{k-1}\}$ of P (see Figure 15); the left child and the right child of P_i are S_i and P_{i+1}, respectively (for any $1 \leq i < k - 1$); the left child and the right child of P_{k-1} are S_{k-1} and S_k, respectively. If a vertex P has a single child in JT, then a copy of P is added as its second child, thus obtaining a strictly binary tree. The body of the query Q' is the conjunction of the vertices of JT' (i.e., $ATOMS(Q')$ coincides with the set of vertices of JT'). Condition (c) is enforced by trivial atom renaming (in both Q' and JT') and relation duplication.

It is easy to see that the entire process can be performed by a logspace machine. □

7.2. SOLVING JTREE IN PARALLEL. In this section, we propose a parallel algorithm for deciding JTREE. An instance of JTREE is an acyclic query instance $\mathscr{T} = (Q, JT, \mathbf{db})$ where the query Q is Boolean (it has no head atom). Given an instance $\mathscr{T} = (Q, JT, \mathbf{db})$ of JTREE, our algorithm first computes (in logspace) its equivalent NF version $\mathscr{T}' = (Q', JT', \mathbf{db}')$. The instance $\mathscr{T}'$ is then transformed into a new data structure, called *e-join tree* (*extended join tree*), which is well suited for parallel execution. A vertex p of an e-join tree T is a pair $\langle Rel_T(p), Sch_T(p)\rangle$, where $Sch_T(p)$ is a relation schema, and $Rel_T(p)$ is a relation instance over $Sch_T(p)$; the attributes of $Sch_T(p)$ are named and partitioned in two distinguished sets $R_T(p)$ and $I_T(p)$. The e-join tree $ET(\mathscr{T}')$ of $\mathscr{T}'$ has exactly the same tree shape as JT' (i.e., it is isomorphic to JT' under a renaming of the vertices). For each vertex (atom) $P = r(\mathbf{u})$ of JT', the schema and the relation of the corresponding vertex p in $ET(\mathscr{T}')$ are defined as follows:

Schema: $I_{ET(\mathscr{T}')}(p)$ is empty, and $R_{ET(\mathscr{T}')}(p)$ is the set of all attributes of the schema of r; each attribute of p keeps the name of the corresponding variable of $r(\mathbf{u})$.[10]

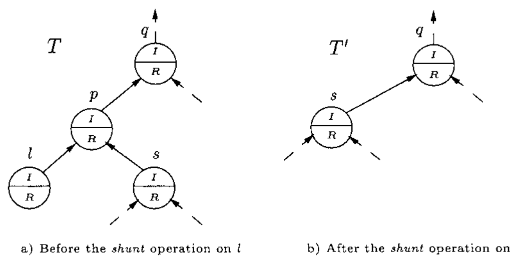
Relation: $Rel_{ET(\mathscr{T}')}(p)$ contains precisely the tuples of relation r of $\mathbf{db}'$ (i.e., $Rel_{ET(\mathscr{T}')}(p) = r$).

Thus, compared to its corresponding vertex $r(\mathbf{u})$ of the join tree JT', a vertex p of $ET(\mathscr{T}')$ additionally stores the relation instance r associated with $r(\mathbf{u})$.

LEMMA 7.2. *Let $\mathscr{T} = (Q, JT, \mathbf{db})$ be an instance of JTREE in normal form, and $ET(\mathscr{T})$ be the corresponding e-join tree. Then, Q is true on* **db** *if and only if the natural join of all relations stored in the vertices of $ET(\mathscr{T})$ is not empty.*

The proof of this lemma is straightforward, and thus omitted.

[10] Note that, since $\mathscr{T}'$ is in normal form, the list $\mathbf{u}$ of terms of $r(\mathbf{u})$ consists solely of distinct variables.



T
T'
q
p
l
s
I
R
a) Before the *shunt* operation on *l*
b) After the *shunt* operation on *l*

FIG. 16. *Shunt* operation applied to leaf *l*.

Our parallel method for evaluating JTREE is based on a *tree contraction* technique that closely resembles the method of Karp and Ramachandran [1990] for the *expression evaluation problem*. The algorithm transforms an e-join tree in stages in such a way that an n-vertices tree T is contracted into a 3-vertices one in $\lceil log\, n \rceil - 1$ stages. At each stage, a local operation, called *shunt*, is applied in parallel to half of the leaves of T.

The main novelty of our method is the shunt operation, first defined here for databases, that guarantees the correctness of the query result, while keeping the size of intermediate relations small (quadratic).

Let l be a leaf of an e-join tree T, p the parent of l, s the other child of p, and q the parent of p (see Figure 16). The shunt operation applied to l results in a new contracted tree T' in which l and p are deleted, s is transformed (as specified below) and takes the place of vertex p (i.e., it becomes child of q). The transformed version of s in T' is defined as follows.

$$R_{T'}(s) = R_T(s).$$

$$I_{T'}(s) = (R_T(q) \cap Sch_T(p)) - R_T(s).$$

$$Rel_{T'}(s) = \prod_{Sch_{T'}(s)} (Rel_T(l) \rtimes Rel_T(p) \bowtie Rel_T(s)).$$

where $\bowtie$ and $\rtimes$ denote the natural join operator and the semi-join operator [Ullman 1989], respectively.

Intuitively, as p is deleted, the join attributes of (the schemas of) p and q (i.e., $Sch_T(p) \cap R_T(q)$, since $Sch_T(p) \cap I_T(q) = \emptyset$ by construction, as proved in Corollary 7.6) must be kept in the new e-join tree, in order to guarantee the soundness of the result. To this end, the attributes of $Sch_T(p) \cap R_T(q)$ missing in s are added to the schema of s (in $I_{T'}(s)$). Thus, after the shunt, $I_{T'}(s)$ stores the "witnesses" of the p-tuples joining to both s and l, that are relevant for the join to q.

Our tree contraction algorithm for JTREE, named *DB-SHUNT*, is shown in Figure 17. After the normalization of the input instance $\mathcal{T}$ and the computation of the e-join tree T, the leaves of T are numbered from left to right, and the **while** loop performs the tree contraction. At each iteration, the shunt operation is applied to the odd-numbered leaves of T in a parallel fashion; to avoid concurrent changes on the same vertex, left and right leaves are processed in two distinct steps ((4a) and (4b), respectively). Observe that the shunt operation is applied only to leaves of depth (distance from the root) greater than one. If a

```
DB-SHUNT ALGORITHM
Input: An instance $\mathcal{J} = (Q, JT, \mathbf{db})$ of JTREE.
Output: The answer to $Q$ on db.
 begin
(1)    Compute the NF version $\mathcal{J}'$ of $\mathcal{J}$;
(2)    $T := ET(\mathcal{J}')$;
       Let $\lambda$ be the number of leaves in $T$;
(3)    Label the leaves of $T$ in order from left to right as $1, \ldots, \lambda$;
(4)    while depth($T$)> 1 do
       (a)  in parallel apply the shunt operation to all odd-numbered leaves that
            are the left children of their parent, and have depth greater than 1;
       (b)  in parallel apply the shunt operation to all odd-numbered leaves that
            are the right children of their parent, and have depth greater than 1;
       (c)  shift out the rightmost bit in the labels of all remaining leaves;
       end (* while *)
(5)    if the join of all relations stored in the vertices of $T$ is non-empty then output "true"
       else output "false"
end.
```

FIG. 17. A parallel tree contraction algorithm for JTREE.

leaf becomes a child of the root, then it is "frozen" until the end of the **while** loop (and its numbering is irrelevant from that point onward). The **while** loop terminates when T is contracted to three vertices (i.e., it gets depth one). The join of the relations stored in these three vertices is eventually computed to check whether the query is true or not (Q is true if and only if the result of this join is not empty).

Example 7.3. Consider the following query Q_6 over the database shown in Figure 18:

$$\begin{aligned} Q_6\colon ans \leftarrow\ & \mathsf{p}_1(X,Y) \wedge \mathsf{p}_2(X,Z,V) \wedge \mathsf{p}_3(X,Z) \wedge \mathsf{p}_4(Z,S) \wedge \mathsf{p}_5(Y,T,X) \\ & \wedge \mathsf{p}_6(X,Y,L) \wedge \mathsf{p}_7(T,W) \wedge \mathsf{p}_8(T,R) \wedge \mathsf{p}_9(W,O). \end{aligned}$$

Figure 19 shows the e-join tree T_0, resulting after steps (1)–(3) of DB-SHUNT. This figure shows also the relation schemas of the vertices of T_0. For each p_i ($1 \leq i \leq 9$), the upper part of the circle labeled by p_i contains the attributes in $I_{T_0}(p_i)$, while its lower part contains the attributes in $R_{T_0}(p_i)$. The relation instance $Rel_{T_0}(p_i)$ associated with a vertex p_i of T_0 is the input relation for p_i shown in Figure 18.

Step (4a) applies the shunt operation in parallel to vertices p_3 and p_6 (which are the leaves numbered 1 and 3, respectively). This step contracts T_0 into the e-join tree T_1^a depicted in Figure 20(a). Note that vertices p_2, p_3, p_5, and p_6 are eliminated, p_4 and p_7 are changed, while all other vertices remain untouched. Figure 21 shows the relations of vertices p_4 and p_7 resulting from the execution of this step. Step (4b) performs a shunt operation on vertex p_9, which eliminates p_7 and p_9, yielding the e-join tree T_1^b shown in Figure 20(b) and modifying the relation of p_8 as in Figure 22. Since T_1^b has depth 1, the **while** loop exits. Step (5) performs the join of the relations of p_1, p_4 and p_8. Figure 23 shows the result of this join. Since this result is non-empty, DB-SHUNT outputs "true."

Before proving the correctness of DB-SHUNT, we state some structural properties of the e-join trees.

p_1	X	Y
	x_1	y_1
	x_2	y_1
	x_1	y_2

p_2	X	Z	V
	x_1	z_1	v_1
	x_2	z_2	v_2

p_3	X	Z
	x_1	z_1
	x_2	z_2
	x_1	z_2

p_4	Z	S
	z_1	s_1
	z_2	s_2
	z_3	s_3

p_5	Y	T	X
	y_1	t_1	x_1
	y_2	t_1	x_1

p_6	X	Y	L
	x_1	y_1	ℓ_1
	x_2	y_1	ℓ_2

p_7	T	W
	t_1	w_1
	t_2	w_1
	t_1	w_2

p_8	T	R
	t_1	r_1
	t_2	r_2

p_9	W	O
	w_1	o_1
	w_2	o_2
	w_1	o_3

FIG. 18. Input database for query Q_6 in Example 7.3.

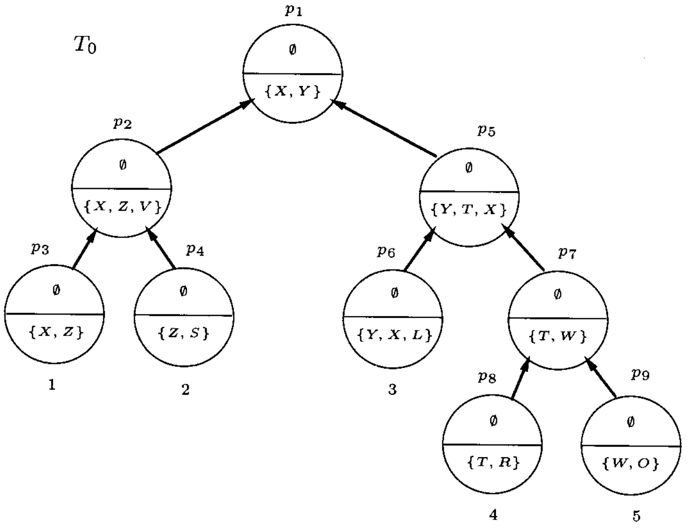


FIG. 19. Initial e-join tree T_0 for query Q_6 in Example 7.3.

LEMMA 7.4. *Let T be the e-join tree computed at a given step of DB-SHUNT, and q and q' be two vertices of T, such that q' is the parent of q. Then,*

$$I_T(q) \subseteq R_T(q').$$

PROOF. We proceed by induction on the execution steps of DB-SHUNT.

Basis (*Initialization of T*). T is initialized by instruction (2) of DB-SHUNT, where it is set to $ET(\mathcal{T}')$. At this step, $I_T(q) \subseteq R_T(q')$ trivially holds, because $I_T(q) = \emptyset$ in $ET(\mathcal{T}')$.

Induction step. Let q'' be the parent of q at the previous step of the algorithm. If $q'' = q'$, then neither q nor q' have been involved in any shunt operation at the last step. Hence, they are unaltered and the statement holds by the induction hypothesis. (Note that a vertex q changes its parent *only if* q is the sibling of a leaf l on which a shunt is applied.) If $q'' \neq q'$, then a shunt has been applied on

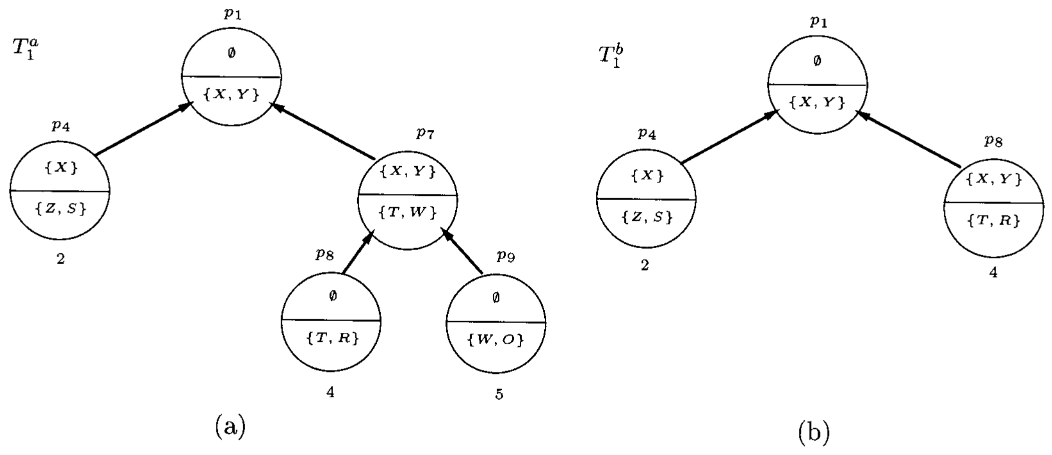


FIG. 20. (a) e-join tree T_1^a, and (b) e-join tree T_1^b in Example 7.3.

$Rel_{T_1^a}(p_4)$	Z	S	X
	z_1	s_1	x_1
	z_2	s_2	x_2

$Rel_{T_1^a}(p_7)$	T	W	X	Y
	t_1	w_1	x_1	y_1
	t_1	w_2	x_1	y_1

FIG. 21. Relations $Rel_{T_1^a}(p_4)$ and $Rel_{T_1^a}(p_7)$ in Example 7.3.

$Rel_{T_1^b}(p_8)$	T	R	X	Y
	t_1	r_1	x_1	y_1

FIG. 22. Relation $Rel_{T_1^b}(p_8)$ in Example 7.3.

$Rel_{T_1^b}(p_1) \bowtie Rel_{T_1^b}(p_4) \bowtie Rel_{T_1^b}(p_8)$	X	Y	Z	S	T	R
	x_1	y_1	z_1	s_1	t_1	r_1

FIG. 23. Join for Step (5) in Example 7.3.

the sibling of q at the last step; while q' has been left untouched. Therefore, from the definition of shunt, it easily follows that $I_{\mathrm{T}}(q) \subseteq R_{\mathrm{T}}(q')$. □

Importantly, e-join trees have the following property, which is closely related to Condition 2 of the definition of an ordinary join tree (see Section 2.3).

THEOREM 7.5. *Let T be the e-join tree computed at a given step of DB-SHUNT, and q_1 and q_2 be two vertices of T. For every vertex q on the (unique) path from q_1 to q_2 in T, the following properties hold.*

(*a*) *If $X \in R_T(q_1) \cap R_T(q_2)$, then $X \in R_T(q)$.*
(*b*) *If $X \in Sch_T(q_1) \cap Sch_T(q_2)$, then $X \in Sch_T(q)$.*

PROOF

(a) At the initial step, $T = ET(\mathcal{T}')$ satisfies property (a) because T is directly obtained from the join tree JT' of the JTREE instance $\mathcal{T}'$. To show that the property is kept during the **while** iterations, we assume that an e-join tree T satisfies property (a), and demonstrate that (a) holds on the e-join tree T' obtained by the execution of step (4)(a) or step (4)(b) on T. Let $X \in R_{\mathrm{T}'}(q_1) \cap$

$R_{T'}(q_2)$ be a common attribute of two vertices q_1 and q_2 of T'; moreover, let $\mathcal{P}$ and $\mathcal{P}'$ be the path connecting q_1 to q_2 in T and T', respectively, and $q \in \mathcal{P}'$. Since the shunt operations neither add new vertices nor alter their relative order (if p is an ancestor of p' in T', then the same holds in T), each vertex occurring in $\mathcal{P}'$ occurs in $\mathcal{P}$ as well. Thus, $q \in \mathcal{P}$. By hypothesis, we know that $X \in R_T(q)$ (because property (a) holds in T). Consequently, $X \in R_{T'}(q)$, as $R_T(p) = R_{T'}(p)$ for each vertex p of T'.

(b) Let q_1 and q_2 be two vertices of T and $X \in Sch_T(q_1) \cap Sch_T(q_2)$. Moreover, let q be a vertex on the path $\mathcal{P}$ from q_1 to q_2 in T. If $X \in R_T(q_1) \cap R_T(q_2)$, then $X \in R_T(q)$ by property (a), and (b) holds because $Sch_T(q) \supseteq R_T(q)$. If $X \in I_T(q_1) \cap R_T(q_2)$, then, from Lemma 7.4, $X \in R_T(q)$ where q is the parent of q_1 in T. Hence, from property (a) on the path from q to q_2, we derive that $X \in R_T(q) \subseteq Sch_T(q)$. Similar proof arguments apply to the cases $X \in R_T(q_1) \cap I_T(q_2)$ and $X \in I_T(q_1) \cap I_T(q_2)$. □

COROLLARY 7.6. *Let T be the e-join tree computed at a given step of DB-SHUNT, and s, p, and q be three vertices of T such that q is the parent of p and p is an ancestor of s in T. Then, the following properties hold.*

(*a*) $R_T(q) \cap Sch_T(s) \subseteq R_T(p)$.
(*b*) $I_T(q) \cap Sch_T(p) = I_T(q) \cap Sch_T(s) = \emptyset$.

PROOF

(a) From the definition of Sch_T, by distributivity, we have that

$$R_T(q) \cap Sch_T(s) = (R_T(q) \cap R_T(s)) \cup (R_T(q) \cap I_T(s)).$$

Let s' be the parent of s in T. From Lemma 7.4, $I_T(s) \subseteq R_T(s')$. Moreover, by virtue of Theorem 7.5(a), we have both $R_T(q) \cap R_T(s) \subseteq R_T(p)$ and $R_T(q) \cap R_T(s') \subseteq R_T(p)$. Therefore,

$$\begin{aligned} &R_T(q) \cap Sch_T(s) \\ &\quad = (R_T(q) \cap R_T(s)) \cup (R_T(q) \cap I_T(s)) \subseteq R_T(p) \cup (R_T(q) \cap R_T(s')) \subseteq R_T(p). \end{aligned}$$

(b) If q is the root of T, then Item (b) trivially holds, because $I_T(q) = \emptyset$ ($I_T(q)$ is initialized to the empty set and it is never modified during the execution of the algorithm). Otherwise, we denote the parent of q in T by q'. By definition of $Sch_T(p)$, we have that

$$\begin{aligned} I_T(q) \cap Sch_T(p) &= I_T(q) \cap (R_T(p) \cup I_T(p)) \\ &= (I_T(q) \cap R_T(p)) \cup (I_T(q) \cap I_T(p)). \end{aligned}$$

From Lemma 7.4, we have $I_T(p) \subseteq R_T(q)$. Therefore, $I_T(q) \cap I_T(p)$ is contained in $I_T(q) \cap R_T(q)$, which is empty by definition of Shunt. Thus,

$$\begin{aligned} (I_T(q) \cap R_T(p)) \cup (I_T(q) \cap I_T(p)) &\subseteq (I_T(q) \cap R_T(p)) \cup (I_T(q) \cap R_T(q)) \\ &= I_T(q) \cap R_T(p). \end{aligned}$$

By Lemma 7.4, $I_T(q) \subseteq R_T(q')$. Consequently,

$$I_T(q) \cap R_T(p) = I_T(q) \cap I_T(q) \cap R_T(p) \subseteq I_T(q) \cap R_T(q') \cap R_T(p).$$

By Theorem 7.5(a), $R_T(q') \cap R_T(p) \subseteq R_T(q)$. Hence,

$$I_T(q) \cap R_T(q') \cap R_T(p) \subseteq I_T(q) \cap R_T(q).$$

As noted above, $I_T(q) \cap R_T(q)$ is empty. Thus,

$$I_T(q) \cap Sch_T(p) = \emptyset.$$

We prove now that also $Sch_T(q) \cap Sch_T(s)$ is empty. By Theorem 7.5(b), $Sch_T(q) \cap Sch_T(s) \subseteq Sch_T(p)$. Therefore,

$$I_T(q) \cap Sch_T(s) = I_T(q) \cap Sch_T(q) \cap Sch_T(s) \subseteq I_T(q) \cap Sch_T(p) = \emptyset. \quad \square$$

We are now in the position to prove the correctness of DB-SHUNT.

THEOREM 7.7. *The Algorithm DB-SHUNT is correct.*

PROOF. Given a vertex p of an e-join tree T, we denote by $val(p, T)$ the projection on $Sch_T(p)$ of the join of all relations stored in the vertices of T_p—the subtree of T rooted at p; moreover, $val(T)$ denotes $val(\sigma, T)$, where σ is the root vertex of T.

Let $(Q, JT, \mathbf{db})$ be the input to DB-SHUNT, and let T be the e-join tree computed after step (1) and step (2) of the algorithm. By Lemma 7.1 and Lemma 7.2, it immediately follows that $val(T) \neq \emptyset$ if and only if Q evaluates to true on **db**. Thus, to prove the correctness of the DB-SHUNT algorithm, it is sufficient to show that $val(T)$ does not change during the **while** loop of the algorithm. (Note that the root vertex of T remains unaltered during the computation; thus, the schema of the relation $val(T)$ is always the same.)

Let q be a vertex of T. If q is a nonleaf vertex, whose left child and right child are p_1 and p_2, respectively, we define

$$l\text{-}in(q, T) = \prod_{R_T(q)} val(p_1, T) \qquad \text{and} \qquad r\text{-}in(q, T) = \prod_{R_T(q)} val(p_2, T).$$

If q is a leaf vertex, then $l\text{-}in(q, T) = U^{|R_T(q)|} = r\text{-}in(q, T)$, where U is the universe of the database, and $|R_T(q)|$ is the arity of the schema $R_T(q)$.

Then, $val(q, T)$ can be redefined in terms of $l\text{-}in$ and $r\text{-}in$.

$$val(q, T) = l\text{-}in(q, T) \bowtie Rel_T(q) \bowtie r\text{-}in(q, T). \tag{1}$$

Indeed, from Corollary 7.6, $R_T(q)$ contains every attribute X of $Sch_T(q)$ which belongs to (the schema of) any other vertex in the subtree T_q. Consequently, the projection on $R_T(q)$, appearing in the definition of $l\text{-}in(q, T)$ and $r\text{-}in(q, T)$, does not cause any loss of information as far as the join to $Rel_T(q)$ is concerned.

The following invariance property of $l\text{-}in$ and $r\text{-}in$, whose formal proof is given in Appendix B, will allow us to prove the theorem immediately.

CLAIM A. *Let T' be the e-join tree resulting from the execution of either step (4)(a) or step (4)(b) on an e-join tree T, and q be a vertex of T'. Then,*

$$l\text{-}in(q, T') = l\text{-}in(q, T) \qquad \textit{and} \qquad r\text{-}in(q, T') = r\text{-}in(q, T).$$

Observing that neither the schema nor the relation of the root vertex of T change during the computation, the statement of the theorem follows immediately from Claim A and Eq. (1). □

We next evaluate the complexity of the DB-SHUNT algorithm with respect to the parallel database model introduced above.

We first need to determine the size of the initial e-join tree. In the following, for any data structure S, $\|S\|$ denotes the size of S in terms of storage volume (i.e., the number of bits needed for its encoding).

LEMMA 7.8. *Given an instance* $\mathcal{T} = (Q, JT, \mathbf{db})$ *of JTREE, let* $\mathcal{T}' = (Q', JT', \mathbf{db}')$ *be its NF version, and* $T = ET(\mathcal{T}')$ *be the e-join tree of* $\mathcal{T}'$. *Then, the following holds.*

(1) —$\|JT'\| = O(\|JT\|)$;
—$\|Q'\| = O(\|Q\|)$;
—$\|\mathbf{db}'\| = O(\|\mathbf{db}\| \cdot \|Q\|)$;
—*for each relation* r *in* $\mathbf{db}'$, $\|r\| \leq d$.

(2) —$\|T\| = O(\|\mathbf{db}\| \cdot \|Q\|) = O(\|\mathcal{T}\|^2)$;
—*for each relation* r *stored in* T, $\|r\| \leq d$.

where d *is the maximum size of the relations in* $\mathbf{db}$.

PROOF

(1) The "binarization" (see proof of Lemma 7.1) of JT may at most double the number of vertices of the join tree. Thus, also the numbers of atoms in Q may be doubled (at most). No further new atoms and vertices are needed; thus, $\|JT'\| = O(\|JT\|)$, and $\|Q'\| = O(\|Q\|)$.

Relation duplication occurs if and only if two atoms with the same predicate occur in the query. Thus, $\|\mathbf{db}'\| = O(\|\mathbf{db}\| \cdot \|Q\|)$. Each new relation in $\mathbf{db}'$ is obtained by a select-project operation on a relation r of $\mathbf{db}$, so that its size cannot exceed $\|r\|$.

(2) The vertices of the e-join tree T of $\mathcal{T}'$ are exactly the relations in $\mathbf{db}'$ (with negligible additional schema information). □

THEOREM 7.9. *Let* $(Q, JT, \mathbf{db})$ *be an instance of JTREE. By applying the DB-SHUNT algorithm,* $(Q, JT, \mathbf{db})$ *can be solved on a parallel DB-machine*

(*a*) *by a sequence of parallel steps of length* $O(\log n)$;
(*b*) *by using* $O(n)$ *relational processors*;
(*c*) *involving* $O(n)$ *joins in total*;
(*d*) *with* $O(n)$ *intermediate relations, having size* $O(d^2)$,

where n *is the number of atoms in* Q, *and* d *is the size of the largest relation in* $\mathbf{db}$.

PROOF. First, observe that the number n' of vertices in the initial e-join tree $T = ET(\mathcal{T}')$ is equal to the number of query atoms in the NF version $\mathcal{T}'$ of $\mathcal{T}$, which is $O(n)$ by Lemma 7.8.

Instructions (1)–(3) of DB-SHUNT can be executed in $O(\log n)$ parallel steps of the DB-machine by using $O(n)$ relational processors. The most expensive database operation involved in these instructions is relation-duplication (that can be needed in instruction (2); it is feasible in the specified bounds as follows. In the worst case, we have to make n copies of the same relation. To this end, one relational processor first makes one copy, the two available relations are then given to two processors obtaining four copies, and so on. After $(\log n) - 1$ steps, we have $2^{\log n} = n$ copies by having used $n/2$ processors.

The **while** cycle (instruction (4)) is executed $\log((n' + 1)/2)$ times, that is, $O(\log n)$ times (recall that n' is the number of vertices in the initial e-join tree $T = ET(\mathcal{T}'))$. Indeed, the number of leaves of T is $(n' + 1)/2$, half of the leaves are deleted after each iteration, and no new leaves are created. Each single shunt operation involved in instruction (4)(a) or instruction (4)(b) of the **while** loop can be executed in one step by one processor of the DB-machine (as it involves a constant number of database operations). The processor in charge of the shunt on leaf l works on l, on the sibling of l and on the parent of l, and takes, as additional input, a copy of the schema of the grandparent of l. Thus, the execution of (one of) instructions (4)(a) and (4)(b) of DB-SHUNT can be fully parallelized on the DB-machine (no concurrency arises), and performed in one (parallel) step. Consequently, one complete **while** iteration requires two steps, to execute instruction (4)(a) and instruction (4)(b) in the specified order, and at most $(n' + 1)/4 = O(n)$ relational processors of the DB-machine. One final step of the DB-machine performs instruction (5). Hence, the entire execution of DB-SHUNT requires $O(\log n)$ (parallel) steps of the DB-machine, where each step employs $O(n)$ relational processors.

The number of joins involved in the complete execution of the **while** loop is exactly $n' - 3$, because each shunt operation performs two joins and eliminates two vertices from the e-join tree (note that exactly $(n' - 3)/2$ shunt operations are executed as, after the **while** loop, the tree contains exactly three vertices). Thus, the total number of joins is $O(n)$.

Concerning part (d) of the theorem, it is clear that the number of intermediate relations is at most $n' = O(n)$ (the number decreases during the **while** execution).

To prove the size bound $O(d^2)$, we prove that, at each step of the **while**, for any vertex s of the current e-join tree T, the following two conditions hold.

$$(\mathbf{g}_1)\ \Big\|\prod_{R_T(s)} Rel_T(s)\Big\| \leq d; \qquad \text{and} \qquad (\mathbf{g}_2)\ \Big\|\prod_{I_T(s)} Rel_T(s)\Big\| \leq d.$$

These conditions clearly entail $\|Rel_T(s)\| \leq d^2$, since

$$Rel_T(s) \subseteq \prod_{R_T(s)} Rel_T(s) \times \prod_{I_T(s)} Rel_T(s),$$

where $\times$ denotes Cartesian product.

($\mathbf{g}_1$). At the initial step, T is initially equal to $ET(\mathcal{T}')$, and $\|\Pi_{R_T(s)}\ Rel_T(s)\| \leq d$ by Lemma 7.8.

During the computation, condition (g_1) holds, because $R_T(s)$ remains always the same and the number of its tuples (which are kept in $\Pi_{R_T(s)}\ Rel_T(s))$ is monotonically decreasing (because of possible joins to other relations).

($\mathbf{g}_2$). If s is the root of T, then property g_2 trivially holds, as the root never changes during the **while** loop (no shunt can be applied to the root or to its children). Suppose now that s is not the root.

CLAIM. *Let q be the parent of s in T. Then,*

$$\|\prod_{R_T(q)} Rel_T(s)\| \leq d. \tag{2}$$

PROOF OF CLAIM. (Recall that projection attributes that are not part of the schema of the relation to be projected are irrelevant. Thus, $\Pi_{R_{\mathrm{T}}(q)}\ Rel_{\mathrm{T}}(s) = \Pi_{R_{\mathrm{T}}(q) \cap Sch_{\mathrm{T}}(s)}\ Rel_{\mathrm{T}}(s)$.)
We proceed by induction on the execution steps.
The basis is trivial.

Induction step. Let T be the e-join tree obtained by the execution of step (4)(a) or step (4)(b) on e-join tree T' which satisfies property (2). If q is the parent of s already in T', then property (2) holds by the induction hypothesis (as s has been left untouched by the last execution step). Otherwise, let p and l be the parent of s and the sibling of s in T', respectively. s takes the place of p in T because of a shunt operation applied on l (like in Figure 16). By definition of shunt,

$$Rel_{\mathrm{T}}(s) = \prod_{Sch_{\mathrm{T}}(s)} Rel_{\mathrm{T'}}(l) \rtimes Rel_{\mathrm{T'}}(p) \bowtie Rel_{\mathrm{T'}}(s).$$

Thus,

$$\|\prod_{R_{\mathrm{T}}(q)} Rel_{\mathrm{T}}(s)\| \leq \|\prod_{R_{\mathrm{T}}(q)} Rel_{\mathrm{T'}}(p) \bowtie Rel_{\mathrm{T'}}(s)\|.$$

On the other hand, by Corollary 7.6, $Sch_{\mathrm{T'}}(s) \cap R_{\mathrm{T'}}(q) \subseteq R_{\mathrm{T'}}(p)$. Hence, $Sch_{\mathrm{T'}}(s) \cap R_{\mathrm{T}}(q) \subseteq R_{\mathrm{T'}}(p)$, as $R_{\mathrm{T}}(q) = R_{\mathrm{T'}}(q)$. It follows that $Sch_{\mathrm{T'}}(s) \cap R_{\mathrm{T}}(q) \subseteq Sch_{\mathrm{T'}}(p)$, as $R_{\mathrm{T'}}(p) \subseteq Sch_{\mathrm{T'}}(p)$. Thus, we obtain the following chain of equations and inequations:

$$\|\prod_{R_{\mathrm{T}}(q)} Rel_{\mathrm{T'}}(p) \bowtie Rel_{\mathrm{T'}}(s)\| \leq \|\prod_{R_{\mathrm{T}}(q)} \left(Rel_{\mathrm{T'}}(p) \bowtie \prod_{R_{\mathrm{T}}(q) \cap Sch_{\mathrm{T'}}(s)} Rel_{\mathrm{T'}}(s) \right)\| \tag{3}$$

$$\|\prod_{R_{\mathrm{T}}(q)} \left(Rel_{\mathrm{T'}}(p) \bowtie \prod_{R_{\mathrm{T}}(q) \cap Sch_{\mathrm{T'}}(s)} Rel_{\mathrm{T'}}(s) \right)\| = \|\prod_{R_{\mathrm{T}}(q)} (Rel_{\mathrm{T'}}(p) \ltimes \prod_{R_{\mathrm{T}}(q) \cap Sch_{\mathrm{T'}}(s)} Rel_{\mathrm{T'}}(s))\| \tag{4}$$

$$\|\prod_{R_{\mathrm{T}}(q)} \left(Rel_{\mathrm{T'}}(p) \ltimes \prod_{R_{\mathrm{T}}(q) \cap Sch_{\mathrm{T'}}(s)} Rel_{\mathrm{T'}}(s) \right)\| \leq \|\prod_{R_{\mathrm{T}}(q)} Rel_{\mathrm{T'}}(p)\| \tag{5}$$

$$\|\prod_{R_{\mathrm{T}}(q)} Rel_{\mathrm{T'}}(p) \bowtie Rel_{\mathrm{T'}}(s)\| \leq \|\prod_{R_{\mathrm{T}}(q)} Rel_{\mathrm{T'}}(p)\|. \tag{6}$$

In particular, Inequations (3) and (5) follow from basic relational algebra laws. Equation (4) holds because the schema of the right term of the join is a subset of the schema of the left term of the join (above we pointed out that $Sch_{T'}(s) \cap R_{\mathrm{T}}(q) \subseteq Sch_{T'}(p)$). Inequation (6) is obtained by the Inequations (3) and (5) and by Eq. (4) by transitivity.

Finally, from the induction hypothesis, we have

$$\Big\| \prod_{R_{\mathrm{T}}(q)} Rel_{\mathrm{T}'}(p) \Big\| \leq d.$$

By Lemma 7.4, $I_{\mathrm{T}}(s) \subseteq R_{\mathrm{T}}(q)$. Thus, property g_2 follows immediately from the claim above. In turn, g_1 and g_2 imply item (d) of the theorem. □

Let us discuss how the DB-SHUNT algorithm can be adapted to solve Boolean queries of bounded treewidth, bounded degree of cyclicity, or bounded query-width, once a suitable decomposition of the query is provided. First note that it is sufficient to consider bounded query-width. Indeed, as mentioned in Section 6, query-width generalizes the concept of treewidth, and, moreover, by Lemma 6.17, for each hinge-tree decomposition HD of a Boolean query Q having degree of cyclicity c, a query decomposition of width c of Q can be computed in logspace, and thus in a highly parallel fashion, from HD.

Let Q be a Boolean conjunctive query over a database **db**, and $\langle T, \lambda \rangle$ be a query decomposition of Q of width at most k, for some constant k. As shown in the proof of Theorem 6.4, we can compute in logspace an instance (Q', JT, **db**$'$) of JTREE equivalent to (Q, **db**, $\langle T, \lambda \rangle$). In terms of parallel database operations, this computation is very simple. In fact, for each vertex v of the query-decomposition tree T, it is sufficient to perform the natural join of the (at most k) relations corresponding to the atoms in $\lambda(v)$. In our parallel database machine model, computing this natural join corresponds to a sequence of at most $k - 1$ single joins and thus requires at most $k - 1$ sequential steps. This can be done in parallel for each vertex of T. After this preprocessing phase, the DB-SHUNT algorithm is used to evaluate the instance (Q', JT, **db**$'$). Thus, in terms of parallel steps, the total cost of evaluating the original instance (Q, **db**, $\langle T, \lambda \rangle$) is $O(k - 1 + \log n) = O(\log n)$.

7.3. A Parallel Algorithm for Non-Boolean ACQs. In this section, we present a parallel algorithm ACQ that extends DB-SHUNT to the case of non-Boolean queries. Answering acyclic conjunctive queries requires exponential time in the worst case, since Q(**db**) may contain an exponential number of tuples. The problem is tractable for *bounded-projection queries*, that is, conjunctive queries whose head atoms have arity bounded by some fixed constant. For bounded-projection queries, the size of the output relation corresponding to the atom *ans* is polynomial in the size of the input database instance **db**. Even though the algorithm ACQ presented in this section is correct for general acyclic conjunctive queries, it is guaranteed to use polynomial workspace only in case of bounded-projection queries.

The ACQ algorithm takes a triple (Q, JT, **db**) as the input, where JT is a join tree for the (non-Boolean) acyclic conjunctive query Q on **db**. We denote by $Out(Q)$ the set of projection attributes of Q (i.e., the attributes corresponding to the variables occurring in the head of Q). The output of the algorithm is a relation *ans* over attributes $Out(Q)$, representing the answer of Q on **db**.

The method is very similar to the algorithm for JTREE of Section 7.2. The main difference is that projection attributes are never deleted, but they are preserved during all the computation. To this end, the relation schema $Sch_{\mathrm{T}}(p)$ of a vertex p of an e-join tree T, is now partitioned in three (instead of two)

```
ALGORITHM ACQ
Input: A triple $(Q, JT, \mathbf{db})$ where $JT$ is a join tree for the acyclic conjunctive query $Q$ on $\mathbf{db}$.
Output: The answer to $Q$ over $\mathbf{db}$.
 begin
(1)    Compute the NF version $(Q', JT', \mathbf{db}')$ of $(Q, JT, \mathbf{db})$;
(2)    $T := ET((Q', JT', \mathbf{db}'))$;
       Let $\lambda$ be the number of leaves in $T$;
(3)    Label the leaves of $T$ in order from left to right as $1, \ldots, \lambda$;
(4)    while $depth(T) > 1$ do
         (a)  in parallel apply the shunt operation to all odd-numbered leaves that
              are the left children of their parents, and have depth greater than 1;
         (b)  in parallel apply the shunt operation to all odd-numbered leaves that
              are the right children of their parents, and have depth greater than 1;
         (c)  shift out the rightmost bit in the labels of all remaining leaves;
       end (* while *)
       Let $p$ be the root of $T$, and let $p'$ and $p''$ be the children of $p$ in $T$;
(5)    output $\prod_{Out(Q)}(Rel_T(p) \bowtie Rel_T(p') \bowtie Rel_T(p''))$
end.
```

FIG. 24. A parallel algorithm for ACQ.

distinguished sets of attributes: $R_T(p)$, $I_T(p)$, and $O_T(p)$. The set $O_T(p)$ will contain projection attributes (occurring in the schema of the output relation *ans*).

The e-join tree $T = ET(\mathcal{T})$ is initialized as for JTREE. Indeed, for each vertex p of T, $O_T(p)$ is set to $\emptyset$. The shunt operation is redefined as follows.

Let l be a leaf of an e-join tree T, p the parent of l, s the other child of p, and q the parent of p. The *shunt operation* applied to l results in a new contracted e-join tree T' in which l and p are deleted, and s is transformed (as specified below) and takes the place of vertex p (i.e., it becomes child of q). The schema and the relation of the transformed version of s in T' are specified below.

$$R_{T'}(s) = R_T(s).$$

$$I_{T'}(s) = (R_T(q) \cap Sch_T(p)) - R_T(s).$$

$$O_{T'}(s) = ((Sch_T(l) \cup Sch_T(p) \cup Sch_T(s)) \cap Out(Q)) - (R_{T'}(s) \cup I_{T'}(s)).$$

$$Rel_{T'}(s) = \prod_{Sch_{T'}(s)} (Rel_T(l) \bowtie Rel_T(p) \bowtie Rel_T(s)).$$

where $Out(Q)$ is the set of attributes of the output relation *ans*.

Thus, each projection attribute occurring in a deleted vertex (p or l) is kept in $Rel_{T'}(s)$ (if it does not belong to $R_{T'}(s) \cup I_{T'}(s)$, then it is added to $O_{T'}(s)$).

Figure 24 shows a parallel algorithm for the computation of acyclic conjunctive queries.

The Algorithm ACQ works essentially in the same time and space bounds of DB-SHUNT, specified in Theorem 7.9. The only difference is an increase of the size of the intermediate relations which is in Theorem 7.9. The only difference is an increase of the size of the intermediate relations which is $O(d^{2+k})$ for ACQ, where k is the number of projection attributes (i.e., the cardinality of $Out(Q)$) and d is the size of the largest input relation. Thus, the working space of ACQ remains polynomial for bounded-projection queries. Correctness and complexity of ACQ stem from slight modifications of the proofs of Theorem 7.7 and Theorem 7.9, respectively.

As noted before, for queries whose number of projection attributes is not bounded by a constant, the output relation can be exponentially larger than the input; thus, the best algorithm will require workspace polynomial in the combined size of the input instance and of the output relation, as for Yannakakis's sequential algorithm [Yannakakis 1981]. We developed such an "output polynomial" parallel algorithm, together with a (parallel) method for fully reducing a database by a parallel semijoin program. These algorithms are quite sophisticated, and for space reasons we cannot describe them here. The interested reader is referred to Gottlob et al. [2000].

7.4. RELATED PARALLEL ALGORITHMS. As pointed out in Section 5, Constraint Satisfaction (CS) is very similar to conjunctive query evaluation, and therefore parallel algorithms proposed in the literature for solving CS can be used to solve JTREE.

Kasif and Delcher [1994] presented an NC^2 parallel algorithm for acyclic constraint networks having only binary constraints. In the database context, this problem corresponds to evaluating an acyclic conjunctive query Q over a database where all relations are binary (consequently, the hypergraph $H(Q)$ of Q is a tree in this case). A comparison of DB-SHUNT to Kasif and Delcher's algorithm yields the following:

—DB-SHUNT applies to general instances of JTREE while the algorithm by Kasif and Delcher applies only to the restricted case where all database relations are binary.

—Kasif and Delcher's algorithm is not formulated in database terms, and it is not completely obvious how to implement their algorithm on a parallel database machine. (They use Boolean matrix multiplication as primitive.)

—Both approaches have essentially the same parallel complexity (in the PRAM model), when restricted to those instances for which Kasif and Delcher's method is defined, that is, for databases whose relations have arity two. Our method, however, maintains the same complexity even in the unbounded arity case.

A parallel algorithm for more general acyclic networks, admitting also nonbinary constraints, is due to Zhang and Mackworth [1993]. This algorithm is called PTAC.

The main facts we observed while comparing DB-SHUNT to PTAC are the following:

—The algorithm is formulated in a database context, that is, in a setting similar to the one adopted here.

—PTAC is formulated for join-trees where the relations have bounded arities.

—In Zhang and Mackworth [1993], it is implicitly assumed that the size of the database universe is bounded by a constant. In fact it is stated in Zhang and Mackworth [1993], that "*the number of tuples in a relation schema of size $O(\log n)$ is bounded by poly(n)*." Moreover, it is assumed in Zhang and Mackworth [1993] that joins and semi-joins can be performed in constant time.

From these assumptions, it is clear that the authors of Zhang and Mackworth [1993] implicitly assume a bounded universe.

—DB-SHUNT requires neither bounded relation arity nor a fixed database universe.

—By applying arguments similar to those in the proof of Theorem 7.9, we can show that PTAC also works in the case of unbounded arities and unbounded universe (even though this was not proved in Zhang and Mackworth [1993]). However, if applied to databases with unbounded universes and relations of unbounded arities, PTAC requires intermediate relations of size $O(d^3)$, where d is, as in Theorem 7.9, the size of the largest relation in the input database. Recall that DB-SHUNT requires intermediate relations of size $O(d^2)$ only.

—DB-SHUNT requires fewer parallel steps to terminate. The difference, however, consists only of a constant factor. Asymptotically, the algorithms use the same number of parallel steps.

8. *Conclusion and Further Research*

In this paper, we have determined the precise complexity of an important database problem, the evaluation of acyclic Boolean conjunctive queries (ABCQ). We have shown that this problem is complete for LOGCFL, a very low complexity class consisting of highly parallelizable problems. We have thus pinpointed the parallel complexity of this problem, locating it between the parallel complexity classes NC^1 and AC^1. A practical consequence is that one can design AC^1 and NC^2 algorithms for ABCQ and for all equivalent problems, but most likely no NC^1 algorithm.

On the theoretical side, we believe that our results provide new insights concerning the complexity-class LOGCFL. While the previously known complete problems for LOGCFL apparently bear two sources of nondeterminism, namely, the choice of a suitable tree shape and the choice of a locally consistent fill-in for a given tree structure (for details, see Section 1.2), all new problems shown LOGCFL-complete in the present paper bear only the second source of nondeterminism. This leads to a new intuitive characterization of LOGCFL as the class capturing the complexity of filling a given tree (or acyclic data structure) with "locally consistent" data items.

We have shown that the ABCQ problem is equivalent to a number of important database and AI problems. Consequently, we have established LOGCFL-completeness results for all these problems. In particular, we have pinpointed the complexity of acyclic constraint satisfaction, a problem which has received much interest in the literature.

We have generalized our results to problems whose associated hypergraphs are of bounded width. More specifically, we have considered three different generalizations of acyclicity that have recently been studied by various authors: bounded treewidth, bounded query-width, and bounded degree of cyclicity. We have proven that the main decision problems considered in our paper remain LOGCFL-complete under these generalizations. For a comparison of these and further generalizations of acyclicity described in the literature, see Gottlob et al. [2000c].

Finally, we presented parallel database algorithms for evaluating acyclic conjunctive queries. These algorithms exploit a high degree of inter-operation parallelism on parallel databases machines (where it is assumed that intra-operation parallelism is taken care of automatically). This is—to our best knowledge—the first time that the parallelism inherent in acyclic queries is exploited in the context of database query optimization. We believe that our model of parallel databases is rather realistic and that our algorithms can be easily adapted to fit existing parallel database management systems.

The parallel algorithms presented here have been refined and extended in a recent paper [Gottlob et al. 2000]. In particular, in Gottlob et al. [2000], we show that the algorithm ACQ presented here can be improved to work with "output-polynomial" storage-size, that is, requires storage-sized polynomial in the combined size of the input instance and of the output. This is reminiscent of Yannakakis's well-known sequential algorithm that computes the answer to a conjunctive query in output-polynomial time [Yannakakis 1981]. In Gottlob et al. [2000], we also show how to fully reduce a set of relations by a parallel semijoin program.

In the future, we plan to develop parallel algorithms for nondatabase problems such as the acyclic clause subsumption problem. As said, clause subsumption is an essential feature of theorem provers and parallelizing this problem could be of great benefit. Also, constraint satisfaction algorithms for CSPs of bounded width, that run on existing parallel hardware, are a tempting research issue.

Appendixes

A. Proof of Claim A of Theorem 4.3

CLAIM A. *Q evaluates to true over* **db** *if and only if there exist n local substitutions $\lambda_1, \ldots, \lambda_n$, for the atoms $A_1, \ldots, A_n$, respectively, such that,* (*i*) *for $1 \le i \le n$, $A_i\lambda_i \in$* **db**, *and* (*ii*) *for $1 < i \le n$, λ_i agrees with $\lambda_{\pi(i)}$.*

PROOF

Only if Direction. Assume Q evaluates to *true*. Then there exists a (global) substitution ϑ: $var(Q) \rightarrow U$ such that, for $1 \le i \le n$, $A_i\vartheta \in$ **db**. For $1 \le i \le n$, let λ_i be the restriction of ϑ to the domain $var(A_i)$. Clearly all λ_i, λ_j, $1 \le i, j \le n$, mutually agree.

If Direction. Assume local substitutions $\lambda_1, \ldots, \lambda_n$ with the described properties exist. Assume that $X \in var(Q)$ is a variable such that $X \in dom(\lambda_i) \cap dom(\lambda_j)$ for some indices $1 \le i, j \le n$. By Property (2) of the definition of a join forest, each atom on the unique path Π from A_i to A_j contains X. Since this path Π is a chain of nodes where any two consecutive nodes are in a parent-child relationship in T, it follows that, for any consecutive nodes A_r and A_s on Π, $\lambda_r(X) = \lambda_s(X)$. Hence, $\lambda_i(X) = \lambda_j(X)$. Let $\vartheta = \lambda_1 \cup \lambda_2 \cup \cdots \cup \lambda_n$. Clearly, ϑ is a well-defined substitution, and, for all $A_i \in atoms(Q)$, $A_i\vartheta = A_i\lambda_i \in$ **db**. □

B. Proof of Claim A of Theorem 7.7

CLAIM A. *Let T' be the e-join tree resulting from the execution of either step* (4*a*) *or step* (4*b*) *on an e-join tree T, and q be a vertex of T'. Then,*

$$l\text{-}in(q, T') = l\text{-}in(q, T) \qquad \textit{and} \qquad r\text{-}in(q, T') = r\text{-}in(q, T).$$

PROOF. First, observe that q does exist also in T, as no new vertex is created by step (4a) or step (4b). We proceed by induction on the depth of T'_q, that is, on the depth of the subtree of T' rooted at q.

Basis ($depth(T'_q) = 0$). q is a leaf vertex of T'. As a consequence, q is also a leaf of T (it is not subject to any shunt operation in the last step). Thus, by definition of *l-in* and *r-in*, we have that $l\text{-}in(q, T') = U^{|R_T(q)|} = l\text{-}in(q, T)$ and $r\text{-}in(q, T') = U^{|R_T(q)|} = r\text{-}in(q, T)$.

Induction ($depth(T'_q) = d > 0$). We demonstrate the invariance of *l-in* under the execution of step (4a); the proof of the other cases is similar and can be easily derived from this one.

Let T' be the e-join tree resulting from the application of step (4a) to the e-join tree T, and q be a vertex of T', we have to prove that

$$l\text{-}in(q, T') = l\text{-}in(q, T). \tag{7}$$

Let p be the left child of q in T. If p is the left child of q in T' as well, then by the induction hypothesis, we have $l\text{-}in(p, T') = l\text{-}in(p, T)$ and $r\text{-}in(p, T') = r\text{-}in(p, T)$. Moreover, $Rel_T(p) = Rel_{T'}(p)$, as the relation of a vertex that has kept the same parent cannot have been changed by the shunt operation. Consequently, we have also $val(p, T') = val(p, T)$. Hence,

$$l\text{-}in(q, T') = \prod_{R_{T'}(q)} val(p, T') = \prod_{R_T(q)} val(p, T) = l\text{-}in(q, T).$$

(Note that the R part of the schema of a vertex never changes during the computation, that is, for each vertex q of T', $R_{T'}(q) = R_T(q)$ holds.)

Suppose now that p is not the left child of q in T'. Thus, p has been deleted in step (4a) by a shunt operation applied on its left child, say l. Consequently, the sibling of l in T, say s, has become the left child of q in T' and we are in the situation described in Figure 16.

By definition of *l-in*, for the left term of Eq. (7), we obtain

$$l\text{-}in(q, T') = \prod_{R_{T'}(q)} val(s, T') = \prod_{R_{T'}(q)} (Rel_{T'}(s) \bowtie l\text{-}in(s, T') \bowtie r\text{-}in(s, T')).$$

By the induction hypothesis, the rightmost term above yields

$$\prod_{R_{T'}(q)} (Rel_{T'}(s) \bowtie l\text{-}in(s, T) \bowtie r\text{-}in(s, T)). \tag{8}$$

Since $Rel_{T'}(s)$ is obtained by applying a shunt operation on the sibling l of s in T, by definition of shunt, (8) is equal to

$$\prod_{R_{T'}(q)} \left(\left(\prod_{Sch_{T'}(s)} (Rel_T(l) \rtimes Rel_T(p) \bowtie Rel_T(s)) \right) \bowtie l\text{-}in(s, T) \bowtie r\text{-}in(s, T) \right).$$

Since all attributes occurring in $l\text{-}in(s, T)$ and $r\text{-}in(s, T)$ belong to $R_T(s) = R_{T'}(s)$ (and, therefore, to $Sch_{T'}(s)$), $l\text{-}in(s, T)$ and $r\text{-}in(s, T)$ can be moved inside $\Pi_{Sch_{T'}(s)}$. Thus, the above term can be rewritten as

$$\prod_{R_{T'}(q)} \prod_{Sch_{T'}(s)} (Rel_T(l) \rtimes Rel_T(p) \bowtie Rel_T(s) \bowtie l\text{-}in(s, T) \bowtie r\text{-}in(s, T)),$$

which is in turn equal to

$$\prod_{R_{T'}(q)} \prod_{Sch_{T'}(s)} (Rel_T(l) \rtimes Rel_T(p) \bowtie val(s, T)). \tag{9}$$

Since the composition of two projections is equal to one projection on the intersection of the two projection-attribute sets, the composed projection above is equal to one projection on the set of attributes $A = R_{T'}(q) \cap Sch_{T'}(s) = R_{T'}(q) \cap (R_{T'}(s) \cup I_{T'}(s))$. By definition of shunt, $I_{T'}(s) = (R_T(q) \cap Sch_T(p)) - R_T(s)$; moreover, for any vertex x of T', $R_{T'}(x) = R_T(x)$. It follows that

$$A = R_T(q) \cap (R_T(s) \cup ((R_T(q) \cap Sch_T(p)) - R_T(s))).$$

By simple set-algebraic transformations, we get

$$A = R_T(q) \cap (R_T(s) \cup (R_T(q)ths) \cap Sch_T(p))).$$

$$A = (R_T(q) \cap R_T(s)) \cup (R_T(q) \cap Sch_T(p)).$$

Since q is the parent of p, which is in turn the parent of s in T, by Corollary 7.6, $R_T(q) \cap R_T(s) \subseteq R_T(p) \subseteq Sch_T(p)$. Therefore, we have that

$$A = R_T(q) \cap Sch_T(p).$$

Thus, by projection properties, term (9) equals

$$\prod_{R_T(q)} \prod_{Sch_T(p)} (Rel_T(l) \rtimes Rel_T(p) \bowtie val(s, T)).$$

Since $Rel_T(l) \rtimes Rel_T(p) \bowtie val(s, T)$ is equal to the join of all vertices in T_p (the subtree of T rooted at p), from definition of *val*, the expression above is equal to

$$\prod_{R_T(q)} val(p, T) = l\text{-}in(q, T).$$

which closes the chain of equalities and the proof of the claim. □

ACKNOWLEDGMENTS. We thank Hans Tompits and Reinhard Pichler for their useful comments on earlier versions of this paper. Moreover, we are very indebted to the anonymous referees for suggesting a large number of improvements.

REFERENCES

ABITEBOUL, S., HULL, R., AND VIANU, V. 1995. *Foundations of Databases*. Addison-Wesley, Reading, Mass.

ARNBORG, S., LAGERGREN, J., AND SEESE, D. 1991. Problems easy for tree-decomposable graphs. *J. Algorithms 12*, 308–340.

BACHMAIR, L., CHEN, T., RAMAKRISHNAN, C. R., AND RAMAKRISHNAN, I. V. 1996. Subsumption algorithms based on search trees. In *Proceedings of the 21st Colloquium on Trees in Algebra and*

Programming (*CAAP '96*) (Linköping, Sweden, Apr. 22–24). Lecture Notes in Computer Science, vol. 1059. Springer-Verlag, New York, pp. 135–148.

BALCÁZAR, J. L., DÍAZ, J., AND GABARRÓ, J. 1988. *Structural Complexity I*. EATCS Monographs on Theoretical Computer Science. Springer-Verlag, Berlin, Germany.

BÉDARD, F., LEMIEUX, F., AND MCKENZIE, P. 1993. Extensions to Barrington's M-programs. *Theoret. Comput. Sci. 107*, 31–61.

BEERI, C., FAGIN, R., MAIER, D., MENDELZON, A., ULLMAN, J. D., AND YANNAKAKIS, M. 1981. Properties of acyclic database schemes. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing* (*STOC '81*) (Milwaukee, Wis., May 11–13). ACM, New York, pp. 355–362.

BEERI, C., FAGIN, R., MAIER, D., AND YANNAKAKIS, M. 1983. On the desiderability of acyclic database schemes. *J. ACM 30*, 3 (July), 479–513.

BERGE, C. 1976. *Graphs and Hypergraphs*. North-Holland, Amsterdam, The Netherlands.

BERNSTEIN, P. A., AND CHIU, D. W. 1981. Using semi-joins to solve relational queries. *J. ACM 28*, 1 25–40.

BERNSTEIN, P. A., AND GOODMAN, N. 1981. The power of natural semijoins. *SIAM J. Comput. 10*, 4, 751–771.

BORODIN, A., COOK, S. A., DYMOND, P. W., RUZZO, W. L., AND TOMPA, M. 1989. Two applications of inductive counting for complementation problems. *SIAM J. Comput. 18*, 559–578.

CHANDRA, A. K., AND MERLIN, P. M. 1977. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing* (*STOC '77*) ACM, New York, pp. 77–90.

CHANG, C. L., AND LEE, R. C. T. 1973. *Symbolic Logic and Mechanical Theorem Proving*. Academic Press, New York.

CHEKURI, CH., AND RAJARAMAN, A. 2000. Conjunctive query containment revisited. *Theoret. Comput. Sci. 239*, 2, 211–229.

CHASE, K. 1981. Join graphs and acyclic data base schemas. In *Proceedings of the International Conference on Very Large Databases* (*VLDB '81*) (Cannes, France). pp. 95–100.

COOK, S. A. 1971. Characterizations of pushdown machines in terms of time-bounded computers. *J. ACM 18*, 1, 4–18.

COOK, S. A. 1985. A taxonomy of problems with fast parallel algorithms. *Inf. Contr. 64*, 2–22.

COOK, S. A., AND MCKENZIE, P. 1987. Problems complete for deterministic logarithmic space. *J. Algorithms 8*, 385–394.

DAHLHAUS, E. 1990. Chordale Graphen im besonderen Hinblick auf parallele Algorithmen. Habilitationsschrift, Universität Bonn, Bonn, Germany. Unpublished.

DAHLHAUS, E., AND KARPINSKI, M. 1996. The matching problem for strongly chordal graphs is in NC. Tech. Report 855-CS. Institut für Informatik, Universität Bonn, Bonn, Germany.

D'ATRI, A., AND MOSCARINI, M. 1986. Recognition algorithms and design methodologies for acyclic database schemes. In *Advances in Computing Research*, vol. 3, P. Kanellakis, ed. JAI press, London, England, pp. 43–68.

DECHTER, R., AND PEARL, J. 1989. Tree clustering for constraint networks. *Artif. Int. 38*, 353–366.

DEWITT, D. J., AND GRAY, J. 1992. Parallel database systems: The future of high-performance database systems. *Communic. ACM 35*, 6 (June), 85–98.

DWORK, C., KANELLAKIS, P. C., AND MITCHELL, J. C. 1984. On the sequential nature of unification. *J. Logic Prog. 1*, 1, 35–50.

FAGIN, R. 1983. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM 30*, 3, 514–550.

FAGIN, R., MENDELZON, A. O., AND ULLMAN, J. D. 1982. A simplified universal relation assumption and its properties. *ACM Trans. Datab. Syst. 7*, 3 (Sept.), 343–360.

GAREY, M. R., AND JOHNSON, D. S. 1979. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Freeman and Co., New York.

GOODMAN, N., AND SHMUELI, O. 1982. Tree queries: A simple class of relational queries. *ACM Trans. Datab. Syst. 7*, 4 (Dec.), 653–677.

GOODMAN, N., AND SHMUELI, O. 1983. Syntactic characterization of tree database schemas. *J. ACM 30*, 4, 767–786.

GOTTLOB, G., LEONE, N., AND SCARCELLO, F. 2000. Advanced parallel algorithms for processing acyclic conjunctive queries, rules, and constraints. In *Proceedings of the 2000 Conference on Software Engineering and Knowledge Engineering* (*SEKE '00*) (Chicago, Ill., July). pp. 167–176. Full version:

Technical Report DBAI-TR-98/18 (available on the web at http://www.dbai.tuwien.ac.at/staff/gottlob/parallel.ps, or by e-mail from the authors).

Gottlob, G., Leone, N., and Scarcello, F. 2000a. Computing LOGCFL certificates. *Theoret. Comput. Sci.*, to appear.

Gottlob, G., Leone, N., and Scarcello, F. 2000b. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, to appear.

Gottlob, G., Leone, N., and Scarcello, F. 2000c. A comparison of structural CSP decomposition methods. *Artif. Int. 124*, 2, 243–282.

Graham, M. H. 1979. On the universal relation. Tech. Report, Univ. Toronto, Toronto, Ont., Canada.

Greenlaw, R., Hoover, H. J., and Ruzzo, W. L. 1995. *Limits to Parallel Computation: P-Completeness Theory*. Oxford University Press, Cambridge, United Kingdom.

Greibach, S. H. 1973. The hardest context-free language. *SIAM J. Comput. 2*, 4, 304–310.

Gyssens, M. 1986. On the complexity of join dependencies. *ACM Trans. Datab. Syst. 11*, 1 (Mar.), 81–108.

Gyssens, M., Jeavons, P. G., and Cohen, D. A. 1994. Decomposing constraint satisfaction problems using database techniques. *Artif. Int. 66*, 57–89.

Gyssens, M., and Paredaens, J. 1984. A decomposition methodology for cyclic databases. In *Advances in Database Theory*, vol. 2. Plenum Press, New York, pp. 85–122.

Hasan, W., Florescu, D., and Valduriez, P. 1996. Open issues in parallel query optimization. *SIGMOD Record 25*, 3, 28–33.

Hong, W., and Stonebraker, M. 1993. Optimization of parallel query execution plans in XPRS. *Dist. Parall. Datab. 1*, 1, 9–32.

Hopcroft, J. E., and Ullman, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass.

Hull, R. 1983. Acyclic join dependency and data base projections. *J. Comput. Syst. Sci. 27*, 3, 331–349.

Jeavons, P., Cohen, D., and Gyssens, M. 1997. Closure properties of constraints. *J. ACM 44*, 4 (July), 527–548.

Johnson, D. S. 1990. A catalog of complexity classes. In *Handbook of Theoretical Computer Science*, vol. A, chap. 2. J. van Leeuwen, Ed. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, and The MIT Press, Cambridge, Mass.

Jones, N. D. 1997. *Computability and Complexity: From a Programming Perspective*. Foundations of Computing Series. The MIT Press, Cambridge, Mass.

Karp, R. M., and Ramachandran, V. 1990. Parallel algorithms for shared-memory machines. In *Handbook of Theoretical Computer Science*, vol. A, chap. 17. J. van Leeuwen, Ed. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, and The MIT Press, Cambridge, Mass.

Kasif, S., and Delcher, A. L. 1994. Local consistency in parallel constraint satisfaction networks. *Artif. Int. 69*, 307–327.

Klein, P. N. 1996. Efficient parallel algorithms for chordal graphs. *SIAM J. Comput. 25*, 4, 797–827.

Kolaitis, P. G., and Vardi, M. Y. 2000. Conjunctive-query containment and constraint satisfaction. *J. Comput. Syst. Sci. 61*, 302–332.

Kruskal, J. B. 1956. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proc. AMS 7*, 48–50.

Ladner, R., and Lynch, N. 1976. Relativization of questions about log-space reducibility. *Math. Syst. Theory 10*, 19–32.

Lakshmi, M. S., and Yu, P. S. 1990. Effectiveness of parallel joins. *IEEE Trans. Knowl. Data Eng. 2*, 4, 410–424.

Levy, A. Y., Mendelzon, A. O., Sagiv, Y., and Srivastava, D. 1995. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (*PODS '95*) (San Jose, Calif., May 22–25) ACM, New York, pp. 95–104.

Lewis, H. R., and Papadimitriou, C. H. 1982. Symmetric Space-Bounded Computation. *Theoret. Comput. Sci. 19*, 161–187.

Maier, D. 1986. *The Theory of Relational Databases*. Computer Science Press, Rockville, Md.

Malvestuto, F. M. 1986. Modeling large bases of categorical data with acyclic schemes. In *Proceedings of the 1st International Conference on Database Theory* (Rome, Italy).

NAOR, J., NAOR, M., AND SCHAFFER, A. 1989. Fast parallel algorithms for chordal graphs. *SIAM J. Comput. 18*, 2, 327–349.

NISAN, N., AND TA-SHMA, A. 1995. Symmetric logspace is closed under complement. *Chi. J. Theoret. Comput. Sci. 1*.

PAPADIMITRIOU, C. H. 1994. *Computational Complexity*. Addison-Wesley, Reading, Mass.

PAPADIMITRIOU, C. H., AND STEIGLITZ, K. 1982. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, N.J.

PAPADIMITRIOU, C. H., AND YANNAKAKIS, M. 1997. On the complexity of database queries. In *Proceedings of 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (*PODS '97*) (Tucson, Az., May 12–14). ACM, New York, pp. 12–19.

PEARSON, J., AND JEAVONS, P. 1997. A survey of tractable constraint satisfaction problems. Tech. Rep. CSD-TR-97-15. Royal-Holloway, Univer. London, London, England.

QIAN, X. 1996. Query folding. In *Proceedings of the International Conference on Data Engineering* (*ICDE '96*). (New Orleans, La., Feb. 26–Mar. 1) pp. 48–55.

RAJAMARAN, A., SAGIV, Y., AND ULLMAN, J. D. 1995. Answering queries using templates with binding patterns. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (*PODS '95*) (San Jose, Calif., May 22–25). ACM, New York, pp. 105–112.

REIF, J. H. 1984. Symmetric complementation. *J. ACM 31*, 2 (Apr.), 401–421.

ROBERTSON, N., AND SEYMOUR, P. D. 1986a. Graph Minors. II. Algorithmic aspects of tree width. *J. Algorithms 7*, 309–322.

ROBERTSON, N., AND SEYMOUR, P. D. 1986b. Graph Minors. V. Excluding a Planar Graph. *J. Combinat. Theory, Ser. B 41*, 92–114.

RUZZO, W. L. 1980. Tree-size bounded alternation. *J. Comput. Syst. Sci. 21*, 218–235.

RUZZO, W. L. 1981. On uniform circuit complexity. *J. Comput. Syst. Sci. 22*, 365–383.

SACCÀ, D. 1985. Closures of database hypergraphs. *J. ACM 32*, 4 (Oct.), 774–803.

SAGIV, Y., AND SHMUELI, O. 1993. Solving queries by tree projections. *ACM Trans. Datab. Syst. 18*, 3 (Sept.), 487–511.

SKYUM, S., AND VALIANT, L. G. 1985. A complexity theory based on Boolean algebra. *J. ACM 32*, 2 (Apr.), 484–502.

SUDBOROUGH, I. H. 1977. Time and tape bounded auxiliary pushdown automata. In *Mathematical Foundations of Computer Science* (*MFCS '77*). Lecture Notes in Computer Science, vol. 53. Springer-Verlag, New York, pp. 493–503.

SUDBOROUGH, I. H. 1978. On the tape complexity of deterministic context-free languages. *J. ACM 25*, 3 (July), 405–414.

TARJAN, R. E., AND YANNAKAKIS, M. 1984. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J. Comput. 13*, 3, 566–579.

ULLMAN, J. D. 1989. *Principles of Database and Knowledge Base Systems*, Vol II. Computer Science Press, Rockville, Md.

ULLMAN, J. D. 1997. Information integration using logical views. In *Proceedings of the International Conference on Database Theory* (*ICDT '97*). Lecture Notes in Computer Science, vol. 1186. Springer-Verlag, New York, pp. 19–40.

VARDI, M. 1982. Complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing* (*STOC '82*) ACM, New York, pp. 137–146.

VENKATESWARAN, H. 1991. Properties that characterize LOGCFL. *J. Comput. Syst. Sci. 43*, 380–404.

WANKE, E. 1994. Bounded tree-width and LOGCFL. *J. Algorithms 16*, 470–491.

WILSCHUT, A. N., FLOKSTRA, J., AND APERS, P. M. G. 1995. Parallel evaluation of multi-join queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (*SIGMOD '95*) (San Jose, Calif., May 22–25). ACM, New York, pp. 115–126.

WOS, L., OVERBEEK, R., AND LUSK, E. 1991. Subsumption, a sometimes undervalued procedure. In *Computational Logic: Essays in Honor of Alan Robinson*. J. L. Lassez and Gordon Plotkin, Ed. The MIT Press, Cambridge, Mass.

YANNAKAKIS, M. 1981. Algorithms for acyclic database schemes. In *Proceedings of the International Conference on Very Large Data Bases* (*VLDB '81*) (Cannes, France). C. Zaniolo and C. Delobel, Eds. pp. 82–94.

YU, C. T., AND ÖZSOYOĞLU, M. Z. 1979. An algorithm for tree-query membership of a distributed query. In *Proceedings of the IEEE COMPSAC*. IEEE Computer Society Press, Los Alamitos, Calif., pp. 306–312.

YU, C. T., AND ÖZSOYOĞLU, M. Z. 1984. On determining tree-query membership of a distributed query. *Informatica 22*, 3, 261–282.

ZHANG, Y., AND MACKWORTH, A. K. 1993. Parallel and distributed finite constraint satisfaction: Complexity, algorithms and experiments. In *Parallel Processing for Artificial Intelligence*. Elsevier Science Publishers B.V., Amsterdam, The Netherlands.

RECEIVED: OCTOBER 1998; REVISED: OCTOBER 2000; ACCEPTED NOVEMBER 2000