

# Tractable Counting of the Answers to Conjunctive Queries<sup>\*</sup>

Reinhard Pichler and Sebastian Skritek

Technische Universität Wien, {pichler, skritek}@dbai.tuwien.ac.at

**Abstract.** Conjunctive queries (CQs) are one of the most fundamental forms of database queries. In general, the evaluation of CQs is NP-complete. Consequently, there has been an intensive search for tractable fragments. In this paper, we want to initiate a systematic search for tractable fragments of the counting problem of CQs, i.e., the problem of counting the answers to a CQ. We prove several new tractability and intractability results by starting with acyclic conjunctive queries and generalising these results to CQs of bounded hypertree-width.

## 1 Introduction

*Conjunctive queries* (CQs) are one of the most fundamental forms of database queries. They correspond to select-project-join queries in relational algebra and to select-from-where queries in SQL. As such, they are the primary target of most current query optimisation techniques. Moreover, CQs are closely related to Constraint Satisfaction Problems (CSPs) [14], which play an important role in artificial intelligence and reasoning. By slight abuse of notation, we consider CQ evaluation as a *decision problem*. Strictly speaking, there are several (inter-reducible) decision problems related to CQ evaluation, like query containment or asking if a given CQ has at least one solution over some database or asking if a given tuple is part of the answer of the CQ over some database, etc.

Without any restrictions, the evaluation of CQs (and, likewise, solving CSPs) is NP-complete (query complexity) [3]. Consequently, there has been an intensive search for tractable fragments. *Acyclic conjunctive queries* (ACQs) [19] have thus played an important role. The most common characterisation of ACQs is in terms of join trees (for details, see Section 2). ACQs can be evaluated efficiently by performing solely semi-joins in a bottom-up traversal of the join tree. Meanwhile, many further tractable classes of CQs have been identified. In particular, many structural decomposition methods have been developed which are based on some notion of decomposition of the graph or hypergraph structure underlying a conjunctive query and which are used to define some notion of width. *Hypertree decompositions*, which are used to define the *hypertree-width* of a CQ, are a very powerful decomposition method (for details, see Section 2) [11]. In particular, CQs can be efficiently evaluated if their hypertree-width is bounded by some constant. The class of CQs of bounded hypertree-width generalises many

---

<sup>\*</sup> This work was supported by the Vienna Science and Technology Fund (WWTF), project ICT08-032.

other tractable fragments of CQs [9] like CQs of bounded tree-width [17], CQs of bounded query-width [4], and also ACQs. Indeed, ACQs are precisely the CQs whose hypertree-width is equal to 1.

In this paper, we concentrate on the *counting problem* related to CQ evaluation, i.e., the problem of counting the number of answers to a CQ. We shall refer to this problem as the  $\#CQ$  problem. Of course, the  $\#CQ$  problem is intractable. Indeed, it is  $\#P$ -complete for CQs with free variables only (i.e., no existential quantifiers) and even slightly harder (namely  $\# \cdot NP$ -complete) in the general case [1]. Interestingly, the search for tractable fragments of  $\#CQ$  has received very little attention so far even though the count operator is an integral part of the SQL language and is supported by virtually any relational database management system. Very recently, we have shown by an ad hoc algorithm that the query complexity of  $\#CQ$  becomes tractable for CQs of bounded tree-width [16]. However, a *systematic search for tractable fragments of  $\#CQ$*  is completely missing to date. In this paper, we want to close this gap. Our goal is to arrive at a situation comparable to the well-studied decision problem of CQ evaluation, i.e., (i) we want to establish tractability for ACQs, (ii) the corresponding algorithms should be based on semi-joins, and (iii) it should be possible to generalise the tractability results to CQs of bounded hypertree-width.

**Organisation of the paper and summary of results.** In Section 2, we recall some basic notions and results. A conclusion is given in Section 6. The main results of the paper are detailed in Sections 3 – 5, namely:

- *ACQs with free variables only.* In Section 3, we restrict ourselves to ACQs with free variables only. We present our basic algorithm for counting the answers to ACQs of this specific form. Our algorithm only requires semi-joins and simple arithmetic. We thus establish the tractability of the combined complexity of  $\#CQ$  for ACQs without existential quantifiers.
- *Arbitrary ACQs.* In Section 4, we consider  $\#CQ$  for arbitrary ACQs. The tractability of the data complexity is easily shown. For the query complexity, a significant extension of the basic algorithm from Section 3 is required. In contrast, for the combined complexity of the  $\#CQ$  problem for arbitrary ACQs we establish the intractability by proving its  $\#P$ -completeness.
- *Extensions.* All tractability and intractability results discussed above immediately carry over to CQs of bounded hypertree-width. In Section 5, we discuss further extensions and applications of our results. For instance, the  $\#P$ -completeness of the combined complexity of the counting problem of ACQs allows us an easy  $\#P$ -completeness proof for the counting problem of *unions of ACQs* even if these ACQs have no existentially quantified variables at all.

## 2 Preliminaries

**Schemas and instances.** A *relational schema*  $\mathbf{R} = \{R_1, \dots, R_n\}$  is a set of relation symbols  $R_i$  each of a fixed arity  $k$  and with an assigned sequence of  $k$  attributes  $(A_1, \dots, A_k)$ . An *instance*  $I$  over a schema  $\mathbf{R}$  consists of a relation  $R_i^I$  for each relation symbol  $R_i \in \mathbf{R}$ , s.t. both have the same arity. We only consider finite instances here, and denote with  $dom(I)$  the *active domain* of  $I$ . We write

$\mathbf{x}$  for a tuple  $(x_1, \dots, x_n)$ . By slight abuse of notation, we also refer to the set  $\{x_1, \dots, x_n\}$  as  $\mathbf{x}$ . Hence, we may use expressions like  $x_i \in \mathbf{x}$  or  $\mathbf{x} \subseteq X$ , etc. For a tuple  $\mathbf{s} \in R_i^I$ , we also say  $R_i(\mathbf{s}) \in I$ .

**CQs.** A *conjunctive query (CQ)*  $Q$  on a database schema  $\mathbf{R}$  is of the form  $Q: \text{ans}(\mathbf{x}) \leftarrow \exists \mathbf{y} \phi(\mathbf{x}, \mathbf{y})$ , where  $\phi(\mathbf{x}, \mathbf{y}) = \bigwedge_{i=1}^n r_i$  is a conjunction of atoms  $r_i = R_j(\mathbf{z})$ , s.t.  $R_j \in \mathbf{R}$  with arity  $k$ , and  $\mathbf{z} \subseteq \mathbf{x} \cup \mathbf{y}$  with  $|\mathbf{z}| = k$ . We do not consider constants in the query, because a simple preprocessing step allows us to get rid of them. For the same reason, w.l.o.g. we assume no variable to occur twice in any atom [10]. The free variables  $\mathbf{x}$  are also referred to as *distinguished* variables, while the existentially quantified variables  $\mathbf{y}$  are often called *nondistinguished* variables. We denote with  $\text{atoms}(Q)$  the set  $\{r_1, \dots, r_n\}$  of atoms appearing in  $Q$ , and for a set  $\mathcal{A}$  of atoms,  $\text{var}(\mathcal{A})$  denotes the set of variables occurring in  $\mathcal{A}$ . By slight abuse of notation, for a tuple  $\mathbf{x} = (x_1, \dots, x_n)$  of variables and a mapping  $\mu: \mathbf{x} \rightarrow \text{dom}(I)$ , we write  $\mu(\mathbf{x})$  for  $(\mu(x_1), \dots, \mu(x_n))$ . A tuple  $\mathbf{s}$  is called an *answer* or *solution* to a CQ  $Q$  on an instance  $I$  if  $\mathbf{s} = \mu(\mathbf{x})$ , where  $\mu: \mathbf{x} \cup \mathbf{y} \rightarrow \text{dom}(I)$  is a variable assignment on  $\mathbf{x}, \mathbf{y}$ , s.t. for every atom  $R_i(\mathbf{z}) \in \text{atoms}(Q)$ ,  $R_i(\mu(\mathbf{z})) \in I$ . For such a mapping  $\mu$ , we also write  $\mu(Q) \subseteq I$  to emphasise that every atom  $R_i(\mathbf{z}) \in \text{atoms}(Q)$  is sent to an atom in  $I$ . The set  $Q(I)$  of answers to  $Q$  on  $I$  is a relation  $\text{ans}^{Q(I)}$  containing all answers.

In this paper, we study the counting problem  $\#CQ$ , which is defined as follows: Given a conjunctive query  $Q$  over some relational schema  $\mathbf{R}$  and an instance  $I$  for  $\mathbf{R}$ , how many tuples are contained in  $\text{ans}^{Q(I)}$ ?

**ACQs.** There exist several notions of acyclic CQs (ACQs). We restrict ourselves here to the so-called  $\alpha$ -acyclicity [7]. One way to characterise ACQs is via join trees, i.e., a CQ is acyclic, if it has a join tree [7]. A *join tree*  $T$  for a query  $Q$  is a tree  $T = (V, E)$ , where  $V$  is identified with the set of atoms in  $Q$ , and  $E$  is such that for every two atoms  $r_i, r_j \in V$  having variables in common, all atoms on the unique path from  $r_i$  to  $r_j$  contain all variables shared by  $r_i$  and  $r_j$ . Given a CQ it can be efficiently checked if it is acyclic, and if so, also a join tree can be computed efficiently [12, 20]. We often identify a tree  $T$  with its vertex set  $V$ , and write  $t \in T$  for  $T = (V, E)$ ,  $t \in V$ . If we want to stress the difference between the graph structure and the query atoms, for some  $t \in T$  for a join tree  $T$ , we write  $Q_t$  to denote the query atom identified with node  $t$ . Further, given an instance  $I$  to evaluate  $Q$  on, for every  $t \in T$  we denote with  $R_t$  the relation  $R_t^I$  assigned to the relation symbol of  $Q_t$ .

**Hypertree decompositions.** A *hypertree decomposition* [11] of a CQ  $Q$  is a triple  $\langle T, \chi, \lambda \rangle$ , where  $T = (V, E)$  is a tree and  $\chi, \lambda$  are mappings  $\chi: V \rightarrow \mathcal{P}(\text{var}(\text{atoms}(Q)))$  and  $\lambda: V \rightarrow \mathcal{P}(\text{atoms}(Q))$  s.t. (1) for each atom  $r_i \in \text{atoms}(Q)$ , there exists a  $v \in V$  s.t.  $\text{var}(\{r_i\}) \subseteq \chi(v)$ ; (2) for each  $z \in \text{var}(\text{atoms}(Q))$ , the set  $T' = \{v \in V \mid z \in \chi(v)\}$  of  $T$  induces a connected subtree of  $T$ ; (3) for each  $v \in V$ ,  $\chi(v) \subseteq \text{var}(\lambda(v))$ ; (4) for each  $v \in V$ ,  $\text{var}(\lambda(v)) \cap \chi(T_v) \subseteq \chi(v)$ , where  $T_v \subseteq T$  is the complete subtree of  $T$  rooted at  $v$ . The width of a hypertree decomposition is  $\max_{v \in V} |\lambda(v)|$ , and the *hypertree-width* of  $Q$  is the minimum width over all hypertree decompositions of  $Q$ . For a given width  $\omega$ , the existence of a hypertree decomposition of width  $\omega$  can be efficiently decided, and a decomposition can be efficiently computed, if one exists [11].

**Counting complexity.** While decision problems ask if for a given problem instance, *at least one solution* exists, counting problems ask *how many* different solutions exist, see e.g. [15], Chap. 18. The most intensively studied counting complexity class is  $\#\text{P}$ , which contains those function problems which consist in counting the number of accepting computation paths of a non-deterministic polynomial-time Turing machine. In other words,  $\#\text{P}$  captures the counting problems corresponding to decision problems in  $\text{NP}$ . However, there exist also  $\#\text{P}$ -complete problems for which the corresponding decision problem is easy. Classical examples of this “easy to decide, hard to count” phenomenon are  $\#\text{PERFECT MATCHINGS}$  and  $2\text{-SAT}$  [18]: checking if a bipartite graph has a perfect matching or if a 2-CNF formula has a model is easy. But counting the number of perfect matchings or the number of models of a 2-CNF-formula is hard. Note that by  $\#\text{P}$ -completeness we mean completeness w.r.t. Cook reductions, i.e., polynomial-time Turing reductions [15].

### 3 ACQs without Nondistinguished Variables

We first consider CQs of the form  $ans(\mathbf{x}) \leftarrow \phi(\mathbf{x})$ , i.e. only queries that contain no existentially quantified variables. Without further restrictions,  $\#CQ$  is well known to be  $\#\text{P}$ -complete for CQs of this form [5]. However, just as for the decision problem, for ACQs, we can make use of the existence of a join tree to guide the computation. We provide an algorithm, mainly working with semi-joins along the join tree, that shows that the problem is indeed tractable for ACQs. Note that we consider counting w.r.t. set semantics here.

Before describing the algorithm, we have to fix some notation. Given an acyclic CQ  $Q: ans(\mathbf{x}) \leftarrow \phi(\mathbf{x})$  over database schema  $\mathbf{R}$  and instance  $I$ , let  $T$  be a join tree of  $Q$ . Recall that for  $t \in T$ ,  $Q_t$  denotes the query atom identified with  $t$ , and  $R_t$  the relation  $R_t^I$  assigned to the relational symbol of  $Q_t$ . Further let  $T_t$  be the complete subtree of  $T$  rooted at  $t$ , and let  $X_t = var(\{Q_t\})$ . For  $Q_t = R_t(x_1, \dots, x_k)$ , if clear from the context, it is convenient to also write  $Q_t$  to refer to  $(x_1, \dots, x_k)$  only. For a subtree  $T'$  of  $T$ ,  $X_{T'} = \bigcup_{t \in T'} X_t$  and  $Q_{T'} = \bigcup_{t \in T'} Q_t$ . Given  $Q_{T'}$ , the *subquery defined by*  $Q_{T'}$  is the query  $ans(X_{T'}) \leftarrow \bigwedge_{t \in T'} Q_t$ . In the following, we always assume an acyclic CQ  $Q: ans(\mathbf{x}) \leftarrow \phi(\mathbf{x})$  with join tree  $T$  to be evaluated over some database instance  $I$ .

Obviously, every row in  $R_t$  encodes a variable assignment  $\alpha$  on  $X_t$ , hence a partial assignment on  $\mathbf{x}$ . The idea of the algorithm is to compute, for every  $t \in T$ , in a bottom-up traversal of the tree the number of possible extensions  $\mu$  of  $\alpha$  to  $X_{T_t}$  that are valid solutions to the subquery defined by  $Q_{T_t}$ . Formally, for a mapping  $\mu: X_{T_t} \rightarrow dom(I)$  and some  $t \in T$ , we define the *footprint of  $\mu$  on  $t$*  as  $fpr(\mu, t) = \alpha$ , where  $\alpha$  is a variable assignment on  $X_t$  s.t.  $\alpha(x_i) = \mu(x_i)$  for all  $x_i \in X_t$ . For a subtree  $T' \subseteq T$  with root  $t$  and  $\alpha: X_t \rightarrow dom(I)$ , we define the set  $N(T', t, \alpha) = \{\mu: X_{T'} \rightarrow dom(I) \mid fpr(\mu, t) = \alpha \text{ and } \mu(Q_{T'}) \subseteq I\}$  of all possible extensions of the variable assignment  $\alpha$  to an assignment  $\mu$  on the variables  $X_{T'}$  that are solutions to the subquery  $Q_{T'}$ . The intuition behind this set is that for every row  $\rho \in R_t$  (which can be considered as a variable assignment  $\alpha$  on  $X_t$ ),  $N(T_t, t, \alpha)$  contains all possible extensions of  $\alpha$  to  $X_{T_t}$  that are solutions to  $Q_{T_t}$ .

As we cannot afford to compute this set explicitly (which would be the naive algorithm), we just compute and store its cardinality, i.e.  $|N(T_t, t, \alpha)|$ . Hence,

at each node  $t \in T$  we maintain a table  $\hat{R}_t$ , having  $|X_t| + 1$  columns. Thereby the first  $|X_t|$  columns (denoted as  $\hat{R}_{t,\pi(X_t)}$ ) store  $\alpha$ , while the last column stores  $|N(T_t, t, \alpha)|$ . To better distinguish between  $|N(T_t, t, \alpha)|$ , i.e. the value intended to be stored in this column, and the actual value present there, we refer to this column (or value) as  $c_\alpha$ . Note that we are only interested in such assignments  $\alpha$  s.t.  $N(T_t, t, \alpha) \neq \emptyset$ , because otherwise  $\alpha$  can never be extended to an answer to  $Q$ . Hence, for every assignment  $\alpha$  s.t.  $\alpha(Q_t) \notin R_t$ , we need no entry in  $\hat{R}_t$ , and therefore  $\hat{R}_t$  contains exactly one row for every  $\rho \in R_t$  s.t.  $|N(T_t, t, \alpha_\rho)| > 0$ . Thereby  $\alpha_\rho$  denotes the variable assignment defined by  $\rho$ . If clear from the context, we drop the  $\rho$  from  $\alpha_\rho$  and use  $\alpha$  and  $\rho$  interchangeably.

It is easy to see that  $N(T_t, t, \alpha) \cap N(T_t, t, \alpha') = \emptyset$  for  $\alpha \neq \alpha'$ . Hence, the number of answers to the ACQ  $Q$  can be retrieved from the root node  $r$  of  $T$ , by just summing up the counter of all entries in  $\hat{R}_r$ . Indeed, the union  $\bigcup_{\alpha \in R_r} N(T_r, r, \alpha)$  gives all possible variable assignments  $\mu$  on  $X_{T_r} = \mathbf{x}$  s.t.  $\mu(Q_{T_r}) = \mu(Q) \subseteq I$ . By the disjointness of  $N(T_r, r, \alpha)$  for different values of  $\alpha$ ,  $|\bigcup_{\alpha \in R_r} N(T_r, r, \alpha)| = \sum_{\alpha \in R_r} |N(T_r, r, \alpha)|$ , and therefore  $\sum_{\alpha \in R_r} c_\alpha$  gives the number of different solutions. It remains to show how to compute the values for each  $\hat{R}_t$ . The algorithm for this consists of two phases: The initialisation phase, and a bottom-up traversal of the tree.

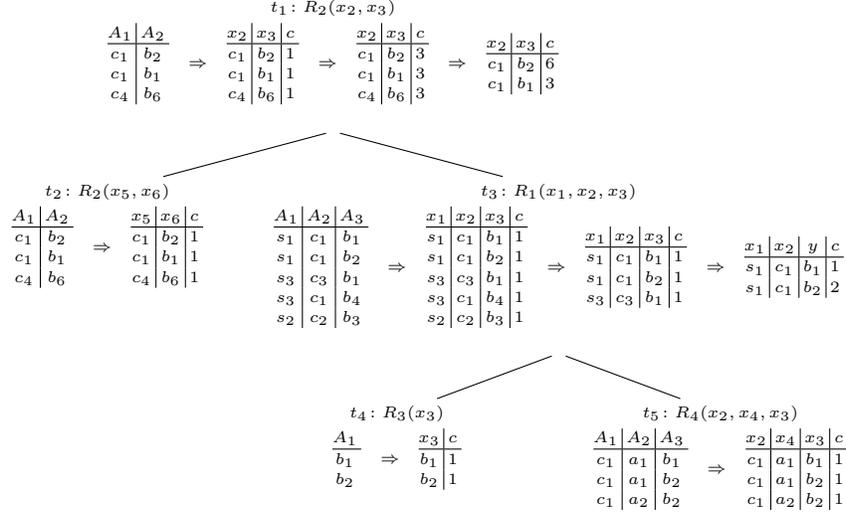
*Initialisation Phase:* For every leaf node  $t \in T$ ,  $\hat{R}_t$  is computed as follows: For every distinct tuple  $\rho \in R_t$ , add a tuple  $(\rho, 1)$  to  $\hat{R}_t$ . Note that for leaf nodes we have  $X_t = X_{T_t}$ . Hence, the only extension  $\mu$  for each  $\alpha$  is  $\alpha$  itself, and  $\hat{R}_t$  therefore contains the correct values.

*Bottom-Up Traversal:* Let  $t \in T$  be a node with children  $t_1, \dots, t_k$  ( $k \geq 1$ ), such that  $\hat{R}_{t_j}$  has already been computed during the bottom-up traversal for every  $j \in \{1, \dots, k\}$ . The idea is now to compute  $\hat{R}_t$  by starting from a relation considering  $t$  only and then, step by step, for every  $j \in \{1, \dots, k\}$ , to include the subtree  $T_j$  (where we use  $T_j$  to abbreviate  $T_{t_j}$ ) by computing the semi-join between the output of the previous step and  $\hat{R}_{t_j}$ . Formally, let  $\hat{R}_t^0$  contain a tuple  $(\rho, 1)$  for every distinct tuple  $\rho \in R_t$ , i.e.  $\hat{R}_t^0$  is defined analogously to  $\hat{R}_{t'}$  for a leaf node  $t' \in T$ . Then, for every child  $j \in \{1, \dots, k\}$ , compute  $\hat{R}_{t,\pi(X_t)}^j = \hat{R}_{t,\pi(X_t)}^{j-1} \ltimes \hat{R}_{t_j,\pi(X_{t_j})}$ . To compute  $c_\alpha^j$  for every row in  $\hat{R}_{t,\pi(X_t)}^j$ , we first define for every  $\alpha \in \hat{R}_{t,\pi(X_t)}^j$  the set  $B_\alpha \subseteq \hat{R}_{t_j,\pi(X_{t_j})}$  containing exactly those  $\beta \in \hat{R}_{t_j,\pi(X_{t_j})}$  that can be joined with  $\alpha$ , i.e.  $B_\alpha = \{\beta \in \hat{R}_{t_j,\pi(X_{t_j})} \mid \beta(x_i) = \alpha(x_i) \text{ for every } x_i \in X_t \cap X_{t_j}\}$ . Then  $c_\alpha^j$  for every row  $\alpha \in \hat{R}_{t,\pi(X_t)}^j$  is set to  $c_\alpha^j = c_\alpha^{j-1} \cdot \sum_{\beta \in B_\alpha} c_\beta$  where  $c_\alpha^{j-1}$  is the value of  $c_\alpha$  in  $\hat{R}_{t,\pi(X_t)}^{j-1}$ . Finally,  $\hat{R}_t = \hat{R}_t^k$ .

The following example will help to illustrate this algorithm.

*Example 1.* Consider an ACQ  $Q: \text{ans}(x_1, x_2, x_3, x_4, x_5, x_6) \leftarrow R_3(x_3) \wedge R_4(x_2, x_4, x_3) \wedge R_1(x_1, x_2, x_3) \wedge R_2(x_2, x_3) \wedge R_2(x_5, x_6)$  and an instance  $I = \{R_1(s_1, c_1, b_1), R_1(s_1, c_1, b_2), R_1(s_3, c_3, b_1), R_1(s_3, c_1, b_4), R_1(s_2, c_2, b_3), R_2(c_1, b_2), R_2(c_1, b_1), R_2(c_4, b_6), R_3(b_1), R_3(b_2), R_4(c_1, a_1, b_1), R_4(c_1, a_1, b_2), R_4(c_1, a_2, b_2)\}$  over the schema  $\mathbf{R} = \{R_1/3, R_2/2, R_3/1, R_4/3\}$ . A possible join tree for  $Q$  is

shown in Fig. 1, where each of the nodes  $t \in \{t_1, \dots, t_5\}$  is annotated with  $Q_t$ . In addition, the pattern of the relations shown at the leaf nodes is “ $R_t \Rightarrow \hat{R}_t$ ”. At the two non-leaf nodes, the pattern is “ $R_t \Rightarrow \hat{R}_t^0 \Rightarrow \hat{R}_t^1 \Rightarrow \hat{R}_t$ ”. The last column in each table contains the counter. The content of the relations is computed in a bottom-up manner as described above. The final result  $|Q(I)| = 9$  can be read off at the root node.



**Fig. 1.** Annotated join tree for the query  $Q$  (Example 1) over  $I$ .

**Theorem 1.** *Suppose that we only consider ACQs  $Q: ans(\mathbf{x}) \leftarrow \phi(\mathbf{x})$  without nondistinguished variables. Then for query  $Q$  with join tree  $T$  over schema  $\mathbf{R}$  and instance  $I$ ,  $\#CQ$  can be solved in time  $O(|Q| \cdot (\max_{t \in T} |R_t|)^2)$ , assuming unit cost for arithmetic operations.*

*Proof (sketch).* First of all, it must be shown that the algorithm is indeed correct. This will be done implicitly in Section 4, where we present an algorithm for counting the number of solutions to an ACQ with nondistinguished variables and prove its correctness. As the algorithm presented above will turn out to be a special case of the one in Section 4, for the correctness we refer to the proof of Theorem 3.

It therefore remains to show the upper bound on the runtime of the algorithm: First of all, note that at each  $t \in T$ ,  $\hat{R}_{t, \pi(X_t)} \subseteq R_t$ . Hence, at each node  $t \in T$ , the space required to store  $\hat{R}_t$  is obviously in  $O(|R_t|)$ . On the other hand, by definition, the size of a join tree of  $Q$  is linear in the size of  $Q$ . Hence the space required to store  $\hat{R}_t$  for all  $t \in T$  is trivially  $O(|Q| \cdot \max_{t \in T} |R_t|)$ . Therefore, as even the naive implementation (with nested loops) of the semi-join needs only quadratic time, by assuming unit cost for the arithmetic operations, the runtime for the algorithm is in  $O(|Q| \cdot (\max_{t \in T} |R_t|)^2)$ .  $\square$

## 4 ACQs with Nondistinguished Variables

For deciding whether a CQ has some solution, it makes no difference whether it contains only free variables, or additional nondistinguished ones. As the goal is just to find one assignment on all variables that maps the query into the instance, all variables can be treated in the same way. In contrast, for the counting problem, this is no longer true, as only variable assignments that differ on the free variables also give different solutions. On the other hand, variable assignments that only differ on the existentially quantified variables must not be counted twice. As already noted in the introduction, under reasonable complexity theoretical assumptions, in the general case, counting becomes harder for queries with nondistinguished variables [1]. In this section, we show that a similar behaviour can be observed for ACQs: First we will show that the problem remains tractable for data- and query complexity, and then we show  $\#P$ -completeness for the combined complexity.

Given an ACQ  $Q: ans(\mathbf{x}) \leftarrow \exists \mathbf{y} \phi(\mathbf{x}, \mathbf{y})$ , the naive approach for counting the number of solutions over some instance  $I$  is to check for each possible variable assignment  $\mu: \mathbf{x} \rightarrow dom(I)$ , whether the (Boolean) ACQ  $Q': ans() \leftarrow \exists \mathbf{y} \phi(\mu(\mathbf{x}), \mathbf{y})$  is satisfied over  $I$ . As  $Q'$  is acyclic, the check can be done in time  $O(|Q| \cdot |I|)$  [19, 8] which immediately leads to tractable data complexity.

**Theorem 2.** *Suppose that we only consider ACQs  $Q: ans(\mathbf{x}) \leftarrow \exists \mathbf{y} \phi(\mathbf{x}, \mathbf{y})$  with nondistinguished variables  $\mathbf{y}$ . Then for query  $Q$  over schema  $\mathbf{R}$  and instance  $I$ ,  $\#CQ$  can be solved in time  $O(|dom(I)|^{|\mathbf{x}|} \cdot |Q| \cdot |I|)$ .*

When considering query complexity, tractability can be reached by some major extensions of the algorithm from the previous section. In the following, we will sketch the necessary changes. Recall that in the last section, given a CQ  $Q$  with join tree  $T$  on some schema  $\mathbf{R}$  with instance  $I$ , the idea was, at each  $t \in T$ , to define a *footprint* (small enough to be stored) for every assignment  $\mu: X_{T_t} \rightarrow dom(I)$ , and to keep track of the number of partial solutions that have this footprint. Now we have to deal with assignments  $\mu = \mu_x \cup \mu_y$  with  $\mu: X_{T_t} \cup Y_{T_t} \rightarrow dom(I)$ , where  $\mu_x$  and  $\mu_y$  are the restrictions of  $\mu$  to  $X_{T_t}$  and  $Y_{T_t}$  resp., and  $Y_t$  and  $Y_{T_t}$  are defined on the nondistinguished variables analogously to  $X_t$  and  $X_{T_t}$ . Therefore we have  $fpr(\mu_x, \mu_y, t) = (\alpha_x, \alpha_y)$  again as the restriction of  $\mu_x$  (resp.  $\mu_y$ ) to  $X_t$  (resp.  $Y_t$ ). However, this time the problem is that the same partial solution  $\mu_x$  may, with different  $\mu_y$ , have different footprints. Hence, if we define the sets  $N(T_t, t, \alpha_x, \alpha_y)$  of partial assignments (on the distinguished variables  $\mathbf{x}$ ) analogously as before, these sets are not necessarily disjoint. We therefore have to group footprints by  $\alpha_x$ , and identify sets of partial solutions by a *combined footprint*, which is a set of footprints  $\{(\alpha_x, \alpha_y^1), \dots, (\alpha_x, \alpha_y^n)\}$ . For convenience, we may also write  $(\alpha_x, \mathcal{I})$  with  $\mathcal{I} = \{\alpha_y^1, \dots, \alpha_y^n\}$  to denote the combined footprint.

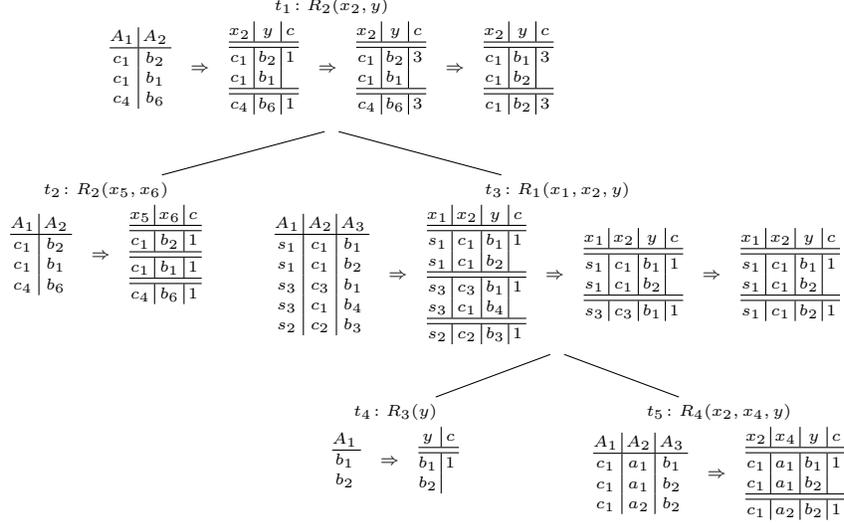
For a subtree  $T_t$  at node  $t \in T$  and a combined footprint  $(\alpha_x, \mathcal{I})$  at node  $t$ , we define  $N(T_t, t, \alpha_x, \mathcal{I})$  as the set of mappings  $\mu_x: X_{T_t} \rightarrow dom(I)$  such that

- (1) for every  $i \in \{1, \dots, |\mathcal{I}|\}$ , there exists a  $\mu_y: Y_{T_t} \rightarrow dom(I)$  s.t.
  - (a)  $fpr(\mu_x, \mu_y, t) = (\alpha_x, \alpha_y^i)$ , and
  - (b)  $\mu(Q_{T_t}) \subseteq I$  (where  $\mu = \mu_x \cup \mu_y$ )
- (2) for all  $\bar{\mu}_y: Y_{T_t} \rightarrow dom(I)$  s.t.  $\mu(Q_{T_t}) \subseteq I$  (where  $\mu = \mu_x \cup \bar{\mu}_y$ ) holds, there exists a  $j \in \{1, \dots, |\mathcal{I}|\}$  s.t.  $fpr(\mu_x, \bar{\mu}_y, t) = (\alpha_x, \alpha_y^j)$

As mentioned above, we cannot afford to store the set  $N(T_t, t, \alpha_x, \mathcal{I})$  at node  $t \in T$ . Hence, we only store  $(\alpha_x, \mathcal{I})$  together with  $|N(T_t, t, \alpha_x, \mathcal{I})|$ , since we are only interested in the number of such assignments. Note that  $(\alpha_x, \mathcal{I})$  fits into a table with  $|X_t| + |Y_t|$  columns (i.e. with schema  $(X_t, Y_t)$ ), containing one row  $\rho_i$  for every  $\alpha_y^i \in \mathcal{I}$ , storing  $\alpha_x$  in the first  $|X_t|$  columns, and  $\alpha_y^i$  in the last  $|Y_t|$  columns. Hence at every node  $t \in T$ , we maintain a set  $\hat{\mathcal{R}}_{T_t} = \{\hat{S}_1, \dots, \hat{S}_n\}$  (for some  $n \in \mathbb{N}$ ; an upper bound on  $n$  will be given below) of relations of the above form. In addition, we maintain for each relation  $\hat{S}_j \in \hat{\mathcal{R}}_{T_t}$  a counter  $c_t^j$  storing  $|N(T_t, t, \alpha_x, \mathcal{I})|$ . Actually, it is convenient to write  $\hat{\mathcal{R}}_t$  short for  $\hat{\mathcal{R}}_{T_t}$ . Only for the correctness proof (in the full version) it will be advantageous to explicitly index  $\hat{\mathcal{R}}$  by a subtree  $T_t$  rather than just by a node  $t$ . Intuitively, every relation  $\hat{S}_j \in \hat{\mathcal{R}}_t$  encodes a set of mappings  $\mu_x: X_{T_t} \rightarrow \text{dom}(I)$ , while each row in  $\hat{S}_j$  encodes a different set of mappings  $\mu_y: Y_{T_t} \rightarrow \text{dom}(I)$  for this  $\mu_x$ , s.t. for each  $\mu = \mu_x \cup \mu_y$ ,  $\mu(Q_{T_t}) \subseteq I$  holds. Note that within each relation, each row has the same value for  $X_t$ , and several relations may contain the same  $\alpha_x$ .

The content of  $\hat{\mathcal{R}}_t$  (for  $t \in T$ ) is again computed in two phases. In an *initialisation* phase, first for every leaf node  $t \in T$ ,  $\hat{\mathcal{R}}_t$  contains exactly one relation for every different value  $\alpha_x$  for  $X_t$  in  $R_t$ , containing one row  $(\alpha_x, \alpha_y)$  for every value  $\alpha_y$  for  $Y_t$  s.t.  $(\alpha_x, \alpha_y) \in R_t$ . The counter for each such relation is set to one. Then, in a *bottom-up traversal* of the tree, for every  $t \in T$  with children  $t_1, \dots, t_k$  s.t.  $\hat{\mathcal{R}}_{t_i}$  has already been computed, we again start with some  $\hat{\mathcal{R}}_t^0$  that is defined like  $\hat{\mathcal{R}}_{t'}$  for leaf nodes  $t' \in T$ , and then include one child after the other as follows: For  $i \in \{1, \dots, k\}$ , let  $\hat{\mathcal{R}}_t^i$  contain all nonempty results  $\hat{S}_1 \times \hat{S}_2$  for all pairs  $(\hat{S}_1 \in \hat{\mathcal{R}}_{t_i}^{i-1}, \hat{S}_2 \in \hat{\mathcal{R}}_{t_i})$ . If  $\hat{S}_1 \times \hat{S}_2$  is nonempty, the value of the counter for the resulting table is the product of the counters for  $\hat{S}_1$  and  $\hat{S}_2$ . Whenever several such pairs give the same result, only one copy of the relations is stored, and their counters are summed up. The number of solutions to  $Q$  is retrieved by summing up the counters for all  $\hat{S}_j \in \hat{\mathcal{R}}_r$  at the root node  $r$  of  $T$ .

*Example 2.* Recall the instance  $I$  and query  $Q$  from Example 1. We will use the same instance  $I$ , and change  $Q$  to  $Q'$  only by removing  $x_3$  from the free variables. (For convenience, we rename it to  $y$ .) I.e.,  $Q': \text{ans}(x_1, x_2, x_4, x_5, x_6) \leftarrow \exists y R_3(y) \wedge R_4(x_2, x_4, y) \wedge R_1(x_1, x_2, y) \wedge R_2(x_2, y) \wedge R_2(x_5, x_6)$ . We use the same join tree as before. It is shown in Fig. 2, where again each node  $t \in \{t_1, \dots, t_5\}$  is annotated with  $Q_t$ . Recall that to deal with nondistinguished variables, we have to maintain sets  $\hat{\mathcal{R}}_t$  of relations. At each node  $t$ ,  $\hat{\mathcal{R}}_t$  is depicted together with its derivation sequence: The pattern shown at the leaf nodes is “ $R_t \Rightarrow \hat{\mathcal{R}}_t$ ”, and for the inner nodes it is “ $R_t \Rightarrow \hat{\mathcal{R}}_t^0 \Rightarrow \hat{\mathcal{R}}_t^1 \Rightarrow \hat{\mathcal{R}}_t$ ”. Thereby, the first row in each table shows the schema, followed by the relations  $\hat{S}_j \in \hat{\mathcal{R}}_t^i$  separated by double horizontal lines. In the first row for each relation  $\hat{S}_j$ , the last column shows the counter for  $\hat{S}_j$ . For instance, at node  $t_3$ , we have two relations  $\hat{S}_1 = \{(s_1, c_1, b_1), (s_1, c_1, b_2)\}$  and  $\hat{S}_2 = \{(s_1, c_1, b_2)\}$  in  $\hat{\mathcal{R}}_t$ . Note that  $\hat{S}_1$  and  $\hat{S}_2$  refer to two different extensions of  $\alpha_x(x_1, x_2) = (s_1, c_1)$  to  $\mu_x(x_1, x_2, x_4)$ . Indeed,  $\hat{S}_1$  corresponds to  $\mu_x(x_4) = a_1$  while  $\hat{S}_2$  corresponds to  $\mu_x(x_4) = a_2$ . Again, the final result  $|Q'(I)| = 6$  can be read off at the root node.



**Fig. 2.** Annotated join tree for the query  $Q'$  (Example 2) over  $I$ .

**Theorem 3.** *Suppose that we only consider ACQs  $Q: \text{ans}(\mathbf{x}) \leftarrow \exists \mathbf{y} \phi(\mathbf{x}, \mathbf{y})$  with nondistinguished variables  $\mathbf{y}$ . Then for query  $Q$  with join tree  $T$  over schema  $\mathbf{R}$  and instance  $I$ ,  $\#CQ$  can be solved in time  $O(|Q| \cdot m^2 \cdot A^m)$  (assuming unit cost for arithmetic operations), where  $m = \max_{t \in T} |R_t|$ .*

*Proof (sketch).* The details of the algorithm and the correctness proof will be given in the full version. We only discuss the runtime here. Note that at some node  $t \in T$ , the values for  $X_t$  need not be different between two different relations  $\hat{S}_i, \hat{S}_j \in \hat{\mathcal{R}}_t$ . Hence for every  $t \in T$ , there might be up to  $2^m$  different elements in  $\hat{\mathcal{R}}_t$ , each of size  $O(m)$  (obviously,  $\hat{S}_i \subseteq R_t$  for every  $\hat{S}_i \in \hat{\mathcal{R}}_t$ ), where  $m = \max_{t \in T} |R_t|$ . Hence there are at most  $O((2^m)^2)$  semi-joins between two nodes, each requiring time at most  $O(m^2)$  even for a naive implementation, which gives an overall runtime estimation of  $O(|Q| \cdot m^2 \cdot (2^m)^2)$ .  $\square$

Theorem 2 and Theorem 3 immediately imply that  $\#CQ$  can be solved efficiently for data- and query complexity. The natural next question is, whether there exists also an algorithm that solves the problem efficiently when considering combined complexity. However, the next theorem shows that such an algorithm is very unlikely to exist.

**Theorem 4.** *Suppose that we only consider ACQs  $Q: \text{ans}(\mathbf{x}) \leftarrow \exists \mathbf{y} \phi(\mathbf{x}, \mathbf{y})$  with nondistinguished variables  $\mathbf{y}$ . Then the combined complexity of  $\#CQ$  is  $\#P$ -complete. The problem remains  $\#P$ -complete, even if  $Q$  contains a single nondistinguished variable only.*

*Proof (sketch).* *Membership* is easy to show. The *hardness* is shown by reduction from the  $\#P$ -complete problem  $\#PERFECT MATCHINGS$  [18], i.e. from the problem of counting the number of perfect matchings in a bipartite graph. A

perfect matching for a bipartite graph  $G = (V, E)$  is a subset  $M \subseteq E$  s.t. every  $v \in V$  is contained in exactly one  $e \in M$ . Let an arbitrary instance of  $\#\text{PERFECT MATCHINGS}$  be given by a bipartite graph  $G = (V, E)$ , where  $V = A \cup B$  with  $A = \{a_1, \dots, a_n\}$ ,  $B = \{b_1, \dots, b_n\}$ , and  $E \subseteq A \times B$ . From this, we create a database instance  $I$  over the schema  $\mathbf{S} = \{e/2, t/2\}$  as follows:  
 $I = \{e(a_i, b_j) \mid \text{for each } (a_i, b_j) \in E\} \cup \bigcup_{i=1}^n \{t(i, x) \mid x \in B \setminus \{b_i\}\}$ ,  
 where, by slight abuse of notation, we use  $a_i$  and  $b_i$  to denote both, vertices in  $V$  and constants in  $I$ . We further define two conjunctive queries  $Q_1, Q_2$  as

$$Q_1: \text{ans}(x_1, \dots, x_n) \leftarrow \bigwedge_{i=1}^n e(a_i, x_i)$$

$$Q_2: \text{ans}(x_1, \dots, x_n) \leftarrow \exists y \bigwedge_{i=1}^n e(a_i, x_i) \wedge \bigwedge_{i=1}^n t(y, x_i)$$

Note that a perfect matching is equivalent to a bijective mapping from  $A$  to  $B$  along the edges in  $E$ . The idea of the reduction is that  $Q_1$  (over  $I$ ) selects all possible mappings of nodes from  $A$  to nodes from  $B$ , while  $Q_2$  selects all those mappings where at least one element from  $B$  is not assigned to an element from  $A$ . Hence,  $Q_1(I) \setminus Q_2(I)$  gives exactly the set of perfect matchings, and because of  $Q_2(I) \subseteq Q_1(I)$ ,  $|Q_1(I)| - |Q_2(I)|$  returns the number of perfect matchings. Note that this is a special case of a Turing reduction, called subtractive reduction [6].

Obviously, this reduction is feasible in polynomial time. Its correctness as well as the membership will be proved in the full version.  $\square$

## 5 Extensions

In this section we discuss some extensions and applications of our main results.

**Unions of conjunctive queries.** For unions of conjunctive queries (UCQs), it is well known that the decision problem is not harder to solve than for CQs. In contrast, for the counting problem, even if we consider UCQs with distinguished variables only, we encounter a similar problem as for CQs with nondistinguished variables: the same answer could be produced in different ways. For UCQs this means that the same answer could appear in several disjuncts. In fact, a slight modification of the proof of Theorem 4 yields the following intractability result.

**Theorem 5.** *Suppose that we only consider unions of acyclic conjunctive queries  $\text{ans}(\mathbf{x}) \leftarrow \bigvee_{i=1}^n \phi_i(\mathbf{x})$  without nondistinguished variables. Then the combined complexity of the counting problem of such UCQs is  $\#\text{P}$ -complete.*

*Proof (idea).* For the *hardness*, recall the hardness part of the proof of Theorem 4. We use the same reduction as described there, only that instead of a single relation  $t/2$ , we use  $n$  relations  $t_i/1$ , and we define  $Q_2$  as  $Q_2: \text{ans}(x_1, \dots, x_n) \leftarrow \bigvee_{j=1}^n (\bigwedge_{i=1}^n e(a_i, x_i) \wedge \bigwedge_{i=1}^n t_j(x_i))$ . We define instance  $I$  also as before, but instead of the content of  $t$ , we have  $\bigcup_{j=1}^n \{t_j(x) \mid x \in B \setminus \{b_j\}\}$ .

Intuitively, we expand the query by instantiating the existentially quantified variable in  $Q_2$ , and create one disjunct for every instantiation. Hence, for the correctness, analogous arguments as in the proof of Theorem 4 apply.  $\square$

**Queries with bounded hypertree-width.** In [11] it was shown that testing if the result of a CQ is nonempty (or whether the result contains a certain tuple) is tractable for queries with bounded *hypertree-width*. Below we also generalise the tractability of  $\#\text{CQ}$  on ACQs to queries with bounded hypertree-width.

**Theorem 6.** *Suppose that we only consider CQs  $Q$  whose hypertree-width is bounded by some constant. Then the following holds: If  $Q$  contains no nondistinguished variables, then  $\#CQ$  is polynomial time solvable (combined complexity).*

*On the other hand, if  $Q$  contains nondistinguished variables, then solving  $\#CQ$  is feasible in polynomial time (data complexity and query complexity), while the problem is  $\#P$ -complete (combined complexity).*

*Proof (idea).* In [11] it was shown that, given a Boolean CQ  $Q$  with hypertree-width  $\omega$  (for some constant  $\omega$ ) over some instance  $I$ , an equivalent ACQ  $Q'$  over an instance  $I'$  can be efficiently constructed, such that the combined size of  $Q'$  and  $I'$  is in  $O(|Q| \cdot m^\omega)$ , where  $m = \max_{S_i \in I} |S_i|$ , and s.t.  $Q(I)$  returns *true* iff  $Q'(I')$  returns *true*. It is easy to verify that this construction preserves the number of solutions. Actually, even  $Q(I) = Q'(I')$  holds. Hence applying the results from the previous sections to  $Q'$  and  $I'$  proves the case.  $\square$

**Counting under bag semantics.** So far, we have only considered counting under set semantics. However, by a slight change of the initialisation step, the algorithm presented in Section 3 works for counting under bag semantics as well. Further, note that the proof of Theorem 4 heavily depends on assuming set semantics for the query. In fact, we can show the following result.

**Theorem 7.** *Suppose that we only consider ACQs. Then, for query  $Q$  with join tree  $T$  over schema  $\mathbf{R}$  and instance  $I$ ,  $\#CQ$  w.r.t. bag semantics can be solved in time  $O(|Q| \cdot \max_{t \in T} |R_t|^2)$  (assuming unit cost for arithmetic operations), no matter whether  $Q$  contains nondistinguished variables or not.*

*Proof (sketch).* First, note that under bag semantics, there is no need to make a distinction between distinguished- and nondistinguished variables: In Section 4, the problem was to avoid multiple counting of the same solution if it can be derived in several ways. However, under bag semantics, this is exactly what we want to do. Hence, we can just consider all variables in  $Q$  as free.

It therefore remains to show that the algorithm from Section 3 can be adapted to bag semantics. In fact, the only thing that needs to be changed is in the initialisation phase: Recall that for every distinct tuple  $\rho \in R_t$ , we add a tuple  $(\rho, 1)$  to  $\hat{R}_t$ . All that needs to be changed is to replace 1 by the number of occurrences of  $\rho \in R_t$ . It can be easily checked that the remaining algorithm can be left unchanged.  $\square$

## 6 Conclusion

In [16], we started to look for a tractable fragment of the  $\#CQ$  problem via the tree-width of the query. In the current paper, we initiated a systematic study of tractable fragments of  $\#CQ$ , considering ACQs and CQs with bounded hypertree-width (which is a more general measure than tree-width). We have presented new algorithms based on semi-joins which allowed us to prove several new tractability results. We also identified limits to this search for tractable fragments by proving the  $\#P$ -completeness of the  $\#CQ$  problem for arbitrary acyclic conjunctive queries. Our results apply to the count operator, which is an

integral part of the SQL language. In the future, we want to extend the search for tractable fragments to other aggregate functions (like minimum, maximum, sum, average) and to the group by construct in SQL.

Another interesting extension of [16] would be the search for a dichotomy result in the style of [13]. Further, extending the results from [2] concerning the expressiveness of counting the answers of CQs to ACQs, and addressing the complexity related questions raised there is another direction for future work.

## References

1. M. Bauland, P. Chapdelaine, N. Creignou, M. Hermann, and H. Vollmer. An algebraic approach to the complexity of generalized conjunctive queries. In *SAT 2004 - Revised Selected Papers*, volume 3542 of *LNCS*, pages 30–45. Springer, 2005.
2. M. Bielecki and J. V. den Bussche. Database interrogation using conjunctive queries. In *Proc. ICDT 2003*, pages 259–269. Springer, 2003.
3. A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. STOC 1977*, pages 77–90. ACM, 1977.
4. C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 239(2):211–229, 2000.
5. N. Creignou and M. Hermann. Complexity of generalized satisfiability counting problems. *Inf. Comput.*, 125(1):1–12, 1996.
6. A. Durand, M. Hermann, and P. G. Kolaitis. Subtractive reductions and complete problems for counting complexity classes. *Theor. Comput. Sci.*, 340(3):496–513, 2005.
7. R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514–550, 1983.
8. J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *J. ACM*, 49(6):716–752, 2002.
9. G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.
10. G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J. ACM*, 48(3):431–498, 2001.
11. G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.
12. M. Graham. On the universal relation. *University of Toronto technical report*, 1979.
13. M. Grohe, T. Schwentick, and L. Segoufin. When is the evaluation of conjunctive queries tractable? In *Proc. STOC 2001*, pages 657–666, 2001.
14. P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *J. Comput. Syst. Sci.*, 61(2):302–332, 2000.
15. C. H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994.
16. R. Pichler and S. Skritek. The complexity of evaluating tuple generating dependencies. In *Proc. ICDT 2011*, pages 244–255. ACM, 2011.
17. N. Robertson and P. D. Seymour. Graph minors. ii. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.
18. L. G. Valiant. The complexity of enumeration and reliability problems. *SIAM J. Comput.*, 8(3):410–421, 1979.
19. M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. VLDB 1981*, pages 82–94. IEEE Computer Society, 1981.
20. C. Yu and M. Ozsoyoglu. An algorithm for tree-query membership of a distributed query. In *Proc. COMPSAC 1979w*, pages 306–312. IEEE, 1979.