Query Answering Exploiting Structural Properties*

Francesco Scarcello
DEIS, Università della Calabria, Italy
scarcello@unical.it

ABSTRACT

We review the notion of hypertree width, a measure of the degree of cyclicity of hypergraphs that is useful for identifying and solving efficiently easy instances of hard problems, by exploiting their structural properties. Indeed, a number of relevant problems from different areas, such as database theory, artificial intelligence, and game theory, are tractable when their underlying hypergraphs have small (i.e., bounded by some fixed constant) hypertree width. In particular, we describe how this notion may be used for identifying tractable classes of database queries and answering such queries in an efficient way.

1. INTRODUCTION

In this paper we deal with the fundamental problem of evaluating queries in relational databases, focusing on recently proposed techniques based on structural properties of the queries. For the sake of simplicity, we consider conjunctive queries (CQs), though most results may be easily extended to more general queries. The class CQ, equivalent in expressive power to the class of Select-Project-Join queries, is probably the most thoroughly analyzed class of database queries. Note that the great interest in conjunctive queries is also due to the fact that CQ evaluation is essentially the same problem as *conjunctive query containment* [6], which is of central importance in *view-based query processing* [2], and *constraint satisfaction*, which is one of the major problems studied in the field of AI (see, e.g., Vardi's survey paper [47] on the interactions between the areas of query evaluation and constraint satisfaction).

Recall that database management systems (DBMSs) have specialized modules, called query optimizers, looking for good ways to deal with any given query. For all commercial DBMSs, such a way is always based on *quantitative methods*: they examine a number of alternative plans for answering a query and then choose the best one, according to some cost model. These planners exploit information on the data, e.g., sizes of relations, indices, and so on. In fact, all of them compute just approximations of optimal query plans, as the optimization problem is NP-hard, in general. See [39] for a short survey of quantitative methods and for further references.

A completely different approach to query answering is based on structural properties of queries, rather than on quantitative information about data values. Exploiting such properties is possible to answer large classes of queries efficiently, that is, with a polynomial-time upper bound. The structure of a query Q is best represented by its *query hypergraph* $\mathcal{H}(Q) = (V, H)$, whose set V of vertices consists of all variables occurring in Q, and where the set H of hyperedges contains, for each query atom A, the set $var(A)$ of all variables occurring in A. As an example, consider the following query

Q_0: $ans \leftarrow s_1(A,B,D) \ \wedge \ s_2(B,C,D) \ \wedge \ s_3(B,E) \ \wedge \ s_4(D,G) \ \wedge s_5(E,F,G) \ \wedge \ s_6(E,H) \ \wedge \ s_7(F,I) \ \wedge \ s_8(G,J)$.

Figure 1 shows its associated hypergraph $\mathcal{H}(Q_0)$.

One of the most important and deeply studied class of tractable queries is the class of *acyclic queries* [5, 7, 9, 14, 28, 33, 37, 48, 49]. It was shown that acyclic queries coincide with the *tree queries* [4], see also [1, 30, 43]. The latter are queries whose query hypergraph has a *join tree* (or *join forest*) (see Section 3 for a formal definition). By well-known results of Yannakakis [48], acyclic conjunctive queries are efficiently solvable. More precisely, all answers of an acyclic conjunctive query can be computed in time polynomial in the combined size of the input and the output. This is the best possible result, because in general the answer of a query may contain an exponential number of tuples. Recall that, for cyclic queries, even computing small outputs, e.g. just one tuple, or checking whether the answer of a query is non-empty (Boolean queries) requires exponential time (unless P $=$ NP) [6].

Therefore, many attempts have been made in the literature for extending the good results about acyclic conjunctive queries to relevant classes of *nearly acyclic* queries. We call these techniques *structural query decomposition methods*,[1] because they are based on the acyclicization of cyclic (hyper)graphs. More precisely, each method specifies how appropriately transforming a conjunctive query into an equivalent tree query (i.e., acyclic query given in form of a join tree), by organizing its atoms into a polynomial number of clusters, and suitably arranging the clusters as a tree (see Figure 1). Each cluster contains a number of atoms. After performing the join of the relations corresponding to the atoms jointly contained in each cluster, we obtain a join tree of an acyclic query which is equivalent to the original query. The resulting query can be answered in output-polynomial time by Yannakakis's algorithm. Thus, in case of a Boolean query, it can be answered in polynomial time. The tree of atom-clusters produced by a structural query decomposition method on a given query Q is referred to as the *decomposition* of Q. Figure 1 also shows two possible decompositions of our example query Q_0. A decomposition of Q can be seen as a query plan for Q, requiring to first evaluate the join of each cluster, and then to process the resulting join tree in a bottom-up fashion (following Yannakakis's algorithm).

*[*] **Database Principles Column.** Column editor: Leonid Libkin, Department of Computer Science, University of Toronto, Toronto, Ontario M5S 3H5, Canada. E-mail:libkin@cs.toronto.edu

[1] In the field of constraint satisfaction, the same notion is known as *structural CSP decomposition method*, cf. [15].

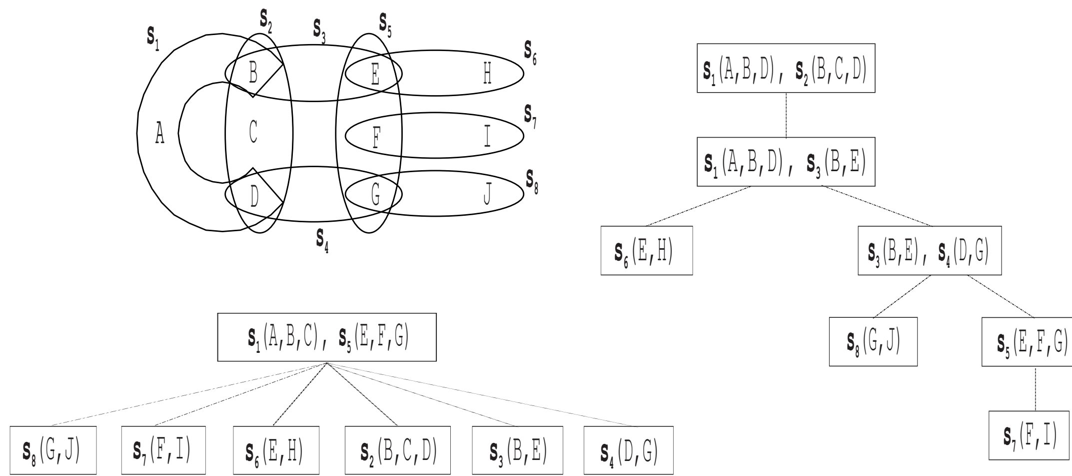

Figure 1: Hypergraph $\mathcal{H}(Q_0)$ (left), two hypertree decompositions of width 2 of $\mathcal{H}(Q_0)$ (right and bottom).

Thus, the efficiency of a structural decomposition method essentially depends on the maximum size of the produced clusters, measured (according to the chosen decomposition method) either in terms of the number of variables or in terms of the number of atoms. For a given decomposition, this size is referred-to as the *width* of the decomposition. For example, if we adopt the number of atoms, then the width of both decompositions shown in Figure 1 is 2. Intuitively, the complexity of transforming a given decomposition into an equivalent tree query is exponential in its width w. In fact, the evaluation cost of each of the (polynomially many) clusters is bounded by the cost of performing the (at most) w joins of its relations, which is in turn bounded by $O(|r_{max}|^{w-1} \log |r_{max}|)$, where $|r_{max}|$ denotes the size of the largest relation r_{max} in the database. The overall cost (transformation+evaluation of the resulting acyclic query) is thus $O(v|r_{max}|^{w-1} \log |r_{max}|)$, where v is the number of vertices of the decomposition tree. It is worthwhile noting that, for queries involving many atoms, exploiting such a structural information may lead to a quite remarkable computational saving. For instance, the above upper bound is $O(7|r_{max}| \log |r_{max}|)$ for the query in Figure 1, whereas typical query answering algorithms would take $O(|r_{max}|^7 \log |r_{max}|)$ time, in the worst case.

In general, a rough upper bound for the cost of answering a given query Q according to any structural method D is given by $O(n^{w+1} \log n)$, where w is the D-width of Q and n is the total size of the input problem, that is, the size of the query and of the database encoding [15]. Therefore, once we fix a bound k for such a width, the structural method D identifies a class of queries that can be answered in polynomial time, namely, the class of all queries having k-bounded D-width (i.e., D-width at most k).[2] The main structural decomposition methods are based on the notions of Biconnected Components [11], Tree Decompositions [35, 7, 28, 8, 25, 10], Hinge Decompositions [26], and Hypertree Decompositions [18, 19, 21, 40].

Among them, the Hypertree Decomposition Method (HYPERTREE) seems to be the most powerful method, as a large class of cyclic queries has a low hypertree-width, and in fact it strongly generalizes all other structural methods [15]. More precisely, this means that every class of queries that is recognized as tractable according to any structural method D (has k-bounded D-width), is also tractable according to HYPERTREE (has k-bounded HYPERTREE-width), and that there are classes of queries that are tractable according to HYPERTREE, but not tractable w.r.t. D (have unbounded D-width). Moreover, for any fixed $k > 0$, deciding whether a hypergraph has hypertree width at most k is feasible in polynomial time, and is actually highly parallelizable, as this problem belongs to LOGCFL [19] (See [36, 17], for properties and characterizations of this complexity class). In fact, it has been conjectured that a class of queries is tractable if and only the cores of their structures have bounded hypertree width (under some widely believed complexity-theoretic assumptions) [24]. The first part of this paper is devoted to the presentation of the main results about hypertree decompositions.

Despite their very nice computational properties, all the above structural decomposition methods, including Hypertree Decomposition, are often unsuited for some real-world applications. For instance, in a practical context, one may prefer query plans (i.e., minimum-width decompositions) which minimize the number of clusters having the largest cardinality. Even more importantly, decomposition methods focus "only" on structural features, while they completely disregard "quantitative" aspects of the query, that may dramatically affect the query-evaluation time. For instance, while answering a query, the computation of an *arbitrary* hypertree decomposition (having minimum width) could not be satisfactory, since it does not take into account important quantitative factors, such as relation sizes, attribute selectivity, and so on. These factors are flattened in the query hypergraph (which considers only the query structure), while their suitable exploitation can significantly reduce the cost of query evaluation.

On the other hand, query optimizers of commercial DBMSs are based solely on quantitative methods and do not care of structural properties at all. Indeed, all the commercial DBMSs restrict the search space of query plans to very simple structures (e.g., left-deep trees), and then try to find the best plans among them, by estimating their evaluation costs, exploiting quantitative information on the input database. It follows that, on some low-width queries with a guaranteed polynomial-time evaluation upper-bound, they may also take time $O(n^\ell)$, which is exponential in the length ℓ of

[2] Intuitively, the D-width of a query Q is the minimum width of the decompositions of Q obtainable by method D.

the query, rather than on its width. On some relevant applications with many atoms involved, this may lead to unacceptable costs. For instance, consider the problem of the population and refreshing of cubes in data warehouse initialization and management. Periodically, a number of batch queries are executed on the reconciled operational database. Note that these queries are typically very different from OLAP queries. Indeed, while the latter queries are executed on star schemes (or similar simple schemes), these populating queries usually span several tables in the reconciled scheme in order to update both dimension and fact tables. Thus, they are very often long queries involving many join operations, plus selections, projections and, possibly, grouping and aggregate operators. In this context, the choice of a good query-execution strategy is therefore particularly relevant, because the differences among execution times can be several orders of magnitude large. In fact, very often such queries are not very intricate and have low hypertree width, though they are not necessarily acyclic.

To overcome the above mentioned drawbacks of both approaches, we proposed an extension of hypertree decompositions, in order to combine this structural decomposition method with quantitative approaches [41]. In the second part of this paper, we review the main results on this generalized notion of HYPERTREE, where hypertree decompositions are equipped with polynomial-time weight functions that may encode quantitative aspects of the query database, or other additional requirements. In general, computing a minimal weighted-decomposition is harder than computing a standard decomposition. However, we present a class of functions, called *tree aggregation functions* (TAFs), which is useful for query optimization and easy to deal with.

We describe how the notion of weighted hypertree decomposition can be used for generating effective query plans for the evaluation of conjunctive queries, by combining structural and quantitative information. We also briefly report some results of an ongoing experimental activity, showing that this hybrid approach may in fact lead to significant computational savings.

2. QUERIES AND ACYCLIC HYPERGRAPHS

We will adopt the standard convention [1, 43] of identifying a relational database instance with a logical theory consisting of ground facts. Thus, a tuple $\langle a_1, \dots a_k \rangle$, belonging to relation r, will be identified with the ground atom $r(a_1, \dots, a_k)$. The fact that a tuple $\langle a_1, \dots, a_k \rangle$ belongs to relation r of a database instance **DB** is thus simply denoted by $r(a_1, \dots, a_k) \in \mathbf{DB}$.

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

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

where $n \geq 0$; $r_1, \dots r_n$ are relation names (not necessarily distinct) of DS; ans is a relation name not in DS; and $\mathbf{u}, \mathbf{u_1}, \dots, \mathbf{u_n}$ 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 $\mathbf{u}\theta$ such that $\theta : var(Q) \longrightarrow U$ is a substitution replacing each variable in $var(Q)$ by a value of U and such that for $1 \leq i \leq n$, $r_i(\mathbf{u_i})\theta \in \mathbf{DB}$. (For an atom A, $A\theta$ denotes the atom obtained from A by uniformly substituting $\theta(X)$ for each variable X occurring in A.)

If Q is a conjunctive query, we define the hypergraph $H(Q) = (V, E)$ associated to Q as follows. The set of vertices V, denoted by $var(H(Q))$, consists of all variables occurring in Q. The set E, denoted by $edges(H(Q))$, contains for each atom $r_i(\mathbf{u_i})$ in the body of Q a hyperedge consisting of all variables occurring in $\mathbf{u_i}$. Note that the cardinality of $edges(H(Q))$ 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 $edges(H(Q))$. For example, the three query atoms $r(X, Y)$, $r(Y, X)$, and $s(X, X, Y)$ all correspond to a unique hyperedge $\{X, Y\}$.

A query Q is acyclic if and only if its hypergraph $H(Q)$ is acyclic or, equivalently, if it has has a join forest. A *join forest* for the hypergraph $H(Q)$ is a forest G whose set of vertices V_G is the set $edges(H(Q))$ and such that, for each pair of hyperedges h_1 and h_2 in V_G having variables in common (i.e., such that $h_1 \cap h_2 \neq \emptyset$), the following conditions hold:

1. h_1 and h_2 belong to the same connected component of G, and
2. all variables common to h_1 and h_2 occur in every vertex on the (unique) path in G from h_1 to h_2.

If G is a tree, then it is called a *join tree* for $H(Q)$.

Intuitively, the efficient behavior of acyclic instances is due to the fact that they can be evaluated by processing any of their join trees bottom-up by performing upward semijoins, thus keeping the size of the intermediate relations small (while it could become exponential, if regular join were performed).

Let us recall the highly desirable computational properties of acyclic queries:

1. Acyclic instances can be efficiently solved. Yannakakis provided a (sequential) polynomial time algorithm for Boolean acyclic queries[3]. Moreover, he showed that the answer of a non-Boolean acyclic conjunctive query can be *computed* in time polynomial in the combined size of the input instance and of the output relation [48].

2. We have shown that answering queries is highly parallelizable on acyclic queries, as this problem (actually, the decision problem of answering Boolean queries) is complete for the low complexity class LOGCFL [18]. Efficient parallel algorithms for Boolean and non-Boolean queries have been proposed in [18] and [16]. They run on parallel database machines that exploit the *inter-operation parallelism* [44], i.e., machines that execute different relational operations in parallel. These algorithms can be also employed for solving acyclic queries efficiently in a distributed environment.

3. Acyclicity is efficiently recognizable: deciding whether a hypergraph is acyclic is feasible in linear time [42] and belongs to the class L (deterministic logspace). The latter result is

[3]Note that, since both the database **DB** and the query Q are part of an input-instance, what we are considering is the *combined complexity* of the query [46].

new: it follows from the fact that hypergraph acyclicity belongs to SL [17], and from the very recent proof that SL is in fact equal to L [34].

3. HYPERTREE DECOMPOSITIONS

We recall the formal definition and the most important results about *hypertree width* and *hypertree decompositions*.

A *hypertree for a hypergraph* $\mathcal{H}$ is a triple $\langle T, \chi, \lambda \rangle$, where $T = (N, E)$ is a rooted tree, and χ and λ are labeling functions which associate to each vertex $p \in N$ two sets $\chi(p) \subseteq var(\mathcal{H})$ and $\lambda(p) \subseteq edges(\mathcal{H})$. The *width* of a hypertree is the cardinality of its largest λ label, i.e., $max_{p \in N}|\lambda(p)|$.

We denote the set of vertices of any rooted tree T by $vertices(T)$, and its root by $root(T)$. Moreover, for any $p \in vertices(T)$, T_p denotes the subtree of T rooted at p. If T' is a subtree of T, we define $\chi(T') = \bigcup_{v \in vertices(T')} \chi(v)$.

Definition 3.1 *[21]*A *generalized hypertree decomposition* of a hypergraph $\mathcal{H}$ is a hypertree $HD = \langle T, \chi, \lambda \rangle$ for $\mathcal{H}$ which satisfies the following conditions:

1. *For each edge $h \in edges(\mathcal{H})$, all of its variables occur together in some vertex of the decomposition tree*, that is, there exists $p \in vertices(T)$ such that $h \subseteq \chi(p)$ (we say that p *covers* h).
2. *Connectedness Condition*: for each variable $Y \in var(\mathcal{H})$, the set $\{p \in vertices(T) \mid Y \in \chi(p)\}$ induces a (connected) subtree of T.
3. For each vertex $p \in vertices(T)$, *variables in the χ labeling should belong to edges in the λ labeling*, that is, $\chi(p) \subseteq var(\lambda(p))$.

A *hypertree decomposition* is a generalized hypertree decomposition that satisfies the following additional condition:

4. *Special Descendant Condition*: for each $p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

The HYPERTREE *width* $hw(\mathcal{H})$ (resp., generalized hypertree width $ghw(\mathcal{H})$) of $\mathcal{H}$ is the minimum width over all its hypertree decompositions (resp., generalized hypertree decompositions).

An edge $h \in edges(\mathcal{H})$ is *strongly covered* in *HD* if there exists $p \in vertices(T)$ such that $var(h) \subseteq \chi(p)$ and $h \in \lambda(p)$. In this case, we say that p strongly covers h. A decomposition HD of hypergraph $\mathcal{H}$ is a *complete decomposition* of $\mathcal{H}$ if every edge of $\mathcal{H}$ is strongly covered in *HD*. From any (generalized) hypertree decomposition *HD* of $\mathcal{H}$, we can easily compute a complete (generalized) hypertree decomposition of $\mathcal{H}$ having the same width.

Note that the notions of hypertree width and generalized hypertree width are true generalizations of acyclicity, as the acyclic hypergraphs are precisely those hypergraphs having hypertree width and generalized hypertree width one. In particular, as we will see in the next section, the classes of conjunctive queries having bounded (generalized) hypertree width have the same desirable computational properties as acyclic queries [19].

At first glance, a generalized hypertree decomposition of a hypergraph may simply be viewed as a clustering of the hyperedges (i.e., query atoms) where the classical connectedness condition of join trees holds. However, a generalized hypertree decomposition may deviate in two ways from this principle: **(1)** A hyperedge already used in some cluster may be reused in some other cluster; **(2)** Some variables occurring in reused hyperedges are not required to fulfill any condition.

For a better understanding of this notion, let us focus on the two labels associated with each vertex p: the set of hyperedges $\lambda(p)$, and the set of *effective* variables $\chi(p)$, which are subject to the connectedness condition (2). Note that all variables that appear in the hyperedges of $\lambda(p)$ but that are not included in $\chi(p)$ are "ineffective" for v and do not count w.r.t. the connectedness condition. Thus, the χ labeling plays the crucial role of providing a join-tree like re-arranging of all connections among variables. Besides the connectedness condition, this re-arranging should fulfill the fundamental Condition 1: every hyperedge (i.e., query atom, in our context) has to be properly considered in the decomposition, as for graph edges in tree-decompositions and for hyperedges in join trees (where this condition is actually even stronger, as hyperedges are in a one-to-one correspondence with vertices of the tree). Since the only relevant variables are those contained in the χ labels of vertices in the decomposition tree, the λ labels are "just" in charge of covering such relevant variables (Condition 3) with as few hyperedges as possible. Indeed, the width of the decomposition is determined by the largest λ label in the tree. This is the most important novelty of this approach, and comes from the specific properties of hypergraph-based problems, where hyperedges often play a predominant role. For instance, think of our database framework: the cost of evaluating a natural join operation with k atoms (read: k hyperedges) is $O(n^{k-1} \log n)$, no matter of the number of variables occurring in the query.

Example 3.2 Consider the following conjunctive query Q_1:

$$\begin{array}{rcl} ans & \leftarrow & a(S, X, X', C, F) \wedge b(S, Y, Y', C', F') \\ & & \wedge\, c(C, C', Z) \wedge d(X, Z) \wedge \\ & & e(Y, Z) \wedge f(F, F', Z') \wedge g(X', Z') \wedge \\ & & h(Y', Z') \wedge j(J, X, Y, X', Y'). \end{array}$$

Let $\mathcal{H}_1$ be the hypergraph associated to Q_1. Since $\mathcal{H}_1$ is cyclic, $hw(\mathcal{H}_1) > 1$ holds. Figure 2 shows a (complete) hypertree decomposition HD_1 of $\mathcal{H}_1$ having width 2, hence $hw(\mathcal{H}_1) = 2$.

In order to help the intuition, Figure 3 shows an alternative representation of this decomposition, called *atom* (or *hyperedge*) *representation* [19]: each node p in the tree is labeled by a set of atoms representing $\lambda(p)$; $\chi(p)$ is the set of all variables, distinct from '_', appearing in these hyperedges. Thus, in this representation, possible occurrences of the anonymous variable '_' take the place of variables in $var(\lambda(p)) - \chi(p)$.

Another example is depicted in Figure 1, which shows two hypertree decompositions of query Q_0 in Section 1. Both decompositions have width two and are complete decompositions of Q_0. □

Let k be a fixed positive integer. We say that a CQ instance I has k-bounded (generalized) hypertree width if $(g)hw(\mathcal{H}(I)) \leq k$. A class of queries has bounded (generalized) hypertree width if there is some $k \geq 1$ such that all instances in the class have k-bounded (generalized) hypertree width.

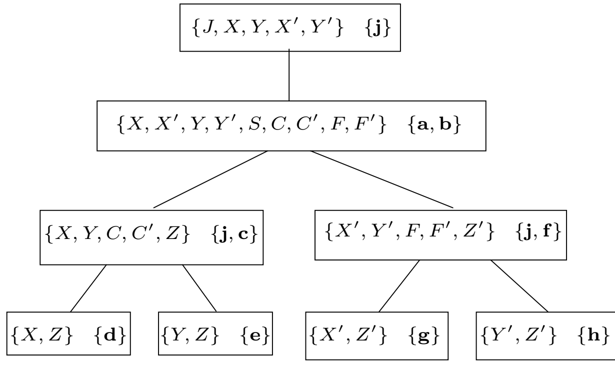

Figure 2: A 2-width hypertree decomposition of hypergraph $\mathcal{H}_1$ in Example 3.2

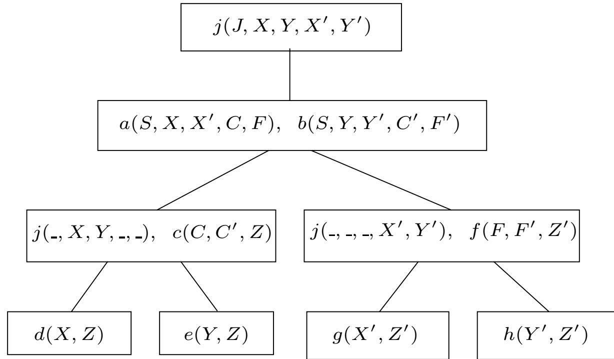

Figure 3: Atom representation of the hypertree decomposition in Figure 2

Clearly enough, choosing a tree and a clever combination of χ and λ labeling for its vertices in order to get a decomposition below a fixed threshold width k is not that easy, and is definitely more difficult than computing a simple tree decomposition, where only variables are associated with each vertex. In fact, the tractability of generalized hypertree width is an interesting open problem, as no polynomial time algorithm is known for deciding whether a hypergraph has generalized hypertree width at most k, for any fixed $k \geq 2$.

It is thus very nice and somehow surprising that dealing with the hypertree width is a very easy task. More precisely, for any fixed $k \geq 1$, deciding whether a given hypergraph has hypertree width at most k is in LOGCFL, and thus it is a tractable and highly parallelizable problem. Correspondingly, the search problem of computing a k-bounded hypertree decomposition belongs to the functional version of LOGCFL, which is $\mathrm{L}^{\mathrm{LOGCFL}}$ [19]. See the Hypertree Decomposition Homepage [40], for available implementations of algorithms for computing hypertree decompositions, and further links to heuristics and other papers on this subject.

Let us briefly discuss the only difference of hypertree decomposition with respect to generalized hypertree decomposition, that is, the *descendant condition* (Condition 4 in Definition 3.1). Consider a vertex p of a hypertree decomposition and a hyperedge $h \in \lambda(p)$ such that some variables $\bar{X} \subseteq h$ occur in the χ labeling of some vertices in the subtree T_p rooted at p. Then, according to this condition, these variables must occur in $\chi(p)$, too. This means, intuitively, that we have to deal with variables in $\bar{X}$ at this point of the decomposition tree, if we want to put h in $\lambda(p)$. For instance, as a consequence of this condition, for the root r of any hypertree decomposition we always have $\chi(r) = var(\lambda(r))$. However, once a hyperedge has been covered by some vertex of the decomposition tree, any subset of its variables can be used freely in order to decompose the remaining cycles in the hypergraph.

To shed more light on this restriction, consider what happens in the related hypergraph-based notions: in query decompositions [7], all variables are relevant; at the opposite side, in generalized hypertree decompositions, we can choose as relevant variables any subset of variables occurring in λ, without any limitation; in hypertree decompositions, we can choose any subset of relevant variables as long as the above descendant condition is satisfied. Therefore, the notion of hypertree width is clearly more powerful than the (intractable) notion of query width, but less general than the (probably intractable) notion of generalized hypertree width, which is the most liberal notion.

For instance, look at Figure 3: the variables in the hyperedge corresponding to atom j in $\mathcal{H}_1$ are jointly included only in the root of the decomposition, while we exploit two different subsets of this hyperedge in the rest of the decomposition tree. Note that the descendant condition is satisfied. Take the vertex at level 2, on the left: the variables j, X' and Y' are not in the χ label of this vertex (they are replaced by the anonymous variable '_'), but they do not occur anymore in the subtree rooted at this vertex. On the other hand, if we were forced to take all the variables occurring in every atom in the decomposition tree, it would not be possible to find a decomposition of width 2. Indeed, j is the only atom containing both pairs X, Y and X', Y', and it cannot be used again entirely, for its variable J cannot occur below the vertex labeled by a and b, otherwise it would violate the connectedness condition (i.e., Condition 2 of Definition 3.1). In fact, every query decomposition of this hypergraph has width 3, while the hypertree width is 2. In this case the generalized hypertree width is 2, as well, but in general it may be less than the hypertree width. However, after a recent interesting result by Adler et al. [3], the difference of these two notions of width is within a constant factor: for any hypergraph $\mathcal{H}$, $ghw(\mathcal{H}) \leq hw(\mathcal{H}) \leq 3ghw(\mathcal{H}) + 1$. It follows that a class of hypergraphs has bounded generalized hypertree width if and only if it has bounded hypertree width, and thus the two notions identify the same set of tractable classes.

Though the formal definition of hypertree width is rather involved, it is worthwhile noting that this notion has very natural characterizations in terms of games and logics [21]:

- **The robber and marshals game (R&Ms game).** It is played by one robber and a number of marshals on a hypergraph. The robber moves on variables, while marshals move on hyperedges. At each step, any marshal controls an entire hyperedge. During a move of the marshals from the set of hyperedges E to to the set of hyperedges E', the robber cannot pass through the vertices in $B = (\cup E) \cap (\cup E')$, where, for a set of hyperedges F, $\cup F$ denotes the union of all hyperedges in F. Intuitively, the vertices in B are those not released by the marshals during the move. As in the monotonic robber and cops game defined for treewidth [38], it is required that the marshals capture the robber by monotonically shrinking the moving space of the robber. The game is

won by the marshals if they corner the robber somewhere in the hypergraph. A hypergraph $\mathcal{H}$ has k-bounded hypertree width if and only if k marshals win the R&Ms game on $\mathcal{H}$.

- **Logical characterization of hypertree width.** Let L denote the existential conjunctive fragment of positive first order logic (FO). Then, the class of queries having k-bounded hypertree width is equivalent to the k-guarded fragment of L, denoted by $\mathrm{GF}_k(\mathrm{L})$. Roughly, we say that a formula Φ belongs to $\mathrm{GF}_k(\mathrm{L})$ if, for any subformula ϕ of Φ, there is a conjunction of up to k atoms jointly acting as a guard, that is, covering the free variables of ϕ. Note that this notion is related to the *loosely guarded fragment* as defined (in the context of full FO) by Van Benthem [45], where an arbitrary number of atoms may jointly act as guards (see also [23]).

3.1 Query Decompositions and Query Plans

In this section we describe the basic idea to exploit (generalized) hypertree decompositions for answering conjunctive queries.

Let $k \geq 1$ be a fixed constant, Q a conjunctive query over a database **DB**, and $HD = \langle T, \chi, \lambda \rangle$ a generalized hypertree decomposition of Q of width $w \leq k$. Then, we can answer Q in two steps:

1. For each vertex $p \in vertices(T)$, compute the join operations among relations occurring together in $\lambda(p)$, and project onto the variables in $\chi(p)$. At the end of this phase, the conjunction of these intermediate results forms an acyclic conjunctive query, say Q', equivalent to Q. Moreover, the decomposition tree T represents a join tree of Q'.
2. Answer Q', and hence Q, by using any algorithm for acyclic queries, e.g. Yannakakis's algorithm.

For instance, Figure 4 shows the tree $\mathcal{JT}_1$ obtained after Step 1 above, from the query Q_1 in Example 3.2 and the generalized hypertree decomposition in Figure 3. E.g. observe how the vertex labeled by atom p_3 is built. It comes from the join of atoms j and c (occurring in its corresponding vertex in Figure 3), and from the subsequent projection onto the variables X, Y, C, C', and Z (belonging to the χ label of that vertex). By construction, $\mathcal{JT}_1$ satisfies the connectedness condition. Therefore, the conjunction of atoms labeling this tree is an acyclic query, say Q_1', such that $\mathcal{JT}_1$ is one of its join trees. Moreover, it is easy to see that Q_1' has the same answer as Q_1 [19].

Step 1 is feasible in $O(m|r_{max}|^{w-1} \log |r_{max}|)$ time, where m is the number of vertices of T, and r_{max} is the relation of **DB** having the largest size. In fact, for Boolean queries, Yannakakis's algorithm in Step 2 does not take more time than Step 1, and thus its cost is an upper bound for the entire query evaluation process. For non-Boolean queries, Yannakakis's algorithm works in time polynomial in the combined size of the input and of the output, and thus we should add to the above cost a term that depends on the answer of the given query (which may be exponential w.r.t. the input size). For instance, if we consider query Q_1, the above upper bound is $O(7|r_{max}| \log |r_{max}|)$, whereas typical query answering algorithms (which do not exploit structural properties) would take $O(|r_{max}|^7 \log |r_{max}|)$ time, in the worst case.

It has been observed that, according to Definition 3.1, a hypergraph may have some (usually) undesirable hypertree decompositions [19], possibly with a large number m of vertices in the decomposition tree. For instance, a decomposition may contain two vertices with exactly the same labels. Therefore, a *normal form* for hypertree decompositions has been defined in [19], and then strengthened in [41], in order to avoid such kind of redundancies. Hypertree decompositions in normal form having width at most k may be computed in time polynomial in the size of the given hypergraph $\mathcal{H}$ (but exponential in the parameter k). The number m of vertices cannot exceed the number of variables in $\mathcal{H}$, and is typically much smaller. Moreover, $\mathcal{H}$ has a hypertree decomposition of width w if and only if it has a normal-form hypertree decomposition of the same width w.

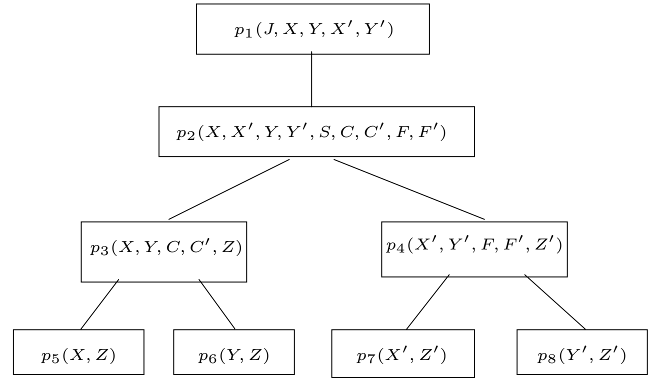

Figure 4: Join tree $\mathcal{JT}_1$ computed for query Q_1'

It follows that, for any fixed $k \geq 1$, the class of all queries having k-bounded hypertree width may be answered in polynomial time (actually, in input-output polynomial time, for non-Boolean queries). Indeed, given a query Q, both computing a hypertree decomposition HD of width at most k of $\mathcal{H}(Q)$, and then answering Q exploiting HD are polynomial-time tasks.

As far as generalized hypertree decompositions are concerned, we currently miss a polynomial-time algorithm for recognizing queries having k-bounded generalized hypertree-width. However, there is a great deal of interest in these decompositions, and some first results are coming. For instance, some very good heuristics for computing generalized hypertree decompositions are described in [29, 32].

4. WEIGHTED HYPERTREE DECOMPOSITIONS

As described in the previous section, given a query Q on a database **DB** and a small-width decomposition HD for Q, we know that there is a polynomial time upper bound for answering Q, while in general this problem is NP-hard and all the available algorithms requires exponential time, in the worst case. However, HD is not just a theoretical indication of tractability for Q. Rather, the above two steps for evaluating Q actually represent a query plan for it, though not completely specified. For instance, no actual join method (merge, nested-loop, etc.) is chosen, but this final more physical phase can be easily implemented using well-known database techniques. We remark that such optimizations are executed just on relations belonging to the same vertex, and hence on w relations at most, if w is the width of HD. Thus, also optimal methods based on dynamic programming or sophisticated heuristics can be employed, as the size of the problem is small.

The remaining interesting problem is before this evaluation phase, where we have to compute a decomposition for $\mathcal{H}(Q)$. Indeed, in general there is an exponential number of hypertree decompositions of a hypergraph. Every decomposition encodes a way of aggregating groups of atoms and arranging them in a tree-like fashion. As far as the polynomial-time upper bound is concerned, we may be happy with any minimum-width decomposition. However, in practical real-world applications we have to exploit all available information. In particular, for database queries, we cannot get rid of information on the database **DB**. Indeed, looking only at the query structure is not the best we can do, if we may additionally exploit the knowledge of relation sizes, attribute selectivity, and so on.

4.1 Minimal Decompositions

In this section, we thus consider hypertree decompositions with an associated weight, which encodes our preferences, and allows us to take into account further requirements, besides the width. We will see how to answer queries more efficiently, by looking for their best decompositions.

Formally, given a hypergraph $\mathcal{H}$, a *hypertree weighting function* (short: HWF) $\omega_{\mathcal{H}}$ is any polynomial-time function that maps each generalized hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ of $\mathcal{H}$ to a real number, called the *weight* of HD.

For instance, a very simple HWF is the function $\omega^{w}_{\mathcal{H}}(HD) = \max_{p \in vertices(T)} |\lambda(p)|$, that weights a decomposition HD just on the basis of its worse vertex, that is the vertex with the largest λ label, which also determines the width of the decomposition.

In many applications, finding such a decomposition having the minimum width is not the best we can do. We can think of minimizing the number of vertices having the largest width w and, for decompositions having the same numbers of such vertices, minimizing the number of vertices having width $w-1$, and continuing so on, in a lexicographical way. To this end, we can define the HWF $\omega^{lex}_{\mathcal{H}}(HD) = \sum_{i=1}^{w} |\{p \in N \text{ such that } |\lambda(p)| = i\}| \times B^{i-1}$, where $N = vertices(T)$, $B = |edges(\mathcal{H})| + 1$, and w is the width of HD. Note that any output of this function can be represented in a compact way as a radix B number of length w, which is clearly bounded by the number of edges in $\mathcal{H}$. Consider again the query Q_0 of the Introduction, and the hypertree decomposition, say HD', of $\mathcal{H}(Q_0)$ shown in Figure 1, on the right. It is easy to see that HD' is not the best decomposition w.r.t. $\omega^{lex}_{\mathcal{H}}$ and the class of hypertree decompositions in normal form. Indeed, $\omega^{lex}_{\mathcal{H}}(HD') = 4 \times 9^0 + 3 \times 9^1$, and thus the decomposition HD'' shown on the bottom of Figure 1 is better than HD', as $\omega^{lex}_{\mathcal{H}}(HD'') = 6 \times 9^0 + 1 \times 9^1$.

Let $k > 0$ be a fixed integer and $\mathcal{H}$ a hypergraph. We define the class $kHD_{\mathcal{H}}$ (resp., $kNFD_{\mathcal{H}}$) as the set of all hypertree decompositions (resp., normal-form hypertree decompositions) of $\mathcal{H}$ having width at most k.

Definition 4.1 *[41]* Let $\mathcal{H}$ be a hypergraph, $\omega_{\mathcal{H}}$ a weighting function, and $\mathcal{C}_{\mathcal{H}}$ a class of generalized hypertree decompositions of $\mathcal{H}$. Then, a decomposition $HD \in \mathcal{C}_{\mathcal{H}}$ is *minimal* w.r.t. $\omega_{\mathcal{H}}$ and $\mathcal{C}_{\mathcal{H}}$, denoted by $[\omega_{\mathcal{H}}, \mathcal{C}_{\mathcal{H}}]$-$minimal$, if there is no $HD' \in \mathcal{C}_{\mathcal{H}}$ such that $\omega_{\mathcal{H}}(HD') < \omega_{\mathcal{H}}(HD)$. □

For instance, the $[\omega^{w}_{\mathcal{H}}, kHD_{\mathcal{H}}]$-$minimal$ decompositions are exactly the k-bounded hypertree decompositions having the minimum possible width, while the $[\omega^{lex}_{\mathcal{H}}, kHD_{\mathcal{H}}]$-$minimal$ hypertree decompositions are a subset of them, corresponding to the lexicographically minimal decompositions described above.

It is not difficult to show that, for general weighting functions, the computation of minimal decompositions is a difficult problem even if we consider just bounded hypertree decompositions [41]. We thus restrict our attention to simpler HWFs.

Let $\langle \mathbb{R}^+, \oplus, \min, \bot, +\infty \rangle$ be a *semiring*, that is, $\oplus$ is a commutative, associative, and closed binary operator, $\bot$ is the neuter element for $\oplus$ (e.g., 0 for $+$, 1 for $\times$, etc.) and the absorbing element for $\min$, and $\min$ distributes over $\oplus$.[4] Given a function g and a set of elements $S = \{p_1, ..., p_n\}$, we denote by $\bigoplus_{p_i \in S} g(p_i)$ the value $g(p_1) \oplus \ldots \oplus g(p_n)$.

Definition 4.2 *[41]* Let $\mathcal{H}$ be a hypergraph. Then, a *tree aggregation function* (short: TAF) is any hypertree weighting function of the form

$$\mathtt{F}^{\oplus,v,e}_{\mathcal{H}}(HD) = \bigoplus_{p \in N} \big(v_{\mathcal{H}}(p) \ \oplus \bigoplus_{(p,p') \in E} e_{\mathcal{H}}(p,p') \big),$$

associating an $\mathbb{R}^+$ value to the hypertree decomposition $HD = \langle (N,E), \chi, \lambda \rangle$, where $v_{\mathcal{H}} : N \mapsto \mathbb{R}^+$ and $e_{\mathcal{H}} : N \times N \mapsto \mathbb{R}^+$ are two polynomial functions evaluating vertices and edges of hypertrees, respectively. □

We next focus on a tree aggregation function that is useful for query optimization. We refer the interested reader to [41] for further examples and applications.

Given a query Q over a database **DB**, let $HD = \langle T, \chi, \lambda \rangle$ be a hypertree decomposition in normal form for $\mathcal{H}(Q)$. For any vertex p of T, let $E(p)$ denote the relational expression $E(p) = \bowtie_{h \in \lambda(p)} \prod_{\chi(p)} rel(h)$, i.e., the join of all relations in **DB** corresponding to hyperedges in $\lambda(p)$, suitably projected onto the variables in $\chi(p)$. Given also an incoming node p' of p in the decomposition HD, we define $v^*_{\mathcal{H}(Q)}(p)$ and $e^*_{\mathcal{H}(Q)}(p,p')$ as follows:

- $v^*_{\mathcal{H}(Q)}(p)$ is the estimate of the cost of evaluating the expression $E(p)$, and
- $e^*_{\mathcal{H}(Q)}(p,p')$ is the estimate of the cost of evaluating the semi-join $E(p) \ltimes E(p')$.

Let $cost_{\mathcal{H}(Q)}$ be the TAF $\mathtt{F}^{+,v^*,e^*}_{\mathcal{H}(Q)}(HD)$, determined by the above functions. Intuitively, $cost_{\mathcal{H}(Q)}$ weights the hypertree decompositions of the query hypergraph $\mathcal{H}(Q)$ in such a way that minimal hypertree decompositions correspond to "optimal" query evaluation plans for Q over **DB**. Note that any method for computing the estimates for the evaluation of relational algebra operations from the quantitative information on **DB** (relations sizes, attributes selectivity, and so on) may be employed for v^* and e^*. For instance, in our experiments described in the next section, we employ the standard techniques described in [12, 13].

[4]For the sake of presentation, we refer to $\min$ and hence to minimal hypertree decompositions. However, it is easy to see that all the results presented in this paper can be generalized easily to any semiring, possibly changing $\min$, $\mathbb{R}^+$, and $+\infty$.

Clearly, all these powerful weighting functions would be of limited practical applicability, without a polynomial time algorithm for the computation of minimal decompositions. Surprisingly, it turns out that, unlike the traditional (non-weighted) framework, working with normal-form hypertree decompositions, rather than with any kind of bounded-width hypertree decomposition, does matter. Indeed, computing such minimal hypertree decompositions with respect to any tree aggregation function is a tractable problem, while it has been proved that the problem is still NP-hard if the whole class of bounded-width hypertree decomposition is considered. A polynomial time algorithm for this problem, called minimal-k-decomp, is presented in [41].

4.2 Some Experiments

We implemented the algorithm cost-k-decomp, which computes a minimal decomposition with respect to the weighting function $cost_{\mathcal{H}(Q)}$ and the class of k-bounded normal-form hypertree decompositions. In this section, we report some results of an ongoing experimental activity on the application of cost-k-decomp to database query evaluation. A more detailed description and further experiments can be found in the full version of [41], currently available at the hypertree decomposition homepage [40]. Our aim here is just to show that Algorithm cost-k-decomp may significantly speed-up the evaluation of database queries having structural properties to be exploited. All benchmark queries are executed on the commercial DBMS *Oracle 8.i* by using either Oracle standard query execution method, or the following technique, based on minimal decompositions: the query plans are generated by the algorithm cost-k-decomp (with k ranging over $(2..5)$), by exploiting the information of the data available from Oracle; the plan execution is then enforced in the DBMS by supplying a suitable translation in terms of views and hints (*NO_MERGE, ORDERED*) to *Oracle 8.i*, which eventually executes the query by its engine, following the desired plan. In both methods, we do not allow indices on database relations, in order to focus just on the less-physical aspects of the optimization task.

We tested the methods with different kinds of queries by varying the hypertree width, the number of query atoms, and the number of variables. Here, we report only the experiments on a set of test queries: we consider again query Q_1 described in Example 3.2, as well as two modifications Q_2 and Q_3, such that Q_2 consists of 8 atoms and 9 distinct variables, and query Q_3 is made of 9 atoms, 12 distinct variables, and 4 output variables. All these queries have width 2. They are evaluated over synthetic data: For each query atom p, we first fix the *size* r_p of the corresponding relation, and we then exploit a random generator that materializes r_p data tuples, by choosing attribute values uniformly at random from a fixed set of possible values. All the experiments were performed on 1600MHz/256MB Pentium IV machine running *Windows XP Professional*. Time measurements for query evaluation in *Oracle 8.i* have been done by using the *SQL Scratchpad* utility. We considered different values for the parameter k. It is worthwhile noting that a higher value of k permits to consider a larger number of hypertree decompositions, and can therefore allow to generate a better plan; but it obviously causes a computational overhead due to a larger search space to be explored by cost-k-decomp. For the experiments reported in this paper, we chose $k = 3$, which seems empirically a good bound to be used in practice for queries with less than 10 atoms. Figure 5 shows the absolute execution times for Oracle and cost-k-decomp over a database of 1500 tuples. It can be observed that, on all considered queries, the evaluation of the query plans generated by our approach is significantly faster than the evaluation which exploits the internal query optimization module of *Oracle 8.i*.

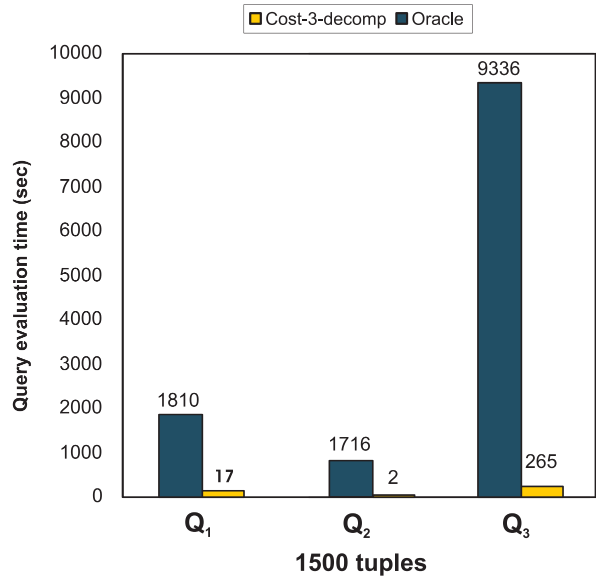

Figure 5: Evaluation time for test queries Q_1, Q_2, and Q_3.

5. CONCLUSION

We described the notion of hypertree width and some of its extensions, and we showed how they can be exploited for identifying and solving efficiently tractable classes of database queries.

Our ongoing work includes an integration of the optimization technique based on minimal decompositions with the query optimizer of the open source DBMS PostgreSQL, as well as a thorough experimentation activity with real queries and databases, loaded with non-random data.

Many interesting questions about structural decompositions are still open and deserve further research. For instance, we do not know if having bounded hypertree width is a necessary condition for a class of queries to be tractable. Moreover, for many real world applications with hundreds of hyperedges, we need good heuristics for computing generalized hypertree decompositions. We refer the interested reader to [22] for a recent graph-theoretic survey on hypertree decompositions, with further results and details on these related issues.

Acknowledgments

The author sincerely thanks Georg Gottlob and Nicola Leone, who worked with him on these issues since 1999, when they jointly defined the notion of hypertree decompositions. Moreover, he thanks Gianluigi Greco for his recent contribution to the weighted extension, and Alfredo Mazzitelli for his valuable work in designing and implementing the tools for experiments.

6. REFERENCES

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] S. Abiteboul and O.M. Duschka. Complexity of Answering Queries Using Materialized Views. In *Proc. of PODS'98*, pp. 254–263, Seattle, Washington, 1998.

[3] I. Adler, G. Gottlob, and M. Grohe. Hypertree-Width and Related Hypergraph Invariants. In *Proc. of EuroComb'05*, Berlin, 2005.

[4] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *J.ACM*, 30(3):479–513, 1983.

[5] P.A. Bernstein and N. Goodman. The power of natural semijoins. *SIAM J.Comput.*, 10(4):751–771, 1981.

[6] A.K. Chandra and P.M. Merlin. Optimal Implementation of Conjunctive Queries in relational Databases. In *Proc. of STOC'77*, pp.77–90, Boulder, Colorado, USA, 1977.

[7] C. Chekuri and A. Rajaraman. Conjunctive query containment revisited. *TCS*, 239(2):211–229, 2000.

[8] R. Dechter. Constraint Processing. Morgan Kaufmann, 2003.

[9] R. Fagin, A.O. Mendelzon, and J.D. Ullman. A simplified universal relation assumption and its properties. *ACM TODS*, 7(3):343–360, 1982.

[10] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *J.ACM*, 49(6):716–752, 2002.

[11] E.C. Freuder. A sufficient condition for backtrack-bounded search. *J.ACM*, 32(4):755–761, 1985.

[12] H. Garcia-Molina, J. Ullman, and J. Widom. *Database system implementation*. Prentice Hall, 2000.

[13] Y.E. Ioannidis. Query Optimization. *The Computer Science and Engineering Handbook*, pp. 1038–1057, 1997.

[14] N. Goodman and O. Shmueli. Tree queries: a simple class of relational queries. *ACM TODS*, 7(4):653–6773, 1982.

[15] G. Gottlob, N. Leone, and F. Scarcello. A comparison of structural CSP decomposition methods. *Artif. Intell.*, 124(2):243–282, 2000.

[16] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Advanced parallel algorithms for processing acyclic conjunctive queries, rules, and constraints. In *Proc. of the Conference on Software Engineering and Knowledge Engineering (SEKE'00)*, pp. 167–176, Chicago, USA, 2000.

[17] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *J.ACM*, 48(3):431–498, 2001.

[18] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions: A survey. In *In Proc. of MFCS'2001*, pp. 37–57, Marianske Lazne, Czech Republic, 2001.

[19] G. Gottlob, N. Leone, and F. Scarcello. Hypertree decompositions and tractable queries. *JCSS*, 64(3):579–627, 2002.

[20] G. Gottlob, N. Leone, and F. Scarcello. Computing LOGCFL Certificates. *TCS*, 270(1-2):761–777, 2002.

[21] G. Gottlob, N. Leone, and F. Scarcello. Robbers, marshals, and guards: game theoretic and logical characterizations of hypertree width. *JCSS*, 66(4):775–808, 2003.

[22] G. Gottlob, M. Grohe, N. Musliu, M. Samer, and F. Scarcello. Hypertree Decompositions: Structure, Algorithms, and Applications. In *Proc. of WG'05*, Metz, France, 2005.

[23] E. Grädel. On the Restraining Power of Guards. *J. Symb. Logic*, Vol. 64,pp. 1719–1742, 1999.

[24] M. Grohe. The Complexity of Homomorphism and Constraint Satisfaction Problems Seen from the Other Side. In *Proc. of FOCS'03*, pp. 552–561, Cambridge, MA, USA, 2003.

[25] M. Grohe, T. Schwentick, and L. Segoufin. When is the evaluation of conjunctive queries tractable? In *Proc. of STOC'01*, pp. 657–666, Heraklion, Crete, Greece, 2001.

[26] M. Gyssens, P.G. Jeavons, and D.A. Cohen. Decomposing constraint satisfaction problems using database techniques. *J. of Algorithms*, 66:57–89, 1994.

[27] D.S. Johnson, A Catalog of Complexity Classes, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pp. 67-161, 1990.

[28] P. G. Kolaitis and M. Y. Vardi. Conjunctive-query containment and constraint satisfaction. *JCSS*, 61(2):302–332, 2000.

[29] T. Korimort. Constraint Satisfaction Problems – Heuristic Decomposition. PhD thesis, Vienna University of Technology, April 2003.

[30] D. Maier. *The Theory of Relational Databases*. Computer Science Press, 1986.

[31] B.J. McMahan, G.Pan, P.Porter, and M.Y. Vardi. Projection Pushing Revisited. In *Proc of EDBT'04*, pp. 441–458, Heraklion, Crete, Greece, 2004.

[32] B. McMahan. Bucket Eliminiation and Hypertree Decompositions. Implementation report, DBAI, TU Vienna, 2004.

[33] C.H. Papadimitriou and M. Yannakakis. On the complexity of database queries. In *Proc. of PODS'97*, pp. 12–19, Tucson, Arizona, USA, 1997.

[34] O. Reingold. Undirected ST-Connectivity in Log-Space, manuscript, 2004, currently available at `http://www.wisdom.weizmann.ac.il/~reingold/publications/sl.ps`

[35] N. Robertson and P.D. Seymour. Graph minors ii. algorithmic aspects of tree width. *J. of Algoritms*, 7:309–322, 1986.

[36] W.L. Ruzzo. Tree-size bounded alternation. *JCSS*, 21:218–235, 1980.

[37] D. Saccà. Closures of database hypergraphs. *J.ACM*, 32(4):774–803, 1985.

[38] P.D. Seymour and R. Thomas. Graph Searching and a Min-Max Theorem for Tree-Width. *J.Comb. Theory B*, 58:22–33, 1993.

[39] F. Scarcello. Answering Queries: Tractable Cases and Optimizations. *Technical Report D2.R3*, Project "Integrazione, Warehousing e Mining di Sorgenti Eterogenee (MURST COFIN-2000)," 2001.

[40] Francesco Scarcello and Alfredo Mazzitelli. The hypertree decompositions homepage, since 2002. `http://wwwinfo.deis.unical.it/~frank/Hypertrees/`

[41] Francesco Scarcello, Gianluigi Greco, and Nicola Leone. Weighted Hypertree Decompositions and Optimal Query Plans. In *Proc. of PODS'04*, pp. 210-221, Paris, 2004.

[42] R.E. Tarjan, and M. Yannakakis. Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs. *SIAM J.Comput.*, 13(3):566-579, 1984.

[43] J. D. Ullman. *Principles of Database and Knowledge Base Systems*. Computer Science Press, 1989.

[44] A.N. Wilschut, J. Flokstra, and P. M.G. Apers. Parallel evaluation of multi-join queries. In *Proc. of SIGMOD'95*, San Jose, CA, USA, 1995.

[45] J. Van Benthem. Dynamic Bits and Pieces. *ILLC Research Report*, University of Amsterdam, 1997.

[46] M. Vardi. Complexity of relational query languages. In *Proc. of STOC'82*, pp. 137–146, San Francisco, CA, USA, 1982.

[47] M. Vardi. Constraint Satisfaction and Database Theory. *Tutorial at PODS'00*, Dallas, Texas, USA, 2000.

[48] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. of VLDB'81*, pp. 82–94, Cannes, France, 1981.

[49] C.T. Yu and M.Z. Özsoyoğlu. On determining tree-query membership of a distributed query. *Infor*, 22(3):261–282, 1984.