Hypertree Decompositions for Query Optimization

Lucantonio Ghionna, Luigi Granata, Gianluigi Greco
University of Calabria
I-87030 Rende, Italy
{ghionna,granata,ggreco}@mat.unical.it

Francesco Scarcello
University of Calabria
I-87030 Rende, Italy
scarcello@unical.it

Abstract

The database community has investigated many structure-driven methods, which guarantee that large classes of queries may be answered in (input-output) polynomial-time. However, despite their very nice computational properties, these methods are not currently used for practical applications, since they do not care about output variables and aggregate operators, and do not exploit quantitative information on the data. In fact, none of these methods has been implemented inside any available DBMS.

This paper aims at filling this gap between theory and practice. First, we define an extension of the notion of hypertree decomposition, which is currently the most powerful structural method. This new version, called query-oriented hypertree decomposition, is a suitable relaxation of hypertree decomposition designed for query optimization, and such that output variables and aggregate operators can be dealt with. Based on this notion, a hybrid optimizer is implemented, which can be used on top of available DBMSs to compute query plans. The prototype is also integrated into the well-known open-source DBMS PostgreSQL. Finally, we validate our proposal with a thorough experimental activity, conducted on PostgreSQL and on a commercial DBMS, which shows that both systems may significantly benefit from using hypertree decompositions for query optimization.

1. Introduction

Defining and implementing advanced optimization strategies are crucial and challenging issues in the design of efficient database management systems (DBMSs).

In fact, optimizers have to find good trade-offs between two contrasting factors. On the one hand, since the cost of evaluating a user query may be exponential in its length, optimizers are asked to compute the best *query plan*, i.e., the strategy for query evaluation that guarantees the best performances over all the possible strategies. On the other hand, since computing an optimal plan is an NP-hard problem, optimizers are forced to restrict the search space of query plans to very simple structures (e.g., left-deep trees), therefore avoiding an exhaustive and expensive search.

In fact, current DBMSs examine a number of alternative plans (but not all) and then choose the most promising one. The choice of these planners is determined by a cost model based on quantitative information collected on the data, e.g., sizes of relations and indices, and attribute selectivity, and are therefore called *quantitative* optimizers. While being generally very efficient, these optimizers cannot guarantee a bound on the query-answering time. In some cases, the approximated query plan is quite far from the ideal one, and query answering does not scale to instances with many join operations.

In order to face this problem and compute the optimal query plan that guarantees a polynomial bound on the answering time, a completely different approach has been investigated since many years in the database theory community. The approach is based on the exploitation of the *structural* properties of these queries, usually represented through their associated hypergraph.

As an example, consider the following query Q_5, extracted from the `TPC-H` specifications (*www.tpc.org/tpch*):

```
SELECT n_name,
       sum(l_extendedprice*(1-l_discount)) AS revenue
FROM customer, orders, lineitem,
     supplier, nation, region
WHERE c_custkey = o_custkey
      and l_orderkey = o_orderkey
      and l_suppkey = s_suppkey
      and c_nationkey = s_nationkey
      and s_nationkey = n_nationkey
      and n_regionkey = r_regionkey
      and r_name = '[REGION]'
      and o_orderdate >= date '[DATE]'
      and o_orderdate < date '[DATE]' + interval '1' year
GROUP BY n_name ORDER BY revenue desc;
```

Figure 1 shows the hypergraph associated with Q_5.

By exploiting the structure of the query hypergraph, it is possible to efficiently compute the optimal query plan for large classes of queries and, based on it, answer these queries with a polynomial-time upper bound guarantee. The

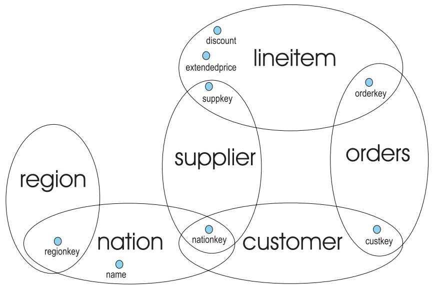


Figure 1. *Hypergraph* $\mathcal{H}(Q_5)$.

most known and studied of these classes consists of those queries whose hypergraphs are *acyclic*. From a well-known result of Yannakakis [12], all the answers to an acyclic query can be computed in time polynomial in the combined size of the input and of the output. This is the best possible result, because in general the answer of a query may contain an exponential number of tuples. Unfortunately, many queries arising in practice are not acyclic, as it is the case for query Q_5, whose associated hypergraph is cyclic.

However, it has been observed that many hypergraphs, though not acyclic, can be "made acyclic" with a limited effort. Indeed, many attempts have been made in the literature for extending the good results about acyclic conjunctive queries to relevant classes of *nearly acyclic* queries, and many *structural decomposition* methods have been proposed based on the notions of Biconnected Components [2], Tree Decompositions [9, 7, 1], Hinge Decompositions [8], and Hypertree Decompositions [5, 6, 10]. Unfortunately, despite their very nice computational properties, structural decomposition methods have not had any serious impact on the design of commercial DBMS optimizers, and the interest for these techniques remained only at a theoretical level, up to now. This is mainly due to the following reasons:

(1) Decomposition methods flatten in the query hypergraph all the "quantitative" aspects of data, such as selectivity and cardinality of relations, whose knowledge, in the practice of query evaluation, may dramatically speed-up the evaluation time.

(2) Decomposition methods do not generally take care of the output of the queries, or of aggregate operators. Indeed, they often assume to deal with Boolean conjunctive queries. While this is useful when studying their theoretical properties (since the basic reasoning tasks can be restated as decision problems), this assumption appears completely unrealistic for real life applications.

(3) No implementation of any of the decomposition methods proposed in the literature is yet available, possibly integrated in the optimization module of some DBMS. Hence, no testing activity has been conducted, and no evidence has been given on the practical usefulness of structural approaches to query answering.

A first step to fill the gap between theory and practice has been made in [11], where the Hypertree Decomposition method, which is in fact the broadest-known tractable generalization of hypergraph acyclicity, has been extended in order to combine this structural decomposition method with quantitative approaches, thereby facing the problem at point **(1)** above.

In this paper, we face the remaining problems, showing that commercial DBMSs may in fact take quite a profit from the exploitation of structural methods. In summary:

▷ We introduce a structural notion, called query-oriented hypertree decomposition, which is a suitable relaxation of hypertree decomposition specially designed for query optimization, and such that output variables and aggregate operators can be dealt with.

▷ We develop a prototype optimizer, based on query-oriented hypertree decompositions, that can be used on top of available DBMSs to compute query plans (as SQL views). We implemented our optimizer directly inside the well-known open-source DBMS PostgreSQL, thereby achieving a complete coupling between structural techniques and quantitative methods.

▷ We report results of an extensive experimentation of our techniques, both with the prototype optimizer used on the top of a leader DBMS system (which will be referred to by the fantasy name CommDB, for licence restrictions), and with the version of PostgreSQL equipped with our structure-driven optimizer. The experiments have been conducted both on standard `TPC-H` queries and on further syntectic datasets.

Organization. The rest of the paper is organized as follows. Section 2 and 3 give some preliminaries on queries, hypergraphs, and hypertree decompositions. In Section 4, we define the notion of query-oriented hypertree decomposition. In Section 5, we describe the optimizer based on this notion. In Section 6, we illustrate the results of our experimental activity, and in Section 7, we draw our conclusions.

2 Queries and Hypergraphs

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

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

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

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 $\mathbf{u_i}$. It is worthwhile noting that the head of the query has no corresponding hyperedge in $H(Q)$.

Moreover, observe that in general the correspondence between atoms of the query and hyperedges of the hypergraph is not one-to-one, because query atoms having exactly the same set of variables in their arguments give rise to one edge in $H(Q)$. However, a one-to-one correspondence can be obtained easily, by adding to each atom a fresh variable, which distinguish it from every other atom in the query. These variables are eventually removed before starting the actual query evaluation. Therefore, for the sake of presentation, in this paper we assume that, if some query atom misses such a peculiar variable, we implicitly perform the above procedure.

A query Q is acyclic if and only if its hypergraph $H(Q)$ is acyclic or, equivalently, if it 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 $edges(H(Q))$ of atoms occurring in the body of Q and whose edges are 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 $H(Q)$.

SQL Queries. In this paper, we are interested in SQL queries, rather than in the simpler conjunctive queries. For the sake of simplicity, we next assume SQL queries without nested statements, and with the equality comparison only. However, we shall discuss how to extend the proposed approach to nested queries and to the other built-in predicates.

Let Q be an SQL query on a database schema *DS*. Then, the conjunctive query $CQ(Q)$ associated with Q is the rule

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

where $r_1, \ldots, r_n$ are all the relations occurring in the FROM statement. The variables are defined by considering each set of attributes A involved together in some equality conditions in the WHERE statement of Q. Indeed, these attributes represent an equivalence class C_A. Correspondingly, $CQ(Q)$ contains a variable X_A, and this variable occurs in the list $\mathbf{u_i}$ of any atom $r_i(\mathbf{u_i})$ such that some attribute $a_j \in A$ is taken from the relation r_i. Moreover, we have a variable X_a also for each attribute a occurring in some statement of Q, but not occurring in any equality condition (e.g., attributes occurring only in the SELECT or in the GROUP BY statements). Then, X_a belongs to the list $\mathbf{u_i}$ of the atom $r_i(\mathbf{u_i})$ corresponding to the relation r_i of Q where a comes from. These are the only variables occurring in $CQ(Q)$. It follows that an atom $r_i(\mathbf{u_i})$ corresponding to a relation r_i in *DS* may have in $CQ(Q)$ a smaller arity than it has in **DB**, because only attributes involved in the query appears in $r_i(\mathbf{u_i})$, through their associated variable in $\mathbf{u_i}$. Finally, the variables $\mathbf{u}$ of $ans(\mathbf{u})$, denoted also as $out(Q)$, are all the variables associated with those attributes occurring in the SELECT statement or in the GROUP BY statement of Q.

The hypergraph $H(Q)$ of an SQL query Q is defined as the hypergraph $H(CQ(Q))$.

Example 1 Consider the TPC-H query Q_5 in the Introduction. Then, its associated conjunctive query $CQ(Q_5)$ is the following:

$$\begin{array}{l} ans(Name, ExtendedPrice, Discount) \leftarrow \\ customer(CustKey, NationKey) \;\wedge\; orders(OrdKey, NationKey) \\ \wedge\; lineitem(SuppKey, OrdKey, ExtendedPrice, Discount) \\ \wedge supplier(SuppKey, NationKey) \;\wedge\; region(RegionKey) \\ \wedge nation(Name, NationKey, RegionKey), \end{array}$$

and its associated hypergraph $\mathcal{H}(Q_5)$ is shown in Figure 1. Note that this hypergraph is not acyclic. □

3. Decompositions and Query Plans

In this section, we recall the basic notions about hypertree decompositions and discuss how they can be used to efficiently answer user queries.

3.1. 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 1 [5] A *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.

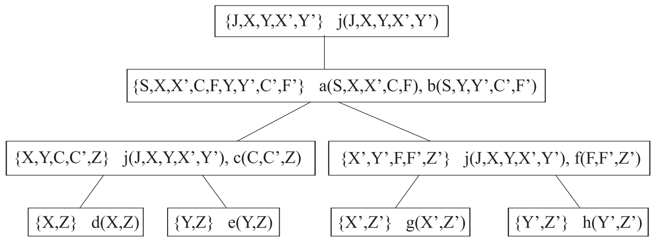
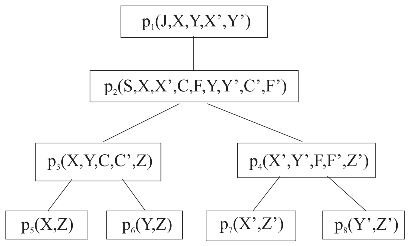

{J,X,Y,X',Y'} j(J,X,Y,X',Y')
{S,X,X',C,F,Y,Y',C',F'} a(S,X,X',C,F), b(S,Y,Y',C',F')
{X,Y,C,C',Z} j(J,X,Y,X',Y'), c(C,C',Z)
{X',Y',F,F',Z'} j(J,X,Y,X',Y'), f(F,F',Z')
{X,Z} d(X,Z)
{Y,Z} e(Y,Z)
{X',Z'} g(X',Z')
{Y',Z'} h(Y',Z')


Figure 2. Left: A hypertree decomposition of $\mathcal{H}(Q_0)$. Right: The tree $\mathcal{JT}_0$ computed for query Q'_0.

3. For each vertex $p \in vertices(T)$, variables in the χ labeling should belong to edges in the λ labeling, that is, $\chi(p) \subseteq var(\lambda(p))$.
4. Special Descendant Condition: for each $p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

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

If we drop the Special Descendant Condition, we get the notion of *Generalized Hypertree Decomposition* [6], which is a more general notion with all the nice properties of hypertree decompositions, but not known to be tractable. Note also that the notion of hypertree width is a true generalization of acyclicity, as the acyclic hypergraphs are precisely those hypergraphs having hypertree width one.

Example 2 Consider the following conjunctive query Q_0:

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

Let $\mathcal{H}(Q_0)$ be the hypergraph associated with Q_0. Then, $\mathcal{H}(Q_0)$ is cyclic, since $hw(\mathcal{H}(Q_0)) = 2$ holds. A hypertree decomposition of width 2 is shown on the left in Figure 2. For each vertex p of the decomposition tree, the set of variables $\chi(p)$ and the atoms in $\lambda(p)$ are reported. □

3.2 Answering Queries through Decompositions

We next describe how a hypertree decomposition of $\mathcal{H}(Q)$ can be used as an effective (logical) query plan for the query Q. More precisely, we can evaluate Q by the following two-steps approach: (S_1) *Build a hypertree decomposition HD for $\mathcal{H}(Q)$*, and (S_2) *Exploiting HD, compute the answers to Q*. Both the steps above are easy when $hw(\mathcal{H}(Q))$ is bounded by a fixed constant k.

In particular, the search problem (S_1) of computing a k-bounded hypertree decomposition belongs to $\mathrm{L}^{\mathrm{LOGCFL}}$, the functional version of LOGCFL [5].[1] As for step S_2, we can answer Q by using *HD* as follows: (S'_2) 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 [5]. (S''_2) Answer Q', and hence Q, by using Yannakakis's algorithm for acyclic queries [12].

[1]Hence, the problem is highly parallelizable, since LOGCFL is not only included in polynomial time, but also in NC^2.

Example 3 Consider again the query Q_0 of Example 2. Figure 2 shows, on the right, the tree $\mathcal{JT}_0$ obtained after Step S'_2 above, when applied to the hypertree decomposition in the left of Figure 2. 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 of the original decomposition), and from the subsequent projection onto the variables X, Y, C, C', and Z (belonging to the χ label of that vertex). By Condition 2 in Definition 1, $\mathcal{JT}_0$ satisfies the connectedness condition. By Condition 1, it follows easily that the conjunction of all atoms labeling this tree is a query equivalent to Q_0. Moreover, it is acyclic, as $\mathcal{JT}_0$ is one of its join trees. □

Step S'_2 is feasible in polynomial time. Indeed, building the acyclic query Q' is feasible in $O(m|r_{max}|^k)$ time, where m is the number of vertices of T, and r_{max} is the relation of the given database **DB** having the largest size. Also, step S''_2 can be efficiently carried out. Indeed, for Boolean conjunctive queries, acyclic instances can be evaluated by processing any of their join trees bottom-up by performing upward semijoins, as in Yannakakis's algorithm: For each vertex p of the decomposition tree, the join of p with each one of its children is computed, and the result is projected onto p's variables. When the root is eventually reached, the answer is *Yes* if the filtered relation at the root is not empty; otherwise, the answer is *No*. Since no intermediate relations are computed, Yannakakis's algorithm takes only $O((m-1)|r_{max}|^k \log |r_{max}|)$.

For a non-Boolean query Q, Yannakakis's algorithm can be adapted to work in time polynomial in the combined size of the input and of the output. The algorithm consists of three steps: (i) the bottom-up evaluation described above, (ii) a specular top down evaluation, and (iii) a final bottom-up evaluation where, instead of taking semijoins as in the previous steps, we compute joins, and project onto the variables in the current vertex plus the output variables in $out(Q)$ that comes from the subtree rooted at that vertex.

4. Hypertree Decompositions for Queries

In this section, we describe a new extension of the notion of hypertree decomposition specifically designed for the query evaluation purposes described in the previous sections. In particular, we show how to compute decomposition trees whose associated query plans guarantee a polynomial time upper bound with a single bottom-up evaluation phase, and we improve step S_2' in Section 3.2, by avoiding the naive construction of the acyclic equivalent instance Q'.

Definition 2 A *query-oriented hypertree decomposition* (short: *q-hypertree decomposition*) of a conjunctive query Q is a hypertree $HD = \langle T, \chi, \lambda \rangle$ of the hypergraph $\mathcal{H}(Q)$ which satisfies the following conditions:

1. For each edge $h \in edges(\mathcal{H})$, there exists $p \in vertices(T)$ such that $h \subseteq \chi(p)$.
2. There exists $p \in vertices(T)$ such that $out(Q) \subseteq \chi(p)$.
3. 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. □

In fact, a q-hypertree decomposition *HD* of a (non-Boolean) query Q is a (generalized) hypertree decomposition of $\mathcal{H}(Q)$, except for the following important features:

a) The decomposition *HD* is forced to have a vertex p that covers all the output variables of Q. Therefore, if we root the decomposition tree at p, the query may be answered by performing only one bottom-up evaluation of the decomposition tree. Indeed, at the end of this procedure, the relation at vertex p, projected onto $out(Q)$, directly provides the answer of Q. Thus, steps (ii) and (iii) described above are no longer needed.

b) Condition 3 of Definition 1 is not required here, that is, some variables in the χ labeling may be not covered by atoms in the λ labeling. This relaxation allows us to perform an important optimization of decomposition-based query plans: we may save join operations at a vertex p, as long as there are variables in $\chi(p)$ whose sets of possible tuples are bounded by atoms in some child of p.

To evaluate a conjunctive query Q on a database **DB**, given a q-hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ for it, we use an adaptation of the Yannakakis's algorithm described next.

A *q-hypertree evaluator* is a procedure that, given as its input $\langle HD, Q, \mathbf{DB} \rangle$, performs the following steps: (P') For each vertex $p \in vertices(T)$, compute the join of the relations associated to atoms in $\lambda(p)$, and project the result onto the variables in $\chi(p)$; (P'') Following a topological order of T, evaluate each vertex p of the decomposition tree by taking the join of the relation at p with each one of its children and projecting the result onto $\chi(p)$'s variables; (P''') Output the projection of the relation at the root of T onto the variables in $out(Q)$.

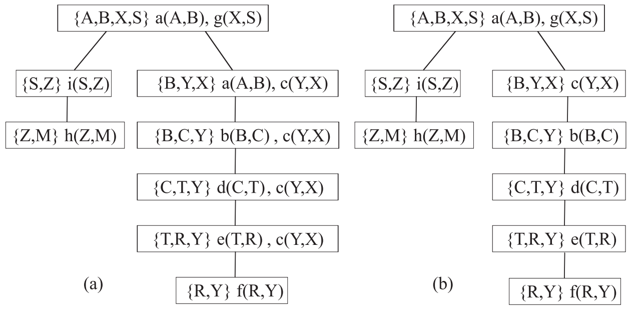


Figure 3. HD_1 (a) and HD_1' (b), in Example 4.

Example 4 Consider the following SQL query Q_1:

```
SELECT A, S, max(X) FROM a,b,c,d,e,f,g,h,i
WHERE a.B=b.B and b.C=d.C and d.T=e.T and e.R=f.R and
      f.Y=c.Y and g.X=c.X and g.S=i.S and h.Z=i.Z
GROUP BY A,S
```

Figure 3 shows two q-hypertree decomposition HD_1 and HD_1' of $CQ(Q_1)$. Note that both of them have width 2, though $hw(\mathcal{H}(Q_1)) = 1$, as the query hypergraph is acyclic. In fact this is the best we can do, because we want to start from a hypertree decomposition like HD_1 (to keep its nice properties), but we have to satisfy also Condition 2 in Definition 2. Intuitively, this condition may introduce cycles in the hypergraph, through the connections involving output variables. This is what we are going to pay, to avoid the additional top-down and its subsequent bottom-up evaluation of the decomposition tree for non-Boolean queries (steps (ii) and (iii) in Section 3.2).

Looking at the simpler hypertree HD_1' in Figure 3, we can see how the feature *(b)* of q-hypertree decompositions works, by allowing the optimizer to get better plans. Compare the right subtrees of the roots of HD_1 and HD_1': The atom a in the first vertex, and the atom c in the subsequent three vertices do not occur in the λ labels of their corresponding vertices in HD_1'. Hence, the preliminary evaluation Step P' requires for HD_1' only one join operation (at the root) instead of 5, as it is for the hypertree HD_1. □

Note that, in general, there are different ways of evaluating Q on **DB** w.r.t. HD, depending on the choice of the topological order of the decomposition tree. Moreover, observe that, at Step P'', we take joins instead of semi-joins, because some variables at a vertex p may be covered by its children, rather than by its own atoms. In fact, differently from the corresponding Step S_2' described in Section 3, the output of Step P' is not an acyclic query equivalent to Q. It follows that, in principle, the size of such intermediate joins—and hence the cost of P''—may increase exponentially, even if the width is bounded by the constant k.

```
Input: A conjunctive query $Q$.
Output: A good q-hypertree decomposition $HD = \langle T, \chi, \lambda \rangle$ of $Q$.
 Compute a "minimal" NF hypertree decomposition $HD$ of $\mathcal{H}(Q)$ such that
     Condition 2 in Definition 2 is satisfied, the width of $HD$ is at most $k$,
     and, for the root $r$ of $HD$, $out(Q) \subseteq \chi(r)$ holds;
 If no such a decomposition exists,
     Output "Failure";
 else
     Execute Optimize $(HD, r)$;
     Output $HD$.
------------------------------------------------------------------
Procedure Optimize(VAR $HD, p \in vert(HD)$)
 For each hyperedge $a \in \lambda(p)$ Do
   If $\exists q \in child(p)$ and $b \in \lambda(q)$ such that $a \cap \chi(p) \subseteq b \cap \chi(q)$
     Then Remove $a$ from $\lambda(p)$;
 For each $q \in child(p)$ Do Optimize($HD$, $q$).
```

Figure 4. ALGORITHM q-HypertreeDecomp.

Thus, a key issue is the computation of a q-hypertree decomposition that can be evaluated efficiently, that is, such that the size of intermediate relations cannot blow-up, and the whole procedure has a polynomial-time upper bound in the combined size of the input and of the output.

Definition 3 A q-hypertree decomposition *HD* of a conjunctive query Q is said *good*, if there is a q-hypertree evaluator that, given as its input $\langle HD, Q, \mathbf{DB} \rangle$, takes polynomial-time in $\|Q\| + \|\mathbf{DB}\| + \|Q(\mathbf{DB})\|$ to compute the answer of Q on **DB**. $\square$

Putting it all together, given an SQL query Q on a database **DB**, our algorithm for evaluating Q by exploiting hypertree decompositions consists of the following steps: (1) we compute the conjunctive query $CQ(Q)$, as described in Section 2; (2) we compute a good q-hypertree decomposition *HD* of $CQ(Q)$; (3) we compute the answer of $CQ(Q)$ on **DB** by means of a suitable q-hypertree evaluation of Q on **DB** w.r.t. *HD*; and (4) we evaluate possible aggregate operators (including group-by computations) working on the answer of $CQ(Q)$.

Step (4) is implemented in any standard way, as, by definition, variables in $out(Q)$ include all variables involved in such aggregate operators. It remains to describe how to implement Step (2) and Step (3), that will be the subjects of the following sections.

4.1. Computing good q-Hypertree Decompositions

An algorithm that given a conjunctive query Q returns a good q-hypertree decomposition of Q is reported in Figure 4. This algorithm depends on a fixed constant k, which bounds the width of the decompositions to be considered (typically, $k = 4$ is enough for database queries). Firstly, the algorithm computes a width-k hypertree decomposition *HD* of $\mathcal{H}(Q)$ that satisfies Condition 2 in Definition 2, if any. Note that in general there is an exponential number of hypertree decompositions of a hypergraph, leading to different query-evaluation performances. Therefore, we have implemented an algorithm (based on the ideas in [11]) that evaluates different hypertrees according to a cost model for physical operators. Specifically, the cost model is based on a number of estimates about the operations on the input database, computed with standard techniques described in [3, 4]. Then, rather than looking at all possible hypertree decompositions, we focus on those hypertree decompositions in having the minimum associated cost, as they correspond to "optimal" query evaluation plans for Q over **DB**. Notably, as shown in [11], computing such a best query plan can be done efficiently, if we consider normal form (NF) decompositions (again, it is feasible in $\mathrm{L}^{\mathrm{LOGCFL}}$).

After the computation of this hypertree decomposition of $\mathcal{H}(Q)$, we execute the *Procedure Optimize*, which simplifies the hypertree *HD* by removing hyperedges from the λ labels, in order to get a more efficient query plan for the query, without giving up the guarantee on the polynomial-time upper bound on the evaluation of $CQ(Q)$.

Theorem 1 *Let $k \geq 1$ be a fixed constant. Given a conjunctive query Q, Algorithm q-HypertreeDecomp in Figure 4 runs in polynomial time, and outputs a good q-hypertree decomposition of Q, or "Failure". The latter output is returned if and only if there is no hypertree decomposition of $\mathcal{H}(Q)$ having width at most k that satisfies Condition 2 in Definition 2.*

For space limitations, we just give a rough idea on how the procedure *Optimize* works, guided by the example in Figure 3. In HD_1, consider the right child p_r of the root, where atom a is removed from $\lambda(p_r)$. In principle, this atom is useful here, because it provides a bound on the possible values for variable B coming from the bottom of the tree (but non for variable A, that does not belong to $\chi(p_r)$). However, B is also contained in the atom b in the child of this vertex. Then, we may think of an equivalent hypertree decomposition where a is replaced by b in this vertex. Now, it is clear that, in any evaluation of the decomposition tree, there is no sense in computing the join operation between the two bs in these adjacent vertices. Indeed, the result of such an operation would be exactly the relation corresponding to the atom b (possibly already filtered by previous join operations executed in the bottom-up evaluation). Thus, a may be replaced by b, and b is useless, whence we can just delete a from that vertex, as far as the polynomial-time upper bound is concerned. Intuitively the bounding effect on the variables in $a \cap \chi(p_r)$ (in this case, only B) is guaranteed by the atom b in the child of p_r.

It is worthwhile noting that, in more complex examples where such a simplified atom has many children, the topological order used in the evaluation of the join tree should take care of the children used for the simplification, that have to be joined with their parent before the other siblings. Otherwise, intermediate relations with exponentially many tuples can be temporary computed.

5. System Architecture

The structural approach described in this article has been implemented and integrated into a prototype system, which can be used either as a stand-alone application, or as a module plugged-in the open source PostgreSQL DBMS.

In the former case, the system rewrites the user query in a set of SQL views (based on its structural decomposition), which can be evaluated on top of any DBMS — in this case, possible statistical information about data should be provided explicitly by the user, and the DBMS optimizer is responsible for the translation of the logical query plan into the physical one.

In the latter case, instead, the decomposition algorithms have been tightly integrated in the PostgreSQL optimizer, so that *(1)* the optimization process is completely transparent to the user, and *(2)* additional information about data can automatically be exploited, to find a good query plan.

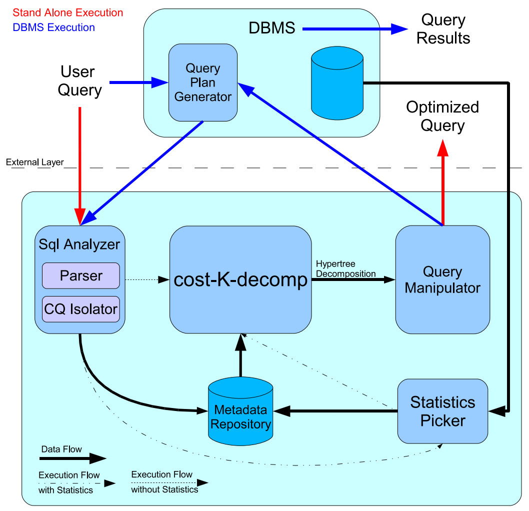


Figure 5. System architecture.

Figure 5 illustrates a functional view of the system architecture. Basically, it is formed by the following modules:

Sql Analyzer. It is responsible for preprocessing the query. First, the *Sql Parser* verifies its syntactical correctness, and then the *Conjunctive Query Isolator* computes the associated query hypergraph, which is the basis for the structural optimization.

Statistics Picker. This module is responsible for collecting the statistics about the relations involved in the query. When the system is coupled with PostgreSQL, these statistics are directly accessible from the DBMS optimizer. Otherwise, i.e., in the stand-alone usage, the user may optionally indicate the cardinality of the involved relations, and the selectivity of their attributes.

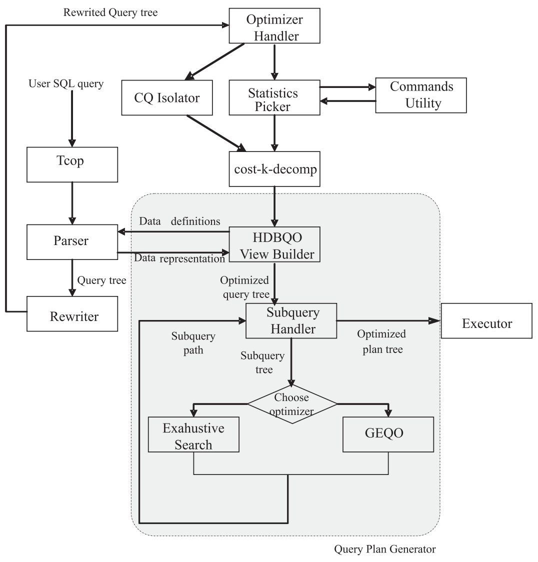


Figure 6. Integration in PostgreSQL.

`cost-k-decomp`. This is the fundamental module of our architecture. It picks from the *Metadata Repository* statistics about data, together with the query hypergraph generated by the *Sql Analyzer*, and produces a q-hypertree decomposition according to the ideas described in Section 4.

Query Manipulator. In the case of the direct integration in PostgreSQL, the module produces a suitable data structure which is used to implement the bottom-up strategy discussed in the paper. Otherwise, i.e., in the stand-alone usage, the query plan is returned to the user in terms of a rewritten SQL query, which can be evaluated on top of any DBMS (possibly, disabling its internal optimizer).

5.1. Tight Coupling with PostgreSQL

Since our system is fairly the first attempt to integrate structural optimizations into the core of standard (quantitative) query optimizers, it is relevant to discuss in more details how this coupling has been practically achieved.

In PostgreSQL, queries are processed as follows (see Figure 6). The *Tcop* module intercepts user requests and forwards them to *Parser*, which performs the syntactical and semantical analysis and produces a structured representation called *Query tree*. The *Rewriter* elaborates the query tree for optimization purposes and sends the result to the *Optimizer handler*, which is in charge of selecting the way the optimization has to be carried out. In the current implementation of PostgreSQL, two distinct and alternative optimizers are available: one performing an *exhaustive search*, and another using a genetic algorithm (*GEQO*).

To make the coupling possible, we modified the *Optimizer handler*, so that the control is no longer directly passed to either of the optimizers. Rather, both the *CQ Isolator* and the *Statistics picker* are invoked. In particular, statistics about data are now collected from the PostgreSQL *Commands Utility*.

Then, the *Query Plan Generator* is invoked: first, the *HDBQO ViewsBuilder* is responsible for building an optimized query tree based on the q-hypertree decomposition produced by cost-k-decomp, expressed in terms of nested SQL subqueries; then, each subquery is processed by the *HDBQO SubQueryHandler*, which is in charge of its execution on top of the built-in PostgreSQL optimizer.

6. Experimental Results

Compared Methods. Query plans for a number of queries were generated and their performances were compared with those produced by CommDB and PostgreSQL. Specifically, we used CommDB to evaluate the performances of our stand-alone architecture, and PostgreSQL to assess the advantages of a direct coupling inside a DBMS.

Benchmark Queries and Data. Tests included different kinds of queries. For each of them, we varied the hypertree width, the number of involved relations, their cardinality, and the selectivity of their attributes. Our attention was primarily devoted to test the optimization strategies on the standard *TPC-H* benchmarks. In addition, we considered the following kinds of query:

- *Acyclic Queries.* These are acyclic queries whose hypergraph has the form of a line: $q(\mathbf{y}) \leftarrow p_1(\mathbf{x_1}), p_2(\mathbf{x_2}), ..., p_n(\mathbf{x_n})$, where $\mathbf{x_i}$ denotes the set of variables occurring in the query atom p_i. We considered queries such that $\mathbf{x_i} \cap \mathbf{x_{i+1}} \neq \emptyset$, for any $1 \leq i < n$, and $\mathbf{x_i} \cap \mathbf{x_j} = \emptyset$, for any $i \notin \{j+1, j-1\}$. We experimented with queries whose length n ranges from 2 to 10.

- *Chain Queries.* They are the simplest cyclic variation of the above lines, where the first and the last query atoms have a variable in common ($\mathbf{x_1} \cap \mathbf{x_n} \neq \emptyset$).

TPC-H queries were evaluated over data generated by the *dbgen* tool provided by TPC. For the other kinds of queries, synthetic data were used, which has been generated randomly by using an uniform distribution over a fixed range of values, and setting the desired values for the cardinality of each relation and the selectivity of each attribute.

All experiments were performed on a 2,66Ghz Pentium 4 laptop, equipped with 512 Mb of ram and a 5400 rpm hard disk, running Windows XP Professional.

6.1. Experimenting with CommDB

We executed TPC-H queries on CommDB, both with and without its standard optimizer, to execute queries according to the q-hypertree decomposition method (*q-HD*). In the latter case, we report the *total* execution time, i.e., the summation of the stand-alone optimization time and of the CommDB evaluation time.

Figure 8 shows results for two TPC-H queries, Q_5 and Q_8, having hypertree width 2 (that is, two cyclic queries). For these queries, the use of statistics for *q-HD* had no impact on the computed query plans, which means that, for these queries and according to our cost model, exploiting the structure was estimated more important than exploiting the information on the database. Thus, the results shown in this figure for *q-HD*, are in fact the same as those obtained without any information on the data, that is, by using *q-HD* as a purely structural method.

The picture is completely different for CommDB: standard execution time, when it is not allowed to use statistics on the data, dramatically grows with the database size, and the evaluation quickly becomes infeasible. Even when CommDB is allowed to exploit statistics on the data, the use of q-hypertree decomposition (*q-HD*) improves the query evaluation performances. Of course, such good results are closely related with the peculiarities of queries Q_5 and Q_8,

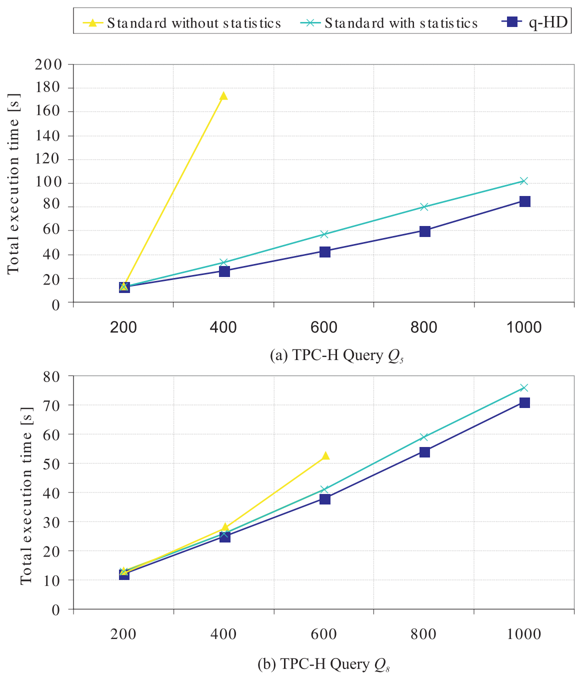


Figure 8. Execution time on TPCH-Queries: CommDB vs cost-k-decomp. Execution Times with database size ranging from 200mb to 1000mb: (a) Query Q_5. (b) Query Q_8.

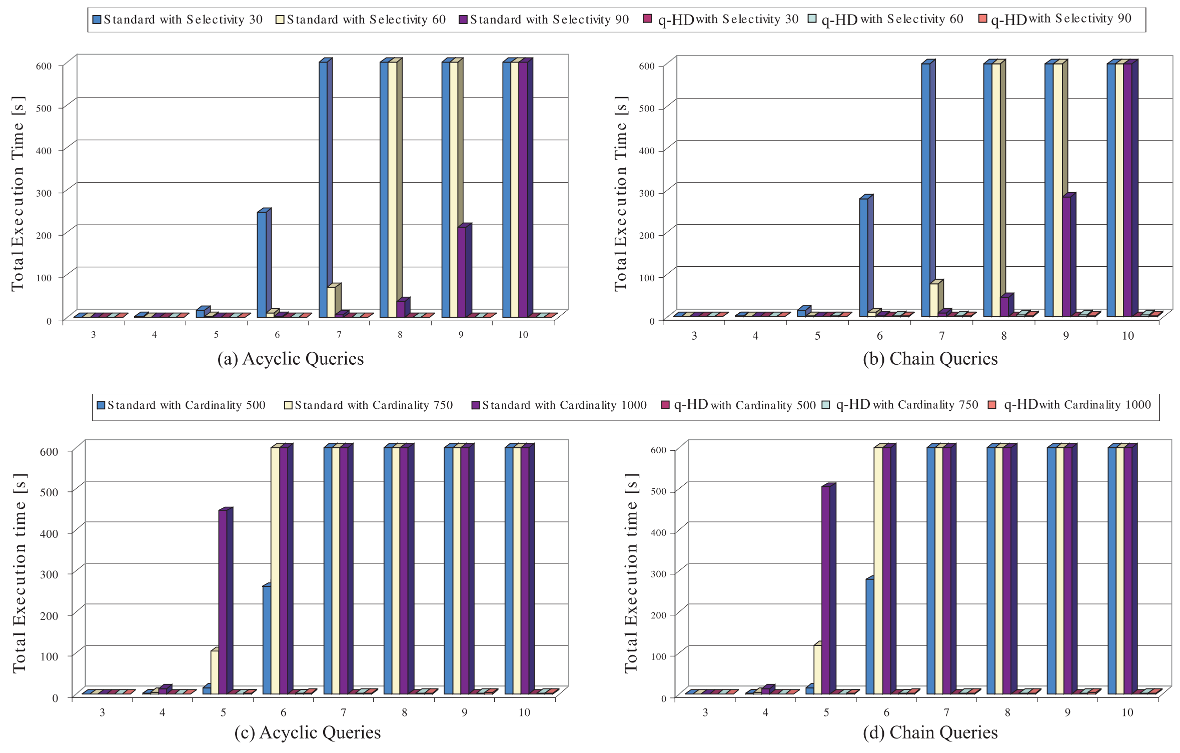


Figure 7. Execution times w.r.t. number of body atoms: (a) and (b) for attributes selectivity values 30, 60 and 90 — cardinality 500. (c) and (d) for databases of 500, 750 and 1000 tuples — selectivity 30.

which are cyclic and involve many join operations. In fact, on queries where the structure plays instead a marginal role, *q-HD* used as a purely structural method is generally not competitive with CommDB exploiting statistics.

Anyway, it is relevant to notice that gathering statistics is expensive (for 1GB, 800 seconds are needed) while building a structure-based query plan takes an average time of 1.5 seconds—not affected by the database size, and usually leads to good performances. Hence, besides the cases of long or cyclic queries like Q_5 and Q_8, *q-HD* could be very useful in all those applications where statistics are not available (or not yet).

Experimental results for Acyclic and Chain queries further confirmed the above intuitions. The results are depicted in Figure 7, which reports query execution times, varying the number of body atoms and with different values for cardinality of relations and selectivity of attributes—where CommDB is able to use statistics on the data.

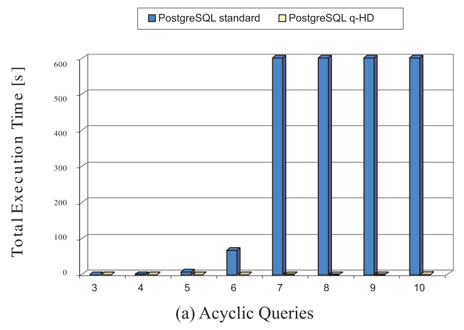
Notice that, for queries with 10 atoms in the body, CommDB executions do not terminate after more than 10 minutes while the *q-HD* driven executions take just a few seconds. This evidences that, when the size of the query grows (especially if combined with its its intricacy), current DBMS optimizers often fail in finding good query plans and structural decomposition methods can significantly improve their performances.

6.2. Experimenting with PostgreSQL

All the experiments described above have been repeated with our prototype directly implemented inside PostgreSQL 8.3. The relative gain turned out to be even higher than for the tests with CommDB, since in this scenario query evaluation can benefit of both the structural methods and the quantitative statistics about data.

As an example, we reported in Figure 9 the execution times on Acyclic and Chain queries, for a synthetic database where each relation contains 450 tuples and whose attributes have selectivity 60 (by lowering the selectivity, the gain of the structural approach is even more evident).

The reader may notice that the basic PostgreSQL optimizer performs quite poorly when compared with CommDB, since evaluating an acyclic query with 6 body atoms takes about 80 second in this scenario, while it is feasible in a few seconds by CommDB (cf. Figure 7.(a)). However, when the structural methods are integrated in PostgreSQL (*q-HD*), we get some quite surprising results, since its query evaluation nicely scales up to 10 body atoms, while CommDB (without the use of structural optimizations) does not terminate after 10 minutes, even for 8 atoms only.



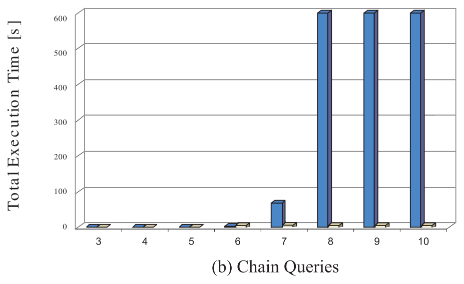


Figure 9. PostgreSQL: Execution times w.r.t. number of body atoms — selectivity 60, cardinality 450.

Finally, in the last set of experiments, we explore the benefits of using the procedure *Optimize* in Figure 4, and hence of exploiting feature (b) of q-hypertree decompositions. The results for chain queries are shown in Figure 10 (over the same dataset as in Figure 9).

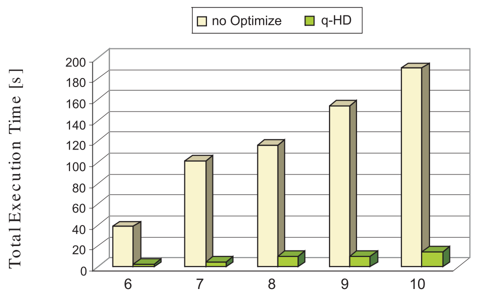


Figure 10. Impact of Procedure *Optimize*.

7. Conclusion

We have introduced the query-oriented hypertree decomposition, which is variation of the hypertree decomposition suitably designed for query optimization. Queries having good q-hypertree decompositions can be evaluated efficiently, with a polynomial time upper bound (in the combined size of the input and the output). Based on this notion, we implemented a hybrid optimizer, which exploits both structural and quantitative information, because q-hypertree decompositions are in fact computed starting from minimal hypertree decompositions [11].

The prototype optimizer can be used either on top of available DBMSs to compute query plans (expressed as SQL views), or as an internal module of the open-source DBMS PostgreSQL. Results of a throughout experimentation showed that the use of such an hybrid optimization, where the query structure is taken into account, is very beneficial, in particular for queries involving many join operations, or when statistics are not (yet) available.

Many interesting issues are left for future work. For instance, a more efficient computation of aggregate predicates (they can be included in the cost model), dealing with any kind of nested queries, and implementing the hybrid optimizer for other open-source DBMSs.

Acknowledgments

Partially supported by ICAR-CNR within the Project *High quality software* (ICT.P05.002), and by the Public-Private Laboratory *Open know tech*, funded by MIUR.

References

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

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

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

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

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

[6] 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.

[7] 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.

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

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

[10] Francesco Scarcello and Alfredo Mazzitelli. The hypertree decompositions homepage, since 2002: `http://www.deis.unical.it/scarcello/Hypertrees/`

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

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