D2I

Integrazione, Warehousing e Mining di sorgenti eterogenee

Programma di ricerca (cofinanziato dal MURST, esercizio 2000)

Answering Queries:
Tractable Cases and Optimizations

FRANCESCO SCARCELLO

D2.R3 **30 aprile 2001**

Sommario

Answering queries is computationally very expensive, and many approaches have been proposed in the literature to face this fundamental problem. Some of them are based on optimization modules that exploit quantitative information on the database instance, while other approaches exploit structural properties of the query hypergraph. For instance, acyclic queries can be answered in polynomial time, and also query containment is efficiently decidable for acyclic queries.

In this report, we review both quantitative and structural methods for optimizing query answering and identifying tractable classes of queries. Moreover, we provide a formal comparison of structural methods.

Tema	Tema 2: Estrazione di informazioni distribuite sul WWW
Codice	D2.R3
Data	30 aprile 2001
Tipo di prodotto	Rapporto tecnico
Numero di pagine	22
Unità responsabile	CS
Unità coinvolte	CS
Autore da contattare	Francesco Scarcello Dipartimento di Elettronica, Informatica e Sistemistica Università della Calabria Via P. Bucci 41/C, 87030 Rende (CS), Italy scarcello@deis.unical.it

Answering Queries: Tractable Cases and Optimizations

Francesco Scarcello

30 aprile 2001

Abstract

Answering queries is computationally very expensive, and many approaches have been proposed in the literature to face this fundamental problem. Some of them are based on optimization modules that exploit quantitative information on the database instance, while other approaches exploit structural properties of the query hypergraph. For instance, acyclic queries can be answered in polynomial time, and also query containment is efficiently decidable for acyclic queries.

In this report, we review both quantitative and structural methods for optimizing query answering and identifying tractable classes of queries. Moreover, we provide a formal comparison of structural methods.

1 Introduction

The efficiency of query answering critically depends on both the database design and the query-execution strategy. In this paper, we assume all the designing database issues are solved and we will deal only with the latter aspect. That is, given a database **db** and a query Q, find the best possible way to compute the answer of Q on **db**, denoted by $Q(\mathbf{db})$. Note that answering a query is a very difficult task, as the problem of deciding whether $Q(\mathbf{db})$ is not empty is NP-complete [9, 23], even if Q is a conjunctive query. Moreover, even computing a single tuple in the result of Q is NP-hard, and computing all the output relation is exponential, in general. All the available algorithms for solving these problems require exponential time, in the size of Q and **db**.

However, given a query, there are many execution plans that a DBMS can choose to compute its answer. All these plans are equivalent in terms of output relation. However, the differences among their execution times can be several orders of magnitude large. Therefore, all DBMS have a module, called *query optimizer*, that examines a number of alternative plans for answering a given query and then choose the best one, according to some cost model. Actually, the optimizer performs two main tasks. First, it *rewrites* the query into an equivalent one that can be easily answered, by pushing selections and projections, replacing views by their definitions, flattening out nested queries and so on. Then, it starts the *planning* phase, where it selects a query execution plan, taking into account the actual query costs for the specific database and DBMS. We call *quantitative methods* all these query planners based on quantitative information about sizes of relations, indices, and so on.

Note that computing an optimal plan is an NP-hard problem and hence it is unlikely to find an efficient algorithm for selecting the best plans. Indeed, all the commercial DBMS just compute approximations optimal query plans.

A completely different approach to query answering is based on the identification of some structural properties of queries, rather than on quantitative information about data values. Exploiting such properties is possible to answer large classes of queries in polynomial time. One of the most important and best studied class of tractable queries is the class of *acyclic queries* [7, 11, 12, 17, 18, 24, 42, 48, 50, 54, 55, 68, 70]. It was shown that acyclic queries coincide with the *tree queries* [4], see also [1, 47, 64]. The latter are queries which are representable by a *join tree* (or *join forest*) (see Section 4.2, for a formal definition).

By well-known results of Yannakakis [68], 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 size of the answer may be exponentially greater than the size of the input. 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).

It is worthwhile noting that a number of very important problems, which are equivalent (via logspace transformations) to evaluating Boolean Conjunctive queries, are efficiently solvable in the acyclic case. In particular, given two queries Q_1 and Q_2, checking whether Q_1 is contained in Q_2 is feasible in polynomial time, and actually highly parallelizable, if Q_1 is acyclic [26]. This is a fundamental problem for answering queries using views, in the data warehouse framework.

The good results about acyclic conjunctive queries extend to very relevant classes of *nearly acyclic* queries, such as queries whose associated primal graph has *bounded treewidth* [53], a *cutset* [13] of bounded size, or a bounded *degree of cyclicity* [33]. Thus, a number of query answering techniques based on such structural properties have been proposed. Conceptually, all of them may be viewed as composed of two phases:

1. trasform the given query into an equivalent acyclic query, and

2. solve this acyclic query.

We call these techniques *decomposition methods*, because they solve queries by decomposing them into acyclic ones.

In this paper, we deal with both quantitative and structural approaches to query answering. In particular, we will focus on conjunctive queries, the most basic, most important, and best-studied type of database queries.

For quantitative methods, in Section 3, we give a brief description of each kind of approach, and provide references to the literature for further details, implementation issues, and comparisons. Then, we consider structural methods. In Section 4, we review acyclic queries and their desirable computational properties. In Section 5 we describe the most important approaches to query evaluation (and hence query containment) based on the decomposition of cyclic queries into acyclic queries. Finally, in Section 6, we compare these methods. We draw our conclusions and some directions for future work in Section 7.

2 Preliminaries and Basic Concepts

For a background on databases, conjunctive queries, etc., see [1, 47, 64]. We define only the most relevant concepts here.

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

In this paper, we will adopt the standard convention [1, 64] of identifying a relational database instance with a logical theory consisting of ground facts. Thus, a tuple $\langle a_1, \ldots a_k \rangle$, belonging to relation r, will be identified with the ground atom $r(a_1, \ldots, a_k)$. The fact that a tuple $\langle a_1, \ldots, a_k\rangle$ belongs to relation r of a database instance $\mathbf{db}$ is thus simply denoted by $r(a_1, \ldots, a_k) \in \mathbf{db}$.

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

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

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

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

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

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

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

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

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

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

Example 2.1 Consider a relational database with the following relation schemas:

```
works(Emp#, Proj#, Assigned)
manages(Emp#, Proj#, Assigned)
relative(Emp1, Emp2)
```

Assume we want to know if there exists an employee working in a project managed by his/her relative. This question can be represented by the following query Q_1:

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

Another question is: does there exist a manager who has a relative working in the company? This question can be expressed by means of the following conjunctive query Q_2:

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

Decision problems such as the *query-of-tuple* problem (i.e., checking whether a given tuple belongs to the result of a CQ), the *containment problem* for CQs, and the *clause subsumption* problem are all equivalent via simple logspace transformations (see [26]) to the *evaluation problem* of Boolean CQs.

3 Quantitative Methods

One central component of a query optimizer is its search strategy or enumeration algorithm. The enumeration algorithm of the optimizer determines which plans to enumerate, and the classic enumeration algorithm is based on dynamic programming. This algorithm was pioneered in IBM's System R project [57], and it is used in most query optimizers today. Dynamic programming works very well if all queries are standard SQL-92 queries, the queries are moderately complex, and only simple textbook query execution techniques are used by the database system. Dynamic programming, however, does not work well if these conditions do not hold; e.g., if the database system must support very complex applications like SAP R/3 whose queries often involve many tables [16, 40] or new query optimization and execution techniques need to be integrated into the system in order to optimize queries in a distributed and/or heterogeneous programming environment [35, 43]. In these situations, the search space of query optimization can become very large and dynamic programming is not always viable because of its very high complexity. In general, there is a tradeoff between the complexity of an enumeration algorithm and the quality of the plans generated by the algorithm. Dynamic programming represents one extreme point: dynamic programming has exponential time and space complexity and generates *optimal* plans. Other algorithms have lower complexity than dynamic programming, but these algorithms are not able to find as low-cost plans as dynamic programming. Since the problem of finding an optimal plan is NP-hard [37, 56], implementors of query optimizers will probably always have to take this fundamental tradeoff between algorithm complexity and quality of plans into account when they decide which enumeration algorithm to use.

Due to its importance, a large number of different algorithms have already been developed for query optimization in database systems. All algorithms proposed so far fall into one of three different classes or are combinations of such basic algorithms [60]. In the following, we will briefly discuss each class of algorithms; a more complete overview and comparison of many of the existing algorithms can be found in [59].

3.1 Exhaustive search

All published algorithms of this class have exponential time and space complexity and are guaranteed to find the best plan according to the optimizer's cost model. The most prominent representative of this class of algorithms is (bottom-up) dynamic programming [57], which is currently used in most database systems.

This algorithm works in a bottom-up way as follows. First, dynamic programming generates so-called access plans for every table involved in the query. Typically, such an access plan consists of one or two operators, and there are several different access plans for a table. In the second phase, dynamic programming considers all possible ways to join the tables. First, it considers all two-way join plans by using the access plans of the tables as building blocks and calling the joinPlans function to build a join plan from these building blocks. From the two-way join plans and the access plans, dynamic programming then produces three-way join plans. After that, it generates four-way join plans by considering all combinations of two two-way join plans and all combinations of a three-way join plan with an access plan. In the same way, dynamic programming continues to produce five-way, six-way join plans and so on up to n-way join plans. In the third phase, the n-way join plans are massaged so that they become complete plans for the query; e.g., project, sort,or group-by operators are attached, if necessary. Note that in every step of the second phase, dynamic programming uses the same function to produce more and more complex plans using simpler plans as building blocks. Just as there are usually several alternative access plans, there are usually several different ways to join two tables (e.g., nested loop joins, hash joins, etc.) and the joinPlans function will return a plan for every alternative join method. The beauty of dynamic programming is that it discards inferior building blocks after every step (pruning). While enumerating two-way join plans, for example, dynamic programming would consider an $A \bowtie B$ plan and a $B \bowtie A$ plan, but only the cheaper of the two plans would be retained, so that only the cheaper of the two plans would be considered as a building block for three-way, four-way, ... join plans involving A and B. Pruning is possible because the $A \bowtie B$ plan and the $B \bowtie A$ plan do the same work; if the $A \bowtie B$ plan is cheaper than the $B \bowtie A$ plan, then any complete plan for the whole query that has $A \bowtie B$ as a building block (e.g., $C \bowtie (A \bowtie B)$) will be cheaper than the same plan with $B \bowtie A$ as a building block (e.g., $C \bowtie (B \bowtie A)$). As a result of pruning, dynamic programming does not enumerate inferior plans such as $C \bowtie (B \bowtie A)$ and runs significantly faster than a naive exhaustive search.

Note tht such an algorithm enumerates all bushy plans. However, with slight modifications, this algorithm could also be used to enumerate only so-called left-deep plans, as originally proposed in [57]. See [59], for a discussion about "left-deep vs. bushy" (see, e.g.,).

Efficient ways to implement dynamic programming have been proposed in [49] and [65]. Other representatives of this class are A* search [41] and transformation-based techniques (with top-down dynamic programming) such as those used in EXODUS, Volcano, and some commercial systems [30, 31, 52].

3.2 Heuristics

Typically, the algorithms of this class have polynomial time and space complexity, but they they typically produce worse plans [59]. Representatives of this class of algorithms are *minimum selectivity* and other greedy algorithms [60, 58, 59], the KBZ algorithm [45]; and the AB algorithm [62].

Basically, at each step, a partially determined order is extended by choosing the most promising relational operation to be executed, according to some preference criterion. Obviously, the quality of plans produced by the greedy algorithm strongly depends on the plan evaluation function that guides the preference criterion.

3.3 Randomized algorithms

Various variants of randomized algorithms have been proposed in [39], [61], [38], [46], [22], and [59]. The big advantage of randomized algorithms is that they have constant space overhead. The running time of most randomized algorithms cannot be predicted because these algorithms are indeterministic; typically, randomized algorithms are slower than heuristics and dynamic programming for simple queries and faster than both for very large queries. The best known randomized algorithm is called 2PO and is a combination of applying iterative improvement and simulated annealing [38]. In many situations, 2PO produces good plans. However, there are situations in which 2PO produces plans that are orders of magnitude more expensive than an optimal plan.

3.4 Iterative Dynamic Programming

This technique is based on iteratively applying dynamic programming and can be seen as a combination of dynamic and greedy programming [44]. The essence of this heuristic is that instead of fully enumerating all query processing plans, a resource limit is established (defined by a parameter k). During each dynamic programming stage, all query processing plans are enumerated up to k-way joins. At that point, one or more of the best plans are chosen. These partial plans will be used as building blocks to initiate the next stage of dynamic programming, which also produces query-processing plans until the resource limit is reached. The sequence of dynamic programming stages continues until a complete plan is generated. In [44], a number of important variations of this algorithm are described.

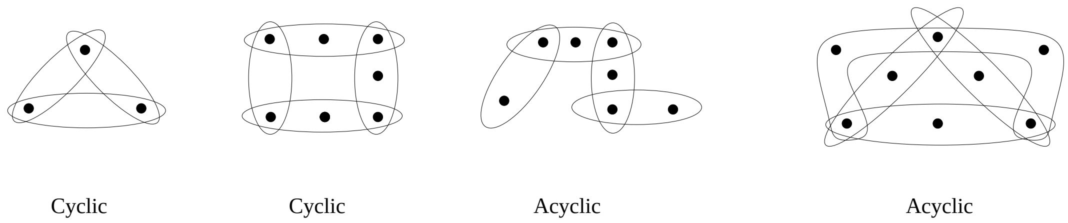


Figure 1: Cyclic and acyclic hypergraphs

4 Acyclic Conjunctive Queries

4.1 Acyclic Hypergraphs

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

Given a hypergraph $H = (V, E)$, the *GYO-reduct* $GYO(H)$ [32, 69] is the hypergraph obtained from H by repeatedly applying the following rules as long as possible:

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

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

It was noted [4] that the order of rule applications does not matter and that the final hypergraph $GYO(H)$ is unique.

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

There exist various equivalent characterizations of acyclic hypergraphs [3, 4, 10, 25, 36, 47].

In Figure 1, examples of cyclic and acyclic hypergraphs are exhibited.

Acyclicity as defined here is the usual concept of acyclicity in the context of databases. It is referred to as α-acyclicity in [17]. This is the least restrictive concept of hypergraph acyclicity among all those defined in the literature. There are several more restrictive notions of acyclicity such as, e.g., β-acyclicity, γ-acyclicity, and Berge-acyclicity [5, 17]. In particular, the following chain of implications holds for any hypergraph H: H is Berge-acyclic $\Rightarrow$ H is γ-acyclic $\Rightarrow$ H is β-acyclic $\Rightarrow$ H is acyclic (i.e., α-acyclic). However, in general, none of the reverse implications holds.

4.2 Join trees

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

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

Example 4.1 Consider again query Q_1 and Q_2 of Example 2.1. The hypergraphs $H(Q_1)$ and $H(Q_2)$ corresponding to queries Q_1 and Q_2, respectively, are depicted in Figure 2. It is easy to see that $H(Q_1)$ is a cyclic hypergraph, while $H(Q_2)$ is an acyclic hypergraph. As a consequence, Q_1 is a cyclic query and Q_2 is an acyclic query.

Figure 2: Hypergraphs $H(Q_1)$ and $H(Q_2)$

Acyclic conjunctive queries were the object of a large number of investigations [7, 12, 17, 18, 24, 48, 50, 54, 55, 68]. In particular, it was shown that the class of acyclic queries coincides with the class of *tree queries* ([4], see also [1, 47, 64]). The latter are queries which are representable by a join tree (or join forest).

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

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

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

Figure 3 shows a join tree for the query Q_3 in Example 4.2.

Example 4.2 Consider the following query:

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

A join tree for this query is shown in Figure 3.

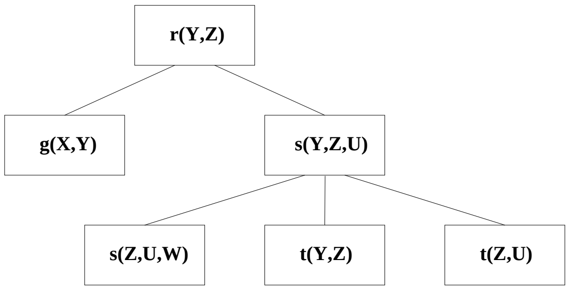


Figure 3: A join tree for the acyclic query Q_3 in Example 4.2

The concept of hypergraph acyclicity plays a very important role in database theory [1, 47, 64]. Indeed, acyclic queries have highly desirable computational properties:

1. Acyclic instances can be efficiently solved. Yannakakis provided a (sequential) polynomial time algorithm solving BCQ on acyclic queries[1] [68].
2. We have recently shown that BCQ are highly parallelizable on acyclic queries, as they are complete for the low complexity class LOGCFL [26]. Efficient parallel algorithms for

[1]Note that, since both the database **db** and the query Q are part of an input-instance of BCQ, what we are considering is the *combined complexity* of the query [66].

Boolean and non-Boolean queries have been proposed in [26] and [28]. They run on parallel database machines that exploit the *inter-operation parallelism* [67], 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, and a join tree of an acyclic hypergraph is efficiently computable. A linear-time algorithm for computing a join tree is shown in [63]; an L^{SL} method has been provided in [26].

4. The result of a (non-Boolean) acyclic conjunctive query Q can be *computed* in time polynomial in the combined size of the input instance and of the output relation [68].

Intuitively, the efficient behaviour 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 small the size of the intermediate relations (which could become exponential if regular join were performed).

5 Decomposition Methods

The important speed-up obtainable on acyclic instances stimulated several research efforts towards the identification of wider classes of queries having the same desirable properties as acyclic CQs. A number of different methods based only on the structure of the associated hypergraph have been proposed both in the database and in the AI communities (as the problem of answering conjunctive queries is equivalent to the constraint satisfaction problem). We call these methods *decomposition methods*, because each one provides a decomposition which transforms any hypergraph to an acyclic hypergraph.

In the following sections, we will give a brief description of the main decomposition methods considered in database theory and in artificial intelligence. Next, we introduce some useful definitions and notations.

Let $\mathcal{H}_I = (V, H)$ be the hypergraph of a CQ instance I. The *primal graph* of I is a graph $G = (V, E)$, having the same set of variables (vertices) as $\mathcal{H}_I$ and an edge connecting any pair of variables $X, Y \in V$ such that $\{X, Y\} \subseteq h$ for some $h \in H$.

Note that if all relations of a CQ have arity two, then its associated hypergraph is identical to its primal graph.

Let $\mathcal{H}$ be a hypergraph. For any set of edges $H' \subseteq edges(\mathcal{H})$, let $var(H') = \bigcup_{h \in H'} h$. Without loss of generality, we assume that $var(H) = var(\mathcal{H})$, i.e., every variable in $var(\mathcal{H})$ occurs in at least one edge of $\mathcal{H}$, and hence, any hypergraph can be simply represented by the set of its edges. Moreover, we assume without loss of generality that all hypergraphs under consideration are both *connected*, i.e., their primal graph consists of a single connected component, and *reduced*, i.e., no hyperedge is contained in any other hyperedge. All our definitions and results easily extend to general hypergraphs.

Let $\mathcal{HS}$ be the set of all (reduced and connected) hypergraphs. A *decomposition method* (short: DM) D associates to any hypergraph $\mathcal{H} \in \mathcal{HS}$ a parameter *D-width($\mathcal{H}$)*, called the *D width* of $\mathcal{H}$.

The decomposition method D ensures that, for fixed k, every CQ instance I whose hypergraph $\mathcal{H}_I$ has D-width $\leq k$ is polynomially solvable, i.e., it is solvable in $p(\|I\|) = O(\|I\|^{O(1)})$ time, where $\|I\|$ denotes the size of I. For any CQ instance I, the size of I is defined in the standard way, i.e., as the number of bits needed for encoding I by listing, for each atom in I, its relation schema and all tuples occurring in its relation instance.

For any $k > 0$, the *k-tractable class* $C(D, k)$ of D is defined by

$$C(D, k) = \{\mathcal{H} \mid D\text{-width}(\mathcal{H}) \leq k\}.$$

Thus, $C(D,k)$ collects the set of CQ instances which, for fixed k, are polynomially solvable by using the strategy D. Typically, the polynomial $p(\|I\|)$ depends on the parameter k. In particular, for each D, there exists a function f such that, for each k, each instance $I \in C(D,k)$ can be transformed in time $O(\|I\|^{O(f(k))})$ into an equivalent *acyclic* CQ instance. (It follows that all problems in $C(D,k)$ are polynomially solvable.)

Every DM D is complete with respect to $\mathcal{HS}$, i.e., $\mathcal{HS} = \bigcup_{k\geq 1} C(D,k)$. Note that, by our definitions, it holds that D-width$(\mathcal{H}) = \min\{k \mid \mathcal{H} \in C(D,k)\}$.

All tractable classes based on restricted structure that we have studied in the literature fit into this framework. We next describe how the notion of width is defined in the decomposition methods we consider in this paper. Detailed descriptions of these methods can be found in the corresponding reference (see below) and in many surveys on this subject, e.g., [51, 13].

5.1 Biconnected Components (short: BICOMP) [20]

Let $G = (V,E)$ be a graph. A vertex $p \in V$ is a *separating vertex* for G if, by removing p from G, the number of connected components of G increases. A biconnected component of G is a maximal set of vertices $C \subseteq V$ such that the subgraph of G induced by C is connected and remains connected after any one-vertex removal, i.e., has no separating vertices.

It is well known that, from any graph G, we can compute in linear time a vertex-labeled tree $\langle T,\chi\rangle$, where the labeling function χ is a bijective function that associates to each vertex of the tree T a set of vertices S of G, such that S is either a biconnected component of G, or a singleton containing a separating vertex for G. There is an edge $\{p,q\}$ in the tree T, if $\chi(p)$ is a biconnected component of G and $\chi(q)$ contains a separating vertex for G belonging to the component $\chi(p)$, i.e., $\chi(q) \subseteq \chi(p)$, holds. We say that $\langle T,\chi\rangle$ is the BICOMP decomposition of G.

For a hypergraph $\mathcal{H}$, the BICOMP decomposition of $\mathcal{H}$ is the BICOMP decomposition of its primal graph, and the *biconnected width* of $\mathcal{H}$, denoted by BICOMP-width$(\mathcal{H})$, is the maximum number of vertices over the biconnected components of the primal graph of $\mathcal{H}$.

Example 5.1 Figure 4.a shows a hypergraph $\mathcal{H}_b$ and Figure 4.b its primal graph. The vertices G, C, D, and E are the separating vertices of this primal graph. Note that the maximum number of vertices over its biconnected components is 3, and thus BICOMP-width$(\mathcal{H}) = 3$. Figure 5 shows the BICOMP decomposition of $\mathcal{H}_b$.

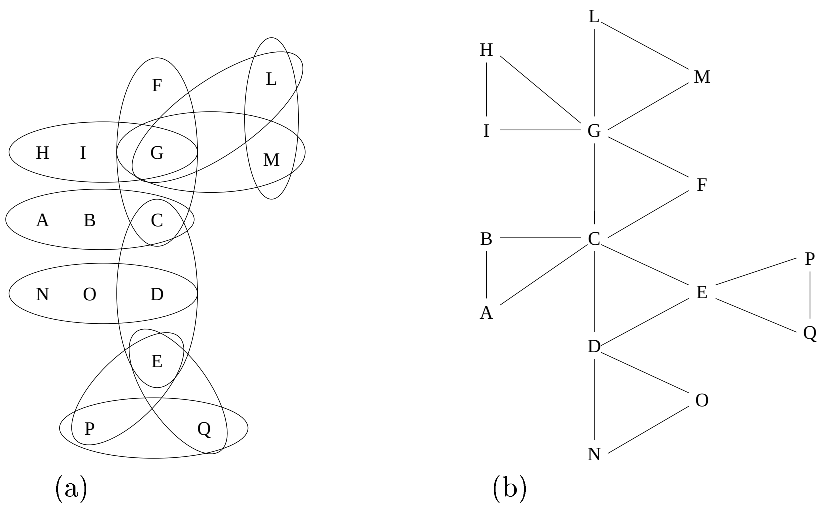


Figure 4: (a) The hypergraph $\mathcal{H}_b$, and (b) its primal graph

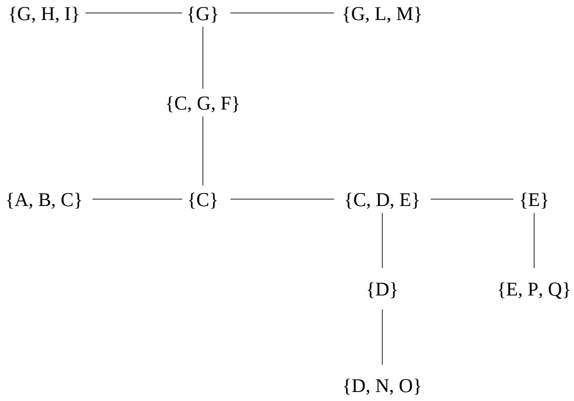
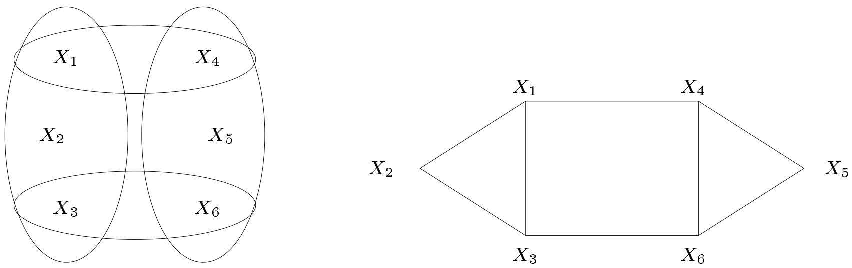


Figure 5: The BICOMP decomposition of the hypergraph $\mathcal{H}_b$ in Example 5.1

5.2 Tree Clustering (short: TCLUSTER) [15]

The *tree clustering* method is based on a triangulation algorithm which transforms the primal graph $G = (V, E)$ of any CQ instance I into a chordal graph G'. The acyclic hypergraph $\mathcal{H}(G')$ having the same set of vertices as G' and the maximal cliques of G' as its hyperedges is a TCLUSTER decomposition of $\mathcal{H}_I$. Intuitively, the hyperedges of $\mathcal{H}(G')$ are used to build the atoms (with their associated relations) of an acyclic CQ I' equivalent to I. The width of the TCLUSTER decomposition $\mathcal{H}(G')$ is the maximum cardinality of its hyperedges. The *tree-clustering width* (short: TCLUSTER *width*) of $\mathcal{H}_I$ is 1 if $\mathcal{H}_I$ is an acyclic hypergraph; otherwise, it is equal to the minimum width over the TCLUSTER decompositions of $\mathcal{H}_I$.

Example 5.2 Consider the hypergraph $\mathcal{H}_{tc}$ shown in Figure 6.a. Figure 6.b shows its primal graph.



(a) (b)

Figure 6: (a) The hypergraph $\mathcal{H}_{tc}$, and (b) its primal graph

This graph can be triangulated as shown in Figure 7.a. If we associate a hyperedge to each maximal clique of this triangulated graph, we get the acyclic hypergraph shown in Figure 7.b. This acyclic hypergraph is a TCLUSTER decomposition of $\mathcal{H}_{tc}$ of width 3. Moreover, it is easy to see that there is no TCLUSTER decomposition for $\mathcal{H}_{tc}$ having a smaller width, and hence the TCLUSTER width of $\mathcal{H}_{tc}$ is 3.

5.3 Treewidth (TREEWIDTH) [53]

A *tree decomposition* of a graph $G = (V, E)$ is a pair $\langle T, \chi \rangle$, where $T = (N, F)$ is a tree, and χ is a labeling function associating to each vertex $p \in N$ a set of vertices $\chi(p) \subseteq V$, such that the following conditions are satisfied:

1. for each vertex b of G, there exists $p \in N$ such that $b \in \chi(p)$;

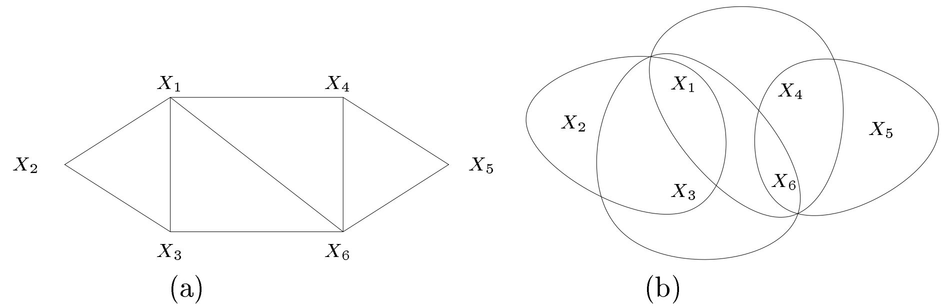


Figure 7: (a) A triangulation of the primal graph of $\mathcal{H}_{tc}$, and (b) a `TCLUSTER` decomposition of $\mathcal{H}_{tc}$

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

The *width* of the tree decomposition $\langle T, \chi \rangle$ is $\max_{p \in N} |\chi(p) - 1|$. The *treewidth* of G is the minimum width over all its tree decompositions. The `TREEWIDTH` of a hypergraph $\mathcal{H}$ is 1 if $\mathcal{H}$ is an acyclic hypergraph; otherwise, it is equal to the treewidth of its primal graph. As pointed out below, `TREEWIDTH` and `TCLUSTER` are two equivalent methods.

Example 5.3 Consider again the hypergraph $\mathcal{H}_{tc}$ in Example 5.2. Figure 8 show a tree decomposition of $\mathcal{H}_{tc}$ having width 2. It follows that the treewidth of $\mathcal{H}_{tc}$ is 2 as only hypergraphs having acyclic primal graphs have treewidth 1.

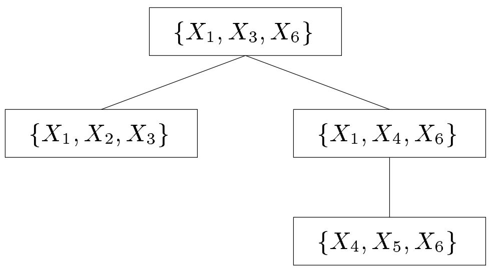


Figure 8: A tree decomposition of hypergraph $\mathcal{H}_{tc}$ in Example 5.2

5.4 Hinge Decompositions (short: `HINGE`) [34, 33]

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

Let $\mathcal{H} \in \mathcal{HS}$ and let H be either $edges(\mathcal{H})$ or a proper subset of $edges(\mathcal{H})$ containing at least two edges. Let $C_1, ..., C_m$ be the connected components of $\mathcal{H}$ with respect to H. Then, H is a *hinge* if, for $i = 1, ..., m$, there exists an edge $h_i \in H$ such that $var(edges(C_i)) \cap var(H)) \subseteq h_i$. A hinge is *minimal* if it does not contain any other hinge.

A *hinge decomposition* of $\mathcal{H}$ is a tree T such that all the following conditions hold: (1) the vertices of T are minimal hinges of $\mathcal{H}$; (2) each edge in $edges(\mathcal{H})$ is contained in at least one vertex of T; (3) two adjacent vertices A and B of T share precisely one edge $L \in edges(\mathcal{H})$; moreover, L consists exactly of the variables shared by A and B (i.e., $L = var(A) \cap var(B)$);

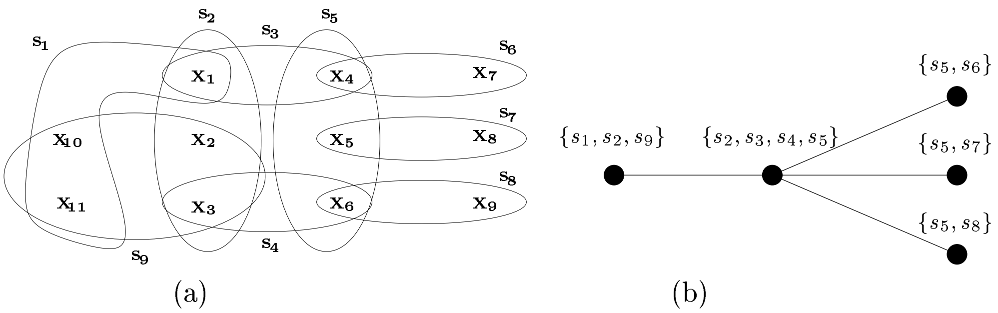


Figure 9: (a) Hypergraph $\mathcal{H}_{hg}$, and (b) a hinge-tree decomposition of $\mathcal{H}_{hg}$

(4) the variables of $\mathcal{H}$ shared by two vertices of T are entirely contained within each vertex on their connecting path in T.

It was shown in [34] that, for any CQ instance I, the cardinality of the largest vertex of any hinge decomposition of $\mathcal{H}_I$ is an invariant of $\mathcal{H}_I$, and is equal to the cardinality of the largest minimal hinge of $\mathcal{H}_I$. This number is called the *degree of cyclicity* of $\mathcal{H}_I$. We will also refer to it as the `HINGE` *width* of $\mathcal{H}_I$.

Example 5.4 Consider the following query I_{hg}:

$$\begin{aligned} ans \leftarrow \quad & s_1(X_1, X_{10}, X_{11}) \wedge s_2(X_1, X_2, X_3) \wedge s_3(X_1, X_4) \wedge s_4(X_3, X_6) \wedge s_5(X_4, X_5, X_6) \wedge \\ & s_6(X_4, X_7) \wedge s_7(X_5, X_8) \wedge s_8(X_6, X_9) \wedge s_9(X_2, X_3, X_{10}, X_{11}). \end{aligned}$$

Figure 9 shows the corresponding hypergraph $\mathcal{H}_{hg}$, which is clearly cyclic. The minimal hinges of $\mathcal{H}_{hg}$ are $H_1 = \{s_1, s_2, s_9\}$, $H_2 = \{s_2, s_3, s_4, s_5\}$, $H_3 = \{s_5, s_6\}$, $H_4 = \{s_5, s_7\}$, $H_5 = \{s_5, s_8\}$, $H_6 = \{s_3, s_6\}$, and $H_7 = \{s_4, s_8\}$, where s_i denotes the set of variables occurring in the atom s_i, for $1 \leq i \leq 9$.

Since the cardinality of the largest minimal hinge of $\mathcal{H}_{hg}$ (hinge H_2) is 4, it follows that the `HINGE` width of $\mathcal{H}_{hg}$ is 4. Figure 9.b shows a `HINGE` decomposition of $\mathcal{H}_{hg}$.

5.5 Hinge Decomposition + Tree Clustering (short: HINGE$^{\texttt{TCLUSTER}}$) [33]

It has been observed [33] that the minimal hinges of a hypergraph can be further decomposed by means of the triangulation technique of the above-described tree-clustering method. This leads to a new decomposition method, that we call `HINGE`$^{\texttt{TCLUSTER}}$, which combines `HINGE` and `TCLUSTER` and can be formally defined as follows. Let $T = (N, E)$ be a hinge tree of a hypergraph $\mathcal{H}$. For any hinge $H \in N$, let $w(H)$ be the minimum of the cardinality of H and the `TCLUSTER` width of the hypergraph $(var(H), H)$. The `HINGE`$^{\texttt{TCLUSTER}}$ width of $\mathcal{H}$ with respect to T is $\max_{H \in N}\{w(H)\}$. A `HINGE`$^{\texttt{TCLUSTER}}$ decomposition of $\mathcal{H}$ with respect to T is an acyclic hypergraph $\mathcal{H}'$ having the same set of vertices as $\mathcal{H}$, and whose set of edges is obtained from T and $\mathcal{H}$ as follows. For each hinge $H \in N$, if $w(H) = |H|$, then $\mathcal{H}'$ contains an edge $var(H)$; otherwise, $\mathcal{H}'$ contains the edges of any `TCLUSTER` decomposition of the (sub)hypergraph $(var(H), H)$ having width $w(H)$.

The `HINGE`$^{\texttt{TCLUSTER}}$ *width* of $\mathcal{H}$ is the minimum `HINGE`$^{\texttt{TCLUSTER}}$ width over all its `HINGE`$^{\texttt{TCLUSTER}}$ decompositions.

Example 5.5 Consider again the atoms of Example 5.4 and the hinge-tree decomposition for the hypergraph $\mathcal{H}_{hg}$ shown in Figure 9.b. From this hinge-tree decomposition, we construct a `HINGE`$^{\texttt{TCLUSTER}}$ decomposition $\mathcal{H}'_{hg}$ of $\mathcal{H}_{hg}$.

Consider the sub-hypergraph $(var(H_1), H_1)$ corresponding to the minimal hinge H_1 occurring in this hinge-tree decomposition. The primal graph of the hypergraph $(var(H_1), H_1)$ is a clique containing the vertices X_1, X_2, X_3, X_{10}, and X_{11}, thus it is easy to see that the `TCLUSTER` width of this hypergraph is 5. However, the hinge H_1 contains three edges, hence we get $w(H_1) = 3$,

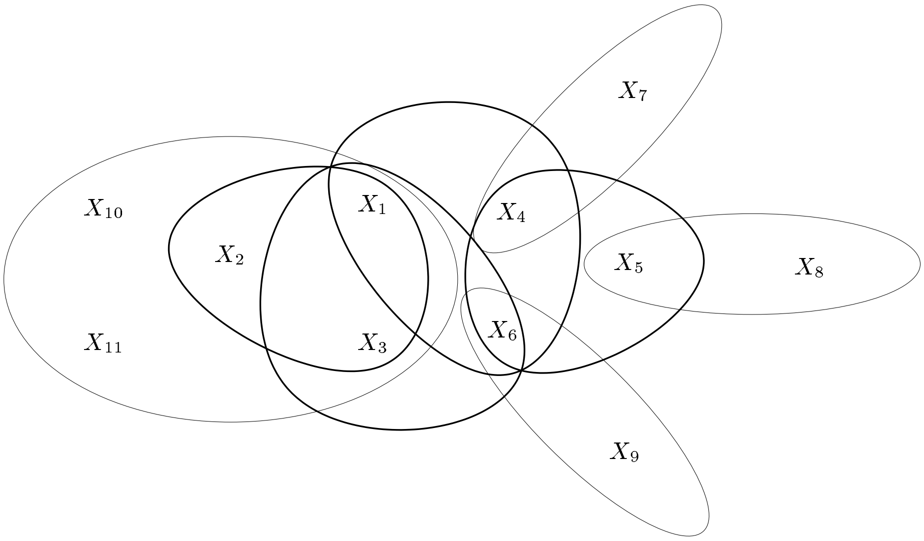


Figure 10: A $\texttt{HINGE}^{\texttt{TCLUSTER}}$ decomposition of hypergraph $\mathcal{H}_{hg}$ in Example 5.4

and the $\texttt{HINGE}^{\texttt{TCLUSTER}}$ decomposition $\mathcal{H}'_{hg}$ contains the edge $\{X_1, X_2, X_3, X_{10}, X_{11}\}$ with all the variables occurring in H_1.

A different situation concerns the sub-hypergraph $(var(H_2), H_2)$ corresponding to the minimal hinge H_2. This hypergraph is identical to hypergraph $\mathcal{H}_{tc}$ in Example 5.2. We observed that $\mathcal{H}_{tc}$ has TCLUSTER width 3, which is smaller than $|H_2| = 4$, and hence $w(H_2) = 3$ holds. This means that, in this case, it is convenient to further decompose $(var(H_2), H_2)$ using the TCLUSTER decomposition method, and the $\texttt{HINGE}^{\texttt{TCLUSTER}}$ decomposition $\mathcal{H}'_{hg}$ contains all the edges belonging to the TCLUSTER decomposition of $\mathcal{H}_{tc} = (var(H_2), H_2)$ shown in Figure 6.

Similarly, for $i \in \{4, 5, 6\}$, the sub-hypergraphs $(var(H_i), H_i)$ corresponding to the other hinges occurring in the hinge-tree decomposition at hand are acyclic hypergraphs. Therefore, $w(H_i) = 1$ holds, because the TCLUSTER width of acyclic hypergraphs is 1.

The resulting $\texttt{HINGE}^{\texttt{TCLUSTER}}$ decomposition $\mathcal{H}'_{hg}$ of $\mathcal{H}_{hg}$ is the acyclic hypergraph shown in Figure 10. The thickest edges in this figure come from the TCLUSTER decomposition of $(var(H_2), H_2)$. Recall that both $w(H_1)$ and $w(H_2)$ are 3, which is the maximum value over the hinges occurring in the given HINGE decomposition of $\mathcal{H}_{hg}$. Thus, the width of $\mathcal{H}'_{hg}$ is 3, and it is easy to verify that there is no other $\texttt{HINGE}^{\texttt{TCLUSTER}}$ decomposition having smaller width. It follows that the $\texttt{HINGE}^{\texttt{TCLUSTER}}$ width of $\mathcal{H}_{hg}$ is 3.

5.6 Cycle Cutset (short: CUTSET) [13]

A *cycle cutset* of a hypergraph $\mathcal{H}$ is a set $S \subseteq var(\mathcal{H})$ such that the subgraph of the primal graph of $\mathcal{H}$ (vertex-)induced by $var(\mathcal{H}) - S$ is acyclic. That is, after deleting the vertices in S, the primal graph of $\mathcal{H}$ becomes acyclic. The CUTSET width of $\mathcal{H}$ is 1 if $\mathcal{H}$ is acyclic; otherwise, it is the minimum cardinality over all its possible cycle cutsets.

Example 5.6 The hypergraph $\mathcal{H}_b$ shown in Figure 4.a has CUTSET width 4. Indeed, $\{G, C, D, E\}$ is a cycle cutset of this hypergraph, and any smaller set of vertices does not allow to break all the cycles in its primal graph (see Figure 4.b). As another example, consider the hypergraph $\mathcal{H}_{tc}$ shown in Figure 6. The CUTSET width of $\mathcal{H}_{tc}$ is 2, because there is no cycle cutset of cardinality 1, while there are cycle cutsets of cardinality 2, e.g., the set $\{X_1, X_4\}$.

5.7 Hypertree Decompositions

A new class of tractable conjunctive database queries, which generalizes the class of acyclic queries, has recently been identified [27]. This is the class of queries having a bounded-width hypertree decomposition [27]. Deciding whether a given query has this property is feasible in polynomial time and even highly parallelizable. We will call HYPERTREE the decomposition method based on this notion.

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

Definition 5.7 A *hypertree decomposition* of a hypergraph $\mathcal{H}$ is a hypertree $HD = \langle T, \chi, \lambda \rangle$ for $\mathcal{H}$ which satisfies all the following conditions:

1. for each edge $h \in edges(\mathcal{H})$, there exists $p \in vertices(T)$ such that $var(h) \subseteq \chi(p)$ (we say that p *covers* h);
2. 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 $p \in vertices(T)$, $\chi(p) \subseteq var(\lambda(p))$;
4. for each $p \in vertices(T)$, $var(\lambda(p)) \cap \chi(T_p) \subseteq \chi(p)$.

Note that the inclusion in Condition 4 is actually an equality, because Condition 3 implies the reverse inclusion.

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 hypertree decomposition HD of hypergraph $\mathcal{H}$ is a *complete decomposition* of $\mathcal{H}$ if every edge of $\mathcal{H}$ is strongly covered in HD.

The *width* of a hypertree decomposition $\langle T, \chi, \lambda \rangle$ is $max_{p \in vertices(T)} |\lambda(p)|$. The HYPERTREE *width* $hw(\mathcal{H})$ of $\mathcal{H}$ is the minimum width over all its hypertree decompositions. A c-width hypertree decomposition of $\mathcal{H}$ is *optimal* if $c = hw(\mathcal{H})$.

The acyclic hypergraphs are precisely those hypergraphs having hypertree width one. Indeed, any join tree of an acyclic hypergraph $\mathcal{H}$ trivially corresponds to a hypertree decomposition of $\mathcal{H}$ of width one. Furthermore, if a hypergraph $\mathcal{H}'$ has a hypertree decomposition of width one, then, from this decomposition, we can easily compute a join tree of $\mathcal{H}'$, which is therefore acyclic [27].

Remark 5.8 From any hypertree decomposition HD of $\mathcal{H}$, we can easily compute a complete hypertree decomposition of $\mathcal{H}$ having the same width. For any "missing" edge h, choose a vertex q of T such that $var(h) \subseteq \chi(q)$ (such a vertex must exist by Condition 1), and create a new vertex p as a child of q with $\lambda(p) = h$ and $\chi(p) = var(h)$. Assuming the use of suitable data structures, this computation can be done in $O(\|\mathcal{H}\| \cdot \|HD\|)$ time, where $\|HD\|$ denotes the size of a hypertree decomposition, i.e., the number of bits needed for encoding HD (that is, for encoding the rooted tree of HD and, for each vertex v of this tree, the labelings χ and λ for v, encoded as a list of variables and a list of edge identifiers, respectively).

Intuitively, if $\mathcal{H}$ is a cyclic hypergraph, the χ labeling selects the set of variables to be fixed in order to split the cycles and achieve acyclicity; $\lambda(p)$ "covers" the variables of $\chi(p)$ by a set of edges.

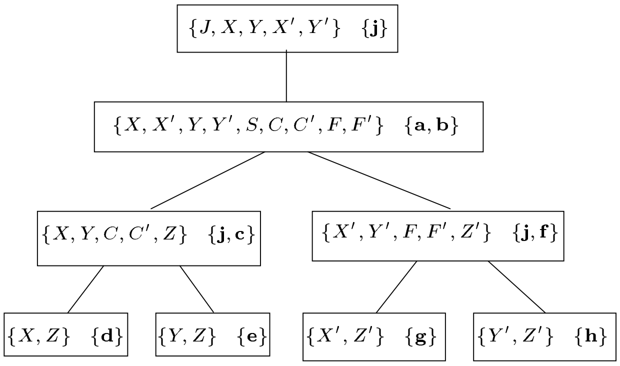


Figure 11: A 2-width hypertree decomposition of $\mathcal{H}_1$

Example 5.9 Consider the following conjunctive query:

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

Let $\mathcal{H}_1$ be its corresponding hypergraph. Since $\mathcal{H}_1$ is cyclic, $hw(\mathcal{H}_1) > 1$ holds. Figure 11 shows a (complete) hypertree decomposition of $\mathcal{H}_1$ having width 2, hence $hw(\mathcal{H}_1) = 2$.

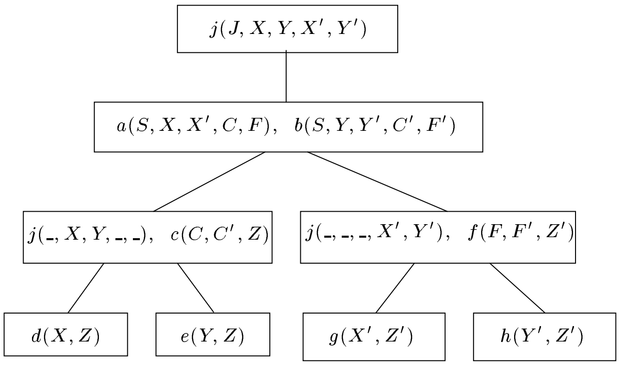


Figure 12: Hyperedge representation of hypertree decomposition HD_1

In order to help the intuition of what a hypertree decomposition is, we also present an alternative representation, called *hyperedge representation*. (Also, "atom representation.") Figure 12 shows the hyperedge representation of the hypertree decomposition HD_1 of $\mathcal{H}_1$. Each node p in the tree is labeled by a set of hyperedges representing $\lambda(p)$; $\chi(p)$ is the set of all variables, distinct from '_', appearing in these hyperedges. Thus, the anonymous variable '_' replaces the variables in $var(\lambda(p)) - \chi(p)$.

Using this representation, we can easily observe an important feature of hypertree decompositions. Once an 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. For instance, the variables in the hyperedge corresponding to atom j in $\mathcal{H}_1$ are jointly included only in the root of the decomposition. If we were forced to take all the variables in every vertex where j occurs, it would not be possible to find a decomposition of width 2. Indeed, in this case, any choice of two hyperedges per vertex yields a hypertree which violates the connectedness condition for variables (i.e., Condition 2 of Definition 5.7).

Let k be a fixed positive integer. We say that a CQ instance I has k-bounded `HYPERTREE` width if $hw(\mathcal{H}_I) \leq k$, where $\mathcal{H}_I$ is the hypergraph associated to I. From the results in [27], it follows that k-bounded hypertree width is efficiently decidable, and that a hypertree decomposition of width k can be efficiently computed (if any).

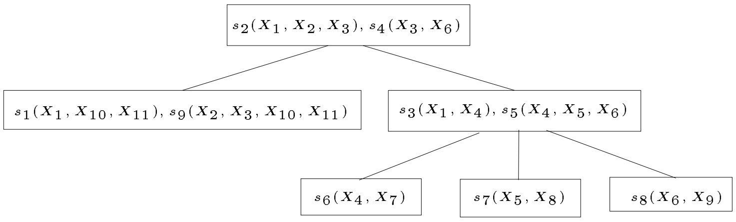


Figure 13: A hypertree decomposition of hypergraph $\mathcal{H}_{hg}$ in Example 5.4

Example 5.10 Consider again the CQ instance I_{hg} in Example 5.4. Figure 13 shows the hyperedge representation of a width 2 hypertree decomposition of its hypergraph $\mathcal{H}_{hg}$. It follows that $hw(\mathcal{H}_{hg}) = 2$, because $\mathcal{H}_{hg}$ is cyclic. Thus, I_{hg} has 2-bounded HYPERTREE width and, more generally, k-bounded HYPERTREE width for any integer $k > 1$.

5.8 Further Methods

Another generalizations of acyclic hypergraphs is the class of hypergraphs having *bounded query width* [11]. This notion yields a very wide class of tractable CQs (resp. CSPs). However, no efficient algorithm for computing a query decomposition of fixed width was known. In fact, it has been recently proved that checking whether a query has width $\leq k$, for some fixed constant k, is NP-complete [27].

Further interesting methods that do not explicitly generalize acyclic hypergraphs are based on a notion of *consistency* as used in [19, 20]. Dechter and Pearl [14] introduced the notion of *induced width* w^* which is – roughly – the smallest width k of any graph G' obtained by triangulation methods from the primal graph G of a hypergraph such that G' ensures $k+1$-consistency. Graphs having induced width $\leq k$ can be also characterized as *partial k-trees* [21] or, equivalently, as graphs having treewidth $\leq k$ [2]. It follows that, for fixed k, checking whether $w^* \leq k$ is feasible in linear time [8]. The approach based on w^* is referred to as the w^*-Tractability method [13]. Note that this method is implicitly based on hypergraph acyclicity, given that the used triangulation methods enforce chordality of the resulting graph G' and thus acyclicity of the corresponding hypergraph. It was noted [15, 13] that, for any given hypergraph $\mathcal{H}$, TCLUSTER-width$(\mathcal{H}) = w^*(\mathcal{H}) + 1$.

6 Comparing Decomposition Methods

For comparing decomposition methods we introduce the relations $\preceq$, $\rhd$, and $\prec\!\!\!\prec$ defined as follows:

$\underline{D_1 \preceq D_2}$ (in words, D_2 *generalizes* D_1) if there exists $\delta \geq 0$ such that, for every $k > 0$, $C(D_1, k) \subseteq C(D_2, k+\delta)$. Thus, $D_1 \preceq D_2$ implies that every class of CQ instances which is tractable according to D_1 is also tractable according to D_2.

Note that the constant δ above allows us to get rid of small differences among tractability classes that should be irrelevant in the comparison. E.g., it is known that TCLUSTER and TREEWIDTH are equivalent methods and one would expect TCLUSTER to generalize TREEWIDTH (as well as vice versa). However, for any $k > 1$, $C(\texttt{TREEWIDTH}, k) \not\subseteq C(\texttt{TCLUSTER}, k)$, because the treewidth is defined through the cardinality of the vertex-labeling minus one. Rather, $C(\texttt{TREEWIDTH}, k) \subseteq C(\texttt{TCLUSTER}, k+1)$ holds. Thus, by taking $\delta = 1$, we easily get TREEWIDTH $\preceq$ TCLUSTER.

$\underline{D_1 \rhd D_2}$ (D_1 *beats* D_2) if there exists an integer k such that, for every m, $C(D_1, k) \not\subseteq C(D_2, m)$. To prove that $D_1 \rhd D_2$, it is sufficient to exhibit a class of hypergraphs contained in some $C(D_1, k)$ but in no $C(D_2, j)$ for every $j \geq 0$.

Intuitively, $D_1 \rhd D_2$ means that, at least on some class of CQ instances, D_1 outperforms D_2 with respect to tractability, because these instances are tractable according to D_1, but not according to D_2. For such classes, using D_1 is thus better than using D_2.

$\underline{D_1 \lll D_2}$ if $D_1 \preceq D_2$ and $D_2 \rhd D_1$. In this case we say that D_2 *strongly generalizes* D_1.

This means that D_2 is really the more powerful method, given that, whenever D_1 guarantees polynomial runtime for query answering, then also D_2 guarantees tractable query answering, but there are classes of queries that can be answered in polynomial time by using D_2 but are not tractable according to D_1.

Mathematically, $\preceq$ is a *preorder*, i.e., it is reflexive, transitive, but not antisymmetric. We say that D_1 *is* $\preceq$*-equivalent to* D_2, denoted $D_1 \equiv D_2$, if both $D_1 \preceq D_2$ and $D_2 \preceq D_1$ hold. Note that, on the other hand, $\lll$ is transitive and antisymmetric, but not reflexive.

The decomposition methods D_1 and D_2 are *strongly incomparable* if both $D_1 \rhd D_2$ and $D_2 \rhd D_1$. Note that, if D_1 and D_2 are strongly incomparable, then they are incomparable with respect to the relations $\preceq$ and $\lll$, too.

Figure 14 shows a representation of the hierarchy of DMs determined by the $\lll$ relation. Each element of the hierarchy represents a DM, apart from that containing *Tree Clustering*, w^*, and *Treewidth* which are grouped together because they are $\preceq$-equivalent.

Theorem 6.1 ([29]) *For each pair D_1 and D_2 of decompositions methods represented in Figure 14, the following holds:*

- *There is a directed path from D_1 to D_2 iff $D_1 \lll D_2$, i.e., iff D_2 strongly generalizes D_1.*
- *D_1 and D_2 are not linked by any directed path iff they are strongly incomparable.*

Hence, Fig. 14 gives a complete picture of the relationships holding among the different methods.

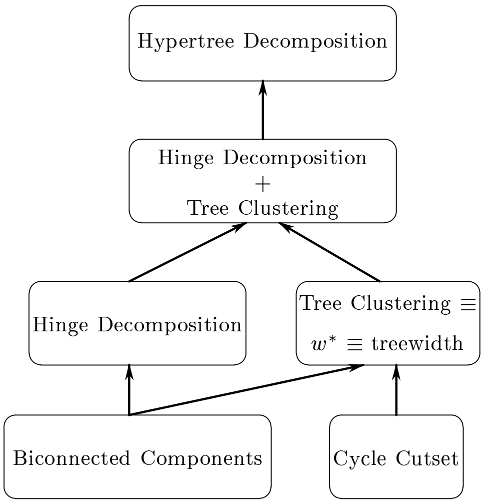


Figure 14: Query Tractability Hierarchy

7 Conclusions and Future Work

In this paper, we have described the most important approaches to query evaluation. We considered both query planning techniques based on quantitative information and evaluation strategies based on some structural properties of queries. Moreover, we provided a formal comparison of the most important structural methods.

It is worthwhile noting that quantitative and structural methods have been completely separated worlds. In particular, structural properties of queries have been deeply investigated in the literature, but most commercial DBMS only use quantitative methods for optimizing queries. However, many meaningful queries are acyclic or close to acyclic queries. Therefore, it may happen that such systems take a large amount of time for answering queries that are structurally simple and easily solvable.

Thus, an interesting line of research concerns the integration of ideas and techniques from both approaches to query answering. For instance, structural notions like hypertree width could be suitably extended in order to take into account quantitative information about size of relations, availability of indices, and so on.

References

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[2] S. Arnborg, J. Lagergren, and D. Seese. Problems easy for tree-decomposable graphs. *Journal of Algorithms*, 12:308–340, 1991.

[3] C. Beeri, R. Fagin, D. Maier, A. Mendelzon, J.D. Ullman, and M. Yannakakis. Properties of acyclic database schemas. In *ACM Symposium on Theory of Computing (STOC '81)*, Milwaukee, Wisconsin, pp.355–362, 1981.

[4] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, July 1983.

[5] C. Berge. *Graphs and Hypergraphs.* North-Holland, The Netherlands, 1976.

[6] P.A. Bernstein and D.W. Chiu. Using Semi-Joins to Solve Relational Queries. *Journal of the ACM*, 28(1):25–40, 1981.

[7] P.A. Bernstein and N. Goodman. The power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

[8] H.L. Bodlaender. Treewidth: Algorithmic Techniques and Results. In *Proceedings of MFCS'97*, Bratislava. LNCS 1295, Springer, pp. 19-36, 1997.

[9] A.K. Chandra and P.M. Merlin. Optimal Implementation of Conjunctive Queries in relational Databases. In *ACM Symposium on Theory of Computing (STOC'77)*, pp.77–90, 1977.

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

[11] Ch. Chekuri and A. Rajaraman. Conjunctive Query Containment Revisited. *Theoretical Computer Science*, 239(2):211–229, 2000.

[12] A. D'Atri, and M. Moscarini. Recognition algorithms and design methodologies for acyclic database schemes. In *Advances in Computing Research*, 3:43–68, P. Kanellakis editor, JAI press, London, 1986.

[13] Rina Dechter. Constraint networks. In Stuart C. Shapiro, editor, *Encyclopedia of Artificial Intelligence*, pages 276–285. Wiley, 1992. Volume 1, second edition.

[14] Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence*, 34(1):1–38, 1987.

[15] Rina Dechter and Judea Pearl. Tree clustering for constraint networks. *Artificial Intelligence*, 38(3):353–366, April 1989.

[16] Jochen Doppelhammer, Thomas Höppler, Alfons Kemper, and Donald Kossmann. Database performance in the real world - tpc-d and sap r/3 (experience paper). In Joan Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*, pages 123–134. ACM Press, 1997.

[17] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30(3):514–550, 1983.

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

[19] E.C. Freuder. A sufficient condition for backtrack-free search. *Journal of the ACM*, 29(1):24–32, 1982.

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

[21] E.C. Freuder. Complexity of K-Tree Structured Constraint Satisfaction Problems. *Proceedings of AAAI'90*, 1990.

[22] César A. Galindo-Legaria, Arjan Pellenkoft, and Martin L. Kersten. Fast, randomized join-order selection - why use transformations? In Jorge B. Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases, September 12-15, 1994, Santiago de Chile, Chile*, pages 85–95. Morgan Kaufmann, 1994.

[23] M.R. Garey and D.S. Johnson. *Computers and Intractability. A Guide to the Theory of* NP*-completeness*. Freeman and Comp., NY, USA, 1979.

[24] N. Goodman, and O. Shmueli. Tree queries: a simple class of relational queries. *ACM Transactions on Database Systems*, 7(4):653–677, 1982.

[25] N. Goodman, and O. Shmueli. Syntactic characterization of tree database schemas. *Journal of the ACM*, 30(4):767–786.

[26] Georg Gottlob, Nicola Leone, and Francesco Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 43(3), 2001.

[27] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems – PODS'99*, pages 21–32, May 31st – June 2nd 1999. To appear in *Journal of Computer and System Sciences.*

[28] G. Gottlob, N. Leone, and F. Scarcello. Advanced Parallel Algorithms for Processing Acyclic Conjunctive Queries, Rules, and Constraints. In *Proceedings of the 2000 Conference on Software Engineering and Knowledge Engineering (SEKE'00)*, pp. 167–176, Chicago, July, 2000. Full version: Technical Report DBAI-TR-98/18, available on the web at `http://www.dbai.tuwien.ac.at/staff/gottlob/parallel.ps`, or by email from the authors.

[29] G. Gottlob, N. Leone, and F. Scarcello. A Comparison of Structural CSP Decomposition Methods. *Artificial Intelligence*, 124(2):243–282, Elsevier, 2000.

[30] Goetz Graefe and David J. DeWitt. The exodus optimizer generator. In Umeshwar Dayal and Irving L. Traiger, editors, *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, California, May 27-29, 1987*, pages 160–172. ACM Press, 1987.

[31] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 209–218. IEEE Computer Society, 1993.

[32] M.H. Graham. On the universal relation. Tech. Report, Univ. of Toronto, Toronto, Canada, 1979.

[33] M. Gyssens, P.G. Jeavons, and D.A. Cohen. Decomposing constraint satisfaction problems using database techniques. *Artificial Intelligence*, 66:57–89, 1994.

[34] M. Gyssens, and J. Paredaens. A Decomposition Methodology for Cyclic Databases. In *Advances in Database Theory*, volume 2, pp. 85-122. Plenum Press, New York, 1984.

[35] Laura M. Haas, Donald Kossmann, Edward L. Wimmers, and Jun Yang. Optimizing queries across diverse data sources. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 276–285. Morgan Kaufmann, 1997.

[36] R. Hull. Acyclic join dependency and data base projections. *Journal of Computer and System Sciences*, 27(3):331–349, 1983.

[37] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *TODS*, 9(3):482–502, 1984.

[38] Y. E. Ioannidis and Younkyung Kang. Randomized algorithms for optimizing large join queries. In Hector Garcia-Molina and H. V. Jagadish, editors, *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, May 23–25, 1990, Atlantic City, NJ*, volume 19(2) of *SIGMOD Record (ACM Special Interest Group on Management of Data)*, pages 312–321, New York, NY 10036, USA, June 1990. ACM Press.

[39] Yannis E. Ioannidis and Eugene Wong. Query optimization by simulated annealing. In Umeshwar Dayal and Irv Traiger, editors, *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data 1987 annual conference, San Francisco, May 27–29, 1987*, pages 9–22, New York, NY 10036, USA, 1987. ACM Press.

[40] Alfons Kemper, Donald Kossmann, and Florian Matthes. Sap r/3: A database application system (tutorial). In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, page 499. ACM Press, 1998.

[41] Alfons Kemper, Guido Moerkotte, and Klaus Peithner. A blackboard architecture for query optimization in object bases. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 543–554. Morgan Kaufmann, 1993.

[42] P. G. Kolaitis and M. Y. Vardi. Conjunctive-Query Containment and Constraint Satisfaction. *Proceedings of Symposium on Principles of Database Systems (PODS'98)*, pp.205–213, Seattle, Washington, 1998. Full version: *Journal of Computer and System Sciences*, 61:302–332, 2000.

[43] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[44] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: a new class of query optimization algorithms. *ACM Transactions on Database Systems*, 25(1):43–82, 2000.

[45] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi, editors, *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, pages 128–137. Morgan Kaufmann, 1986.

[46] Rosana S. G. Lanzelotte, Patrick Valduriez, and Mohamed Zaït. On the effectiveness of optimization search strategies for parallel execution spaces. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 493–504. Morgan Kaufmann, 1993.

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

[48] F.M. Malvestuto. Modelling large bases of categorical data with acyclic schemes. In *First International Conference on Database Theory*, Rome, Italy, 1986.

[49] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*, pages 314–325. Morgan Kaufmann, 1990.

[50] C.H. Papadimitriou and M. Yannakakis. On the Complexity of Database Queries. In *Proceedings of Symposium on Principles of Database Systems (PODS'97)*, pp.12–19, Tucson, Arizona, 1997.

[51] J. Pearson and P. Jeavons. A survey of tractable constraint satisfaction problems. Technical Report CSD-TR-97-15, Royal Holloway, University of London, 1997.

[52] Arjan Pellenkoft, César A. Galindo-Legaria, and Martin L. Kersten. The complexity of transformation-based join enumeration. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 306–315. Morgan Kaufmann, 1997.

[53] N. Robertson and P.D. Seymour. Graph Minors II. Algorithmic aspects of tree width. *Journal of Algorithms*, 7:309–322, 1986.

[54] D. Saccà. Closures of Database Hypergraphs. *Journal of the ACM*, 32(4):774–803, 1985.

[55] Y. Sagiv and O. Shmueli. Solving Queries by Tree Projections. *ACM Transactions on Database Systems*, 18(3):487–511, 1993.

[56] Wolfgang Scheufele and Guido Moerkotte. On the complexity of generating optimal plans with cross products. In *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, May 12-14, 1997, Tucson, Arizona*, pages 238–248. ACM Press, 1997.

[57] P.G. Selinger, M.M. Astrahan, R.A. Lorie, and T.G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD '79)*, pages 23–34, Boston MA, May 30 – June 1 1979. ACM Press.

[58] Eugene J. Shekita, Honesty C. Young, and Kian-Lee Tan. Multi-join optimization for symmetric multiprocessors. In Rakesh Agrawal, Seán Baker, and David A. Bell, editors, *19th International Conference on Very Large Data Bases, August 24-27, 1993, Dublin, Ireland, Proceedings*, pages 479–492. Morgan Kaufmann, 1993.

[59] Michael Steinbrunn, Guido Moerkotte, and Alfons Kemper. Heuristic and randomized optimization for the join ordering problem. *VLDB Journal*, 6(3):191–208, 1997.

[60] Arun N. Swami. Optimization of large join queries: Combining heuristic and combinatorial techniques. In James Clifford, Bruce G. Lindsay, and David Maier, editors, *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, May 31 - June 2, 1989*, pages 367–376. ACM Press, 1989.

[61] Arun N. Swami and Anoop Gupta. Optimization of large join queries. In Haran Boral and Per-Åke Larson, editors, *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, June 1-3, 1988*, pages 8–17. ACM Press, 1988.

[62] Arun N. Swami and Balakrishna R. Iyer. A polynomial time algorithm for optimizing join queries. In *Proceedings of the Ninth International Conference on Data Engineering, April 19-23, 1993, Vienna, Austria*, pages 345–354. IEEE Computer Society, 1993.

[63] 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 Journal on Computing*, 13(3):566-579, 1984.

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

[65] Bennet Vance and David Maier. Rapid bushy join-order optimization with cartesian products. In H. V. Jagadish and Inderpal Singh Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 35–46. ACM Press, 1996.

[66] M. Vardi. Complexity of Relational Query Languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC '82)*, pp. 137–146, 1982.

[67] A.N. Wilschut, J. Flokstra, and P. M.G. Apers. Parallel evaluation of multi-join queries. In *Proceedings of SIGMOD'95*, San Jose, CA USA, pp.115–126, 1995.

[68] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th Conference on Very Large Databases, Morgan Kaufman pubs. (Los Altos CA), Zaniolo and Delobel(eds)*, September 1981.

[69] C.T. Yu, and M.Z. Özsoyoğlu. An algorithm for tree-query membership of a distributed query. In *Proceedings of IEEE COMPSAC*, pp.306–312, 1979.

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