Proc. CAAP83 8th Colloquium on Trees in Algebra and Programming, Springer-Verlag Lecture Notes in Computer Science 159 (1983), ed. G. Ausiello and M. Protasi, pp. 65-89.

ACYCLIC DATABASE SCHEMES (OF VARIOUS DEGREES): A PAINLESS INTRODUCTION

Ronald Fagin
IBM Research Laboratory
San Jose, California 95193

ABSTRACT: Database schemes (which, intuitively, are collections of table skeletons) can be viewed as hypergraphs. (A *hypergraph* is a generalization of an ordinary undirected graph, such that an edge need not contain exactly two nodes, but can instead contain an arbitrary nonzero number of nodes.) Unlike the situation for ordinary undirected graphs, there are several natural, nonequivalent notions of acyclicity for hypergraphs (and hence for database schemes). A large number of desirable properties of database schemes fall into a small number of equivalence classes, each completely characterized by the degree of acyclicity of the scheme. This paper is intended to be an informal introduction, in which the focus is mainly on the originally studied (and least restrictive) degree of acyclicity.

Categories and Subject Descriptors: F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic; G.2.2 [**Discrete Mathematics**]: Graph Theory - *Graph algorithms* and *Trees*; H.2.1 [**Database Management**]: Logical Design - *Normal forms* and *Schema and subschema*; H.3.3. [Information Storage and Retrieval]: Information Search and Retrieval - *Query formulation.*

General terms: Algorithms, Design, Languages, Management, Theory

Additional Key Words and Phrases: acyclic, hypergraph, database scheme, relational database

1. Introduction

This paper is intended to be an informal introduction to acyclic database schemes (of various degrees). The first such notion of acyclic (database) scheme to be introduced goes under various names, including *acyclic scheme* [BFMMUY], *tree scheme* [GS1], and *α-acyclic* scheme [Fa3]. In the first part of this paper, we shall refer to them simply as "acyclic (database) schemes". There have now been a number of papers published about acyclic database schemes ([ADM], [BDM], [BFMMUY], [BFMY], [BB], [Ch], [DM], [Fa3], [FMU], [GS1], [GS2], [GS3], [GST], [GP], [Ha], [Hu], [MU], [Sa], [TY], [Ya]).

The organization of this paper is as follows. In the first part of the paper (Sections 2-8), we discuss acyclic database schemes and their desirable properties. The results we describe are due to Beeri, Fagin, Maier, Mendelzon, Ullman, and Yannakakis [BFMMUY]. Two of these sections, namely Sections 6 and 7 (dealing with the formal definition of acyclicity and with join dependencies) are more technical, and may be skipped without real loss for many readers. For a more detailed and precise description of what we shall discuss in the Sections 2-8, the reader is referred to [BFMY]. In the second part of the paper (Sections 9 and 10), we describe other, even more restrictive degrees of acyclicity (due to Fagin [Fa3]), and their even nicer properties.

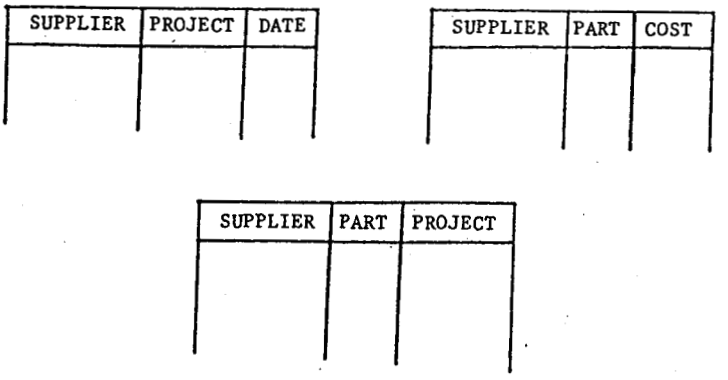

Figure 1.1

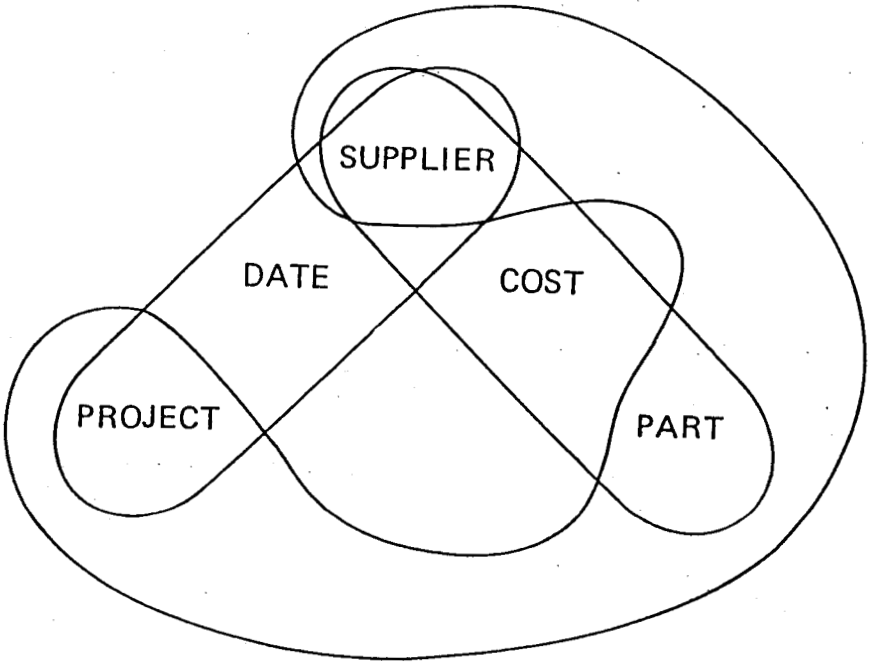

Figure 1.2

A *database scheme* can be thought of as a collection of table skeletons (as in Figure 1.1). When the tables are filled in with entries, we have a *(relational) database* [Co1]. It is convenient to associate a hypergraph with each database scheme. A *hypergraph* is a pair ($\mathcal{N}$, $\mathcal{E}$), where $\mathcal{N}$ is a finite set of *nodes*, and $\mathcal{E}$ is a set of *edges* (or *hyperedges*), which are arbitrary nonempty subsets of $\mathcal{N}$. An ordinary undirected graph (without self-loops) is, of course, a hypergraph where every edge has exactly two nodes. We shall identify a hypergraph by only mentioning its edges, and tacitly assume that the nodes are precisely those that appear in some edge. The hypergraph of Figure 1.2 corresponds to the database scheme of Figure 1.1. The correspondence should be clear: for example, there is a {SUPPLIER,PART,COST} edge in the hypergraph of Figure 1.2 because of the {SUPPLIER,PART,COST} "relation scheme" in Figure 1.1, and so on.

Consider now the two database schemes of Figures 1.3 and 1.4, each consisting of three relation schemes. The only difference between them is that the second database scheme has a D in the last relation scheme where the first has an E. Although this seems on the surface to be a minor difference, it turns out that the first scheme is *acyclic*, or desirable, while the second is *cyclic*, or undesirable. Thus, we shall see that the first scheme enjoys a number of desirable properties that the second does not.

To help understand why we call the first scheme acyclic and the second cyclic, consider the associated hypergraphs of the two schemes, in Figure 1.5. The first hypergraph should, intuitively, look somehow acyclic (in fact, it looks like a long snake), while the second hypergraph should look cyclic (in fact, somewhat like a snake biting its tail). It indeed turns out that by our definitions, the first hypergraph is acyclic and the second is cyclic.

A number of basic, desirable properties of relational database schemes turn out to be equivalent to acyclicity. These properties were defined and studied by a number of researchers, in quite different contexts. It is somewhat remarkable that all of these properties are equivalent (in fact, all are equivalent to acyclicity). We shall discuss some of these properties in this paper. On the other hand, cyclic schemes exhibit certain pathological properties.

The theory is more elegant in the acyclic case. Thus, by assuming acyclicity, it is possible to prove theorems that would not hold in general. Also, there are efficient algorithms available in the acyclic case that are not available in the general case. Thus, there are problems that are NP-complete in general (as proven, for example, in [HLY] and [MSY]), but which have polynomial-time algorithms in the acyclic case [Ya]. We shall see an example later.

Finally, we note that there is a simple, efficient algorithm for determining acyclicity. We exhibit this algorithm in the next section. For pedagogical reasons, we shall postpone the formal definition of acyclicity until later (Section 6). For now, the reader can take the algorithm of the next section as defining acyclicity.

2. Graham's algorithm for determining acyclicity

We now give an algorithm for determining whether a hypergraph (and hence the corresponding database scheme) is acyclic. It is called *Graham's algorithm*, in honor of Marc Graham, who showed [Gr]

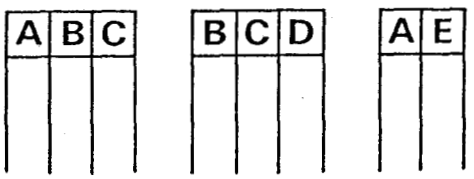

Figure 1.3

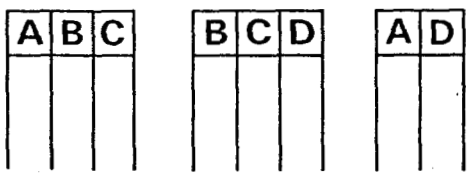

Figure 1.4

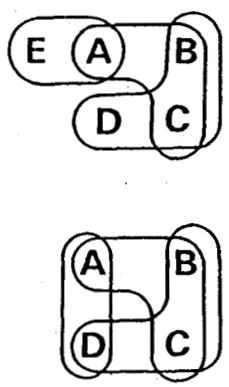

Figure 1.5

that if a hypergraph was accepted by his algorithm, then this was sufficient to imply a certain nice property ("Every pairwise consistent database is consistent"; see Section 3.) It actually turns out that acceptance by Graham's algorithm is not just sufficient but also necessary for this property. Graham's algorithm was also used, independently, by Yu and Ozsoyoglu [YO].

We shall describe Graham's algorithm by example. Consider the hypergraph of Figure 2.1. Somewhat surprisingly, it turns out that this hypergraph is acyclic, even though it seems to contain a "cycle"; we shall come back to this point at the end of this section.

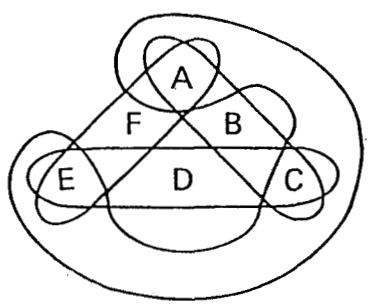

Figure 2.1

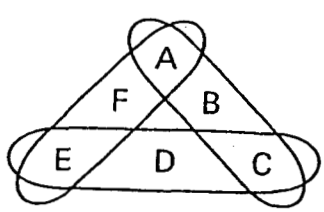

Figure 2.2

We begin the algorithm by writing the edges, one underneath the other:

A	B	C			
		C	D	E	
A				E	F
A		C		E	

(For convenience, we have put common vertices in the same column.)

There are two types of operations that we perform repeatedly, in any order, until neither can be applied. The first operation is to delete an "isolated" node, that is, a node that appears in exactly one row. In our example, we thereby delete nodes B, D, and F, since each is isolated. We are left with:

A	C	
	C	E
A		E
A	C	E

The second operation is to delete a row if its nodes are all in some other row. Thus, the first (AC) row is contained in the last (ACE) row, and so we delete the AC row:

	C	E
A		E
A	C	E

Similarly, we delete the new first and second rows, since each is contained in the new third row. We are left with a single row:

A	C	E

We now apply the first operation, since each of A, C, and E is an isolated node. We are then left with the empty set.

We say that Graham's algorithm *succeeds* if (as in our example) it terminates with the empty set; otherwise, it *fails*.

Theorem 2.1 [BFMY]. A hypergraph is acyclic precisely if Graham's algorithm succeeds, with the hypergraph as input.

For the hypergraph of Figure 2.1, Graham's algorithm succeeds, and so the hypergraph is acyclic.

It is instructive to see an example in which Graham's algorithm fails (and so the hypergraph is cyclic). This time, we apply the algorithm to the hypergraph of Figure 2.2. This hypergraph contains three of the four edges of the hypergraph of Figure 2.1. The algorithm begins with

A	B	C			
		C	D	E	
A				E	F

After deleting the isolated nodes B, D, and F, we are left with:

A	C	
	C	E
A		E

The algorithm now halts, since no node is isolated, and no row is a subset of another row. Since what is left is not the empty set, the hypergraph is cyclic.

Note that the acyclic hypergraph of Figure 2.1 has a cyclic subhypergraph, namely, the hypergraph of Figure 2.2. (A *subhypergraph* of a hypergraph is simply the hypergraph consisting of a subset of the edges.) This counterintuitive phenomenon does not happen with ordinary graphs: that is, it is not possible for a subgraph of an ordinary acyclic graph to be cyclic. In Sections 10 and 11, we shall discuss other degrees of acyclicity for hypergraphs, where this counterintuitive phenomenon does not occur.

We close this section by remarking that Tarjan and Yannakakis [TY] have recently obtained an even more efficient algorithm for determining acyclicity than Graham's algorithm. Their algorithm runs in linear time.

3. Consistency

We are now ready to discuss some of the desirable properties of database schemes that are equivalent to acyclicity. We begin by considering the concept of *consistency*. In doing so, we shall make several definitions by example. (Remember, this is an informal paper. For formal definitions, see [BFMY].)

Consider the following three relations:

A	B	C
0	0	1
1	0	1
2	3	4

B	C	D
0	1	1
3	4	5

A	D
0	1
1	1
2	5

Figure 3.1

(For typographical convenience, we are now writing relations as in Figure 3.1, rather than with vertical bars, as in Figure 1.3.)

It is a natural and useful question to ask whether a database (a set of relations) is *consistent*, that is, whether the relations in the database are all projections of one fixed ("universal") relation. For the database in Figure 3.1, the answer is "Yes", since, as we shall see, each of the relations is a projection of the relation in Figure 3.2.

A	B	C	D
0	0	1	1
1	0	1	1
2	3	4	5

Figure 3.2

For example, the ABC relation of Figure 3.1 is the projection of the relation r of Figure 3.2 onto its ABC columns, since the result of "ignoring" the D entries in r is the ABC relation of Figure 3.1. (The order of the tuples, or rows, is even the same for both, although this is simply a coincidence. Since a relation is a *set* of tuples, the order of the tuples does not matter.) Similarly, the BCD relation of Figure 3.1 is the projection of r onto its BCD columns, since both contain precisely the tuples (0,1,1) and (3,4,5). Although it appears at first glance that the projection of r onto its BCD columns contains *two* copies of (0,1,1), this is not the case, since a relation is a set, and so there are no duplicates. Finally, the AD relation of Figure 3.1 is the projection of r onto its AD columns. So, the database in Figure 3.1 is indeed consistent. Intuitively, these relations can be pieced together into a single big relation.

We now show an example of an *inconsistent* database.

A	B	C
0	0	0
1	1	1

B	C	D
0	0	0
1	1	1

A	D
0	1
1	0

Figure 3.3

It is easy to verify, by a simple ad hoc argument, that the database in Figure 3.3 is not consistent. For, the (0,0,0) tuple of the ABC relation and the (0,0,0) tuple of the BCD relation together imply that an ABCD relation r that has each of them as projections would have a (0,0,0,0) tuple. However, this would imply that the AD relation which is a projection of r would have a (0,0) tuple, which the AD relation of Figure 3.3 does not.

It is easy to see that a pair of relations is consistent if they agree on their common part. For example, the ABC and BCD relations of Figure 3.1 are consistent, since they have the same BC projections, which consist precisely of the tuples (0,1) and (3,4). (Once again, duplicate tuples do not matter.) Let us say that a database is *pairwise consistent* if each pair of relations is consistent. It is clear every consistent database is pairwise consistent. It would be very nice if the converse were true, since then there would be a simple test for consistency, namely, pairwise consistency. Unfortunately, however, the converse does not hold. For, it is easy to verify that the database of Figure 3.3 is pairwise consistent; but, as we already noted, it is not consistent. In fact, we already knew that there could be no simple test for consistency, since determining consistency is known to be an NP-complete problem [HLY].

However, in the acyclic case our desired converse holds, that is, pairwise consistency and consistency are equivalent.

Theorem 3.1 [BFMY]. If the scheme is acyclic, then a database is consistent if and only if it is pairwise consistent.

Can there be any *cyclic* schemes for which consistency and pairwise consistency are equivalent? The answer is no.

Theorem 3.2 [BFMY]. If the scheme is cyclic, then there is a database that is pairwise consistent but not consistent.

Putting Theorems 3.1 and 3.2 together, we see that a scheme is acyclic if and only if every pairwise consistent database is consistent. Therefore (using also the fact that consistency implies pairwise consistency), it follows that a scheme is acyclic if and only if checking pairwise consistency is an algorithm for deciding consistency. This gives us one of the promised desirable properties that is equivalent to acyclicity.

Note that acyclicity of a scheme can be checked once and for all (by Graham's algorithm). Then, if the scheme is acyclic, there is an easy test for consistency.

There is one viewpoint on what we have just discussed that should be emphasized. In the *unrestricted* case (where we do not assume acyclicity), determining consistency is an NP-complete problem. However, in the *acyclic* case, there is a polynomial-time algorithm for checking consistency (namely, checking pairwise consistency). This gives us an example of an NP-complete problem that has a polynomial-time algorithm if the scheme is acyclic. Yannakakis [Ya] has found a number of other such problems.

4. Semijoins

We now consider a completely different type of problem, and once again find a desirable property that is equivalent to acyclicity. Assume that we have a geographically distributed database, with several sites, where each relation in the database is located at exactly one site. For example, assume that the first relation below is in Rome, and the second relation below is in San Jose.

A	B	C	D	E	F	G
0	0	1	2	3	4	5
1	3	9	0	6	3	17
2	1	17	4	19	2	8

Figure 4.1: The Rome relation

A	B	V	W	X	Y	Z
0	0	101	102	103	104	105
0	0	201	202	203	204	205
3	6	14	91	3	6	47

Figure 4.2: The San Jose relation

Assume for our example that the relations contain many tuples (only some of which are shown). We have also chosen the relation schemes to get across the idea that both relations contain many columns.

The most important way to combine the information from several relations is to take the *join*. For example, if one relation contains information about EMPLOYEEs and DEPARTMENTs, and a second contains information about DEPARTMENTs and LOCATIONs, then by joining the two relations, we can relate EMPLOYEEs and LOCATIONs. For the relations as written above, the join is as below:

A	B	C	D	E	F	G	V	W	X	Y	Z
0	0	1	2	3	4	5	101	102	103	104	105
0	0	1	2	3	4	5	201	202	203	204	205

Figure 4.3: The join of the Rome and San Jose relations

Formally, the *join* r ⋈ s of two relations r and s with attributes R and S respectively is a relation with attributes R∪S, and which consists of all tuples t such that (i) t[R], the projection of t onto R, is in r, and (ii) t[S] is in s.

There are several possible strategies for obtaining the join of the Rome and San Jose relations. One strategy is simply to ship the entire Rome relation to San Jose, and take the join in San Jose (or vice-versa). Another strategy is called the *semijoin strategy*. We shall describe how it works in the case of our example.

Step 1. Ship the AB projection of the San Jose relation to Rome (since A and B are the attributes that the two relations have in common). Note that, even though several tuples in the San Jose relation have (0,0) as their AB projection, only one copy of (0,0) is shipped to Rome.

Step 2. Prune the Rome relation by removing tuples whose AB projection is not in the relation shipped to Rome in Step (1). For example, since (0,0) comes to Rome from San Jose, we do *not* remove the Rome tuple that begins with (0,0). However, if (1,3) does not come to Rome from San Jose, then we remove the Rome tuple that begins (1,3) from the Rome relation. We call the final result of pruning the Rome relation in this manner the *semijoin* of the Rome relation with the San Jose relation (written r ⋉ s, where r is the Rome relation and s is the San Jose relation.)

Step 3. Ship this pruned Rome relation to San Jose, and take the join.

It is easy to see that the result of the pruning operation in step (2) is to remove from the Rome relation those tuples that will not participate in the join (that is, none of these removed tuples will be the projection of a tuple in the join.) Under natural assumptions, this semijoin strategy saves quite a bit of data transmission, since only a few columns of the "wide" San Jose relation were shipped to Rome in step (1), and (hopefully) only a fraction of the Rome tuples were shipped to San Jose in step (3).

The semijoin strategy has been analyzed extensively ([BC], [BG]). This strategy has actually been implemented in the SDD1 system of the Computer Corporation of America [R*], and in the RAP database machine of the University of Toronto [OSS].

Let us define a *semijoin program* to be a linear sequence of semijoin statements, such as

$r_3 := r_3 \ltimes r_7$
$r_7 := r_7 \ltimes r_4$
$r_2 := r_2 \ltimes r_1$
$r_7 := r_7 \ltimes r_3$
•
•
•

For example, the first statement might say to prune the Rome relation with the San Jose relation (as we did above), the second might say to prune the San Jose relation with the New York relation, and so on.

Theorem 4.1 ([BFMY], [BG]) Assume that the database scheme is acyclic. Then there is a semijoin program that fully reduces all of the relations in the database. By "fully reduce", we mean that every tuple that is left after the pruning process participates in the join, that is, is a projection of a tuple in the join. In fact, there is even such a semijoin program that is short (it consists of only 2n-2 semijoins, where n is the number of sites).

In other words, if the scheme is acyclic, then the semijoin strategy is useful. However, the situation is completely different in the cyclic case. Before we state a theorem, let us look at an example. Consider the database in Figure 2.5 above. The join of all of the relations is empty, but no relation can be pruned by a semijoin. For example, the semijoin of the first relation with the second relation just gives back the first relation, that is, no tuples are removed. Thus, semijoins do not help us to prune. This is typical in the cyclic case:

Theorem 4.2 ([BFMY], [BG]) Assume that the database scheme is cyclic. Then no semijoin program is guaranteed to fully reduce all of the relations in the database.

We can summarize Theorems 4.1 and 4.2 by saying that a database scheme is acyclic if and only if there is a semijoin program that fully reduces all of the relations in the database. This can be stated loosely by saying that a scheme is acyclic if and only if the semijoin strategy is a useful strategy. Bernstein and Goodman [BG] help make precise this loose statement.

5. Monotone join expressions

We now consider a different problem, involving "monotone join expressions". It is helpful to begin with an example.

Consider the following scenario. A user desires to take the join of four relations r_1, r_2, r_3, and r_4. The following might happen. He might first form $r_1 \bowtie r_2$, which might have, say, a thousand tuples. Then he might join the result with r_3, to obtain $r_1 \bowtie r_2 \bowtie r_3$, a relation with, say, a million tuples. He might finally join the result with r_4, to obtain his desired answer $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$, which might have only ten tuples. Thus, even though the result he was seeking had only ten tuples, he might have had an intermediate result with a million tuples. We now discuss "monotone join expressions", which prevent this unpleasant behavior.

A *join expression* is a parenthesization, such as

$$(R_1 \bowtie R_4) \bowtie (R_2 \bowtie R_3),$$

where we can think of the R_i's as relation variables. The join expression is said to be *monotone* with respect to a database r_1, r_2, ... if every (binary) join that appears in the join expression is over consistent relations (when r_1 is substituted for R_1, etc.) For example, the above join expression is monotone with respect to the database r_1, r_2, r_3, r_4 if

(a) r_1 and r_4 are consistent,
(b) r_2 and r_3 are consistent, and
(c) $(r_1 \bowtie r_4)$ and $(r_2 \bowtie r_3)$ are consistent.

The significance of being monotone is that when a pair of relations are consistent, then "no tuples are lost" in taking the join, that is, the resulting relation has each of the relations that were joined as projections (and in particular, the number of tuples does not decrease in taking the join). A *monotone join expression* is a join expression that is monotone with respect to every pairwise consistent database.

Theorem 5.1 A database scheme is acyclic if and only if there is a monotone join expression.

6. The formal definition of acyclicity

Up to now, we have been taking Graham's algorithm as defining acyclicity. We now give a graph-theoretic definition. We note that (a) the author is mindful of his promise of a "painless introduction", and that (b) this section is probably painful for most readers. Therefore, the reader is advised that he may skip this section without real loss.

Recall that a hypergraph is a pair $(\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is a finite set of nodes, and $\mathcal{E}$ is a set of edges (or hyperedges), which are arbitrary nonempty subsets of $\mathcal{N}$. We shall identify a hypergraph by mentioning only its edges, and tacitly assuming that the nodes are precisely those that appear in some edge.

A *path* from node s to node t is a sequence of $k \geq 1$ edges $E_1,...,E_k$ such that

(i) s is in E_1,
(ii) t is in E_k, and
(iii) $E_i \cap E_{i+1}$ is nonempty if $1 \leq i < k$.

We also say the above sequence of edges is a path from E_1 to E_k.

Two nodes are *connected* if there is a path from one to the other. Similarly, two edges are connected if there is a path from one to the other. A set of nodes or edges is connected if every pair is connected. The *connected components* are the maximal connected sets of edges.

Let $(\mathcal{N}, \mathcal{E})$ be a hypergraph. Its *reduction* $(\mathcal{N}, \mathcal{E}')$ is obtained by removing from $\mathcal{E}$ each edge that is a proper subset of another edge. A hypergraph is *reduced* if it equals its reduction, that is, if no edge is a subset of another edge.

Let $\mathcal{F}$ be a set of edges, and let $\mathcal{M}$ be the set of nodes that appear in one or more of the edges in $\mathcal{F}$. We say that $\mathcal{F}$ is *closed* if for each edge E of the hypergraph, there is an edge F in $\mathcal{F}$ such that $E \cap \mathcal{M} \subseteq F$. For example, {G,H,I} in Figure 6.1 is a closed set of edges. Thus, the intersection of edge K with $G \cup H \cup I$ is contained in edge H; similarly, the intersection of edge J with $G \cup H \cup I$ is contained in G, and the intersection of each of edges L and M with $G \cup H \cup I$ is contained in I. However, {L,M} is *not* a closed set of edges, if nodes x and y are present, as drawn in Figure 6.1. For, the intersection of edge I with $L \cup M$ is contained in neither L nor M. Note that every set of edges in an ordinary undirected graph is automatically closed.

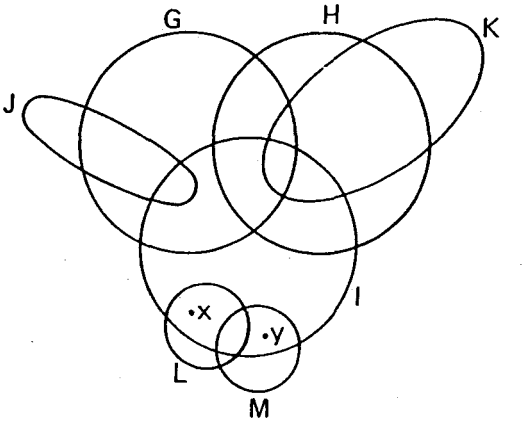

Figure 6.1

Let $\mathcal{F}$ be a connected, reduced set of edges, and let E and F be in $\mathcal{F}$. Let $Q = E \cap F$. We say that Q is an *articulation set* of $\mathcal{F}$ if the result of removing Q from every edge of $\mathcal{F}$, that is, $\{E-Q: E \in \mathcal{F}\} - \{\emptyset\}$, is not connected. It is clear that an articulation set in a hypergraph is a generalization of the concept of an articulation point in an ordinary undirected graph (where we think of the articulation point as the intersection of a pair of edges).

A reduced hypergraph is said to be acyclic if every closed, connected set of at least three edges has an articulation set. A hypergraph is said to be cyclic or acyclic precisely if its reduction is.

Example 6.1: It is straightforward to verify that Figure 2.1 shows an acyclic hypergraph. Its edges are ABC, CDE, EFA, and ACE. An articulation set for the set of all edges is $ABC \cap ACE = AC$, since the result of removing A and C from each edge is to leave the set of edges B, DE, EF, and E, which is not connected (B is disconnected from the others). Note that the set of edges {ABC, CDE, EFA} has no

articulation set. However, this set is not closed, so there is no contradiction of our assertion that the hypergraph of Figure 2.1 is acyclic. □

7. Acyclic join dependencies

This section is intended for those interested in database dependency theory. It is likely to be a painful section for anyone else.

We say ([ABU], [Ri]) that a relation r with attributes $R_1 \cup ... \cup R_n$ obeys the *join dependency* $\bowtie \{R_1, ..., R_n\}$ if $r = \bowtie \{r_1, ..., r_n\}$, where $r_i = r[R_i]$, for $1 \le i \le n$. It follows that the join dependency $\bowtie \{R_1, ..., R_n\}$ holds for the relation r if and only if r contains each tuple t for which there are tuples $w_1, ..., w_n$ of r (not necessarily distinct) such that $w_i[R_i] = t[R_i]$ for each i ($1 \le i \le n$). As an example, the relation r in Figure 7.1 below violates the join dependency $\bowtie$ {AB, ACD, BC}.

A	B	C	D
0	1	0	0
0	2	3	4
5	1	3	0

Figure 7.1

For, let w_1, w_2, w_3 be, respectively, the tuples (0,1,0,0), (0,2,3,4), and (5,1,3,0) of r; let R_1, R_2, R_3 be, respectively, AB, ACD, and BC; and let t be the tuple (0,1,3,4); then $w_i[R_i] = t[R_i]$ for each i ($1 \le i \le n$), although t is not a tuple in the relation r. However, it is straightforward to verify that the same relation r obeys, for example, the join dependency $\bowtie$ {ABC, BCD, ABD}.

To help us understand the semantics of join dependencies, it is helpful to consider an example (from [FMU]).

Example 7.1 Assume that the attributes are C(ourse), T(eacher), R(oom), H(our), S(tudent), and G(rade). Assume a "universal" relation over these attributes, consisting of tuples (c,t,r,h,s,g) for which teacher t "teaches" course c; course c "meets in" room r "at hour" h; and student s "is getting" grade g "in" course c. Thus, the universal relation is {(c,t,r,h,s,g): P_1(t,c) and P_2(c,r,h) and P_3(s,g,c)}, where the P_1 predicate gives the "teaches" relation, etc. It follows easily that the universal relation obeys the join dependency $\bowtie$ {TC, CRH, SGC}. In general, a necessary and sufficient condition for a relation to obey a join dependency is that the relation is "built up" of corresponding predicates [FMU]. □

A *multivalued dependency* $X \twoheadrightarrow Y$ is (equivalent to) a "binary" join dependency $\bowtie$ {XY,XZ}, where Z is the set of attributes not in $X \cup Y$. For extensive discussions of the value of multivalued dependencies, see [Fa1] or [Za]. Since multivalued dependencies are simple special cases of join dependencies, it is a desirable property of a join dependency for it to be equivalent to a set of multivalued dependencies. Moreover, it is easy to test (by sorting and counting) whether a given multivalued dependency holds for a relation; however, the problem of whether a given join dependency holds for a given relation is NP-

complete ([MSY]). The join dependency $\bowtie \{R_1, ..., R_n\}$ is said to be acyclic precisely if the hypergraph $\{R_1, ..., R_n\}$ is acyclic.

Theorem 7.2 [FMU] A join dependency is acyclic if and only if it is equivalent to a set of multivalued dependencies.

It is known [BFMY] that an acyclic join dependency $\bowtie \{R_1, ..., R_n\}$ is in fact equivalent to a set of at most n-1 multivalued dependencies (where n is the number of R_i's), and that there is a polynomial-time algorithm for finding such a set of n-1 multivalued dependencies. As an example, the acyclic join dependency $\bowtie$ {ABC, CDE, EFA, ACE}, which corresponds to the acyclic hypergraph of Figure 2.1, is equivalent to the set {AC$\twoheadrightarrow$DEF, CE$\twoheadrightarrow$ABF, AE$\twoheadrightarrow$BCD} of multivalued dependencies.

8. Summary so far

The theorems we have stated show the following:

Theorem 8.1 The following conditions on $\mathbf{R} = \{R_1, ..., R_n\}$ are equivalent.

(1) **R** is an acyclic hypergraph.
(2) Graham's algorithm succeeds with input **R**.
(3) Every pairwise consistent database over **R** is consistent.
(4) There is a semijoin program that fully reduces every database over **R**.
(5) **R** has a monotone join expression.
(6) The join dependency $\bowtie \mathbf{R}$ is equivalent to a set of multivalued dependencies.

We note that there are a number of other conditions that we have not mentioned in this introductory paper that are also equivalent to acyclicity.

9. Other degrees of acyclicity

For the rest of this paper, let us refer to the "acyclicity" that we have been discussing as "α-acyclicity". We now discuss even more restrictive degrees of acyclicity, which thereby enjoy even nicer properties.

By a *subhypergraph* of a hypergraph, we mean a subset of the edges. Similarly, a *subscheme* of a database scheme contains a subset of the relation schemes. The definition of α-acyclicity has the counterintuitive property that a subhypergraph of an α-acyclic hypergraph may be α-cyclic. As an example, the acyclic hypergraph of Figure 2.1 has the cyclic subhypergraph of Figure 2.2. This strange phenomenon does not happen for ordinary graphs. Let us define [Fa3] a hypergraph (respectively, database scheme) to be *β-acyclic* if every subhypergraph (respectively, subscheme) is α-acyclic. It is then easy to see that every subhypergraph of a β-acyclic hypergraph is β-acyclic; thus, this counterintuitive phenomenon does not happen for β-acyclic hypergraphs. Note that the hypergraph of Figure 2.1 is α-acyclic but not β-acyclic. Of course, a hypergraph (or scheme) is said to be *β-cyclic* precisely if it is not β-acyclic.

Theorem 9.1 [Fa3] Let $\mathcal{P}$ be any of the equivalent desirable properties of α-acyclic database schemes. A database scheme is β-acyclic if and only if every subscheme enjoys property $\mathcal{P}$.

Thus, β-acyclic schemes are of interest because of the importance of α-acyclic schemes, and because of the naturalness of considering subschemes. As an example of Theorem 9.1, let $\mathcal{P}$ be the property that there is a semijoin program that fully reduces every database over **R**. This property $\mathcal{P}$ can be stated loosely by saying that "semijoins are helpful if we desire to take the join of all of the relations in the database". Theorem 9.1 then tells us that a database scheme is β-acyclic if and only if semijoins are helpful whenever we desire to take the join of an *arbitrary* subset of the relations in the database.

It turns out that there are a number of equivalent graph-theoretic definitions of β-acyclicity. For example:

Theorem 9.2 [Fa3] A hypergraph is β-cyclic if and only if there is a sequence $(S_1, x_1, S_2, x_2, \ldots, S_m, x_m, S_{m+1})$ such that

(i) $x_1,\ldots,x_m$ are distinct nodes of the hypergraph;
(ii) $S_1,\ldots,S_m$ are distinct edges of the hypergraph, and $S_{m+1} = S_1$;
(iii) $m \geq 3$, that is, there are at least 3 edges involved; and
(iv) x_i is in S_i and S_{i+1} (for $1 \leq i \leq m$) and in no other S_j.

Also, we note the following:

Theorem 9.3 [Fa3] There is a polynomial-time algorithm for determining β-acyclicity.

10. γ-acyclicity

There is an even more restrictive degree of acyclicity which is useful for databases, called γ-acyclicity. Every γ-acyclic scheme (or hypergraph) is β-acyclic, and each β-acyclic scheme (or hypergraph) is α-acyclic, but neither reverse implication holds. Since γ-acyclicity is the most restrictive assumption, every γ-acyclic scheme enjoys all of the desirable properties that any of the others do.

We shall defer the definition of γ-acyclicity until the end of this section. We now discuss some of the desirable properties of γ-acyclic schemes. It is convenient to begin with an example. Assume that the database scheme consists of three relation schemes: a EMP__WORK relation scheme with attributes EMP (for "employee"), DEPT (for "department"), and SAL (for "salary"); a DEPT__INFO relation scheme with attributes DEPT, CITY, and MGR; and an EMP__HOME relation scheme with attributes EMP, STREET, CITY, and CHILD. See Figure 10.1 for an example of one tuple in each relation. In this example, there are two distinct "{EMP, CITY} relationships". One, which has the tuple (Fagin, San Jose), relates an employee to the city where he works. The other, which has the tuple (Fagin, Los Gatos), relates an employee to the city where he lives. (In general, if X is a set of attributes, such as {EMP, CITY}, then by an *X relationship*, we mean the result of taking a "connected join" of some of the relations in the database, and then projecting onto X. In a *connected join*, the set of relation schemes of the relations that are joined is connected.)

EMP_WORK:

EMP	DEPT	SAL
Fagin	CS	$200K

DEPT_INFO:

DEPT	CITY	MGR
CS	San Jose	Peled

EMP_HOME:

EMP	STREET	CITY	CHILD
Fagin	162 Loma Alta	Los Gatos	Joshua

Figure 10.1

EMP_WORK:

EMP	DEPT	SAL
Fagin	CS	$200K

DEPT_INFO:

DEPT	WORK_CITY	MGR
CS	San Jose	Peled

EMP_HOME:

EMP	STREET	HOME_CITY	CHILD
Fagin	162 Loma Alta	Los Gatos	Joshua

Figure 10.2

We have seen that in Figure 10.1, there is not a unique {EMP, CITY} relationship. However, assume that we were to rename the attribute CITY in the DEPT__INFO relation scheme to be WORK__CITY, and the attribute CITY in the EMP__HOME relation scheme to be HOME__CITY (see Figure 10.2). There is now a unique {EMP, WORK__CITY} relationship, which includes the tuple (Fagin, San Jose), and a unique {EMP, HOME__CITY} relationship, which includes the tuple (Fagin, Los Gatos). Although the database scheme of Figure 10.1 is γ-cyclic (and even α-cyclic), the database scheme of Figure 10.2 is γ-acyclic.

Theorem 10.1 [Fa3] A database scheme is γ-acyclic if and only if for every set X of attributes, and for every consistent database over the scheme, there is a unique X-relationship.

Knowing that relationships are unique make it possible to greatly simplify the form of queries. Thus, the simplest SQL [C*] query to find all EMPs associated with the WORK__CITY San Jose for the database scheme of Figure 10.2 is:

```
SELECT EMP
FROM EMP__WORK, DEPT__INFO
WHERE EMP__WORK.DEPT = DEPT__INFO.DEPT
AND DEPT__INFO.WORK__CITY = 'San Jose'.
```

However, by uniqueness, it is possible instead to unambiguously pose the query

```
SELECT EMP WHERE WORK__CITY = 'San Jose'.        (10.1)
```

The result is obtained by finding the unique {EMP, WORK__CITY} relationship, and then selecting out those tuples where the CITY entry is 'San Jose'. The desirability of being able to pose queries such as (10.1), with such a simple syntax, has been discussed by Ullman [Ul]. Not only is the query (10.1) easier to pose and simpler to understand than the SQL query, but also the system has a great deal of flexibility in optimizing how to find the result of the query. The system's choice of which relations to join (if there are several possibilities) might depend, for example, on which indices are present.

Languages such as SQL are considered "high-level", since it is not necessary to explicitly state the access paths (such as which indices to utilize). Similarly, we have seen that in a γ-acyclic database scheme, it is possible to make use of a still higher-level language, in which it is not even necessary to specify which relations must be joined to obtain the answer the user desires.

We now discuss another desirable property of database schemes which is equivalent to γ-acyclicity. Recall that we stated earlier (Section 5) that a database scheme is α-acyclic if and only if some join expression is monotone. We have the following stronger result for γ-acyclic schemes. (In this theorem, a *connected* join expression is one for which every pair of relations that are joined have at least one attribute in common; thus, no Cartesian products are taken.)

Theorem 10.2 [Fa3] A database scheme is γ-acyclic if and only if every connected join expression is monotone.

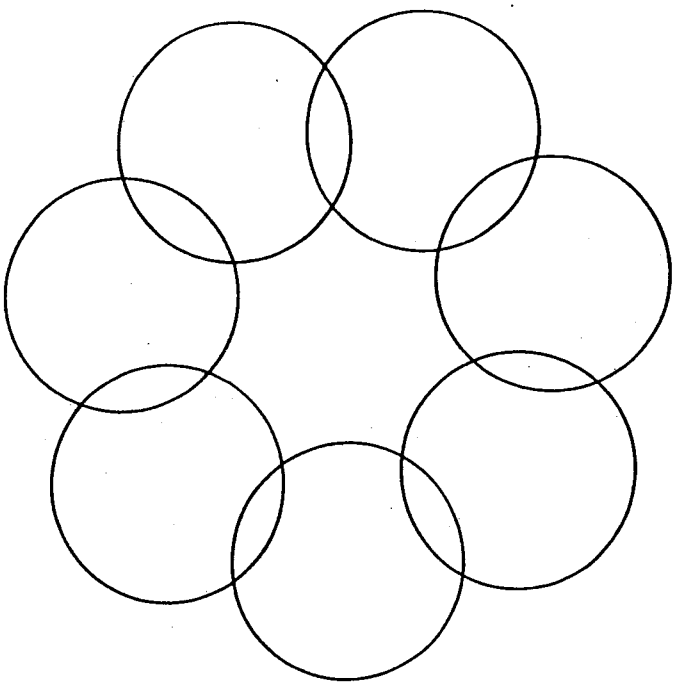

Figure 10.3

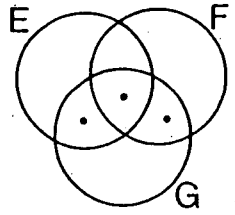

Figure 10.4

Thus, for α-acyclic schemes, there is *some* monotone join expression; for γ-acyclic schemes, *every* reasonable (i.e., connected) join expression is monotone.

Theorem 10.2 gives an example of how γ-acyclicity can help the *system*. For, the system can decide how to take joins, and it is guaranteed to be a monotone-increasing join (if the database is consistent, and if the system does not try to take joins "foolishly", that is, by taking a Cartesian product.) On the other hand, Theorem 10.1 gives an example of how γ-acyclicity can help the *user*, since it is possible for the user to use a very high-level query language.

We close this section by defining γ-acyclicity. A hypergraph is *γ-acyclic* if it has no pair (E,F) of incomparable, nondisjoint edges such that in the hypergraph that results by removing $E \cap F$ from every edge, what is left of E is connected to what is left of F. (E and F are said to be *incomparable* if $E \not\subseteq F$ and $F \not\subseteq E$.) Again, a hypergraph (or scheme) is *γ-cyclic* precisely if it is not γ-acyclic. As in the case of β-acyclicity, several equivalent definitions of γ-acyclicity are known [Fa3]; we shall give another one, involving hidden configurations, along with an algorithm for determining γ-acyclicity. This algorithm was discovered by D'Atri and Moscarini [DM], and is in the spirit of Graham's algorithm. We remark that there is also a characterization of γ-cyclicity that is very much like the characterization given in Theorem 9.2 for β-cyclicity.

Let $(S_1, ..., S_m, S_{m+1})$ be a sequence of sets, where $S_1, ..., S_m$ are distinct, and where $S_{m+1} = S_1$. Let us call S_i and S_{i+1} *neighbors* $(1 \leq i \leq m)$; note in particular that S_m and S_1 are neighbors. Let us call $(S_1, ..., S_m, S_{m+1})$ a *pure cycle* if $m \geq 3$ (that is, at least three sets are involved), and if whenever $i \neq j$, then $S_i \cap S_j$ is nonempty if and only if S_i and S_j are neighbors. Thus, a pair is nondisjoint precisely if it is a neighboring pair. Furthermore, if $m=3$, then we assume also that $S_1 \cap S_2 \cap S_3$ is empty. If $m \geq 4$, then the comparable assumption (that is, the assumption that $S_1 \cap ... \cap S_m$ is empty) is unnecessary, since it is a consequence of our other assumptions. A pure cycle with seven edges appears in Figure 10.3, where two edges have nonempty intersection if and only if they are shown to intersect in Figure 10.3.

Theorem 10.3 [Fa3] A hypergraph is γ-cyclic precisely if it contains at least one of two kinds of "forbidden configurations" of edges: either a pure cycle, or else a set of three edges that intersect at least as shown as in Figure 10.4. (By the latter, we mean that in Figure 10.4, there is at least one node in $E \cap F \cap G$, there is at least one node in $(E \cap G)-F$, and there is at least one node in $(F \cap G)-E$. Other intersections involving combinations of E, F, and G may also occur.)

It is not obvious from these characterizations that there is a polynomial-time algorithm for determining γ-acyclicity. However, this is the case.

Theorem 10.4 ([DM], [Fa3]) A hypergraph is γ-acyclic if and only if the following algorithm says that it is.

Algorithm for determining γ-acyclicity: Apply the following operations repeatedly, in any order, until none can be applied:

(a) if a node is isolated (that is, if it belongs to precisely one edge), then delete that node;

(b) if an edge is a singleton (that is, if it contains exactly one node), then delete that edge (but do not delete the node from other edges that might contain it);

(c) if an edge is empty, then delete it;

(d) if two edges contain precisely the same nodes, then delete one of these edges;

(e) if two nodes are edge-equivalent, then delete one of them from every edge that contains it. (We say that two nodes are *edge-equivalent* if they are in precisely the same edges.)

The algorithm clearly terminates. If the end result is the empty set of edges, then the original hypergraph is γ-acyclic; otherwise, it is γ-cyclic.

Note: We shall often apply rule (d) implicitly, by simply dealing at all times with a *set* of edges (which has the effect of automatically removing duplicates). Also, it is natural to apply rule (c), the deletion of an empty edge, implicitly.

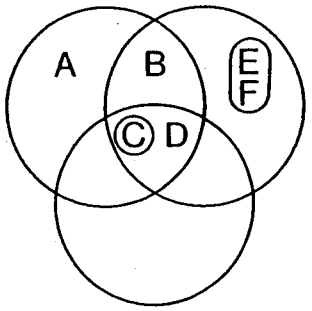

Figure 10.5

Example 10.5 Let us apply this algorithm to the hypergraph of Figure 10.5. The edges are:

	B	C	D	E	F
A	B	C	D		
		C			
		C	D		
				E	F

(For convenience, we have put common vertices in the same column.) Node A is isolated, and edge {C} is a singleton, so both are deleted, by rules (a) and (b). This leaves us with

	B	C	D	E	F
	B	C	D		
		C	D		
				E	F

Nodes E and F are edge-equivalent, and so, by rule (e), we delete F from both edges that contain it. Similarly, nodes C and D are edge-equivalent, and so we delete D from all three edges that contain it. We are left with

B C E
B C
C
E

The third and fourth edges above are singletons, and so they are eliminated. This leaves

B C E
B C

Node E is isolated; after it is deleted, we are left with

B C
B C

These edges are identical, so we delete one by rule (d). We are left with

B C

Both nodes are now isolated, and so they are deleted. We are left with a single empty edge, which is deleted by rule (c). The end result is the empty set of edges, and so the original hypergraph is γ-acyclic. □

11. Closing viewpoint

Various degrees of acyclicity are each equivalent to various desirable properties for database schemes. The choice of database scheme (the "database design") might be influenced by trying to obtain some such desirable property. This is quite analogous to "normalization" of a single relation schema (the relation scheme along with its set of dependencies) ([Co2], [Fa2]). For, there is a hierarchy of normal forms for relation schemas, each normal form being more restrictive than its predecessor. Codd has argued that we should not *insist* that a relation schema be in a given normal form. Rather, the database designer should be aware of the issues, and have a warning flag that if the relation schema is not in a given normal form, then certain problems may arise. An identical comment applies to the question of whether a database scheme should obey a given degree of acyclicity. Fortunately, there is a polynomial-time algorithm for determining the degree of acyclicity of a database scheme. Shel Finkelstein [Fi] has observed that an especially appropriate situation in which to strive for, say, γ-acyclic schemes, is in the case of a user view, where the number of attributes is small enough that it might be quite reasonable to obtain γ-acyclicity (perhaps by renaming a few attributes). Then the user can have the advantages of γ-acyclicity, such as the ability to use a very high-level language.

12. Acknowledgment

The author is grateful to Moshe Vardi for helpful comments.

REFERENCES

[ABU] Aho, A. V., C. Beeri, and J. D. Ullman, "The theory of joins in relational databases," ACM Trans. on Database Systems 4,3 (1979), 297-314.

[ADM] Ausiello, G., A. d'Atri, and M. Moscarini, "Minimal coverings of acyclic database schemata", Proc. ONERA-CERT Toulouse Workshop on Logical Bases for Data Bases (Dec. 1982).

[BDM] Batini, C., A. D'Atri, and M. Moscarini, "Formal tools for top-down and bottom-up generation of acyclic relational schemata", Proc. 7th Int. Conf. on Graph-Theoretic Concepts in Computer Science, Linz (1981).

[BFMMUY] Beeri, C., R. Fagin, D. Maier, A. O. Mendelzon, J. D. Ullman, and M. Yannakakis, "Properties of acyclic database schemes," Proc. Thirteenth Annual ACM Symposium on the Theory of Computing, 355-362 (1981).

[BFMY] Beeri, C., R. Fagin, D. Maier, and M. Yannakakis, "On the desirability of acyclic database schemes," J. ACM, to appear.

[BC] Bernstein, P. A. and D. W. Chiu, "Using semi-joins to solve relational queries", J. ACM 28, 1 (Jan. 1981), 25-40.

[BG] Bernstein, P. A. and N. Goodman, "The power of natural semijoins" SIAM J. Computing 10,4 (Nov. 1981), 751-771.

[BB] Biskup, J. and H. H. Bruggemann, "Towards designing acyclic database schemas", Proc. ONERA-CERT Toulouse Workshop on Logical Bases for Data Bases (Dec. 1982).

[C*] Chamberlin, D.D., M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner, and B. W. Wade, "SEQUEL 2: A unified approach to data definition, manipulation, and control," IBM J. of Research and Development 20,6 (Nov. 1976), 560-575.

[Ch] Chase, K., "Join graphs and acyclic data base schemes", Proc. 1981 Very Large Data Bases Conf., 95-100.

[Co1] Codd, E. F., "A relational model for large shared data banks," Comm. ACM 13,6 (1970), 377-387.

[Co2] Codd, E. F., "Further normalization of the database relational model," Courant Computer Science Symposia 6: Data Base Systems, (May 24-25, 1971), Prentice-Hall, 65-98.

[DM] D'Atri, A. and M. Moscarini, "Acyclic hypergraphs: their recognition and top-down vs bottom-up generation," Consiglio Nazionale Delle Ricerche, Istituto di Analisi dei Sistemi ed Informatica, R.29 (1982).

[Fa1] Fagin, R., "Multivalued dependencies and a new normal form for relational databases," ACM Trans. on Database Systems 2,3 (1977), 262-278.

[Fa2] Fagin, R., "A normal form for relational databases that is based on domains and keys," ACM Trans. on Database Systems 6,3 (Sept. 1981), 387-415.

[Fa3] Fagin, R., "Degrees of acyclicity for hypergraphs and relational database schemes". To appear, J. ACM.

[FMU] Fagin, R., A. O. Mendelzon, and J. D. Ullman, "A simplified universal relation assumption and its properties," ACM Trans. on Database Systems 7,3 (Sept. 1982), 343-360.

[Fi] Finkelstein, S., private communication.

[GS1] Goodman, N., and O. Shmueli, "The tree property is fundamental for query processing", Proc. First ACM SIGACT-SIGMOD Principles of Database Systems (1982), Los Angeles, 40-48.

[GS2] Goodman, N., and O. Shmueli, "Transforming cyclic schemas into trees", Proc. First ACM SIGACT-SIGMOD Principles of Database Systems (1982), Los Angeles, 49-54.

[GS3] Goodman, N., and O. Shmueli, "Tree queries: a simple class of queries," ACM Trans. on Database Systems 7,4 (Dec. 1982), 653-677.

[GST] Goodman, N., O. Shmueli, and Y. C. Tay, "GYO reductions, canonical connections, and cyclic schemas and tree projections", Proc. Second ACM SIGACT-SIGMOD Principles of Database Systems (1983), Atlanta, 267-278.

[Gr] Graham, M. H., "On the universal relation," Technical Report, Univ. of Toronto (Sept. 1979).

[GP] Gyssens, M. and J. Paredaens, "A decomposition methodology for cyclic databases", Proc. ONERA-CERT Toulouse Workshop on Logical Bases for Data Bases (Dec. 1982).

[Ha] Hanatani, Y., "Eliminating cyclies in database schemas", Proc. ONERA-CERT Toulouse Workshop on Logical Bases for Data Bases (Dec. 1982).

[HLY] Honeyman, P., R. E. Ladner, and M. Yannakakis, "Testing the universal instance assumption," Inf. Proc. Letters, 10:1 (1980), 14-19.

[Hu] Hull, R., "Acyclic join dependency and database projections," USC Technical Report (June 1981).

[MSY] Maier, D., Y. Sagiv, and M. Yannakakis, "On the complexity of testing implications of functional and join dependencies", J. ACM 28,4 (Oct. 1981), 680-695.

[MU] Maier, D. and J. D. Ullman, "Connections in acyclic hypergraphs," Proc. First ACM SIGACT-SIGMOD Principles of Database Systems (1982), Los Angeles, 34-39.

[OSS] Ozkarahan, E. A., S. A. Schuster, and K. C. Sevcik, "Performance evaluation of a relational associative processor", ACM Trans. on Database Systems 2,2 (June 1977), 175-196.

[Ri] Rissanen, J., "Theory of relations for databases - a tutorial survey." Proc. 7th Symp. on Math. Found. of Comp. Science, Lecture Notes in Comp. Science, 64 (1978), Springer-Verlag, 537-551.

[R*] Rothnie, J. B. Jr., P. A. Bernstein, S. Fox, N. Goodman, M. Hammer, T. A. Landers, C. Reeve, D. W. Shipman, and E. Wong, "Introduction to a system for distributed databases (SDD-1)", ACM Trans. on Database Systems 5,1 (Mar. 1980), 1-17.

[Sa] Sacca, D., "On the recognition of coverings of acyclic database hypergraphs", Proc. Second ACM SIGACT-SIGMOD Principles of Database Systems (1983), Atlanta, 297-304.

[TY] Tarjan, R. E., and Yannakakis, M., "Simple linear-time algorithms to test chordality of graphs, test acyclicity of hypergraphs, and selectively reduce acyclic hypergraphs", Bell Labs Technical Report (March 1982), Murray Hill, New Jersey.

[Ul] Ullman, J. D., "The U.R. strikes back," Proc. First ACM SIGACT-SIGMOD Principles of Database Systems (1982), Los Angeles, 10-22.

[Ya] Yannakakis, M., "Algorithms for acyclic database schemes," Proc. 1981 Very Large Data Bases Conf., 82-94.

[YO] Yu, C.T. and M.Z. Ozsoyoglu, "An algorithm for tree-query membership of a distributed query", Proc. 1979 IEEE COMPSAC, 306-312.

[Za] Zaniolo, C., Analysis and design of relational schemata for database systems, Ph.D. Dissertation, Tech. Rep. UCLA-ENG-7669, UCLA, July 1976.