Introduction

In this paper we propose a systematic classification of the types of relationships that can occur between any two relations in a relational data base [9, 14].

Many types of relationships can occur in relational data bases [12, 13]. Nevertheless, although relationships are fundamental in data base theory [2, 4, 8], and practice [20, 21, 27, 31, 32], there appears to have been no systematic attempt in the literature to classify them. There are probably two explanations for this neglect. First, the basic relational database manipulation languages, such as DSL Alpha [12, 13], SQL [10, 20, 21], QUEL [27, 28], and relational algebra [14, 15] can be used with any relationship type without any requirement of distinction or classification. A second explanation is that there is a significant body of theoretical data base research devoted to elimination of the need for navigation between relations, that is, of the need to specify relationships between relations in any way in language expressions [19, 22, 23, 29]. This type of research is based on the idea of manipulating a universal relation, formed from the underlying relations of the data base by means of relational algebraic join operations [1, 3, 14, 18]. A possible motive for the universal relation approach is that it has proven difficult to specify many important relationship types using the basic relational languages and dialects [6, 31].

In the commercial world, where nowadays the files of non relational data bases have become relational in structure, relationships between relations are used extensively [5, 15, 30]. In addition, relationships are an integral part of the entity/relationship approach [11]. Relationships are also necessary with natural quantifier relational languages [6]. Consequently, there is reason to believe that relationships between relations are important, so that there are good grounds for both research directed towards a better understanding of them, and the development of techniques that make complex relationships more amenable to use with database manipulation languages.

There exists a class of higher level relational data base languages that make use of relationships in a manner that is far more concise than with conventional data base languages. They do this by permitting the use of the wide range of natural quantifiers of natural languages. An example of this type of langauge is SQL/NQ [4, 6, 7], a research language that is upward compatible with SQL. Languages of this type require that relationships be specified outside of language expressions, either in the schema, in host programs, or interactively, and it is research into such specifications that has led the author to the classification proposed in this paper.

1.1 Notation

We use upper case letters for attributes or attribute concatenations, except where otherwise stated. Upper case bold is used for relation names. Relation schemes are implied throughout, but are not named [28]. We use subscriped lower case for attribute values within a tuple. Thus the relation scheme [T, U, V, W] could give rise to a relation (instance) **T**(T, U, V, W), which may have a tuple t(t_1, u_1, v_1, w_1). The primary key attribute (or any canditdate key attribute) is underscored, and for reader convenience will always have the same upper case letter as the relation name.

1.2 The relationship concept with relational data bases

No formal definition of a relationship between relations appears to have been proposed in the literature, although the term is used frequently. The following definition is consistent with the largely intuitive use of the concept to date.

Definition 1

Consider a data base with a set of relations [**A**, **B**, **C**, ...]. There is a relationship between any arbitrary pair of relations (**A**, **B**) of this data base iff, by a process of relational algebraic operations, involving only relations from in [**A**, **B**, **C**, ...], it is possible to generate a relation **R**(A, B, ...), with A, B as minimal attributes, and where R is not necessarily a relation of the database. **R** may be empty.

We refer to **R** as a relationship relation, and essentially this paper deals with a systematic classification of the kinds of **R** that can be generated from relational data bases.

1.3 Overview of major classification components

The main classification is into primitive and composite (non primitive) relationships. Briefly, a relationship between **A** and **B** is primitive iff it as the direct result of a join on attributes of **A** and **B** alone. The major components of the classification are thus:

Class 1 **Primitive relationships**

(a) Simple one-to-many, or simple 1:n relationships

(b) Co-relationships

Class 2 **Composite relationships**

(a) Composite one-to-many, or composite 1:n relationships

(b) Simple matrix, or simple many-to-many relationships

(c) Composite matrix, or composite many-to-many relationships

Note that the list of primitive and composite relationships above is not exahustive. Those listed are merely the more common. In addition, most of the above association types can be further classified into cyclic and non cyclic versions, as will be analysed in later sections. Thus it is legitimate, in this context, to refer to a cyclic (or recursive),many-to-many relationship.

2. CLASS 1 - PRIMITIVE RELATIONSHIPS

Before looking at the different kinds of primitive relationship in detail, we need more formal definitions of both primitive and composite relationships.

Suppose that we use the symbol ¶ for algebraic projection, and the symbol * for any arbitrary join operation, whether equality based or otherwise [1, 3, 14]. Consider any two relations **A, B** from a database.

Definition 2 Iff two relations **A, B** are such that $\P_{A,B}(\mathbf{A} * \mathbf{B})$ generates a relation **R**(A,B), which may be empty, then the relationship described by **R** is primitive.

The possibility that **R** may be empty is reasonable. Admittedly, with specific relation instances, if **R** is empty, then it will not be possible to pair or associate any tuples from those instances. This does not invalidate the generally accepted notion of a relationship, however, for a relationship is generally accepted to be a constant property of a pair of relations, and not just instances of the relations. It cannot be that a simple update to a relation could also eliminate a relationship. Suppose that we are using an equijoin on attributes F in **A** and **G** in **B**. Suppose that with * as an equijoin $\P_{A,B}(\mathbf{A} * \mathbf{B})$ generates an non empty **R**. Sometime later, following update of the two relations, it is possible that G and F will be disjoint sets, so that (**A** * **B**) and consequently **R** will be empty. If we do not admit that the relationship still exists when **R** is empty, then we must accept that a simple update to the data base can eliminate, or create a relationship. Our definition above does not permit this.

Definition 3 A relationship that is not primitive is composite.

Theorem 1 There will be a composite relationship between **A** and **B** iff

$$\P_{A,B}(\mathbf{A} * \mathbf{C} * \mathbf{D} * \ldots * \mathbf{B}) = \mathbf{T}(A,B)$$

where **T** can be an empty relationship relation.

Proof Since **T** exists, even if empty, there is a relationship between **A and B**, by Definition 1. Since **T** is not generated by a projection of a join of **A** with **B**, by Definitions 2 and 3, **T** must describe a composite relationship.

Corollary There is a composite relationship between relations **A** and **B** if **A** and **B** are at either end of a chain of primitive relationships.

Proof Let **X** ⊛ **X** denote the equijoin of any relation **X** with X as the join attribute.

$$\P_{A,B}(\mathbf{A} * \mathbf{C} * \mathbf{D} * \ldots \mathbf{M} * \mathbf{B}) =$$

$$\P_{A,B}(\mathbf{A} * \mathbf{C} \circledast \mathbf{C} * \mathbf{D} \circledast \mathbf{D} * \ldots \mathbf{M} \circledast \mathbf{M} * \mathbf{B}) =$$

$$\P_{A,B}[(\P_{A,C}(\mathbf{A} * \mathbf{C})) \circledast (\P_{C,D}(\mathbf{C} * \mathbf{D})) \ldots \circledast (\P_{M,B}(\mathbf{M} * \mathbf{B}))]$$

which forms a chain of primitive relationships, by Definition 2.

Theorem 2 There is a primitive relationship between **A** and **B**, iff **A** and **B** each have an attribute drawn on the same domain.

Proof If eac h of **A** and **B** has a common domain, then a join operation is allowed. If we denote $*_{G,C}$ as the join operation based on join attributes G and C, then if G and C are the common domain attributes of **A** and **B**, the computation $\P_{A,B}(\mathbf{A} *_{G,C} \mathbf{B})$ will yield a relation **R(A, B)**.

2.1 Simple one-to-many relationships

In the following, whenever we use the term attribute, it is to be assumed that the term attribute concatenation could also apply.

Consider a pair of attributes from **A, B** drawn on a common domain.

Definition 4 If at least one of the common domain attributes in a primitive relationship is a candiate key, that is, if one attribute has only unique values, then the relationship is simple one-to-many (1:n).

This means that if A is one of the common domain attributes, and the other one is some attribute G in **B**, then the relationship relation **R**(A,B) is equal to $\P_{A,B}(\mathbf{A} *_{A,G} \mathbf{B})$, where $*_{A,G}$ denotes an equijoin on the attributes A, G. Since G values are not unique in **B**, it follows from the mechanism of an equijoin that B values are unique, so that given a B value in an **R** tuple, the A value is determined (but

not vice versa). Accordingly, **R** implies a function r:B → A.

The one-to-many relationship between **A** and **B** is displayed in Figure 1, using a directed graph. A node shows a relation with its attributes, and the association is depicted by an edge that is directed from the key attribute A to the other join attribute G. This graphical depiction, and the arrow direction, is based more on tradition than anything else [4, 5, 15, 28, 31].

The relationship can also be usfully depicted using a co-ordinate system, and the relationship relation **R**(A,B). The B values are assumed laid out along the x-axis (Figure 2) in ascending G value order, that is, the function implicit in **R**(A,B) is assumed derived from $\Pi_{A,B}(\mathbf{A} *_{A,G} \mathbf{B}^{G})$, where $\mathbf{B}^{G}$ is simply **B** with the tuples sorted in ascending G value order.

A trivial case of one-to-many relationship occurs when G is also a (candidate) key attribute, so that G values, as well as A values, are unique. In such a case the function r:B → A implicit in **R**(A,B) is simply a trivial one-to-one function, not included in the classification as a distinct relationship type.

2.2 Co-relationships

Another type of primitive relationship between any two relations **A**, **B** occurs when neither of the two attribute in the pair with a common domain are candidate key attributes.

Definition 5 A primitive relationship between any two relations **A**, **B**, is a co-relationship, iff neither of the common domain attributes supporting the relationship is a primary or candidate key.

In other words, the relationship is supported by a pair of attributes, each with values are that are not unique. Thiis type of relationship has similarities with the much investigated concept of a multivalued dependency.

Lemma 1 The relationship relation **R** for a co-relationship between any two relations **A**, **B** is partitioned by the values of the attributes supporting the relationship.

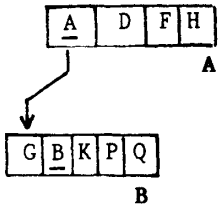


Figure 1

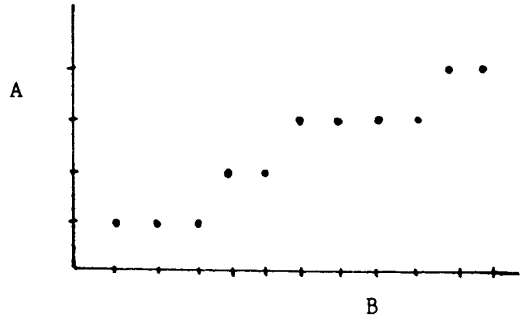


Figure 2

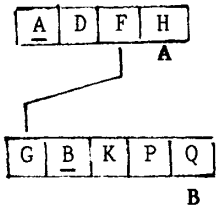


Figure 3

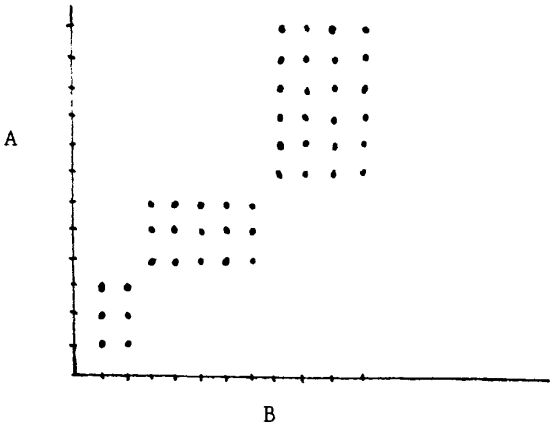


Figure 4

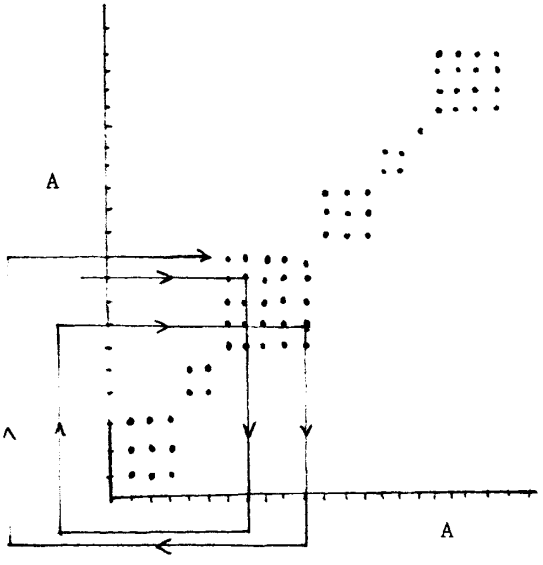


Figure 5

Proof Suppose that F in **A**, and G in **B** are the attributes supporting the relationship. Let $[a_1, a_2 \ldots a_i]$ be the set of A values in all those **A** tuples with a F value g_k, and similarly $[b_1, b_2, \ldots b_j]$ is the set of B values in all those **B** tuples with an equal G value g_k.

If we now take $\P_{A,B}(\mathbf{A} *_{F,G} \mathbf{B})$, then in order to form the relationship relation **R**(A, B), each of the set of A values $[a_1 \ldots a_i]$ must be paired with each of the set of B values $[b_1, \ldots b_j]$ to give us a relation $[(a_1,b_1), (a_1,b_2), \ldots (a_i,b_j)]$ that is a subset of **R**(A,B). No other tuples of **R** can contain either $[a_1, \ldots a_i]$ values or $[b_1, \ldots b_j]$ values, so that **R** must contain a subset of tuples for each value common to both F and G, that is, for each member of the set $\P_F\mathbf{A} \quad \P_G\mathbf{B}$. It follows that each common F and G value gives rise to a block of **R** tuples, and thus a partition of **R**.

Theorem 3 For a co-relationship between relations **A** and **B**, within a block of the partition of **R**: Iff tuples (a_1, b_1) and (a_2, b_2) exist, then tuples (a_1, b_2) and (a_2, b_1) also exist.

Proof By Lemma 1, the attribute value supporting the relationship between **A** and **B** tuples within a block of a partition of **R** is a constant g_k. Accordingly, there exists **A** tuples $(a_1, \ldots g_k)$, $(a_2, \ldots g_k)$ and **B** tuples $(b_1, \ldots g_k)$, $(b_2, \ldots g_k)$. If we apply the operation for generating the relationship relation **R** to these tuples, namely $\P_{A,B}(\mathbf{A} *_{F,G} \mathbf{B})$, then we must get tuples (a_1, b_2) and (a_2, b_1) in **R**.

This is reminiscent of the condition for a multivalued dependency. In fact, if we add the common F and G attribute to **R**, then this ternary relation will contain a multivalued dependency, as proved in [7]. In otherwords, if we rename the F attribute in **A** as G, and form the relation $\mathbf{R_m}$ from $\P_{A,G,B}(\mathbf{A} *_G \mathbf{B})$, where $*_G$ denotes a natural join on the G attribute, then it can be shown that for $\mathbf{R_m}$ the following condition holds:

"Iff tuples (a_1, g_k, b_1) and (a_2, g_k, b_2) exist, then tuples (a_1, g_k, b_2) and (a_2, g_k, b_1) also exist."

This is the well known condition for a multivalued dependency.

Returning to the relationship relation **R** for a co-relationship between relations **A** and **B**, supported by non key attributes F and G, a graphical display, as in Figure 3, is useful. Because of the defining condition for **R**(A, B), there does not exist an implicit function r:A → B or r:A ↛ B. However, if we order the tuples of **R** so that tuples of the same partition block are adjacent, and use a coordinate system to display $\P_A(\mathbf{R})$ versus $\P_B(\mathbf{R})$, we get an interesting geometry of rectangles (Figure 4). Each rectangle represents a block of the partition of **R**, and a point within a rectangle represents a pair of related tuples.

Example Each tuple of **A** described a carrier based aircraft, identified by A. The attribute F in **A** identifies the carrier on which the aircraft is based. Each tuple of **B** identifies a crew member of a carrier, and G in **B** identifies the carrier to which the crew member is assigned. If crew-member b_4 and aircraft a_2 are respectively assigned to and based on the same carrier g_k, then they are related, as are the tuples describing them. The relationship is clearly not as "strong" as that of a one-to-many relationship, but is clearly important nevertheless.

2.3 Equivalence relationship

An equivalence relationship is essentially an implicitly cyclic relationship with no corresponding non cyclic version. A relationship is cyclic if is between two identical relations. The relationship is non complex.

Definition 6 A primitive relationship between two identical relations is an equivalence relationship iff a common attribute in each of the two relations supports the relationship, and that attribute is neither a primary nor candidate key.

Theorem 4 The relationship relation **R**(A, A) for an equivalence relationship of the relation **A** supported by the attribute G is an equivalence relation, (in the mathematical sense of the term), such that there is a block of a partition of the relation for each G value.

Proof We have $\mathbf{R}(A, A) = \pi_{A,A}(\mathbf{A} *_G \mathbf{A})$

Suppose that we have the set of A values $[a_1, \ldots a_i]$ for a given G value g_k. Then in forming **R** tuples, each of the set of A values $[a_1, \ldots a_i]$ is paired with every member of the set, to give the relation:

$[(a_1,a_1), (a_1, a_2), \ldots (a_i, a_i)]$

which is a subset of **R**(A, A). No other tuples can contain $[a_1, \ldots a_i]$ values. It follows that each G value gives rise to a block of **R** tuples and thus a partition of **R**. It follows that **R** is an equivalence relation.

The equivalence relationship is really the special case of a co-relationship between two identical relations (**A**, **A**) on a common attribute (G). Although it is not usual to display an equivalence relation on a cordinate system, it is intructive to do so in this case, as it gives a geometrical perspective on the difference between co-relationships and equivalence relationships.

In Figure 5 we order the tuples of the equivalence relation **R**(A, A) so that tuples with A values from **A** tuples with the same G value lie adjacent, and then display A versus A on the cordinate system. We see that we get a system of squares, with each square representing a block of the partition of **R**. A point within a square represents a pair of tuples from **A**. The cyclic nature of the relationship should also be apparent. If we take a given tuple within a block of the partition (or point in a square of the co-ordinate display), such as (a_2,a_5), then we can find another tuple (a_5, a_3), and another (a_3, a_9), and so on in an undending sequence, as illustrated in Figure 5. Note, however, that given the partition of **R**, the sequence remains within a block of the partition.

2.4 Cyclic (or recursive) one-to-many relationships

Definition. Where **B** = **A** in a simple one-to-many relationship between **A** and **B**, then the relationship is cyclic (or recursive) one-to-many.

In order to be able to distinguish the relations in a cyclic one-to-many relationship, let us refer to the two related relations as **A** and $\mathbf{A}_G$, where the relationship is supported by the attribute A in **A**, and G in $\mathbf{A}_G$. The primary key attribute in $\mathbf{A}_G$ is then A_G iin conformance with the convention we are using for primary keys.

The relationship relation $\mathbf{R}(A, A_G)$ is then generated from

$$\P_{A,\ A_G}(\mathbf{A} *_{A,A_G} \mathbf{A}_G)$$

and this relationship relation contains the function $r:A_G \rightarrow A$. Since there can be many related $\mathbf{A}_G$ tuples for a single A tuple, the relationship is clearly 1:n in nature.

The relationship can be displayed as a directed graph, as in Figure 6a, where we draw two (identical apart from the name) versions of **A**, and in Figure 6b, where we use only one node (with two names **A**, and $\mathbf{A}_G$). It is also useful to display the relationship relation using a cordinate system, as in Figure 7. Here we assume that **R** was generated, using the expression above, from a version of $\mathbf{A}_G$ where the tuples are arranged in ascending or descending G order.

If we take any A value a_k, then, because of the 1:n nature of therelationship, there will in general exist a set of **R** tuples $[(a_k, a_1), (a_k, a_2), \ldots (a_k, a_j)]$, indicating a number of related $\mathbf{A}_G$ tuples for any given **A** tuple. But for any of these $\mathbf{A}_G$ tuples, such as that identified by a_2, for example, if regarded as an **A** tuple, then in turn it will be further related with a futther set of $\mathbf{A}_G$ tuples, and so on in cyclic fashion, as is also illustrated in Figure 7.

Note that if the relationship relation **R** contains the tuples (a_1, a_7) and (a_7, a_{13}), then the **A** tuple identified by a_1 will be related to the

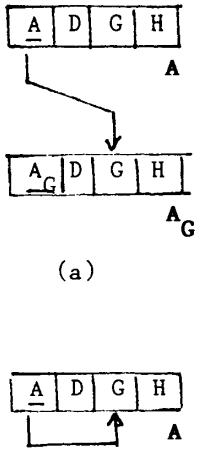


Figure 6

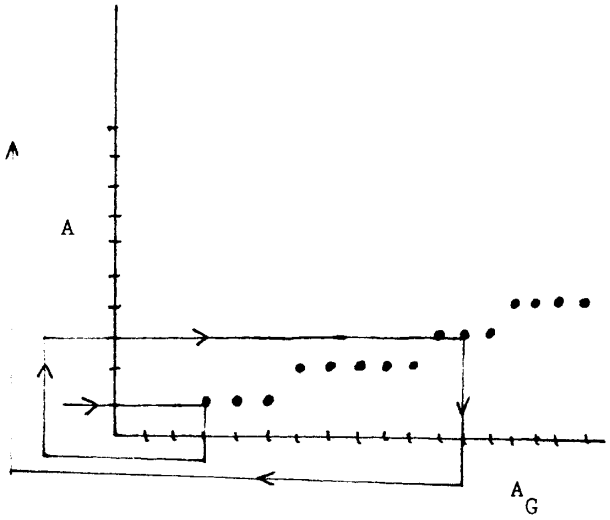


Figure 7

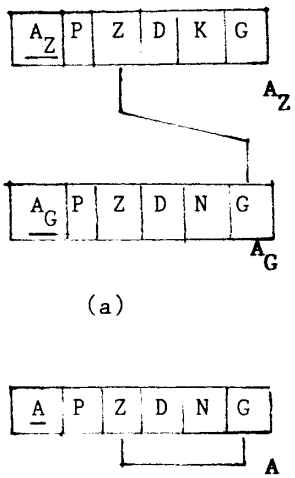


Figure 8

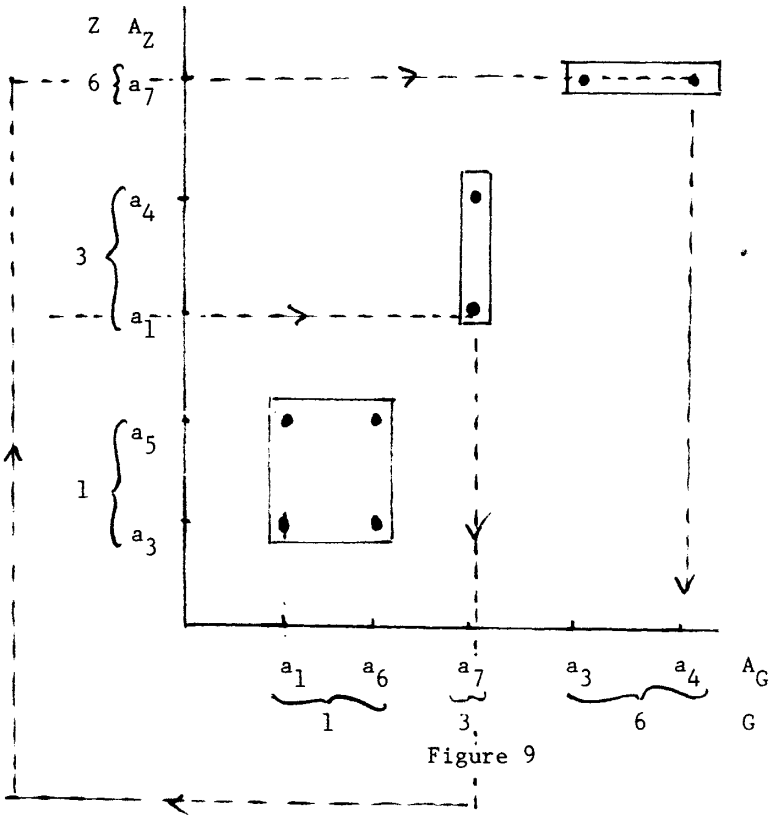


Figure 9

A tuple identified by a_{13}. However this relationship is not primitive, since the tuple (a_1, a_{13}) is not a member of $\mathbf{R}(A, A_G)$ generated from $\P_{A,A_G}(\mathbf{A} *_{A,G} \mathbf{A_G})$. This tuple can only be generated from:

$$\mathbf{R}(A, A_G) *_{A,A_G} \mathbf{R}(A, A_G)$$

confirming that the relationship involved (or sub or outer relationship) is non primitive, that is, it is composite. This is an admitted flaw in our so far "clean" classification of relationships into primitive and composite relationships. However, it is the only one, and is relatively minor. The difficulties resulting from other classifications attempted by the author are far greater.

Example The classic example is where an **A** tuple describes an employee in an organization, A gives the employee's identifying number and G give the identifying number of the employee's immediate superior. An employee is thus primitively (in a cyclic manner) related to his immediate superior, as are the corresponding tuples, but is compositely related to his immediate superior's superior, as are the corresponding tuples.

2.5 Cyclic co-relationships

Definition 8 When **B** = **A** in any co-relationship between **A** and **B**, then the relationship is a cyclic co-relationship, provided the attributes supporting the relationship are not the same.

It should thus be clear that we have a cyclic co-relationship between two identical relations **A, A** when the relationship is supported by two non key attributes Z and G drawn on the same domain. To distinguish the relations for analysis purposes we refer to the relations as $\mathbf{A_Z}$ and $\mathbf{A_G}$. An $\mathbf{A_Z}$ tuple is cyclic co-related to an $\mathbf{A_G}$ tuple if the Z value in the $\mathbf{A_Z}$ tuple has the same value as the G value in the $\mathbf{A_G}$ tuple. The relationship relation **R** is formed:

$$\mathbf{R}(A_Z, A_G) = \P_{A_Z,A_G}(\mathbf{A_Z} *_{Z,G} \mathbf{A_G})$$

From lemma 1 it follows that **R** is also partitioned, with a block of **R** tuples

for each common value of Z and G. From Theorem 3, within a block of the partition of **R**, that is, for a single value z_k of both Z and G, the following condition holds:

"Iff tuples (a_{z1}, a_{g1}) and (a_{z2}, a_{g2}) exist, then so do tuples (a_{z1}, a_{g2}) and (a_{z2}, a_{g1})."

And, despite the symmetry of the situation, if (a_{z1}, a_{g1}) is in a block of a partition of **R**, (a_{g1}, a_{z1}) is not necessarily in **R**. The existence of (a_{z1}, a_{g1}) means that the Z value of the **A** tuple identified by a_{z1} equals the G value of the **A** tuple identified by a_{g1}; however, there is no reason for the Z value of the tuple identified by a_{g1} to be also equal to the G value of the tuple identified by a_{z1}, the condition for the existence of (a_{i1}, a_{g1}) in **R**. This is best seen by the example of **A** and **R** below, which will also be useful again later for illustrating some further properties of this relation.

A	Z	G
a_1	3	1
a_3	1	6
a_6	9	1
a_5	1	6
a_7	6	3
a_4	3	6

A, $\mathbf{A_Z}$, $\mathbf{A_G}$

A_Z	A_G
a_3	a_1
a_5	a_1
a_3	a_6
a_5	a_6
a_1	a_7
a_4	a_7
a_7	a_3
a_7	a_4

$\mathbf{R}(A_Z, A_G)$

It should be clearly understood that a tuple (a_5, a_1) in **R**, for example, means that the Z value of the a_5 tuple in **A**, equals the G value of the a_1 tuple in **A**. In the example above, **R** has three partition blocks whose tuples all satisfy the existence rule given above, which followed from Theorem 3.

The relationship is displayed graphically in Figure 8a, 8b. If the tuples

of **R** are arranged so that all tuples belonging to the same block of the partition are adjacent, as in the example above, then when we use a co-ordinate system to display $\Pi_{A_Z}(\mathbf{R})$, or A_Z, versus $\Pi_{A_G}(\mathbf{R})$, or A_G, we once more get a system of rectangles (Figure 9), each rectangle representing a block of the partition of **R**.

The cyclic nature of the relationship also displayed in Figure 9. Suppose that in **R** we have a tuple (a_5, a_1), as in the example above, within a particular block of the partition of **R**, or rectangle (Figure 9, beginning at line linking points), then it is possible that there will exist a further tuple where the A_Z value is a_1, such as (a_1, a_7) in the example above, and then a tuple (a_7, a_4), and so on. This chain of related tuples will not in general remain within the original block of the partition or rectangle, as shown in Figure 9, since Z and G values within a single **A** tuple are not necessarily the same, and will indeed normally be different. This leads to an interesting theorem.

Theorem 5 A chain of related **R** tuples in the relationship relation **R** of a relation **A** participating in a cyclic co-relation, may be finite iff the set difference $\Pi_Z(\mathbf{A_Z}) - \Pi_G(\mathbf{A_G})$ is non empty.

Proof Suppose that $\Pi_z(\mathbf{A_Z}) - \Pi_G(\mathbf{A_G})$ is empty. Suppose that we have an arbitrary tuple (a_p, a_q) from **R**. There must therefore exist an **A** tuple witha G value equal to the Z value of the tuple identified by a_q. If this new **A** tuple is identified by a_r, then there exists an **R** tuple (a_q, a_r). In a similar manner there must exist an **R** tuple (a_r, a_s), and so on, so that the chain is infinite.

Now suppose that $\Pi_Z(\mathbf{A_Z}) - \Pi_G(\mathbf{A_G})$ is non empty, and that we have an arbitrary tuple (a_p, a_q). This time we cannot guarantee that there exists an **A** tuple with a G value equal to the Z value of the tuple identified by a_q, because, since the set of Z values in **A** does not match the set of G values in **A**, the Z value of the **A** tuple

identified by a_q may not exist in the set of G values. Accordingly, a chain may be finite.

As an example, using the tuples of the instance of **R** given earlier, the chain

(a_3, a_1); (a_1, a_7); (a_7, a_3); (a_3, a_6); $(a_6, ?)$

is finite, since the **A** tuple identified by a_6 has a Z value (9) that does not occur as a G value. Removal of this tuple from **A** would make the sets of Z and G value equal, so that all chains would be infinite.

Cyclic co-relationships occur reasonably frequently in data bases, although the relationships normally have a very low level of significance in practice.

Example A tuple of relation **A** describes a stolen car; A identifies a car, Z gives the (U.S.) state in which the car was stolen, and G gives the state of the dealership that originally dold the car. Thus a car a_5 stolen in state 1 is related to a car a_6 originally sold in state 1. The relationship will often have a significance that is merely coincidental, although in police or other kinds of investigations, the relationship may occasionally have more significance.

2.6 Co-relationships and significance levels

Although the topic is largely beyond the scope of this paper, and is thoroughly discussed elsewhere [7, 8, 9], , co-relationships may be further classified in accordance with the level of semantic significance involved. The lowest level of semantic significance is the coincidental level. Coincidental co-relationships are by far the most common. Informally, if two tuples are coincidentally co-related, the relationship is pure coincidence, and has no further significance. In the example given earlier about crew-members and aircraft assigned to carriers, the co-relationship between the crew-member and aircraft relations (supported by the carrier attribute G) would normally be coincidental. Thus the fact that a given grew-member and a given aircraft were assigned to the same carrier would be mere coincidence, in the normal sense of the term. However, if it were the

case that for each carrier, all crew members could operate the message decoders on all aircraft on the carrier, and only those aircraft, then the co-relationship would have a greater (and very meaningful) level of significance. The different levels of significance are defined in [7, 8].

It is shown elsewhere that a co-relationship can give rise to a connection trap [7]. Essentially, a connection trap will occur whenever users assume either a level of significance for the co-relationship that is too high, or a level of significance that is correct but is not supported by the data in the relations. In addition, there is a fundamental theorem about co-relationships and polygonal join dependencies, proved in [8]. Polygonal join dependencies are defined in [9]. This theorem states that for a closed chain of existential co-relationships, there will be a connection trap for any user attempting to extract reliable information from a complete join of the relations in the chain, unless the join contains a polygonal join dependency of order equal to the number of relations in the chain.

3 CLASS 2 - COMPOSITE RELATIONSHIPS

As stated by Theorem 2, a composite relationship between two relations **A** and **B** will involve a chain of primitive relationships. Since we can have different kinds of primitive relationships in the chain, different kinds of composite relationships are possible.

3.1 Composite 1:n relationships

Definition There is a composite 1:n relationship between relations **A** and **B**, with an implicit function r:B -> A, iff there exists a non empty set of relations [$\mathbf{C_1}$, $\mathbf{C_2}$, ... $\mathbf{C_n}$], with simple 1:n-relationships:

between **A** and $\mathbf{C_1}$, with implicit function r_1:C_1 -> A,

...

between $\mathbf{C_n}$ and **B**, with implicit function r_{n+1}:B -> C_n.

Composite 1:n relationships are quite simple in concept, are very common, and almost always meaningful. Readers can no doubt envisage many common examples. The relationship relation **R**(A, B) will clearly be given by:

$$\mathbf{R}(A, B) = \Pi_{A,B}(\mathbf{A} * C_1 * C_2 * \ldots * C_n * \mathbf{B})$$

with the attributes supporting the individual 1:n relationships as join attributes.

The composite 1:n relationship is a 1:n relationship,since each of the (primitive) 1:n relationships are additive. The relationship can be depicted by a directed graph, where any pair of connected nodes denotes a 1:n relationship, as in Figure 10.

3.2 Many-to-many or matrix relationships

Definition 10 There is a many-to-many relationship between **A** and **B**, when there exists a relation **M**, such that there is either a simple 1:n or composite 1:n relationship between:

(a) **A** and **M**, with implicit function r:M -> A, and

(b) **B** and **M**, with implicit function s:M -> B.

Many-to-many relationships are well known, and are often called matrix or n:m relationships. The Supplier-Parts matrix association due to Date [15] is almost the classic example. The relationship is fundamentally n:m, since for a given **A** tuple there must be many related **M** tuples and thus many related **B** tuples, and the converse.

It is clear that we can distinguish between many-to-many relationships where the relationships between **A** and **M**, and between **B** and **M** are merely simple 1:n, and the many types where one or both of these underlying relationships is a composite 1:n relationship. This is illustrated by the graps in Figures 11a and 11b. When we use a cordinate system to display the relationship, the result is a matrix, no matter how the $\Pi_A(\mathbf{R})$, $\Pi_B(\mathbf{R})$ are ordered.

Unlike other relationships so far discussed, the relationship relation for a many-to-many relationship is a ternary relation of the form **R**(A, M, B). A relationship relation of the form **R**(A, B) is insufficient. The difficulty arises as follows. Suppose that we have related **A** and **B** tuples, identified by a_1

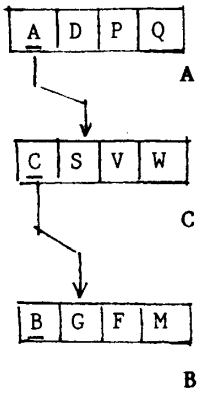
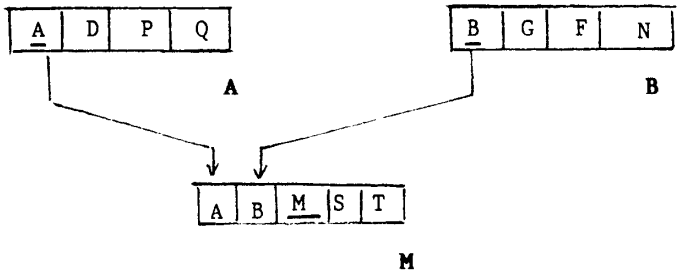
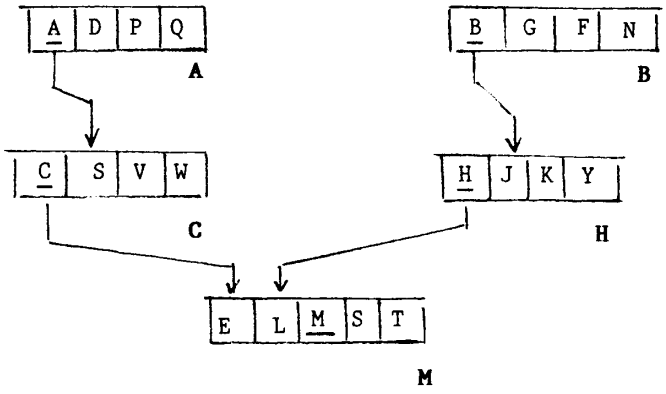


Figure 10



(a)



(b)

Figure 11

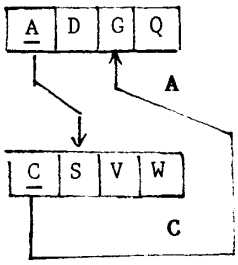


Figure 12

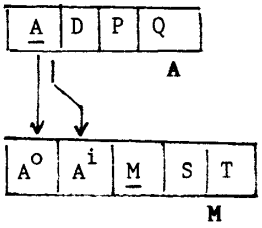


Figure 13

and b_1. For the a_1 tuple there can be many 1:n or composite related **M** tuples, for example, those identified by m_1, m_2, m_3, and m_4. Similarly, for the b_1 tuple in **B**, there can be many related **M** tuples, for example those identified by m_2, m_3, m_4, m_5, and m_6. Consequently, since the intersection of these two sets of **M** tuples are the tuples identified by m_2, m_3, and m_4, it follows that the a_1 tuple in **A** is matrix related with the b_1 tuple in **B** in three distinct ways, via m_2, via m_3, and via m_4.m_2, m_3, and m_4. Each of these tuple associations can be significantly different, and will need to be identified, so that the relationship relation **R** must contain the tuples:

$(a_1, m_2\ b_1)$

(a_1, m_3, b_1)

(a_1, m_4, b_1)

Example Suppose that an **A** tuple describes a supplier (of industrial part types), that a **B** tuple describes an industrial part type, and that an **M** tuple describes a shipment of a part by a supplier. Then supplier a_1 can ship part type b_1 in shipment m_2, and in shipment m_2 and in shipment m_3. There are thus three associations of supplier a_1 with part type b_1 and these are physically distinct and significant, and must be distinguished.

Because there can be more than one association between a given pair of **A** and **B** tuples, the term matrix relationship is something of a misnomer. In mathematics, given a matrix X_{nm}, there will be but one value of X_{nm} for given values of the n and m matrix coordinates (corresponding to A and B values), whereas we can have many associated **M** tuples, for given **A** and **B** tuples,

Obtaining the relationship relation **R**(A, B, B) is a simple matter when there are only (primitive) simple 1:n relationships between **A** and **M** and between **B** and **M**. Clearly, we must have:

$$\mathbf{R}(A, M, B) = \Pi_{A,M,B}(\mathbf{M})$$

In the case of composite 1:n relationship between either **A** and **M**, or between **B**

M, or both, the generation is more complex. The method requires a sequence of joins, beginning with **A** and ending with **B**, followed by a projection on A , M and B, or:

$$\mathbf{R}(A, M, B) = \Pi_{A,M,B}(\mathbf{A} * \mathbf{L}_1 * \ldots * \mathbf{L}_n * \mathbf{M} * \mathbf{R}_m * \ldots \mathbf{R}_1 * \mathbf{B})$$

The extensive computing needed to generate the relationship relation for cases like this has an important corrollary for language designers. Suppose that we are dealing with a conventional SQL system, and have a retrieval expression involving the relationship between **A** and **B**, where the relationship is a matrix relationship involving composite l:n relationships in the relationship chain. The entire chain will have to be specified in SQL, and where the chain takes the form **A, L_1, L_2, M, R_2, R_1, B,** we could have a retrieval request that is easily specified in English, such as:

> Retrieve the A value for each **A** tuple with a D attribute value 4, and where the majority of the (matrix) related **B** tuples have a P value of 10.

This would require the following lengthy SQL expression:

```
SELECT A FROM A, XA
WHERE D = 4
AND [SELECT COUNT(*) FROM B
     WHERE P = 10 AND
       B IN [SELECT B FROM R1
             WHERE R1 IN [SELECT R1 FROM R2
               WHERE R2 IN [SELECT R2 FROM M
                 WHERE L2 = [SELECT L2 FROM L2
                   WHERE L1 = [SELECT L1 FROM L1
                     WHERE A = XA.A]]]]]]
  >
 [SELECT COUNT(*) FROM B
  WHERE P ≠ 10 AND
  B IN [SELECT B FROM R1
```

```
WHERE R1 IN [SELECT R1 FROM R2
   WHERE R2 IN [SELECT R2 FROM M
      WHERE L2 = [SELECT L2 FROM L2
         WHERE L1 = [SELECT L1 FROM L1
            WHERE A = XA.A]]]]]]
```

In addition, each time a query involving this type of relationship is submitted, a computation needed for generating at least the equivalent of the relationship relation **R**(A, D, M, P, B) must be carried out. Since we have:

$$\mathbf{R}(A, D, M, P, B) = \Pi_{A,D,M,P,B}(\mathbf{A} * \mathbf{L_1} * \mathbf{L_2} * \mathbf{M} * \mathbf{R_2} * \mathbf{R_1} * \mathbf{B})$$

and since the above SQL expression is equivalent to:

```
SELECT A FROM R, XR
   WHERE D = 4
      AND [SELECT COUNT (*) FROM R
           WHERE P = 10
           AND A = XR.A]

          [SELECT COUNT(*) FROM R
           WHERE P ≠ 10
           AND A = XA.A]
```

the computation will be lengthy. [It is here that the universal relation approach has something to offer [19, 23], since the required result is also obtained from the short SQL expression above with the universal relation formed by a join of **A**, $\mathbf{L_1}$, $\mathbf{L_2}$, **M**, $\mathbf{R_2}$, $\mathbf{R_1}$ and **B** substituted for **R**(A, D, M, P, B) in the SQL expression.]

An alternative is the SQL/N approach, with an expression close in syntax to the original English-language expression:

```
SELECT A FROM [EACH] A [TUPLE]
WHERE D = 4
AND FOR MOST R RELATED B [TUPLES] (P = 10)
```

Here words surrounded by [] can be omitted, FOR MOST is a natural quantifier, and **R** is a schema specified name for the relationship relation.

R(A, M, B) for the matrix relationship between **A** and **B**. Execution of this SQL/N expression can be made very fast if **R** is stored in the storage schema. Otherwise **R** will have to be computed, a lengthy process, using information in the schema that specifies exactly what **R** is.

3.3 Cyclic composite 1:n relationships

Definition 11 When **B** = **A** in a composite 1:n association between **A** and **B**, the relationship is a cyclic (or recursive) composite 1:n relationship.

The relationship is very similar to the simple cyclic 1:n relationship (Section 2.4). To be able to distinguish the two identical relations, we refer to them as **A** and $\mathbf{A_G}$, where the relationship is supported by A in **A** and G in $\mathbf{A_G}$. The primary keys are then A in **A** and A_G in $\mathbf{A_G}$.

The relationship relation **R**(A, A_G) is then generated from:

$$\P_{A,A_G}(\mathbf{A} *_{A,G_1} \mathbf{R_1} *_{R_1,G_2} \mathbf{R_2} * \cdots \mathbf{R_n} *_{R_n,G} \mathbf{A_G})$$

where in relation $\mathbf{R_p}$, R_p is the primary key attribute, and G_p is an attribute drawn on the same domain as the primary key attribute R_{p-1} of $\mathbf{R_{p-1}}$. The relationship has thus an implicit function $r:A_G \rightarrow A$. It is depicted as a directed graph in Figure 12.

A given **A** tuple identified by a_k will be related with many **A** (that is, $\mathbf{A_G}$) tuples, and each of these in turn with many **A** tuples, exactly as with simple cyclic 1:n relationships.

Unfortunately, while of some interest theoretically, genuine cyclic composite 1:n relationships (not contrived) are rare indeed in practice, so the utility of this type of relationship is marginal at best.

3.4 Cyclic matrix (or n:m) relationships

Definition 12 When **B** = **A** in a matrix relationship between **A** and **B**, the relationship is cyclic (or recursive) matrix (or n:m).

Cyclic matrix relationships, in contrast to cyclic composite 1:n relationships, are quite important in practice, but normally only in those cases where the relationships between **A** and **M** are simple 1:n. The author has found no useful examples of a cyclic

matrix relationship involving simple 1:n relationships between **A** and **M**.

Using $\mathbf{A}^{o}$ and $\mathbf{A}^{i}$ to distinguish two otherwise identical instances of **A**, for the practical case where there are only (primitive) simple 1:n relationships between **A** and **M**, the relationship relation **R** is given by:

$$\mathbf{R}(A^{o}, M, A^{i}) = \Pi_{A^{o},M,A^{i}} (\mathbf{A}^{o} * \mathbf{M} * \mathbf{A}^{i})$$

$$= \Pi_{A^{o},M,A^{i}} (\mathbf{M}(A^{o}, M, A^{i}, \ldots))$$

A unique feature of this relationship is that it generates two different cascades of related tuples, for any given **A** tuple. By convention, in the direction $\mathbf{A}^{o}$ to $\mathbf{A}^{i}$, this cascade of related tuples is called an explosion, and in the opposite direction an implosion.

An explosion is generated as follows. Given an **A** tuple, identified as a_1^o, it will be associated with many $\mathbf{A}^{i}$ tuples, identified by the set of keys $[a_1^i, a_2^i, \ldots a_n^i]$. If we now consider each of these $\mathbf{A}^{i}$ tuples as a $\mathbf{A}^{o}$ tuple, then in turn each of these will be associated with many $\mathbf{A}^{i}$ tuples, and so on in a cascade of related tuples. Conversely, if we considered that original **A** tuple as an $\mathbf{A}^{i}$ tuple, identified by a_1^i (so that a_1^o and a_1^i are identical), then there are many $\mathbf{A}^{o}$ tuples associated with this tuple, and, taking these $\mathbf{A}^{o}$ tuples as $\mathbf{A}^{i}$ tuples, there are many further $\mathbf{A}^{o}$ tuples associated with them, and so on in a cascade. This is the implosion. Figure 13 shows a direced graph of the relationship.

Example 1 Each tuple of **A** describes a part type, assembly type, subassembly type, subsubassembly type, and so on. Each tuple of **R** shows how an A^{o} (outer) identified part type encloses or phusically contains an A^{i} (inner) identified part type. An explosion of any part type gives the part types it contains, and the part types those part types contain, and so on. Conversely, the implosion of a part type gives the part types that contain it, the part types containing those part types, and so on.

Example 2 Each tuple of **A** describes a joint stock company. Each tuple of **M** shows how a A^o identified (parent) company owns stock in an A^i identified company (immediate subsidiary). For a given company, an explosion shows the immediate subsidiaries, their immediate subsidiaries, and so on. Conversely, an implosion for a given company gives the immediate parents, their parents, and so on.

Cyclic matrix relationships are difficult to manipulate using conventional non procedural languages, such as SQL. The major difficulty is the large number of implicit sub-relationships involved. For example, in the case of the joint stock companies, we can be interested in the association between a company and its immediate subsidiaries, which is one type of subrelationship, or between a company and the subsidiaries of its immediate subsidiaries, which is another type of subrelationship, and so on. Conventional languages like SQL require that the user fully specify, in terms of matching attribute specifications, any such subrelationship within a language expression, which can be very difficult. A full account of this matter is beyond the scope of this paper, which concentrates on the classification of relationships, rather than their manipulation. A complete analysis of these subrelationships has been carried out, and will be presented in a separate report.

3.5 Other composite relationships

Many different types of relationship chains between relations **A** and **B** are possible, so that there are very many different possible types of composite association. However, the author has found no others that are likely to be of any use in practice. We are inclined to call all the remaining relationships obscure associations. Neverthless, a few of these obscure relationships may be of theoretical interest. For example, it is possible to define composite co-relationships, and even cyclic composite co-relationships. These relationships are being analysed as part of the author's current research effort.

4 CONCLUSIONS

The number of different types of relationships that can occur between

relations in a relational data base is surprizingly large, and attests to the great flexibility of the relational approach. In the CODASYL approach [5], even where the conceptual files are relations, only one type of relationship is allowed, namely the equivalent of the simple primitive 1:n relationship; however, it has to be admitted that simple 1:n relationships are by far the most common, and are the building blocks of many of the other types defined in this paper.

The classification proposed in this paper is not exhaustive. However, we believe that the present classification should be of use in practice to designers of data base manipulation languages and systems. In addition, even those concentrating on the universal relation approach could find the classification useful. The author is currently using this calssification as the foundation for the development of the natural quantifier language SQL/NQ, for manipulation of complex relationships with a minimum of specification effort.

REFERENCES

1. Aho, A. V., Beeri, C., and Ullman, J.D. The theory of joins in relational databases, ACM Trans. Database Syst., 4(3), 1979, 317-314.

2. Beeri, C., Kifer, M. An integrated approach to logical design of relational data base schemes, ACM Trans. on Database Syst., 11(2), 1986, 134-158.

3. Bernstein, C.W., and Chiu, C.W. Using semijoins to solve relational queries, J. ACM, 28(1), 1981, 25-40.

4. Bradley, J. An extended owner-coupled set data model and predicate calculus for database management, ACM Trans. Database Syst., 3(4), 1978, 385-416.

5. Bradley, J. Database Management in Business, 2nd Ed., Holt, Rinehart & Winston, New York, 1987.

6. Bradley, J. SQL/N and attribute/relation associations implicit in functional dependencies, Int. J. Computer & Information Science, 12(2), 1983

7. Bradley, J. Co-relationships, levels of significance, and the source of the connection trap in relational data bases. Research Report No. 86/250/24, Univ. of Calgary, Calgary, Alberta, Canada, 22 pages.

8. Bradley, J. Polygonal Join dependencies, closed co-relationship chains, and the connection trap in relational data bases. Research Report No. 87/256/04, Univ. of Calgary, Calgary, Alberta, Canada, 1987, 18 pages.

9. Bradley, J. Join dependencies in relational data bases and the geometry of spatial grids. Computer Journal, 29(4), 1986, 378-380.

10. Chamberlin, D.D., et al. SEQUEL 2: A unified approach to data definition, manipulation and control, IBM J. Res. & Dev., 20(6), 1976, 560-575.

11. Chen, P.P., The entity-relationship model: Towards a unified view of data, ACM Trans. Database Syst. 1(1), 1976, 9-36.

12. Codd, E.F. Further normalization of the database relational model, In "Database Systems", Courant Computer Science Symposium, 6, R. Rustin, Ed., Prentice- Hall , Englewood Cliffs, N.J., 1971, 33-74.

13. Codd, E.F. Relational completeness of database sub-languages, In "Database Systems", Computer Science Symposium 6, R. Rustin, Ed., Prentice-Hall, Englewood

Cliffs, N.J., 1971, 65-98.

14. Codd, E.F. Relational database: A practical design for productivity, CACM, 25(2), 1982, 109-117.

15. Date, C.J. Introduction to database systems, 4th Ed., Addison-Wesley, Reading, Mass.,1985.

16. Fagin, R. Multivalued dependencies and a new normal form for relational databases, ACM Trans. Database Syst., 2(3), 1977, 262-278.

17. Fagin, R., Mendelzon, A.O., and Ullman, J.D. A simplified universal relation assumption and its properties, ACM Trans. Database Syst., 7(3), 1982, 343-360.

18. Fagin, R. Horn clauses and database dependencies, J. ACM 29(4), 1982,952-985.

19. Forth, H. et al. System U: A database based on the universal relation assumption, ACM Trans. Database Syst. 9(3), 1984, 331-347.

20. Kim, W. On optimizing an SQL-like nested query, ACM Trans. Database Syst., 7(3), 1982, 443-469.

21. Kim, W., Gajski, D., Kuck, D.J. A parallel piplined relational query processor, ACM Trans. Database Syst., 9(2), 214-242.

22. Maier, D. The Theory of Relational Databases, Computer Science Press, Potomac, Md., 1983.

23. Maier, D., Ullman, J.D., Vardi, M.Y., On the foundations of the universal relation model, ACM Trans. Database Syst. 9(2), 1984, 283-309.

24. Sadri, F, Ullman, J. D.. Templete dependencies: a large class of dependencies in relational data bases and its complete axiomatization. J. ACM 29(2), 1982, 363-372.

25. Sagiv, Y., et al. An equivalence between relational database dependencies and a fragment of propositional logic, J. ACM 28(3), 1981, 435-453.

26. Sagiv, Y. and Walecka, S.F. Subset dependencies and a completeness result for a subclass of embedded multivalued dependencies, J. ACM 29(1), 1982, 363-372.

27. Stonebraker, M., Wong, E., Kreps, P., and Held, G. The design and implementation

of INGRES, ACM Trans. Database Syst., 1(3), 1976, 189-222.

28. Ullman, J.D., Principles of Database Systems, Computer Science Press, Rockville, MD, 1983.

29. Wald, J.A., and Sorenson, P.G. Resolving the query inference problem using Steiner trees, ACM Trans. Database Syst. 9(3), 348-368.

30. Weiderhold, G. Database Design, McGraw-Hill, New York, 1983.

31. Welty, D. and Stemple, D.W. Human factors comparison of procedural and non procedural query languages, ACM Trans. Database Syst., 6(4), 1981, 626-649.

32. Wilmot, R.B. Foreign keys decrease adaptibility of database designs, CACM, 27(12), 1984, 1237.