Introduction

In this paper we examine closed chains of relations that are linked by co-relationships, and prove a fundamental theorem involving polygonal join dependencies and closed chains where the co-relationships are existentially significant. Join dependencies have been known for some time [1, 4, 8, 14, 15, 16, 18], with most of the published work dealing with their technical characteristics. The join dependencies that we deal with are a large class of join dependencies called polygonal join dependencies [8]. Co-relationships, on the the other hand, have not been much studied by relational theorists, probably because of the relatively undefined status often associated with relationships in data base theory [3, 4, 10, 15]. Indeed, nowhere in the basic theory of relations [15] is there even any mention of the concept of a relationship between relations. Nevertheless, in many theoretical papers the concept of a relationship surfaces frequently, without any precise definition [2, 3, 4, 5, 6]. In this paper we use a reasonable but precise definition that has proved very useful [7, 9].

Furthermore, there appears to have been few systematic attempts to classify the different kinds of relationships that can occur in relational data bases. It is nevertheless from a systematic classification developed by the author [7] that the co-relationship has emerged as a well-defined type of relationship. Research into co-relationships has revealed that for any co-relationship there is a level of semantic significance, and that the well-known connection trap occurs with co-relationships in cases where the user assumes the wrong level of semantic significance [9].

In this paper we go a step further, and examine closed chains of relations, particularly binary relations, where each link of the chain is a co-relationship. This leads us to a fundamental theorem about such closed chains and join dependencies.

1. CO-RELATIONSHIPS AND LEVELS OF SIGNIFICANCE

We begin with a brief review of the concepts of co-relationship and levels of significance, reported on in detail elsewhere [7, 9].

1.1 Notation

Upper case bold letters are used for relation names, that is, instances of relations. Upper case letters are used for relation attributes and attribute concatenations. Relation schemes are implied throughout, but are not named [15]. Subscripted lower case letters are used for attribute values with a tuple, with the convention that if relation scheme [P, Q, S, T] could give rise to a relation **P**(P Q, S, T), a tuple of **P** could be (p_2, q_5, s_1, t_6). The primary key attribute (or attribute concatenation) will usually be underscored, and for reader convenience, will often have the same letter as the relation name, as in the case of **P** and P.

1.1 The co-relationship

A co-relationship is a particular type of relationship that can occur in a data base. In a relational data base we can classify all relationships as either primitive or non primitive. A primitive relationship is defined as follows [7]:

> Primitive relationship definition
>
> There is a primitive relationship between any arbitrary pair of relations (**A**, **B**), iff by means of a single join operation, and no more than one projection operation, it is possible to generate a relation **R**(A, B, ...), with minimum attributes A, B.

The relation **R** can be called a relationship relation, and in less formal terms, if we have relations **A**(A, ...) and **B**(B,...), then there will be a primitive relationship between **A** and **B**, if we construct a relation **R**(A, B) by joining **A** and **B** on a common domain attribute, and taking the projection on A and B, that is:

$$\mathbf{R}(A, B) = \pi_{A,B} (\mathbf{A} *_{C,C} \mathbf{B})$$

where π denotes projection, and $*_{C,C}$ denotes a natural join on common domain attribute C.

Non primitive relationships are simply relationships that are not primitive, in accordance with the above definition. They do not concern us in this paper, and are covered in detail in [9]. There are two major categories of primitive relationship, and these are the common one-to-many (1:n) relationship [2, 5, 6, 10, 11, 18] and the co-relationship. Let C be the common domain attribute used to generate a relationship relation **R**(A, B) for the case of a primitive relationship. We refer to C in either **A**, or **B**, as a relationship supporting attribute, or relationship attribute.

One-to-many (primitive) relationship definition

A primitive relationship between relation **A** and **B** is one-to-many iff one, and only one, of the relationship attributes C is a primary or candidate key.

Co-relationship definition

A primitive relationship is a co-relationship if neither of the relationship attributes C is a primary or candidate key.

In accordance with the above definitions, if we have relations **A**(A, ...) and **B**(A, B, ...), then there is a one to many relationship between **A**, and **B**, such that for any one **A** tuple, there can be many related **B** tuples, all with the same A attribute value. In contrast, if we have relations **A**(A, C, ...) and **B**(B, C, ...), then there is a co-relationship between **A** and **B**, such that for a given C value, every single **A** tuple with that C value is related (co-related) to every single **B** tuple with that same C value.

A co-relationship partitions the co-related relations, as illustrated in Figure 1. The tuples of a partition, whether from relation **A** or **B** all have the same relationship attribute C value. If we display the co-relationship between **A** and **B** using a matrix, each partition of the relationship appears as a rectangle, as shown in Figure 2; within a rectangle of the display, each point denotes a pair of co-related **A** and **B** tuples.

1.3 Co-relationship semantics and levels of semantic significance

To handle the semantics behind co-relationships we need to introduce levels of significance, dealt with in detail in [9]. A brief review of this basic concept is necessary at this point. Consider the following relations:

A	C
a_2	c_1
a_7	c_1
a_1	c_3
a_5	c_3
a_8	c_3
a_3	c_5
a_6	c_5

A

B	C
b_3	c_1
b_9	c_1
b_{11}	c_1
b_4	c_3
b_6	c_3
b_2	c_5

B

Figure 1. Two co-related relations are partitioned by the common relationship support attribute (C)

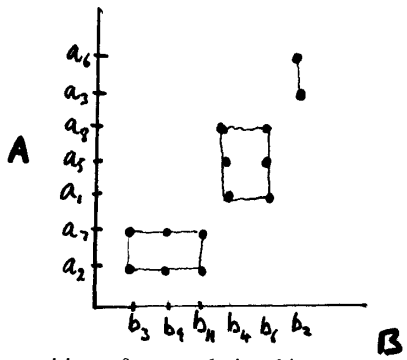

Figure 2. Each partition of a co-relationship appears as a rectangle when the relationship is desplayed as a matrix.

EP(E#, P#); a tuple tells what engineer (E#) works on what project (P#).

EC(E#, C#); a tuple tells what engineer (E#) uses what computer (C#)

Note that neither E#, P#, nor C# is a primary key.

Clearly we have a co-relationship between **EP** and **EC**, supported by the common domain attribute E#. Depending on the semantics involved, we can have the following levels of significance.

(a) **Coincidental significance**

Here, if a pair of **EC** and **EP** tuples have common E# value, it is purely a coincidence and has no further meaning. We could not infer, for example, that because engineer E1 uses computer C7, and engineer E1 works on project P7, that computer C7 has anything to do with project P7. However, even this minimum level of semantic significance for the relationship can be useful in practice. For example, we might want to retrieve the computers used by the engineers who work on project P7, with SQL expression:

```
SELECT C# FROM EC
WHERE E# IN (SELECT P# FROM EP
    WHERE P# = 'P7')
```

(b) **Existential significance**

Here, for a given E# value, some of the **EP** tuples with that E# value can be significantly related to some of the **EC** tuples with that E# value, where the significance level is more meaningful than in the case of pure coincidental significance. For example, it could be that for a given E# value, such as E7, then some of the computers used by E7 might be used on some of the projects worked on by E7. If such were the semantics, then with the retrieval: Retrieve the computers used on project P6 by engineers working on project P7, we still could not use the SQL expression above, since it would retrieve a superset of the required computers. (As we shall see, a person accepting this SQL expression would have fallen into a connection trap.)

(c) **Universal significance**

Here, for a given E# value, all of the **EP** tuples with that E# value are significantly related to all of the **EC** tuples with that E# value, where the semantic level of significance is at a more meaningful level than in the case of pure coincidental significance. For example, it could be that for any given E# value, such as E7, then all of the computers used by E7 are employed on all of the projects worked on by E7. With such semantics then the retrieval: Retrieve the computers used on project P6 by engineers working on P6, would be correctly expressed using the SQL expression given under (a) above.

In addition to coincidental, existential and universal signficiance, two other less important levels of significance can be readily distinguished. These are not important for the purposes of this paper and readers are referred to reference [9].

1.4 Co-relationships and the connection trap

Co-relationship levels of significance enable easy definition of the connection trap. Although levels of significance can give rise to connection traps in a fairly wide variety of ways, as covered in reference [9], the essence of the matter is this: A user will fall into a connection trap if he or she assumes a wrong level of significance. Thus, if the co-relationship between **EP** and **EC** is existential, and a user assumes that all computers used by engineer E4 are used on projects worked on by E4, when in fact, in accordance with existential significance, only some of the computers used by E4 are used on projects worked on by E4, then that user will have assumed a wrong level of significance and will have fallen into a connection trap. However, there are even more sophisticated traps, where closed chains of co-related relations are involved.

2. JOIN DEPENDENCIES AND CYCLIC CO-RELATIONSHIPS

We are now in a position to demonstrate the major point of this paper, namely that polygonal join

dependencies are fundamentally due to closed chains of existentially co-related binary relations, where no connection trap is implicit in a complete join of the relations. Initially we show this for the triangular join dependency.

2.1 Triangular join dependency

The triangular join dependency, so called because it occurs in a relation in which each tuple denotes the three apexes of a triangle in a 1-dimensional grid of triangles, or a set of intersection triangular grids [8], has the following properties: Consider a relation **J**(X, Y, Z); if tuples (–, y_1, z_1) (x_1, –, z_1), and (x_1, y_1, –) occur in **J**, then the tuple (x_1, y_1, z_1) must occur if the relation **J** is to contain a triangular [8] join dependency [15, 16]. It can also be shown that **J** cannot be non loss decomposed into any two projections each with two attributes, that is, any two of relations **XY**(X, Y), **XZ**(X, Z), and **YZ**(Y, Z), where

$$\mathbf{XY}(X, Y) = \pi_{X,Y}(\mathbf{J}(X, Y, Z)), \text{ and so on.}$$

A join of, for example **XY**(X, Y) and **XZ**(X, Z) on join attribute X will not regenerate **J**. In order to regenerate **J**, we must first join any two of the projections of a common join attribute, and then join the result to the third projection on <u>two</u> common join attributes, that is, for example:

Step 1: $\mathbf{XYZ}(X, Y, Z) = (\mathbf{X}, Y) *_X \mathbf{XZ}(X, Z)$, where

Step 2: $\mathbf{J}(X, Y, Z) = \mathbf{XYZ}(X, Y, Z) *_{Y,Z} \mathbf{YZ}(Y, Z)$

These properties can be seen in a geometrical light if the tuples describe triangles in a grid. First we see that if triangles (x_a, y_1, z_1), (x_1, y_a, z_1), and (x_1, y_1, z_a) exist, then geometrically, the triangle (x_1, y_1, z_1) must also exist. Secondly, since a projection on any two attributes will give a relation each of whose tuples denote the side of a triangle, a join of two projections on a common attribute (apex type) will generate spurious triangles that can only be eliminated by a further join (of the third side) [8].

Essentially, we can say that a relation of degree 3 contains a triangular join dependency if it is always a join of all three of its projections. Similarly, a relation of degree 4 will contain a join dependency, the rectagular join dependency, if it must always be a join of all four of its projections. It can be called the rectangular join dependency because it will occur in a relation where each tuple describes a rectangle in a gird of rectangles [8]. Similarly, we can have pentagonal join dependencies, hexagonal join dependencies, and so on, giving a series of polygonal dependencies. It is these polygonal dependencies, particularly the triangular join dependency, that appear likely to occur in practice.

2.2 Cyclic co-relationships

We may define a cyclic co-relationship as follows:

> Any relation **T** participates in a cyclic co-relationship, that is, **T** is co-related to **T**, if **T** is a relation in a closed chain of relations, where each pair of adjacent relations in the chain are co-related.

The simplest chain has two relations. However, this is a special case and does not involve any join dependencies. The simplest case of interest as far as join dependencies are concerned is the closed chain of three co-related binary relations, and we take as an example the relations **EP**(E#, P#), **EC**(E#, C#) and **PC**(P#, C#), with no particular semantics, and with none of the attributes E#, P# or C# being either a primary or candidate key. (Note that each relation must, by definition, be "all key".) We now examine the implications of the different possible levels of significance for these co-relationships, for a three relation chain.

(a) Coincidental significance for each link

Suppose a tuple within any of three relations, for example (e_2, p_4), within **EP**. Such a tuple implies a semantically significant association between e_2 and p_4. Similarly, a tuple (p_5, c_2) in **PC** implies a

semantically significant association between p_5 and c_2, and so on. However, because the co-relationship between **EP** and **EC**, for example, is merely coincidental, for a pair of co-related tuples (c_7, p_3) and (c_7, c_2), there can not be an association of semantic significance between p_3 and c_2, for otherwise the co-relationship would not be merely coincidental. In other words, the relation:

$$\pi_{P\#,C\#} (\mathbf{EP} *_{E\#,E\#} \mathbf{EC}) = \mathbf{X}(P\#, C\#)$$

cannot have any semantic significance. But because of our initial assumption of a chain of relations, there exists the relation **PC**(P#, C#). It therefore follows that **X** and **PC** are disjoint. From this it follows that a join of all three relations **EP**, **EC** and **PC** must be an empty relation. This result can be generalized:

> Theorem 1. In a closed chain of n binary relations, where each adjacent pair of relations on the chain is linked by a co-incidental co-relationship, a join of all the relations results in an empty relation.

The proof should now be obvious to the reader.

(b) **Universal significance for each link**

Consider again the closed chain of relations **EP**, **EC** and **PC**. Taking any adjacent pair of relations, such as **EP** and **EC**, we can be sure that for any E# value e_x, all **EP** tuples containing e_x are significantly related to all **EC** tuples with that e_x value, that is, for any **EP** tuple (e_x, p_x), where there is a **EC** tuple (e_x, c_x), then p_x and c_x must be significantly related, and furthermore, p_x and c_x can be significantly related, only if the E# attribute value is common in the respective **EP** and **EC** tuples. All this follows from the definition of universal significance. More formally, a relation whose tuples give the association between P# and C# values can be obtained from a join of **EP** and **EC**:

$$\mathbf{X}(P\#, C\#) = \pi_{P\#,C\#} (\mathbf{EP} *_{E\#,E\#} \mathbf{EC})$$

But we already have a relation **PC**(P#, C#) within the chain of relations, and this portrays the association between P# and C# values. Because of the definition of universal significance **PC** cannot be a superset of **X**(P#, C#). Hence **PC** is either equal to **X** or a subset of **X**. Either way, **PC** must be redundant, since the tuples it contains can be generated from **EP**, **EC**. In a similar manner, we can show that any of the three relations **EP**, **EC**, and **PC** can be generated from the other two if the links are all universally significant. This result can be generalized further, in that no matter how many relations are in the closed chain, if the links are all universally significant co-relationships, then any one of the relations of the chain can be generated from the other relations (provided we are dealing with binary relations). This gives us the following theorem:

> Theorem 2. A closed chain of binary relations linked by universally significant co-relationships is redundant, in the sense that any relation of the chain can be generated from the remaining relations.

It follows that we need never deal with a closed chain of binary relations linked by universally significant co-relationships, it being sufficient to deal with an open chain of non redundant relations. It is such open chains linked by universally significant co-relationships that when joined give relations with multivalued dependencies [7, 12, 15, 17].

(c) **Existential significance for each link.**

Consider once more the closed chain of relations **EP**, **EC**, and **PC**. If we take any pair of adjacent relations, such as **EP** and **EC**, then we can say that for any E# value e_x, only some of the **EP** tuples with that e_x value will be related to only some of the **EC** tuples with that e_x value. In other words, if we have tuples (e_x, p_x) and (e_x, c_x), if we use a join to form the tuple (e_x, p_x, c_x), and possibly with a projection, we form the tuple (p_x, c_x), we cannot be sure that either of these tuples are valid, in the sense that there is semantic significance, that is, that they are not spurious. This is in accordance with

the definition of existential significance. Thus the relation:

$$\mathbf{X}(P\#, C\#) = \pi_{P\#,C\#} (\mathbf{EP} *_{E\#,E\#} \mathbf{EC})$$

may easily contain spurious tuples. The same will be true for the relation:

$$\mathbf{Y}(P\#, E\#, C\#) = \mathbf{EP} *_{E\#,E\#} \mathbf{EC}$$

However, the relation **PC**(P#, C#) necessarily contains valid tuples that associate P# and C# values correctly. Accordingly, only **Y**(P#, E#, C#) tuples that have pairs of P#, C# values that also occur in **PC**(P#, C#) can be valid, although, as we shall see, this is merely a necessary condition for validity, but not a sufficient one. It follows that a join of **Y**(P#, E#, C#) with **PC**(P#, C#), using both P# and C# as the join attribute, will give rise to a relation that is a subset of **Y** and will thus contain a higher proportion of valid tuples, that is, tuples where the association involving E#, P# and C# values is semantically correct. Thus the relation:

$$\mathbf{PEC}(P\#, E\#, C\#) = \mathbf{PC}(P\#,C\#) *_{(P\#,C\#),(P\#,C\#)} (\mathbf{EP} *_{E\#,E\#} \mathbf{EC})$$

has generally a higher proportion of valid tuples than **Y**(P#, E#, C#).

Exactly why **PEC** may still contain invalid tuples is best understood initially using an example. Assume that **EP** (E#, P#) gives the engineers (E#) that work on projects (P#), that **EC**(E#, C#) gives the engineers (E#) that use computers (C#), and that **PC**(P#, C#) gives the projects (P#) that use computers (C#). The three relations form a closed chain linked by co-relationships, and we further assume existentially significant co-relationships as follows:

1. With **EP**(E#, P#) and **EC**(E#, C#), some of the computers used by a given engineer will be used on some of the projects worked on by that engineer.
2. With **EP**(E#, P#) and **PC**(P#, C#), some of the engineers that work on a given project will use some of the computers used on that project.
3. With **EC**(E#, C#) and **PC**(P#, C#) some of the engineers who use a given computer will work on some of the projects that use that computers.

The question is whether we can determine with certainty what engineers work on what projects using what computers. The answer is that we can not if we merely form the relation **PEC**(P#, E#, C#) as shown above, since this relation may contain invalid tuples. The following should explain why.

If we join **EP**(E#, P#) and **EC**(E#, C#) giving the relation **Y**(P#, E#, C#), in a tuple of **Y** we cannot be sure that the project identified actually uses the computer identified, because of the existential significance. However, if project p_y actually uses the computer c_y then some tuple of **Y**(P#, E#, C#) will be of the form (p_y, –, c_y). That means that **Y**(P#, E#, C#) tuples that do not have P# C# values that occur in **PC**(P#, C#) are clearly invalid and must be eliminated. We can do this with a join of **Y**(P#, E#, C#) with **PC**(P#, C#) on both P# and C# attributes. But some of the tuples in the resulting relation **PEC**(P#, E#, C#) can still be invalid. To see this consider the following semantic situation:

1. Engineer e_1 works on project p_1 using computer c_7.
2. Engineer e_1 works on project p_8 using computer c_1.
3. Engineer e_9 works on project p_1 using computer c_1.

From this we can see that the following tuples of the relations of the chain apply:

E#	P#
e_1	p_1
e_1	p_8
e_9	p_1

EP

E#	C#
e_1	c_7
e_1	c_1
e_9	c_1

EC

P#	C#
p_1	c_7
p_8	c_1
p_1	c_1

PC

First with a join of **EP** and **EC** we form **Y**, and then with a further join of **Y** and **PC** we form **PEC**:

P#	E#	C#
e_1	p_1	c_7
c_1	p_8	c_1
e_9	p_1	c_1
$\mathbf{e_1}$	$\mathbf{p_1}$	$\mathbf{c_1}$
$\mathbf{e_1}$	$\mathbf{p_8}$	$\mathbf{c_7}$

Y

P#	E#	C#
e_1	p_1	c_7
e_1	p_8	c_1
e_9	p_1	c_1
$\mathbf{e_1}$	$\mathbf{p_1}$	$\mathbf{c_1}$

PEC

We see that the last two tuples of **Y** are invalid, but one of these invalid tuples (e_1 p_8 c_7) is eliminated in the final join that forms **PEC**, leaving one invalid tuple in **PEC**.

If the last tuple of **PEC** above had to be valid as well, or, in more general terms, if the relation **PEC**(P#, E#, C#) formed by the two joins always had to contain only valid tuples, then **PEC** would contain a triangular join dependency. This is clear in the case of the example above, since the requirement that tuple (c_1, p_1, c_1) exist if tuples (c_1, p_1, ?), (c_1, ?, c_1) and (?, p_1, c_1) exist is the condition for a triangular join dependency. This leads us to the fundamental theorem of closed chains of existentially co-related binary relations:

> Theorem 3. A join of all the relations of a closed chain of existentially co-related binary relations, where the last join involves the two possible join attributes, will always contain only semantically valid tuples only if the resulting relation contains a polygonal join dependency of order equal to the number of relations in the chain.
>
> Proof. We shall prove the theorem for the case of order 3, that is, a chain of three relations **EP**(E#, P#), **EC**(E#, C#) and **PC**(P#, C#). Suppose that the relation **PEC**(P#, E#, C#) formed from a join of **EP** and **EC** on join attribute E#, followed by a join with **PC** on join attributes P#, C#, contains the tuples (e_x, p_x, ?), (e_x, ?, c_x), and (?, p_x, c_x). By projection, it follows that **EP** must contain a tuple (e_x, p_x) and **EC** a tuple (e_x, c_x), and **PC** a tuple (p_x, c_x). If we join the **EP** and **EC** tuples we get a tuple

(c_x, p_x, c_x) which may not be valid semantically. A necessary, but not sufficient, condition for this tuple to be semantically valid is that there exist a **PC** tuple (p_x, c_x). There does, but (e_x, p_x, c_x) may still be semantically invalid, since the sole reason for the existence of **PC** tuple (p_x, c_x) may be some other valid **PEC** tuple (e_y, p_x, c_x), as originally assumed. However, if (e_x, p_x, c_x) is to be always valid, no matter what the values assigned to c_x, p_x, and c_x, then by the definition of a triangular join dependency, **PEC** must contain such a dependency.

The proof for the general case of a closed chain of length n relations is similar and is left to the reader.

A practical consequence of this theorem is that it is not possible to extract meaningful (other than coincidental) information from two or more existentially co-related relations in such a chain if the complete join of the chain does not contain a join dependency. As an example with the three-relation chain **EP**, **EC**, and **PC**, suppose that we have the query: Find the engineers and projects that use computer c_3. The "obvious" SQL expression:

SELECT E#, P#, C# FROM **EP**, **EC**
WHERE **EP**.E# = **EC**.E# AND **EC**.C# = c_3

is completely wrong, and will normally retrieve invalid data, because only one join is involved. But the more sophisticated SQL expression involving two joins, with the second join having two join attributes:

SELECT E#, P#, C# FROM **EP**, **EC**, **PC**
WHERE **EP**.E# = **EC**.E# AND **EP**.P# = **PC**.P# AND **EC**.C# = **PC**.P#
AND **EC**.C# = c_3

is also wrong, and will normally retrieve data that is invalid. With the first SQL expression the user

has fallen into the common connection trap that occurs with existential co-relationships. With the second SQL expression the user will at least have the consolation of having fallen into a very sophisticated connection trap. The user will avoid the trap with this expression only if **PEC** contains a triangular join dependency.

2.3 Cyclic co-relationships with non binary relations

The difficulties that arise with closed chains of non binary co-related relations are similar to those that occur in the binary case. However, there are many subtle differences, and this topic is relegated to a separate paper.

3. Conclusions

It has been shown that closed chains where the linking relationships are coincidental cannot exist, and that at least one of the relations in a closed chain of universally significant co-relationships is redundant, so that the chain is effectively open. With a chain of binary relations that are linked by existentially significant co-relationships, connection traps are implicit, except in the rare case where a complete join of the chain contains a polygonal join dependency of order equal to the number of relations in the chain. This has been formulated as a fundamental theorem and proved.

References

1. Aho, A.V., Beeri, C., and Ullman, J.D. The theory of joins in relational data bases, ACM Trans. Database Syst., 4(3), 1979, 317-314.
2. Armstrong, W.W. Dependency structure of data base relationships, Proc. IFIP 74, North Holland, Amsterdam, 1974, 580-583.
3. Beeri, C. On the membership problem for functional and multivalued dependencies in relational data bases, ACM Trans. Database Syst., 5(3), 1980, 241-259.
4. Beerie, C., Kifer, M. An integrated approach to logical design of relational data base schemes, ACM. Trans. on Database Syst., 11(2), 1986, 134-158.
5. Bradley, J. An extended owner-coupled set data model and predicate calculus for data base management, ACM. Trans. on Database Syst., 3(4), 1978, 385-416.
6. Bradley, J. SQN/N and attribute/relation associations implicit in functional dependencies, Int. J. Computer and Information, 12(2), 1983.
7. Bradley, J.A. A fundamental classification of associations in relational data bases, Research Report No. 85/204/17, University of Calgary, Alberta, Canada, 32 pages.
8. Bradley, J. Join dependencies in relational data bases and the geometry of spatial grids, Computer Journal, 29(4), 1986, 378-380.
9. Bradley, J. Co-relationships, levels of significance, and the source of the connection trap in relational data bases, Research report No. 86/250/24, University of Calgary, Alberta, Canada.
10. Chen, P.P., The entity-relationship model: Towards a unified view of data, ACM. Trans. on Database Syst., 1(1), 1976, 9-36.
11. Codd, E.F. Relational database: A practical design for productivity, CACM, 25(2), 1982, 109-117.
12. Fagin, R. Multivalued dependencies and a new normal form for relational data bases, ACM Trans. on Database Syst., 2(3), 1977, 262-278.
14. Fagin, R. Horn clauses and data base dependencies, J. ACM 29(4), 1982, 343-360.
15. Maier, D. The Theory of Relational Databases, Computer Science Press, Potomac, Md., 1983.
16. Rissanen, J. Theory of joins for relational data bases - a tutorial Survey. Lect. Notes in Computer Science 64, 537-551, Springer-Verlag, 1979.
17. 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.
18. Wiederhold, G. Database Design, McGraw-Hill, New York, 1983.