RJ2145 (29408) 2/21/78
Computer Science

Research Report

FINDING MISSING JOINS FOR INCOMPLETE QUERIES IN RELATIONAL DATA BASES

C. L. Chang

IBM Research Laboratory
San Jose, California

IBM Research Division
Yorktown Heights, New York · San Jose, California · Zurich, Switzerland

RJ2145 (29408) 2/21/78
Computer Science

FINDING MISSING JOINS FOR INCOMPLETE QUERIES IN RELATIONAL DATA BASES

C. L. Chang

IBM Research Laboratory
San Jose, California

ABSTRACT: In a natural language query system (e.g., RENDEZVOUS) for a relational data base, a problem often arises that the user's query may be translated into an incomplete formal (e.g. DEDUCE) query due to errors, ambiguities or incompleteness in the user's query, or due to the limitations of the natural language analyzer. This paper considers specifically joins which are missing from the incomplete formal query, and proposes an algorithm based on minimal spanning trees for solving this problem.

1. INTRODUCTION

The problem of finding missing joins arose in Codd's RENDEZVOUS project. This project is developing natural language query systems for casual users who wish to interrogate relational data bases [4]. Version 1 of the RENDEZVOUS system is described in [5]. The analyzer, one of the eight components in RENDEZVOUS, translates English into DEDUCE. Joins may be missing from output of the analyzer, if they were also missing or obscured in its source English. In this paper we describe a method for finding joins in case they are missing from a formal query. Hopefully, the method may be useful in other areas besides natural language query systems. For instance, in distributed data bases, a query which used to be complete with respect to an old data base schema may become incomplete when the data base schema is changed (e.g., a relation is decomposed into several relations.) To process the query, one may have to first induce and later evaluate the necessary joins. Some problems related to joins can be found in [1,7], and [7] in particular treats missing joins in terms of conceptual graphs.

The formal query sublanguage used in Version 1 is called DEDUCE [2]. In this paper, we shall consider only queries without subquery structures. The syntax of this type of query is given in the Appendix. As a sample data base, we

use the relational data base taken from [4] which consists of the following relations:

SUPPLIER(SNO, SNAME, SLOC, RATE)

PART(PNO, PNAME, QOH, QOO, PTYPE)

PROJECT(JNO, JNAME, JLOC)

SHIP(SNO, PNO, JNO, QSHIP, DATE).

The SUPPLIER relation has a row for every supplier, giving his serial number, name, location and rating. The PART relation has a row for every part, giving its serial number, name, quantity on hand, quantity on order and type. The PROJECT relation has a row for every project, giving its serial number, name and location. The SHIP relation gives every shipment of a certain quantity (QSHIP) of a part (PNO) from a supplier (SNO) to a project (JNO) on a certain date (DATE).

To understand the problem of missing joins, let us consider an example. Assume we have an incomplete DEDUCE query,

FIND: SUPPLIER(*SNAME) & PART(PNAME=PIPE).

We note that joins are missing from this query. The clauses, SUPPLIER(*SNAME), and PART(PNAME=PIPE), may be joined in any one of the following ways, where x1, x2, x3, x4 and x7 are join variables:

(1) FIND: SUPPLIER(*SNAME, SNO=x1) &
SHIP(SNO=x1, PNO=x4) &
PART(PNAME=PIPE, PNO=x4).

This complete query may be rephrased as: Print the name of every supplier who sent a shipment of a part named pipe.

(2) FIND: SUPPLIER(*SNAME, SNO=x1) &
SHIP(SNO=x1, QSHIP=x7) &
PART(PNAME=PIPE, QOO=x7).

This query may be rephrased as: Print the name of every supplier who sent a shipment having a quantity identical to the quantity on order for a part named pipe.

(3) FIND: SUPPLIER(*SNAME, SLOC=x2) &
PROJECT(JLOC=x2, JNO=x3) &
SHIP(JNO=x3, PNO=x4) &
PART(PNAME=PIPE, PNO=x4).

This query may be rephrased as: Print the name of every supplier who has the same location as a project which receives a shipment of a part named pipe.

Now, to find out which one is correct, the English rephrasings of the above three queries could be presented to the user for him to select his intended one. Because there are often too many possibilities to present to the user all at once, we need a method of selecting the most likely ones first, then the next most likely, and so on. That is, given fragments such as SUPPLIER(*SNAME) and PART(PNAME=PIPE), the problem is how to piece (join) them together to make a complete query. Since, in general, there will be many ways to join them, such as shown in the above example, we need a method that systematically finds all different joins, and assigns priorities to them. In this paper, we shall present a method based on minimal spanning trees for solving this problem.

2. USING MINIMAL SPANNING TREE ALGORITHMS FOR JOINING FRAGMENTS

The techniques we use to piece together fragments are based on the concept of the minimal spanning tree [6]. Given a relational data base, we assign a weight to each pair of joinable attributes to indicate the likelihood that the pair of attributes is intended to be joined. The smaller the weight, the more likely the two attributes are intended to be joined. Usually, joins on keys of entity relations have the lowest weight. We call such joins high-level joins, because there are often terse English idioms for specifying or implying such joins. A join which is not a high-level join is called a low-level join. For our sample data base, the weights for all joinable pairs of the attributes are assigned as in Table 1, where each distinct link is indicated by a unique link number. In addition, we assign zero weight to every pair of attributes that are in the same relation. Using the above assignment of weights, we obtain the graph shown in Fig. 1, where a node represents an attribute of a relation, and an edge indicates that its two attributes are joinable, or they belong to the same relation. In the former case, we assign a non-zero weight, while in the latter case, the edge has zero weight. In the graph, an edge is denoted by a unique number, followed by its weight in parenthesis.

TABLE 1

RELATION	ATTRIBUTE	RELATION	ATTRIBUTE	LINK NUMBER	WEIGHT
SUPPLIER	SNO	SHIP	SNO	1	1
SUPPLIER	SLOC	PROJECT	JLOC	2	1
PROJECT	JNO	SHIP	JNO	3	1
PART	PNO	SHIP	PNO	4	1
PART	QOH	PART	QOO	5	2
PART	QOH	SHIP	QSHIP	6	3
PART	QOO	SHIP	QSHIP	7	2

Insert Fig. 1 about here

Note that there are high-level and low-level joins in Fig.1. For example, Link 1 is a high-level join between SUPPLIER.SNO and SHIP.SNO, and Link 2 is a low-level join between SUPPLIER.SLOC and PROJECT.JLOC. In general, the use of high-level joins for piecing together DEDUCE fragments will have high probability of being correct. That is, when we join nodes in the graph, we would prefer links which have lower weights. This leads us to the use of minimal spanning trees.

In general, the problem can be described as follows: Given a graph such as Fig. 1, and given some fragments which are represented by a subset (not necessarily all) of nodes in the graph, the problem is to find a minimal spanning subtree spanning the subset of nodes. The algorithm we use

is to first find a minimal spanning tree for the whole set of nodes in the graph, then iteratively eliminate leaf nodes of the minimal spanning tree if they are not in the subset. When there are no more leaf nodes that can be eliminated, the remaining tree will be a spanning subtree spanning the subset of nodes. To illustrate the algorithm, we consider an example. Suppose we have the incomplete DEDUCE query,

FIND: SUPPLIER(*SNAME) & PART(PNAME=PIPE).

Then, we first find a minimal spanning tree of Fig. 1 as shown in Fig. 2. Now, since we want to span SUPPLIER.SNAME and PART.PNAME, these two nodes are first marked by the double circles shown in Fig. 2. From the minimal spanning tree of Fig. 2, successively eliminating those leaf nodes which are not double circled, we will obtain a spanning subtree shown in Fig. 3 that spans the subset of nodes, SUPPLIER.SNAME and PART.PNAME.

Insert Fig. 2 and 3 about here

The total weight for this minimal spanning subtree is 2. Using Links 1 and 4 shown in Fig. 3, we obtain a complete DEDUCE query,

```
FIND: SUPPLIER(*SNAME, SNO=x1) &
      SHIP(SNO=x1, PNO=x4) &
      PART(PNAME=PIPE, PNO=x4),
```

which is the same as the one given in (1) of Section 1. This complete DEDUCE query is then rephrased back to the user for

approval. If he rejects the query, the system will find the next spanning subtree that spans the nodes, SUPPLIER.SNAME and PART.PNAME. (All spanning subtrees are ordered in the ascending order of their corresponding weights.) In fact, the DEDUCE queries given in (2) and (3) of Section 1 are based on the next two spanning subtrees having the total weights 3 and 4, respectively shown in Fig. 4 and 5.

Insert Fig. 4 and 5 about here

A heuristic method for finding all spanning subtrees in order is given as follows: (a) Use algorithms such as ones given in [6] to generate spanning trees in order for the whole graph; (b) Use the leaf-eliminating process described above to obtain spanning subtrees. This method has been implemented in APL for RENDEZVOUS [5] and works reasonably well. The method can be used to generate any specified number of spanning subtrees. Usually, the spanning subtrees are in order. However, if they are not in order, they will be sorted by their weights.

3. ACKNOWLEDGMENTS

The author would like to thank E. F. Codd for his suggestion of the problem and encouragement, and J-M. Cadiou for his comments and helpful conversations.

REFERENCES

1. Aho, A. V., Beeri, C., and Ullman, J. D., "The theory of joins in relational data bases," Presented at the 18th Annual Symposium on Foundations of Computer Science, Providence, R. I., Oct. 31 - Nov. 2, 1977.

2. Chang, C. L., "DEDUCE --- A deductive query language for relational data bases," In Pattern Recognition and Artificial Intelligence (edited by C. H. Chen), Academic Press, Inc., New York, 1976, pp. 108-134.

3. Codd, E. F., "A relational model for large shared data banks," Comm. of the ACM, 13(6):377-387, June 1970.

4. Codd, E. F., "Seven steps to RENDEZVOUS with the casual user," Proc. IFIP Working Conference on Data Base Management, (April 1974), North-Holland Publ. Co., Amsterdam, 1974, pp. 179-200.

5. Codd, E. F., Arnold, R. S., Cadiou, J-M., Chang, C. L., and Roussopoulos, N., "RENDEZVOUS Version 1: An experimental English language query formulation system for relational data bases," IBM Research Report RJ2144, San Jose, Ca. (February 1978).

6. Gabow, H. N., "Two algorithms for generating weighted spanning trees in order," TR#CU-CS-078-75, Department of Computer Science, University of Colorado, Boulder, Colorado 80309, August, 1975.

7. Sowa, J. F., "Conceptual graphs for a data base interface," IBM J. RES. DEVELOP., July 1976, pp. 336-357.

APPENDIX. DEDUCE SYNTAX FOR RENDEZVOUS VERSION 1

The complete DEDUCE sublanguage is described in [2]. However, for RENDEZVOUS Version 1 [5], we restrict ourselves to a subclass of DEDUCE queries. In this subclass a query cannot contain subqueries, numerical quantifiers, or functions except COUNT and EXIST, and the same relation cannot appear in more than one clause. The syntax of DEDUCE for this subclass is given as follows:

1. <char> ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N
|O|P|Q|R|S|T|U|V|W|X|Y|Z|-|/
2. <digit> ::= 0|1|2|3|4|5|6|7|8|9
3. <integer> ::= <digit>
| <integer><digit>
4. <string> ::= <char>'
| <digit>
| <string><string>
5. <value> ::= <string>
6. <variable> ::= x<integer>
7. <relation> ::= <char><string>
8. <attribute> ::= <char><string>
9. <comparator> ::= <
| ≤
| >
| ≥
| =
| ≠
10. <tuple> ::= <attribute>=<variable>
| <attribute><comparator><value>
| <tuple>,<tuple>
11. <*tuple> ::= *<attribute>
| *<attribute>=<variable>
| *<attribute><comparator><value>
| <*tuple>,<*tuple>
| <*tuple>,<tuple>
| <tuple>,<*tuple>
12. <clause> ::= <relation>(<tuple>)
13. <*clause> ::= <relation>(<*tuple>)
14. <conjunction> ::= <clause>
| <conjunction>&<conjunction>
15. <*conjunction> ::= <*clause>
| <*conjunction>&<*conjunction>
| <*conjunction>&<conjunction>
| <conjunction>&<*conjunction>
16. <*clause-conjun> ::= <*clause>
| <*clause>&<conjunction>
| <conjunction>&<*clause>
| <conjunction>&<*clause>&<conjunction>
17. <find-type-query> ::= FIND: <*conjunction>
18. <exist-type-query> ::= EXIST: <*clause-conjun>
19. <count-type-query> ::= COUNT: <*clause-conjun>

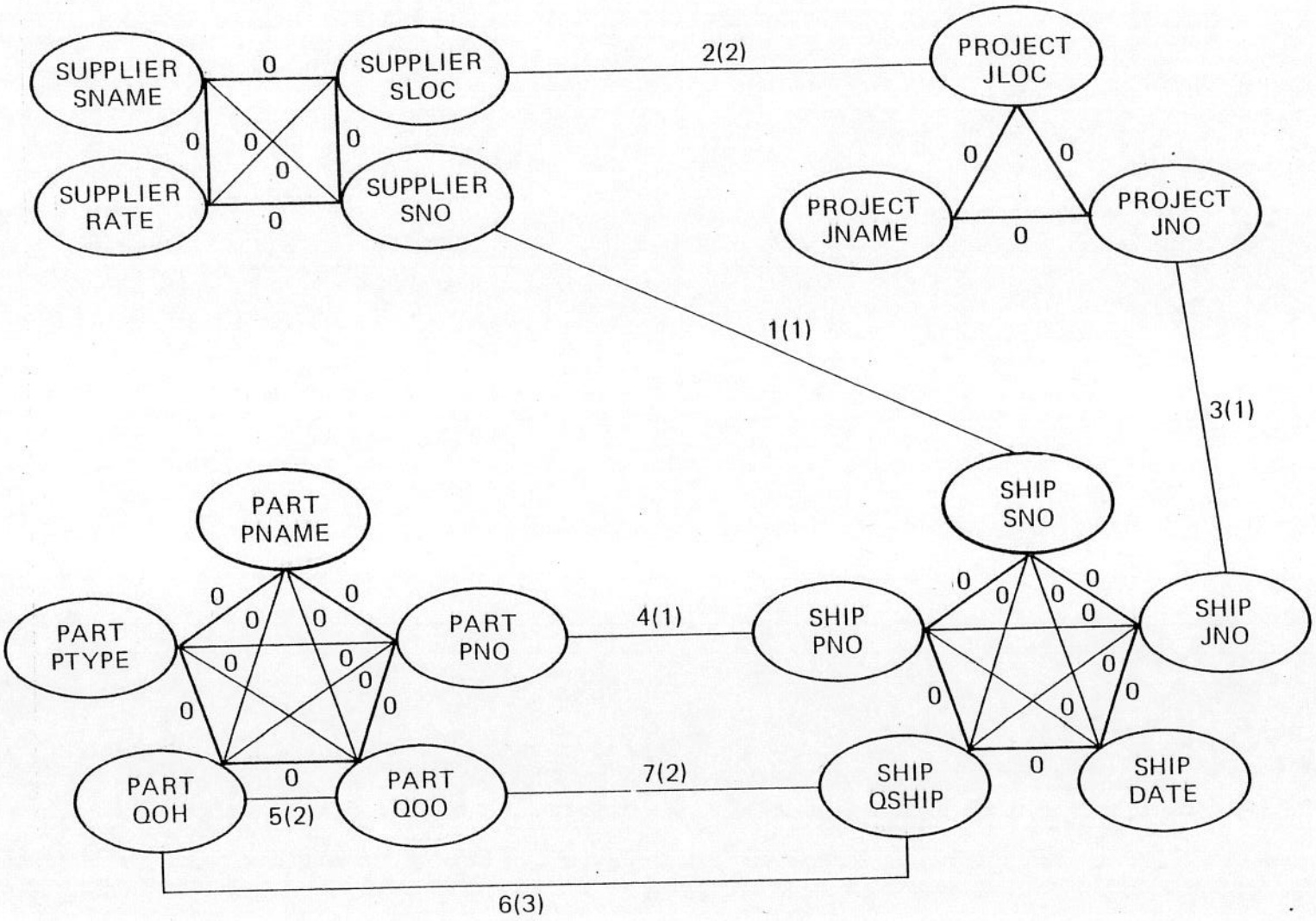

SUPPLIER SNAME
SUPPLIER SLOC
SUPPLIER RATE
SUPPLIER SNO
PROJECT JLOC
PROJECT JNAME
PROJECT JNO
PART PNAME
PART PTYPE
PART PNO
PART QOH
PART QOO
SHIP SNO
SHIP PNO
SHIP JNO
SHIP QSHIP
SHIP DATE
2(2)
1(1)
3(1)
4(1)
5(2)
7(2)
6(3)

Fig. 1

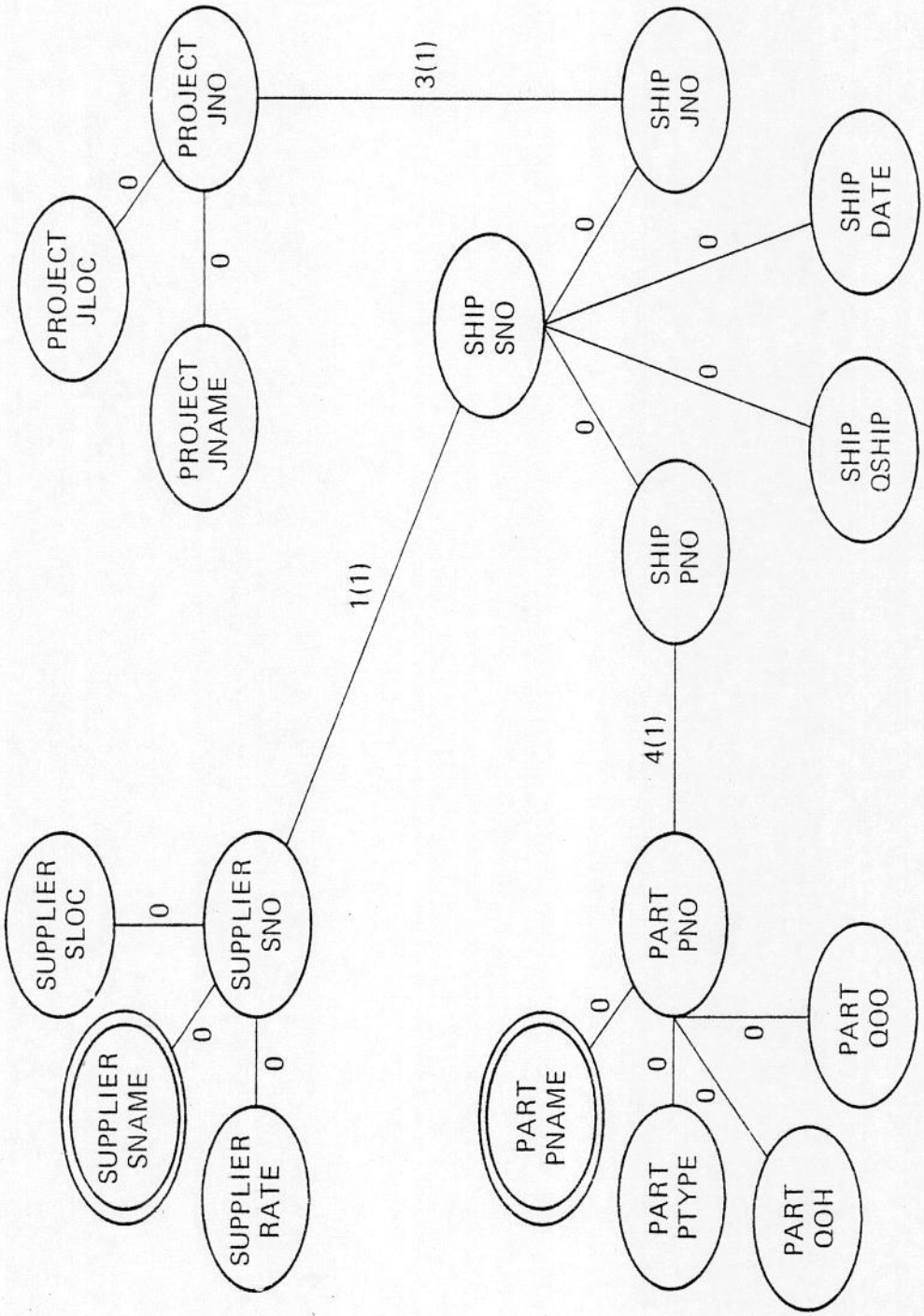
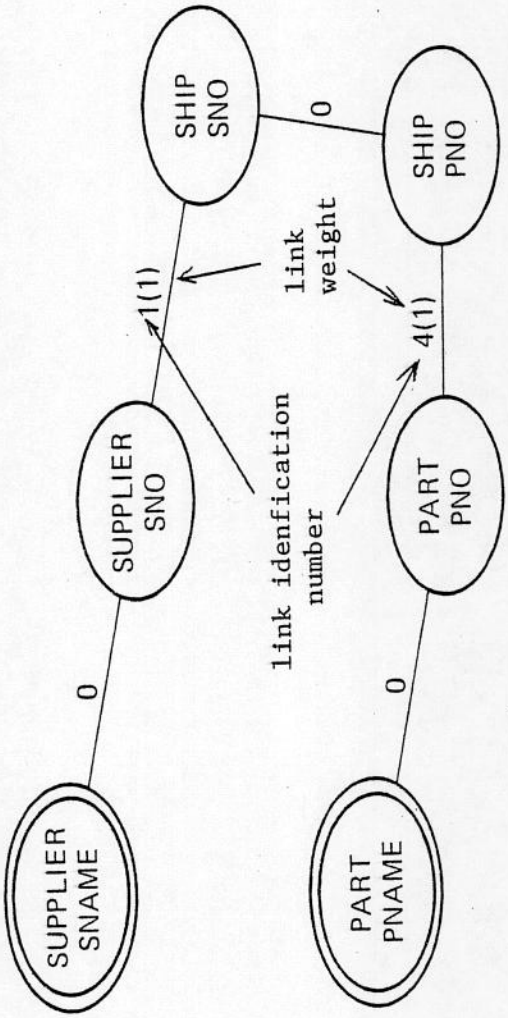
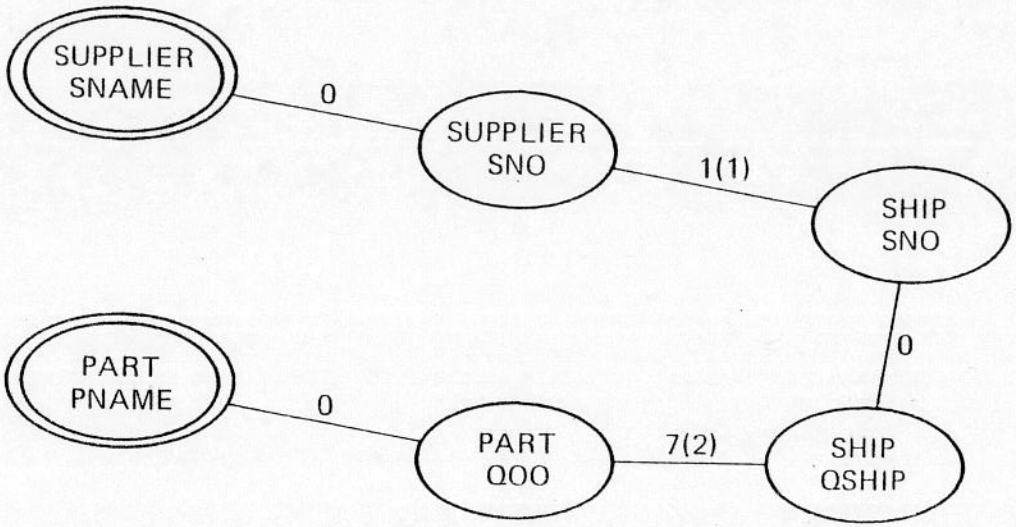

SUPPLIER SLOC
SUPPLIER SNAME
SUPPLIER RATE
SUPPLIER SNO
PROJECT JLOC
PROJECT JNAME
PROJECT JNO
SHIP SNO
SHIP JNO
SHIP DATE
SHIP QSHIP
SHIP PNO
PART PNAME
PART PTYPE
PART QOH
PART QOO
PART PNO
1(1)
3(1)
4(1)
0

Fig. 2



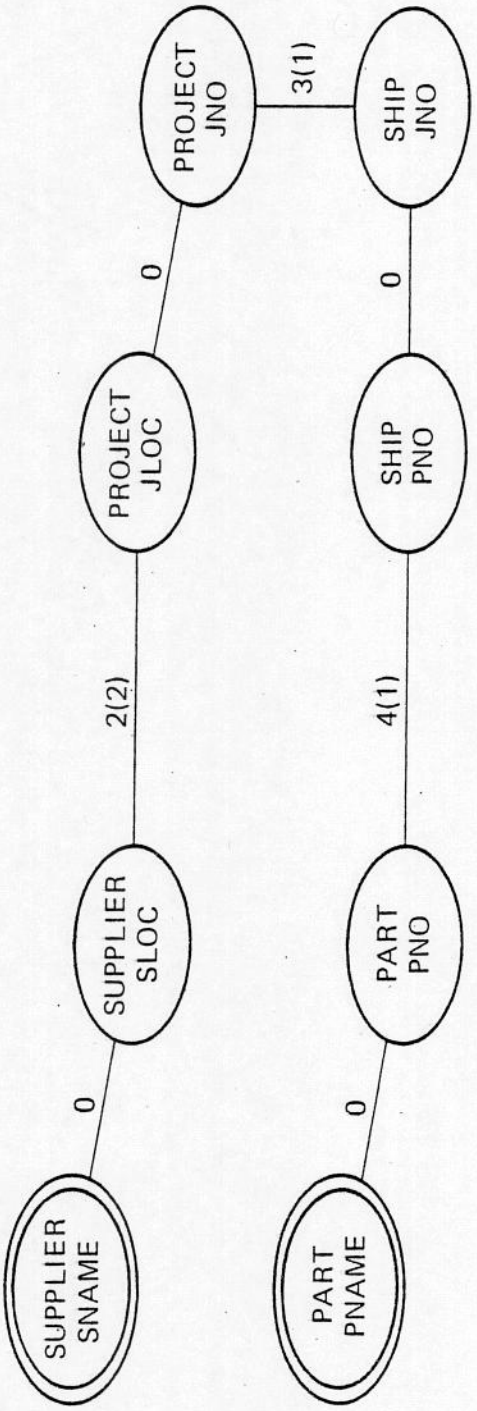
Total weight of the subtree = 0 + 1 + 0 + 1 + 0 = 2

Fig. 3



Total weight of the subtree = 0 + 1 + 0 + 2 + 0 = 3

Fig. 4



Total weight of the subtree = 0 + 2 + 0 + 1 + 0 + 1 + 0 = 4

Fig. 5