Information Systems Vol. 15, No. 2, pp. 173–186, 1990
Printed in Great Britain. All rights reserved

0306-4379/90 $3.00 + 0.00
Copyright © 1990 Pergamon Press plc

ROLES AND THEIR ROLE IN POSING RECURSIVE QUERIES†

SHARON KUCK,[1]‡ ROLAND JOHN,[2] ARND LEWE[3] and MARC NAJORK[4]§

[1]Access to Wisdom, 405 Yankee Ridge Lane, Urbana, IL 61801, U.S.A.
[2]Karolinenstr. 6, 8000 Munich 22, F.R.G.
[3]Spitzwegstr. 30, 6070 Langen, F.R.G.
[4]1304 W. Springfield Ave., Urbana, IL 61801, U.S.A.

(*Received* 25 *January* 1989; *in revised form* 24 *August* 1989; *received for publication* 16 *November* 1989)

Abstract—We show how to use the entity-relationship diagram as a vehicle for specifying the semantics of attributes. A main contribution of this work is a strategy for creating entity-relationship diagrams that make explicit the role of attributes that are over the same domain. Recursive queries can be posed over the universal relation when attributes are over the same domain and play distinct roles. We show how to extend a query language, that poses queries over the universal relation, to include the ability to express recursive queries.

1. INTRODUCTION

We have a vision that has driven this research, namely, that combining into one integerated system the best features of a universal relation interface and a visual display, such as entity-relationship diagrams (abbrev. *ER* diagrams)—our vehicle for database scheme design—will lead to a user-friendly database management system (abbrev. DBMS). A real-world data model with a visual display is essential for user-friendly database scheme design. A graphical query language can then be applied to the graphical database scheme thereby increasing the ability of a naive user to express queries over the database scheme. However, *ER* diagrams have certain shortcomings that can be overcome by adding a universal relation interface to the diagrams. Universal relation interfaces also fall short in that the unique role assumption is too restrictive. In this paper we give solutions to these problems and explicitly show how recursive queries can be posed over the universal relation (also extended *ER* diagrams) which is one of the benefits gleaned from the research on roles.

A constraint placed on a database in order for a universal relation interface to translate a query is the *unique role assumption* which requires an attribute to play a single role within the universal relation scheme. If, for example, attribute *NAME* plays the role of employee name, then it cannot also play the role of the department name in the same universal relation scheme. Doing so would mean that a tuple in the universal relation would either have the employee's name or the department's name in the column for *NAME*, but not both. When we want to represent information about an employee and his department, we must use two distinct attribute names.

ER diagrams have the opposite problem. Every attribute that appears in the diagram is assumed to be playing a distinct role, even when the spelling of two attributes (in two distinct entity sets) is the same. Thus, if *STATE* appears as an attribute in two entity sets, one for products, and one for customers, we would assume that *STATE* will have two distinct values in a tuple that relates products and customers. The database scheme is lacking an important constraint if a customer must buy a product in the state in which the customer lives. In this case, we expect *STATE* to have only one value as the unique role assumption requires. (We elaborate more on this in Example 2.) Moreover, in *ER* diagrams we cannot even assume that the attributes are over the same domain. For example, two occurrences of *STATE* could mean "states in the United States" and "states of employees' health". *ER* diagrams would benefit from the unique role assumption in the ability to capture the semantics of the attributes as these examples show.

With the universal relation scheme assumption (i.e. the unique role assumption more precisely) there is no need for *weak entity sets*¶ [1] whose appearance arises since a distinct attribute may appear in only one

†John and Lewe were supported in part by exchange scholarships between the University of Illinois and the Technische Hochschule Darmstadt, F.R.G.

‡This work was completed while the author was on the faculty of the University of Illinois at Urbana-Champaign; supported in part by IBM Faculty Development Award 1-5-37325.

§Supported in part by the German Academic Exchange Service.

¶An example of a weak entity set is *DEPARTURES* which has a single attribute *DATE*. *DEPARTURES* depends on the many-to-one relationship *INSTANCE-OF* from *DEPARTURES* to *FLIGHTS* for its definition [2]. The key of *FLIGHTS*, namely, *FLIGHTNUM*, together with *DATE* is the key of *DEPARTURES*.

entity set or relationship. The relationship on which a weak entity set depends, is not explicitly stated within the *ER* diagram—it must be gleaned from the written description of the database scheme. Thus, the key of the weak entity set is not apparent in a diagram. With a universal relation interface an attribute could appear repeatedly in a diagram. We have built upon the work of Azar and Pichat [3] who added inclusion dependencies to the *ER* model. Since they do not address roles directly, they do not discuss the benefits we achieve through the results of this paper.

Furthermore, most query languages do not make the universal relation scheme assumption [4–13]. Invariably, within a query, the joins must be specified; this distinguishes the languages from a universal relation interface query language. A *dot notation* is used in [11] and [13]. An explicit join operator is used in [4, 5, 10, 13]. In [7–9], a join is specified by naming a path through the *ER* diagram. Recursive queries, a subject of this paper, can be expressed in [8] and [11].

While the unique role assumption does a fine job of capturing the semantics of attributes that appear in more than one place and play the same role, it does nothing for attributes that are over the same domain and play distinct roles. A criticism of traditional database design, which we see with *ER* diagrams and universal relation interfaces, is that certain queries are unexpressed since the connection (namely that they are over the same domain) between attributes is lost. An example of such attributes can be seen with the *selling agent* and the *buying agent* in a real estate database. Both are real estate agents (i.e. they are over the same domain) and yet they can be distinct people. One has a contract with the home owner (i.e. the seller), the other with a person looking for a new home (i.e. the buyer). When the attributes are named distinctly, an end-user may miss the fact that selling agent and buying agent are over the same domain.

A contribution we make is to show how to relax the unique role assumption in a universal relation interface. We allow an attribute to play more than one role. By using an ISA hierarchy we can *name distinctly* each role played by an attribute. As in the *ER* model, specialization entity sets *inherit* the attributes of the generalization entity set. We provide a means of distinctly naming these attributes in *U*, the universal relation, for each role. Moreover, attributes of entity sets that are determined by the attribute that plays multiple roles are also named distinctly in *U* for each role. Such attributes have distinct values, and so should have separate columns in *U*. This makes the entire database scheme visible for each role an attribute is playing within *U*, and yet, there is an economy of expression since the attribute which plays more than one role and all entity sets determined by it, needs to occur only once in the *ER* diagram.

Maier *et al.* [14] have solved another aspect of the problem of relaxing the unique role assumption. They use the notion of objects to design a database scheme [15]. An *object* represents a unit of retrieval. Consider attribute *A* which is playing a role of attribute *B*. If there is no object containing both *A* and *B*, they could not pose a query about both *A* and *B*. In [14] they show how to use role relationships to make this connection. The solution of [14] can be seen as information hiding. Either *A* or *B* is seen connected to the rest of the database scheme, but not both. Information about *A* is visible and information about *B* is hidden or vice versa. This poses certain problems in their approach as they point out. Certain queries are computable, but not recognized as such by their algorithm for translating queries. An example of such a query is "find all managers and their salary" posed over a database scheme as given in Fig. 1. Salary is always seen as a property of employee which overrides the fact that managers have salaries too. In our approach, we diagrammatically show that employee and manager are over the same domain. We give a method for naming manager-name and manager-salary distinctly from employee-name and employee-salary so that one or the other can explicitly appear in a query. Finally, we retain the notion of functional dependencies which are not explicitly used in their approach.

One advantage we glean from correctly representing roles is the ability to express recursive queries over the universal relation. When two or more attributes are over the same domain and play distinct roles, a query may be interpreted as being recursive by a universal relation interface. As a basic yardstick for measuring the power of relational query languages, Codd introduced the relational algebra and relational calculus [16, 17]. These languages were augmented with aggregate operators [18], a transitive closure operator [19], and a least fixed point operator [20].

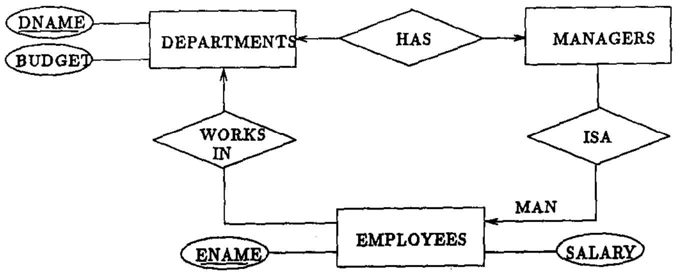

Fig. 1. A sample *ER* diagram.

Agrawal [21] introduced an alpha operator which allowed, in addition to the power of relational algebra, the expression of recursive queries. Recursive queries are more general than transitive closure since aggregations, selections, and other operations can be performed on the transitive closure.

The query language we give is useful for universal relation interfaces such as [3, 15, 22–28] as well as for such a system as Scrabble [29] which is an entity-relationship oriented microcomputer DBMS. Even though Scrabble is based on the functional dependency model [30], it also makes the universal relation scheme assumption [25]. The common trait among these systems is that they allow the user to specify what data is to be retrieved without needing to specify how to retrieve the data. Queries are formulated using only attribute names (and not relation names) in the target-list and in the selection (no range formula appears).

The query language we give is a relational calculus for a database that consists of a single relation. It is a universal relation version of the relational algebra, *Alpha*, given in [21]. Our language is not as powerful as *Alpha*. A recursive query can compute flight plans from New York to Los Angeles using a relation of direct flights, even when there are no direct flights between the two cities. In *Alpha*, once a particular flight plan is obtained, any relational expression can be applied to that flight plan. In our language, we restrict the operations that can be applied to the flight plan to select, project, and join (the attributes in the flight plan range over the universal relation implicitly, so the join operation is implicit). An example of a query requiring a more expressive relational expression is one that asks for flight plans such that every city that exists in the database, occurs in the flight plan.

Our language makes use of the inherent ordering among the tuples in the flight plan. This feature makes certain queries easier to express. We show how to express queries that take advantage of that ordering, for example a query that puts constraints on the layover time between flights.

In this paper we tackle a subset of the research that could lead to accomplishing all our goals as mentioned at the start of this paper. Below, we enumerate the topics of this paper. The first three topics are covered in Section 2.2, the fourth topic in Section 2.5, and the fifth topic in Section 3.

(1) We show how to modify the *ER* diagram so that every occurrence of the same attribute has the same meaning—this is the unique role assumption, a fundamental assumption of a universal relation interface. Once this modification is made, the diagrams are referred to as *Extended ER* diagrams (abbrev. *EER* diagrams).

(2) We show how to relax the unique role assumption of the universal relation interface so that an attribute can play more than one role.

(3) We show how to represent and distinguish between one role and more than one role of an attribute in an *EER* diagram.

(4) We give an algorithm for transforming *EER* diagrams that erroneously convey the impression that two attributes are over distinct domains, to *EER* diagrams that show the two attributes over the same domain playing distinct roles.

(5) With our relaxing of the unique role assumption, recursive queries can now be expressed in a query language that poses queries over the universal relation. We give such a query language. It is a small matter to develop a graphical query language for recursive queries that can be posed over *EER* diagrams that we develop in Section 2.2. We do not give a graphical query language.

2. DOMAINS AND ROLES OF ATTRIBUTES

It is extremely important that the domains and roles of attributes are properly specified. Improper specification of domains and roles can severely limit the allowable queries† or allow meaningless queries to be formulated over the universal relation.‡ We begin by giving an introduction to the *ER* model in Section 2.1. In Section 2.2 we show how to explicitly represent roles of attributes in an *EER* diagram. In Sections 2.3 and 2.4 we show how to determine, through an interactive process with an end-user, whether two attributes are over the same domain and play distinct roles, respectively. The strategies we give are only useful for domains whose elements are character data.§ A more sophisticated strategy is required for numeric data and is outside of the scope of this work. Finally, in Section 2.5, we show how to transform an *EER* diagram D with respect to two attributes A_1 and A_2 which appear to be over distinct domains. The algorithm transforms D into a diagram $\hat{D}$ indicating that A_1 and A_2 are over the same domain and play distinct roles.

2.1. *The entity-relationship model*

In the *ER* model we can represent entities and relationships [33]. An *entity* is a real-world object which is distinguishable. A group of similar entities forms an *entity set*. Entities have *attributes*. Associated with an attribute is a *datatype* (e.g. integer, real) and a *domain* which is a semantic description in a natural language, like English. For example, the

†For instance, unless it is understood that an attribute plays more than one role, recursive queries posed over the universal relation are not possible.

‡If it appeared to the universal relation interface that department name and employee name were over the same domain but playing distinct roles, meaningless recursive queries could be formulated. The number of tuples in each Δ, as described in Section 3, would be one.

§We have implemented this strategy in *ER-Easy*, a graphical interface for the design of *EER* diagrams [31, 32].

domain of an attribute for employee name could be "all names of persons" and the datatype could be "character of length 20".

We can represent a database scheme using an *ER* diagram where entity sets are represented by rectangles and attributes are represented by ovals. A *key* is a set of attributes that distinguish entities of an entity set. In the algorithms of this section, when two (or more) attributes are over the same domain, we determine whether one attribute is a generalization of the other attribute. More precisely, we must consider the entity set of the most general attribute, and not the attribute itself. This question only makes sense if the attribute is a key of the entity set and the value of the attribute is synonymous with the entity being represented. So, we introduce a new term called a *descriptive key* which is reserved for those keys which describe the entity set. Ordinarily, we need not (and do not) distinguish between a descriptive and non-descriptive key. We use the convention of underlining descriptive keys with a solid bar and non-descriptive keys with a dashed bar in *ER* diagrams.

Example 1. In Fig. 2 we see the entity set *DEPARTMENTS* with attributes *DNAME*, *MANAGER*, and *BUDGET*. There are two keys *DNAME* and *MANAGER*, but only *DNAME* is a descriptive key. "Computer Science" easily identifies the department of the authors to the man on the street, whereas "C. W. Gear" would not. ■

A *relationship* is an ordered list of entity sets expressing a real-world correspondence among the entity sets participating in the relationship. In an *ER* diagram the participation of an entity set E in a relationship R is expressed by an edge or an arc from R to E. An arc is directed toward E when a distinct entity, of another entity set participating in R, can be mapped to exactly one entity of E. Three classes of relationships, one-to-one, many-to-one, and many-to-many we represent as illustrated by the "PRESIDENT", "WORKS-IN", and "TAKES" relationships, respectively, of Fig. 3.

The ISA relationship [34–36] from E_1 to E_2 is a built-in relationship that is a special case of a one-to-one relationship. However, unlike one-to-one relationships, we only direct the arc to the more general entity set. An ISA relationship expresses a hierarchy between two entity sets E_1 and E_2. E_2 is a *generalization* of E_1 and also, E_1 is a *specialization* of E_2. E_1 inherits all attributes of E_2. The keys of E_1 are

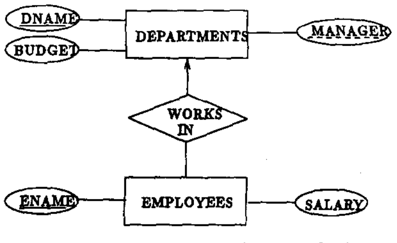

Fig. 2. *MANAGER* as an attribute of the entity set *DEPARTMENTS*.

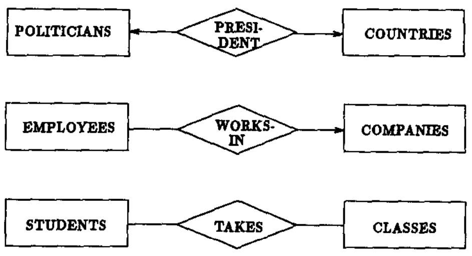

Fig. 3. Relationships.

the keys of E_2. When discussing ISA relationships, we use attribute and entity set interchangeably. In Fig. 4 "PERSONS" is a generalization of "PILOTS" (i.e. every pilot is a person). We label the arc with a *role-name* which is defined in the next section.

Entity sets $E_1, \ldots, E_n$ form a (directed) *path* in the database scheme if for all i $(1 \leq i < n)$, there is a many-to-one or one-to-one relationship from E_i to E_{i+1}. If there is a path from E_i to E_j then E_j is an *ancestor* of E_i.

2.2. *Explicitly representing roles*

We use ISA relationships to explicitly represent each role of an attribute. We distinguish three different semantic relationships that any two attributes can have with respect to each other and will then show how each is represented in the *EER* diagram: (1) Attributes can be over different domains and hence will play different roles. (2) Attributes can have the same domain and play the same role. (3) Attributes can be over the same domain but play distinct roles.

When two attributes have distinct domains then they are required to be named distinctly. For example, attributes for employee name and department name should be distinct.

When two or more attributes are over the same domain and play the same role, the meaning is that, in the universal relation r, a single column for those attributes is sufficient since we expect a tuple of r to agree on all these attributes. We really only have a single attribute, and all the occurrences of the attribute should have the same name in the *EER* diagram. By adding the semantics of the unique role assumption to *ER* diagrams, we are transforming them into *EER* diagrams.

Example 2. For example, consider the *EER* diagram of Fig. 5. *STATE* is an attribute of the entity sets *CUSTOMERS* and *PRODUCTS*. The following semantics can be inferred from the syntax of the diagram. A customer lives in a particular state and a product is priced for sale distinctly in each state. Both occurrences of *STATE* are playing the same role. A customer must buy a product at the price for the state

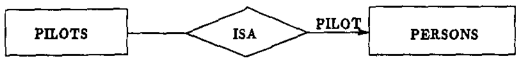

Fig. 4. An ISA relationship.

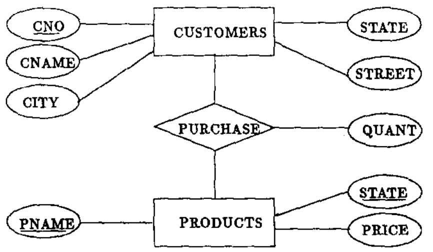

Fig. 5. *STATE* plays the same role in both entity sets.

in which he resides. If customer Jones buys a computer in the state of Illinois for $2000, we know that Jones lives in Illinois, and the computer is priced for sale in Illinois for $2000. ■

When two or more attributes $A_1 \ldots A_n$ are over the same domain and play distinct roles either the roles are explicitly seen as distinctly named attributes or the roles are implicitly defined by distinct paths from an entity set E_i to E_j.† In the former case, either one attribute is a generalization of the other attributes or we can introduce another attribute A_{n+1} which would be a generalization of $A_1 \ldots A_n$.‡ We expect ISA relationships to represent this hierarchy. To create a hierarchy, each A_i $(1 \leqslant i \leqslant n)$, must be a key of an entity set E_i. Such an E_i can be created if it does not already exist. (The algorithm for transforming the *EER* diagram is the topic of Section 2.5.) We call the more general entity set the *multi-role entity set* and the specialization entity sets, *role entity sets*.

Example 3. In the diagram of Fig. 6 it appears as if *FROMCITY* and *TOCITY* are over distinct domains. However, they are over the same domain and play distinct roles. *FROMCITY* could not be a generalization of *TOCITY*, nor vice versa. We can introduce another attribute *CITY* which is a generalization of *FROMCITY* and *TOCITY*. The diagram of Fig. 7 represents the hierarchy with role entity sets *FROMCITIES* and *TOCITIES*, and multi-role entity set *CITIES*. ■

When the roles are implicitly defined by distinct paths from an entity set E_i to E_j, E_j is the multi-role entity set. Two role entity sets are added to the diagram to take the place of E_j on the paths from E_i to E_j. The role entity sets are specializations of the general entity set E_j represented by ISA relationships as we saw before.

Example 4. Consider the *EER* diagram of Fig. 8. There is a path from *FLIGHTS* to *CITIES* through the relationship *FC* and another through the relationship *TC*. This diagram should also be transformed into the diagram of Fig. 7. ■

The attributes of the multi-role entity set *M* are called *multi-role attributes*. For each role that

†A combination of the two may occur. For simplicity, we treat the two cases separately.

‡Each A_i $(1 \leqslant i \leqslant n)$ may occur in several entity sets and play the same role in each entity set. For simplicity, we assume each A_i occurs only once.

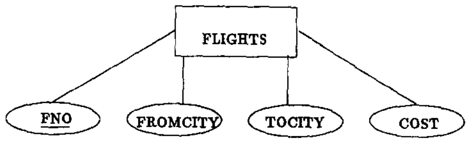

Fig. 6. A *FLIGHTS* entity set with attributes.

M plays, we expect a distinct occurrence of multi-role attributes in the universal relation. (The multi-role attributes themselves occur in the universal relation.) To distinctly name each such attribute, for purposes of this paper, we choose a *role-name* for the role entity set and prefix each multi-role attribute seen in the diagram with the role-name. The result is a set of *role attributes* for each role entity set. Role-names label the arcs emanating from ISA relationships and directed toward the multi-role entity set within the *EER* diagrams.

Example 5. Consider the diagram of Figure 9. *CITY* and *POP* are multi-role attributes. *FROM* and *TO* are role-names labeling the arcs from entity sets *FROMCITIES* and *TOCITIES*, respectively, to the multi-role entity set *CITIES*. Role attributes that we expect in the universal relation scheme are *FROMCITY*, *FROMPOP*, *TOCITY*, and *TOPOP*. ■

Attributes of entity sets that are ancestors of a multi-role entity set are also multi-role attributes and hence, role attributes are created from them for each role entity set.

Example 6. Consider Fig. 9. Given a specific entity *Urbana* of role entity set *FROMCITIES*, there is exactly one entity *Illinois* of entity set *STATES* related to *Urbana*. The reason is because of the many-to-one relationship from multi-role entity set *CITIES* to *STATES* and the ISA-relationship between *FROMCITIES* and *TOCITIES*. Similarly, entity *Los Angeles* of role entity set *TOCITIES* is related to only entity *California* of *STATES*. The *EER* diagram of Fig. 9 would also have role attributes *FROMTAXRATE*, *TOTAXRATE*, *FROMSTATE*, and *TOSTATE*. ■

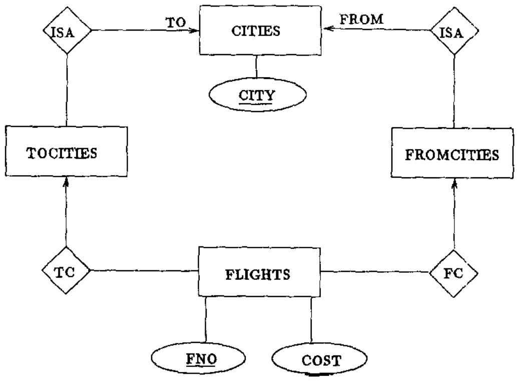

Fig. 7. *FLIGHTS* database scheme augmented with role-defining entity sets.

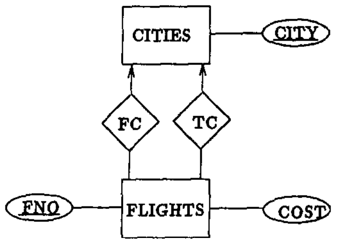

Fig. 8. The two paths from *FLIGHTS* to *CITIES* implies *CITIES* plays two roles.

2.3. Determining same domains

When a user adds an attribute A to the database scheme, he should begin by specifying a datatype t. Then, the user should consider a *domain list* containing those domains already specified for attributes of datatype t. If the domain of A is in the domain list, the user should choose that domain for the new attribute and not enter a second name for the same domain. Otherwise, a new domain d is associated with A and d is added to the domain list for the datatype t.

Suppose that the user chooses an existing domain d_1 which was already associated with an attribute B. To verify whether or not A and B have the same domain, we can ask for sample values a and b for attributes A and B, respectively. Next, we ask the user the questions† (1) Is a an acceptable value for B? (2) Is b an acceptable value for A? When the answer to both questions is "yes", then the two attributes indeed have the same domain. When the answer to both questions is "no", then the domain specification was too broad. In this case, two specifications are needed and each one will describe a smaller set of values than the original.

Example 7. If the user started out by specifying the domain for employee names as "all possible names" and then specified department names as being over the same domain, the original domain specification was too broad. The questions‡ (1) Is Peter Smith an acceptable value for department name? (2) Is Computer Science an acceptable value for employee name? are both answered with a "no". The domain for employee names should be "all possible names of people" and the domain for department names should be "names of academic disciplines". ■

When one answer is "yes" and the other answer is "no", we assume that the domains are the same and then carry out additional inferencing to see if the roles are distinct as discussed in Section 2.4. One is tempted to go beyond establishing the domains as being the same and interpret this response as saying

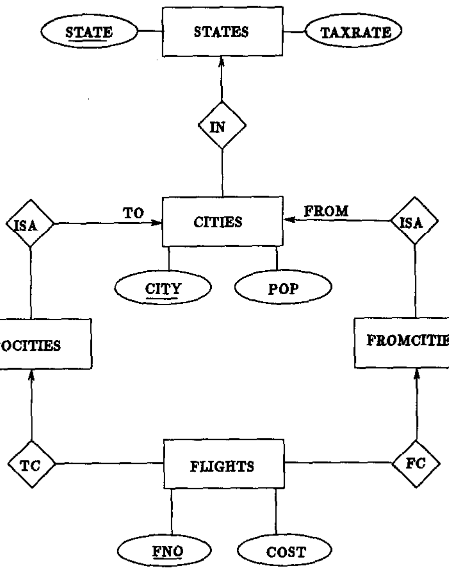

Fig. 9. *FLIGHTS* database scheme augmented with information about *STATES*.

that one attribute, for instance A, is a generalization of the other attribute B. However, the answer we receive is dependent on the sample data and hence it would be possible to obtain a different answer based on a different set of data. For example, if our two attributes are employee names and manager names, the user may choose person names from their own work environment. The two names may be those of a manager and an employee. The answers to our questions could differ, depending on the sample data. This is an instance of attributes playing distinct roles, but to determine this, we need a different type of question which we describe in Section 2.4.

2.4. Determining distinct roles

Only when two attributes A and B are over the same domain does it make sense to ask whether or not A and B play the same role. Suppose that A is an attribute of E_1 and B is an attribute of E_2. E_1 and E_2 are not necessarily distinct. The question is whether or not A and B always are expected to have the same value in a tuple of the universal relation. Worded another way, consider entities e_1 and e_2 of entity sets E_1 and E_2, respectively, such that there is some relationship between e_1 and e_2 in the database. Do we expect A and B to have the same value in e_1 and e_2? If so, then A and B are playing the same role. Otherwise, A and B are playing distinct roles.

Example 8. Consider the *FLIGHTS* entity set of Fig. 6, and the question, "Must a flight have the same value for *FROMCITY* and *TOCITY*?" The answer is "no", so we can assume that *FROMCITY* and *TOCITY* are playing distinct roles. ■

2.5. Transforming a diagram to indicate distinct roles

In this section we start with an *EER* diagram D in which the roles an attribute plays are not visually

†One question should be sufficient to determine whether the attributes are over the same domain. For completeness, we give both possibilities.

‡Peter Smith and Computer Science are sample values given by the user for the attributes for employee name and department name, respectively.

communicated by the *EER* diagram. There are two cases to consider. Either the roles are implicitly defined by distinct paths from an entity set E_i to an entity set E_j, or there are two attributes, A and B, which appear to be over distinct domains, but are actually over the same domain and play distinct roles. We discuss these two cases separately in Sections 2.5.1 and 2.5.2. In practice, the algorithms given in both sections need to be applied since the two cases can occur in a single diagram.

2.5.1. Implicit multiple occurrences of attributes. When there are two distinct paths from entity set E_i to entity set E_j, then the attributes K of the descriptive key of E_j could either be playing one role in the universal relation or two roles. Should K be playing one role, both paths must lead to the same entity of E_j. If K is playing two distinct roles, then each path can lead to distinct entities of E_j. In the latter case, we transform the diagram to make the roles explicit. E_j is the multi-role entity set. We create two role entity sets, R_1 and R_2, each of which takes the place of E_j on one of the original paths originating with E_i. We also create an ISA relationship from R_1 to E_j and from R_2 to E_j.

Example 9. In the *EER* diagram of Fig. 8, there is a path from *FLIGHTS* to *CITIES* through the relationship *FC* and another path through the relationship *TC*. *CITIES* is seen as a multi-role entity set. Two role entity sets are introduced, *FROMCITIES* and *TOCITIES* which take the place of *CITIES* in the relationships *FC* and *TC*, respectively, as seen in the diagram of Fig. 7. ISA relationships are introduced to connect *FROMCITIES* and *TOCITIES* to *CITIES* to complete the transformation. ■

As a last note, the two paths cannot have a relationship in common. Otherwise, we would be creating an ISA hierarchy for the true multi-role entity set, and additional ISA hierarchies for all ancestors of the multi-role entity set.

Example 10. Consider the diagram of Fig. 9. Since *CITIES* is already playing two distinct roles, the attributes of *STATES* are multi-role attributes since *STATES* is an ancestor of *CITIES*. We should not create an ISA hierarchy from *FLIGHTS* to *STATES*, even though there are two paths between the entity sets. *IN* is a relationship shared between the two paths and the above rule prevents us from incorrectly transforming the diagram. ■

2.5.2. Explicit attributes over the same domain playing distinct roles. In this section we start with an *EER* diagram D in which A and B appear to be over distinct domains, that is, the attributes are named distinctly. However, A and B should appear to be over the same domain and play distinct roles. A and B belong to the entity sets E_1 and E_2, respectively, in E. E_1 and E_2 are not necessarily distinct. We give an algorithm to transform D into $\hat{D}$ as described in Section 2.2 for attributes that play more than one role.

We begin by considering the descriptive key of E_1. If A is a descriptive key of E_1, then E_1 could be the multi-role entity set. To determine whether or not E_1 is a generalization of B we could ask the user, "Could B b be an entity of E_1" where b is a value for B† (e.g. considering Fig. 2, we ask, "Could *MANAGER* 'Jones' be an entity of *EMPLOYEES*" as in Example 12). If the answer is yes, then E_1 is the multi-role entity set for $\hat{D}$, otherwise, E_1 is a role entity set.‡ The same process should be carried out for E_2 although only E_1 or E_2 can be the multi-role entity set. If neither E_1 nor E_2 are multi-role entity sets, then we create a multi-role entity set having a multi-role attribute C as a descriptive key. For the rest of this section we will refer to the multi-role entity set as M. If E_1 and/or E_2 are already role entity sets, we will refer to them as R_1 and/or R_2, respectively.

The rest of the algorithm is carried out for both A and B. Without loss of generality, we will use A in our discussion and assume that E_1 was neither a multi-role entity set nor a role entity set. The step for creating the role entity set would be omitted otherwise.

We will create a role entity set R_1 and add a relationship E_1R_1 between E_1 and R_1. If A is a key (but not a descriptive key), then E_1R_1 is one-to-one. If A is not prime (i.e. A does not belong to a key) then E_1R_1 is many-to-one from E_1 to R_1. A should be removed from E_1 in both of these cases.

If A is part of a key of E_1 then there are two possible transformations which will accurately represent the constraints represented by the relationships of the original *EER* diagram. One transformation is to make E_1R_1 a many-to-many relationship from E_1 to R_1. In this case, all the non-key attributes of E_1 become attributes of the relationship E_1R_1. A is removed from E_1. Additionally, R_1 must participate in certain relationships in which E_1 participates. The key of E_1 in the original diagram is split between E_1 and R_1 in the new diagram. R_1 must participate in those relationships, R, in which the key of E_1 is also a key of R (i.e. those in which E_1 participates in R in the "many" sense, that is, there is no incoming arc from R to E_1). This transformation can only be applied if E_1 has one key only. A second possible transformation is to merely make E_1R_1 be a many-to-one relationship from E_1 to R_1. A is *not* removed from E_1 in this instance. Since every attribute, and the roles they play, are named uniquely this simple transformation does not cause ambiguity.§

Finally, an ISA relationship should be created from R_1 to M, the multi-role entity set.

†Sample values, like b, for the attributes would be given by the user.

‡If E_1 is a role entity set, A should eventually be removed from E_1 since a role entity set obtains its descriptive key from the multi-role entity set as in all ISA relationships.

§Hence, the concept of a *weak entity set* is not needed.

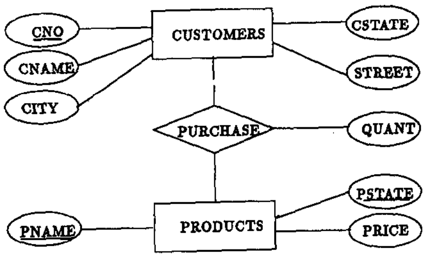

Fig. 10. Attributes *CSTATE* and *PSTATE* play distinct roles.

Example 11. Consider the *EER* diagram of Fig. 10 where the attributes *CSTATE* and *PSTATE* are over the same domain and play distinct roles. (Note, in contrast to Example 2, a customer who lives in Illinois can also buy a product within the state of Michigan.) For this diagram, there are two possible transformations since *PSTATE* is part of the key of *PRODUCTS*. *CSTATE* does not belong to the key of *CUSTOMERS* in Fig. 10 and so, the relationship between *CUSTOMERS* and *CSTATES* is many-to-one after both transformations. After one transformation, as seen in Fig. 11 the relationship *PRICEDIN* between *PRODUCTS* and *PSTATES* is many-to-many. Since *PRODUCTS* participates in relationship *PURCHASE*, *PSTATES* must also participate in *PURCHASE*.† Finally, *PRICE* is transferred from *PRODUCTS* to *PRICEDIN*. The results of the alternative transformation are seen in Fig. 12. Here the transformations are more simple. *PSTATE* and *PRICE* remain in *PRODUCTS* and the relationship *PRICEDIN* is many-to-one from *PRODUCTS* to *PSTATES*. ■

Example 12. Consider the *EER* diagram of Fig. 2. The attributes *MANAGER* and *ENAME* are over the same domain and play distinct roles although the *EER* diagram does not reflect this. Since *ENAME* is a descriptive key of entity set *EMPLOYEES*, *EMPLOYEES* could be a generalization of *MANAGER*. We ask the user "Could *MANAGER* 'Jones' be an entity of *EMPLOYEES*". The answer is "yes", so we create only one role entity set, *MANAGERS*, which is connected to the multi-role entity set *EMPLOYEES* by an ISA relationship, as shown in Fig. 1. Since *MANAGER* is a key, the *HAS* relationship between *DEPARTMENTS* and *MANAGERS* is one-to-one. ■

In summary, we give the algorithm for transforming the *EER* diagram.

Input An *EER* diagram *D* that indicates $A_1 \ldots A_n$ are over distinct domains.

Output An *EER* diagram $\hat{D}$ that indicates $A_1 \ldots A_n$ are over the same domain and play distinct roles.

†The key of *PURCHASE* remains {*CNO*, *PNAME*, *PSTATE*} as it was in Fig. 10.

Algorithm

1. Create a multi-role entity set if none exists. E_j is a multi-role entity set if A_j is a descriptive key of E_j and E_j is generalization of A_i for all i, $1 \leqslant i \leqslant n$, $i \neq j$.
2. Create a role entity set R_i for entity set E_i unless E_i is a role entity set (i.e. A_i is the descriptive key of E_i).
3. Connect role entity set R_i to the multi-role entity set via an ISA relationship for all i, $1 \leqslant i \leqslant n$, except when R_i is the multi-role entity set.
4. Create a relationship E_iR_i from E_i to R_i unless E_i is a multi-role or role entity set. E_iR_i should be one-to-one if A_i is a key, but not a descriptive key, of E_i. E_iR_i should be many-to-one from E_i to R_i if A_i is not prime. In both cases, remove A_i from entity set E_i. When A_i is a proper subset of the sole key of E_i, then either of the following two transformations may be applied. Should E_i have more than one key, then only Transformation 2 should be applied.
   Transformation 1:
   (a) E_iR_i is a many-to-many relationship.
   (b) The non-key attributes of E_i should be transferred from E_i to E_iR_i.
   (c) For every relationship R in which E_i participates in the "many" sense, add an edge from R_i to R.
   (d) Remove A_i from entity set E_i.
   Transformation 2:
   (a) E_iR_i becomes a many-to-one relationship from E_i to R_i.
   (b) A_i is *not* removed from entity set E_i.

3. THE QUERY LANGUAGE

One benefit of the research of Section 2 is the ability to pose recursive queries over the universal relation. In the language we give in Section 3.3 we utilize role names which were introduced in Section 2. Queries are posed over a single universal relation that contains all the attributes in the database scheme. We present the basic query language, without recursion, in Section 3.2. We then present the contribution of this paper in Section 3.3 which is an extension to the basic language to allow the expression of recursive queries.

3.1. Definitions

A *domain* is a set of values. The *Cartesian product* of domains $D_1, \ldots, D_k$ is the set of all k-tuples $(v_1, \ldots, v_k)$ such that v_1 is in D_1, v_2 is in D_2, and so on. A *universal relation* is a subset of the Cartesian product of the domains. A row of the universal relation is a *tuple* and a column is an *attribute*. We give names to the universal relation and to attributes. We will refer to the universal relation by itself as U and also to the universal relation and its attributes a, b, c, as $U(a, b, c)$.

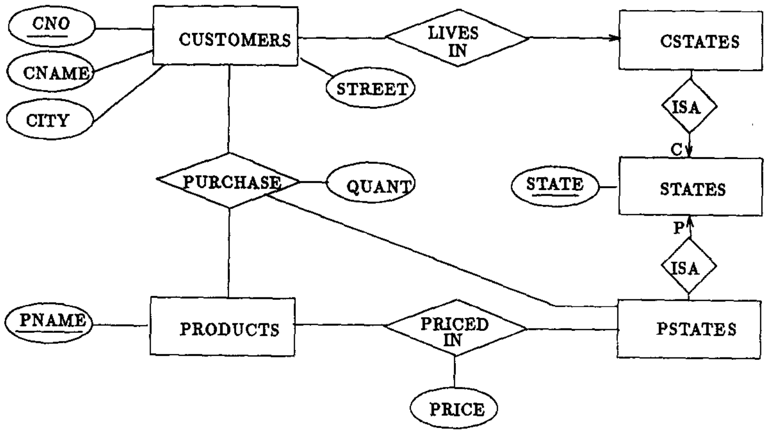

Fig. 11. *CSTATES* and *PSTATES* are added as role entity sets.

Consider a universal relation $U(\ldots, A, \ldots, B, \ldots)$ where the attributes A and B are over the same domain and play distinct roles. The sequences $\ldots$ are place holders for zero or more attributes in U that are not of interest in this discussion. We let $(\ldots, a_i, \ldots, b_i, \ldots)$ represent a tuple in U. A *derivation* of (a_1, b_n) with respect to U is a path

$$\langle(\ldots, a_1, \ldots, b_1, \ldots), \ldots, (\ldots, a_n, \ldots, b_n, \ldots)\rangle$$

where $b_i = a_{i+1}$ for $i = 1, \ldots, n-1$ and the a_i are distinct. We call a_1 and b_n the *endpoints of the derivation* and A and B the *endpoints of the recursion.* We also regard a derivation as an ordered relation δ consisting of the tuples in the path. For all i, j, if $i < j$ then the tuple $(\ldots, a_i, \ldots, b_i, \ldots)$ precedes the tuple $(\ldots, a_j, \ldots, b_j, \ldots)$ in δ. The tuples do not need to be retrieved in this order. The order is an inherent part of the data.

3.2. *The basic query language*

To express SPJ-queries (i.e. select, project, join), we specify an optional list of definitions, a target-list, and a selection [27]. Note that this basic query language does not have a concept of roles. These components of a query are begun with the key words **with**, **retrieve**, and **where**, respectively. The definitions declare new attributes, called *derived-attributes*, which can be calculated from attributes in the database or previously declared derived-attributes. The target-list specifies which attributes, arithmetic function results, and aggregate function results are to be printed. The aggregate functions include **min**, **max**, **avg**, **count**, **sum**. The aggregate functions are always parameterized and produce a single value as a result for each attribute that is used as a parameter.

Example 13. Consider the *EER* diagram of Fig. 6 which has the corresponding universal relation *FLIGHTS* (*FNO*, *FROMCITY*, *TOCITY*, *COST*). The query **retrieve avg**(*COST*) computes the average cost of all flights stored in the database. ■

The selection is an arbitrary logical expression as in FORTRAN or Pascal. The expression can be written in conjunctive normal form which is one or more *conjuncts*. Each conjunct c_j consists of a disjunction of atoms or negated atoms. An atom is built from attributes, derived-attributes, constants, arithmetic operators, and arithmetic comparison operators (e.g. less than, greater than, equal to). When a selection appears in a query, a tuple must satisfy the selection in order to appear in the query result.

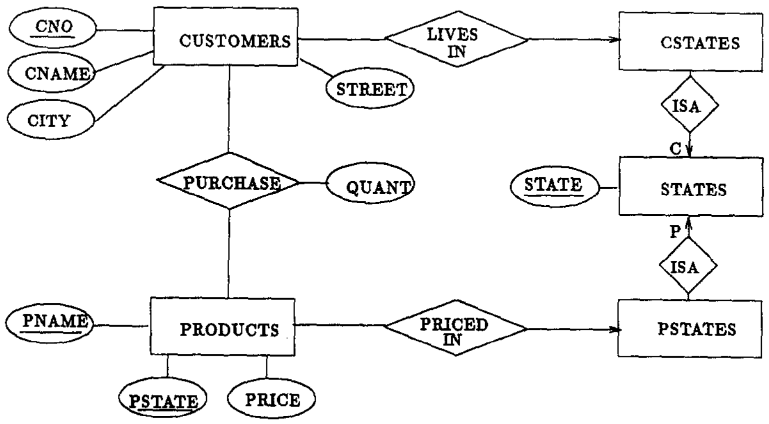

Fig. 12. *CSTATES* and *PSTATES* are added as role entity sets.

Implicit in the query is a *blank tuple variable* that ranges over all the attributes in the universal relation [37].

Example 14. Consider the relation of Example 13. To find the cost, including the tax computed at a rate of 6%, of all flights originating in Urbana, we write

with $TOTALCOST := COST + (COST * .06)$

retrieve $TOCITY\ FROMCITY\ TOTALCOST$

where $FROMCITY$ = "Urbana". ■

3.3. *Expressing recursive queries*

Consider a universal relation U and attributes A and B that are over the same domain and play distinct roles. A recursive query posed over U with respect to A and B is going to produce a set of derivations, $\delta_1, \ldots, \delta_m$. The derivations have a column for every attribute in the universal relation and we name this relation Δ. The query specifies a projection of Δ onto the attributes in a target-list. We refer to this relation as $\dot{\Delta}$. In addition, associated with each derivation, there are values $z_1, \ldots, z_k$ obtained by evaluating recursive aggregate functions $Z_1, \ldots, Z_k$. (We define the syntax and semantics of recursive aggregate functions in Section 3.3.1.) Aggregate function values $a_1, \ldots, a_j$ obtained by evaluating aggregate functions $A_1, \ldots, A_j$ also appear in the result. In summary, the result of the query is a relation, $R(\dot{\Delta}, Z_1, \ldots, Z_k)$ and a set of values for $A_1, \ldots, A_j$.

The syntax of a recursive query posed over U with respect to A and B requires that we specify the endpoints of the recursion. To specify the endpoints for A and B of the recursion we introduce tuple variables t_{init} and t_{final}, respectively. We specify an endpoint for A by equating $t_{\text{init}}.A$ either to a constant value or to the symbol "*". The symbol "*" means to attempt use of all possibilities in the column for A within U as endpoints of derivations. Similarly, to specify an endpoint for B we equate $t_{\text{final}}.B$ either to a constant value or to "*". The expressions for A and B are included as conjuncts in the selection as shown in Example 15.

We use four kinds of tuple variables which are implicitly bound in the following way during a recursive query. Unlike in non-recursive queries, the blank tuple variable is bound to Δ. Attributes do not appear in the result except as a column in Δ. Hence, it makes sense to change the binding of the blank tuple variable to the most commonly used purpose, which is to describe attributes of Δ. Floating tuple variables, $f_1, \ldots, f_j$, which are introduced in Section 3.3.2 are bound to Δ. Recursive aggregate function tuple variables, $r_1, \ldots, r_i$, which are introduced in Section 3.3.1 are bound to Δ. No other tuple variables can be bound to Δ. Tuple variables $t_1, \ldots, t_k$ are bound to U and are restricted from occurring in the target-list. The tuple variables t_{init} and t_{final} must occur in a recursive query and they are bound to U. Once again, they specify the endpoints of the recursion.

FNO	FROMCITY	TOCITY	COST
101	Urbana	Chicago	50
102	Urbana	Dayton	60
103	Urbana	St. Louis	75
200	Chicago	LA	200
201	Chicago	New York	150
300	St. Louis	Chicago	75
301	St. Louis	LA	225
400	Dayton	New York	125

Fig. 13. A relation containing direct flights.

Example 15. Consider the relation given in Fig. 13. The query, "find all flight plans for trips originating in Urbana and terminating in New York" is expressed as

retrieve FNO, $FROMCITY$, $TOCITY$, $COST$

where $t_{\text{init}}.FROMCITY$ = "Urbana" **and** $t_{\text{final}}.TOCITY$ = "New York".

The result of the query is shown in Fig. 14. The result contains three tuples, δ_1, δ_2, and δ_3. We will use the term *flight plan* to describe a tuple in a recursive query result for the flight database throughout the paper. Each tuple in δ_i $(1 \leqslant i \leqslant 3)$, is a *flight*. ■

Example 16. Consider the relation of Example 13. To express the query, "find all flight plans originating in Urbana such that the cost of each flight is less than $100.00" we write

retrieve FNO, $FROMCITY$, $TOCITY$, $COST$

where $t_{\text{init}}.FROMCITY$ = "Urbana" **and** $t_{\text{final}}.TOCITY$ = "*" **and** $COST < 100$.

Since the blank tuple variable is bound to Δ, FNO, $FROMCITY$, $TOCITY$, $COST$ are the columns of $\dot{\Delta}$ and $COST$ is a column of Δ. ■

Example 17. The *EER* diagram of Fig. 15 describes a database that stores distances between two cities, even when there are no direct flights between the cities. The corresponding universal relation is $U(FNO, TIME, CITY, REGION, FROMTIME, TOTIME, COST, FROMCITY,$

	Δ			
	FNO	*FROMCITY*	*TOCITY*	*COST*
δ_1	101	Urbana	Chicago	50
	201	Chicago	New York	150
δ_2	102	Urbana	Dayton	60
	400	Dayton	New York	125
	103	Urbana	St. Louis	75
δ_3	300	St. Louis	Chicago	75
	201	Chicago	New York	150

Fig. 14. A relation showing all flight plans from Urbana to New York.

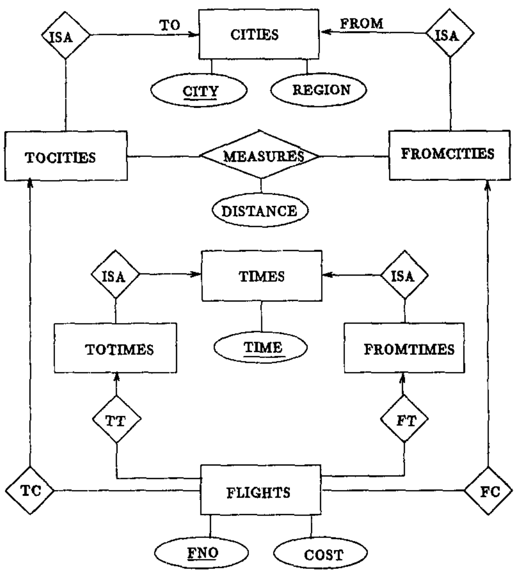

Fig. 15. *FLIGHTS* database scheme including distance between *CITIES*.

FROMREGION, *TOCITY*, *TOREGION*, *DISTANCE*). Consider the query, "find all flight plans from Urbana to New York, but only if the distance between the two cities is less than 700 miles." The query is

retrieve *FNO FROMCITY TOCITY COST*

where t_{init}.*FROMCITY* = "Urbana" **and** t_{final}. *TOCITY* = "New York" **and** t_1.*DISTANCE* < 700 **and** t_1.*FROMCITY* = t_{init}.*FROMCITY* **and** t_1. *TOCITY* = t_{final}.*TOCITY*.

There can be a tuple in *U* that contains Urbana, New York, and the distance even when there is no direct flight between the two cities (in this case, there will be null values for the attributes describing *FLIGHTS*). The tuple variable t_1 is bound to *U*. ∎

3.3.1. Recursive aggregate operators. We define recursive aggregate functions, including, **rmax**, **rmin**, **ravg**, **rsum**, **rcount**, to be those that aggregate values within each δ_i ($1 \leqslant i \leqslant m$). If a recursive aggregate function were to appear in the target-list, we would expect one value for each tuple in the result. The recursive aggregate functions can also appear in the selection.

The aggregate functions introduced in Section 3 are still allowed to appear in the query and they aggregate data from the relation $R(\grave{\Delta}, Z_1, \ldots, Z_k)$. The function **count** gives us *m*, that is, the number of rows in the column for $\grave{\Delta}$. The other aggregate function invocations, **max**(*C*), **min**(*C*), **avg**(*C*), **sum**(*C*) implicitly cause the recursive aggregate function invocation **rsum**(*C*) to be computed in order to produce a single value for each tuple in $\grave{\Delta}$. Then, using these values, the value of the aggregate function invocation is computed. This allows us to compute such values as the average cost of all flight plans, or to find the minimum cost flight plan.

Example 18. Consider the relation of Example 13 and the query, "find all flight plans that originate in Urbana and terminate in New York, the average cost of flights in a flight plan, and the minimum total cost of all flight plans." The query is

retrieve *FNO FROMCITY TOCITY COST* **ravg** (*COST*) **min**(*COST*)

where t_{init}.*FROMCITY* = "Urbana" **and** t_{final}. *TOCITY* = "New York".

The average cost of flights in a flight plan is found by computing the recursive aggregate function **ravg** (*COST*) and the minimum cost of all flight plans is computed by applying the aggregate function **min**(*COST*). The invocation **min**(*COST*) causes the function **rsum**(*COST*) to be computed. ∎

Example 19. Consider the database scheme of Fig. 15 and the query, "Find all flight plans from Urbana to New York such that the distance flown is no more than twice the actual distance between Urbana and New York." The query is

retrieve *FNO FROMCITY TOCITY COST*

where t_{init}.*FROMCITY* = "Urbana" **and** t_{final}. *TOCITY* = "New York" **and** t_1.*FROMCITY* = t_{init}.*FROMCITY* **and** t_1.*TOCITY* = t_{final}. *TOCITY* **and** 2 * t_1.*DISTANCE* ⩾ **rsum** (*DISTANCE*). ∎

We have extended the syntax to include a selection specifically for those tuples to be included in the aggregation. Following a parameter of the aggregate function invocation is a "where" clause. This extension is useful for aggregate functions as well as for recursive aggregate functions.

Example 20. Consider the relation of Example 17 and the query, "find all flight plans from Los Angeles to New York such that there are at least two stops in the midwest". This query appears in [21]. The query is

retrieve *FNO FROMCITY TOCITY*

where t_{init}.*FROMCITY* = "Los Angeles" **and** t_{final}. *TOCITY* = "New York" **and** 2 ⩽ **rcount** (*FNO* **where** *TOREGION* = "MW"). ∎

A recursive aggregate function tuple variable, *r*, can occur within the recursive aggregate function expression, either within the target-list or selection. The tuple variable *r* ranges over the tuples in each δ_i as does the recursive aggregate function.

Example 21. Consider the relation of Example 17 and the query, "find all flights from Urbana to New York such that the layover time is at least 45 min but no more than 90 min." The query is

retrieve *FNO FROMCITY TOCITY COST*

where t_{init}.*FROMCITY* = "Urbana" **and** t_{final}.*TOCITY* = "New York" **and**
rcount (r_1.*TOCITY*) − 1 =
rcount (r_1.*TOCITY* **where** r_2.*FROMTIME* − r_1.*TOTIME* > 45 **and** r_2.*FROMTIME* − r_1.*TOTIME* < 90 **and** r_1.*TOCITY* = r_2.*FROMCITY*). ■

3.3.2. Floating tuple variables. Some queries can be more conveniently expressed when we make use of the inherent ordering among tuples within each δ_i for all i $(1 \leqslant i \leqslant m)$. For example, we can more easily express the layover time between each pair of flights. We introduce *floating tuple variables* that allow us to compare two or more consecutive tuples within a derivation δ_i. We reserve the symbol f, with subscripts, to represent floating tuple variables. If a query contains floating tuple variables $f_1, \ldots, f_k$ then, during the computation of δ_i, the floating tuple variables range over consecutive tuples such that tuple f_j immediately precedes tuple f_k in δ_i for j, $(1 \leqslant j < k)$† and tuple f_k is the newest tuple to be added to δ_i.

In the data definition language (i.e. as part of the database scheme) a relationship among the attributes that are over the same domain and play distinct roles needs to be specified with respect to $f_1, \ldots, f_k$. A relationship between attributes (A, B) would be defined as one or more expressions of the form $f_j.B = f_{j+1}.A$ for all j $(1 \leqslant j < k)$ where A and B are endpoints of the recursion. For example, in a genealogical database scheme with attributes (*CHILD*, *MOTHER*, *FATHER*) over the same domain and playing distinct roles, we may have the relationship specification (f_1.*CHILD* = f_2.*MOTHER*) or (f_1.*CHILD* = f_2.*FATHER*). This would allow a recursive query asking about the hierarchical relationship of all people in the database or only males or only females. For our airline database example we would specify f_1.*TOCITY* = f_2.*FROMCITY*. All such possibilities are used in the evaluation.

Example 22. The query of Example 21 can be more conveniently expressed by using floating tuple variables. The query becomes

retrieve *FNO FROMCITY TOCITY COST*

where t_{init}.*FROMCITY* = "Urbana" **and** t_{final}.*TOCITY* = "New York" **and not** (f_2.*FROMTIME* − f_1.*TOTIME* > 90 **or** f_2.*FROMTIME* − f_1.*TOTIME* < 45). ■

3.3.3. Recursively derived-attributes. We allow the creation of *recursively derived-attributes* of which values are not stored in the database, but are computed from other attributes in the database. Recursively derived-attributes contain expressions that range over Δ.

†That is, $f_i.B = f_{i+1}.A$ where A and B are the endpoints of the recursion.

In the implementation of Ask-easy [27] we allow derived-attributes to be specified at database scheme design time as well as within a query. In the same way, we extend the language to allow *recursively derived-attributes* to be specified at scheme design time and used as if they were attributes in the database scheme. Two advantages are gained by using recursively derived-attributes. First, the expression of a query is less verbose and secondly an end-user can more readily understand the meaning of a recursively derived-attribute over the expression that computes the value of the recursively derived-attribute.

Recursively derived-attributes are only meaningful in a recursive query. The end points of the recursion are expected to be found within a query and so, they need not be specified within a recursively derived-attribute definition. A recursively derived-attribute can be computed by using floating tuple variables.

Example 23. Some useful recursively derived-attributes for the *FLIGHTS* database scheme of Fig. 15 are *NUMBER-FLIGHTS*, *TOTAL-DISTANCE*, *TOTAL-COST*, *MIDWESTSTOPS*, and *LAYOVER*. *LAYOVER* is the only recursively derived-attribute that requires the use of floating tuple variables. The definitions for these recursively derived-attributes are:

with *NUMBER-FLIGHTS*:=**rcount**(*FNO*)
TOTAL-DISTANCE:=**rsum** (*DISTANCE*)
TOTAL-COST:=**rsum** (*COST*)
MIDWESTSTOPS:=**rcount**(*FNO* **where** *TOREGION* = "MW")
LAYOVER:=f_2.*FROMTIME* − f_1.*TOTIME*

We expect an end-user, such as a travel agent, to either query the *EER* diagram of Fig. 16 or the universal relation scheme

U (*FNO*, *TIME*, *FROMTIME*, *TOTIME*, *COST*, *CITY*, *REGION*, *FROMCITY*, *FROMREGION*, *TOCITY*, *TOREGION*, *DISTANCE*, *NUMBER-FLIGHTS*, *TOTAL-DISTANCE*, *TOTAL-COST*, *LAYOVER*, *MIDWESTSTOPS*).

The query of Example 22 can be written as

retrieve *FNO FROMCITY TOCITY COST*

where t_{init}.*FROMCITY* = "Urbana" **and** t_{final}.*TOCITY* = "New York" **and** *LAYOVER* < 90 **and** *LAYOVER* > 45.

The query of Example 19 can be rewritten as

retrieve *FNO FROMCITY TOCITY COST*

where t_{init}.*FROMCITY* = "Urbana" **and** t_{final}.*TOCITY* = "New York" **and** t_1.*FROMCITY* = t_{init}.*FROMCITY* **and** t_1.*TOCITY* = t_{final}.*TOCITY* **and** 2∗ t_1.*DISTANCE* ⩾ *TOTAL-DISTANCE*.

The query of Example 20 can be rewritten as

retrieve *FNO FROMCITY TOCITY*

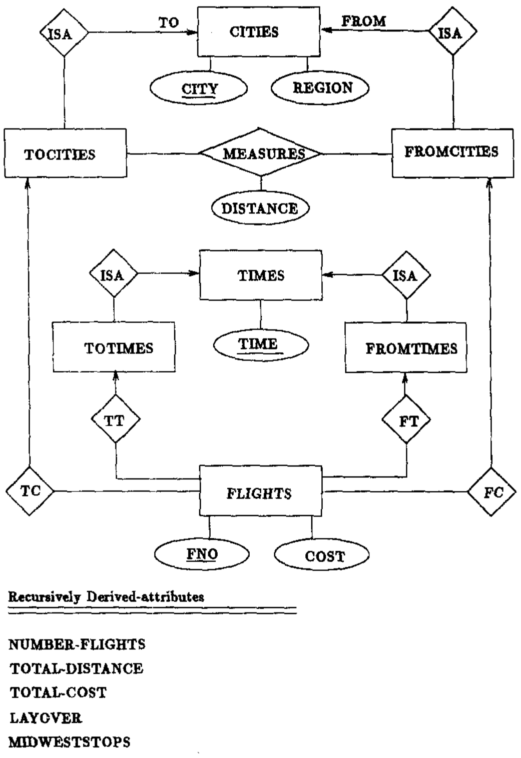

Fig. 16. *FLIGHTS* database scheme with recursively derived-attributes.

where $t_{\text{init}}.FROMCITY$ = "Los Angeles" **and** $t_{\text{final}}.TOCITY$ = "New York" **and** $2 \leqslant MIDWESTSTOPS$. ■

4. CONCLUSION

We have shown how an *EER* diagram can be used to explicitly indicate the attributes that are over the same domain and play distinct roles versus those that play the same role. We have given a strategy for creating such a diagram from a simpler diagram that did not represent the attributes as being over the same domain. Finally, we have given a recursive query language for a universal relation interface.

We have shown that attributes of entity sets that are ancestors of a multi-role entity set are also multi-role attributes and hence, role attributes are created from them for each role entity set. We have omitted discussing many-to-many relationships in which the multi-role entity set participates. For instance, consider the *EER* diagram of Fig. 9. We could add an entity set *CONCERTS*, and a many-to-many relationship between *CITIES* and *CONCERTS*. The question is, are the attributes *X* of *CONCERTS* multi-role attributes or not? (That is, should there be distinct copies of *X* for each role entity set in the universal relation or not?) If *CONCERTS* is *not* a multi-role entity set then must the city in which a concert is playing be the same as the city to (or from) which a flight flys? We believe that the latter situation is the case. However, we must rely on relational design theory to make this point convincing and that is outside the scope of this paper.

Future work includes specifying an algorithm to translate a database with multi-role attributes into a network database scheme, and developing query translation and optimization algorithms for the recursive queries posed over a network database. Also, there are other semantic-packed operators that could be added to the universal relation query language to further achieve the goals as mentioned at the start of this paper. For instance, an *ALL* operator that would allow the expression of such queries as "Find suppliers who supply all parts".

Furthermore, other semantic aid can be provided for the user during database scheme design, such as specifying the functional dependencies. We think the *EER* diagram is a very important interface between the user and a universal relation. The visual display of the entity sets and relationships can aid the user in correctly defining the database scheme.

Interestingly, the language we give for recursive queries is a tuple calculus language for database schemes that have only one relation. In [38], a tuple calculus language is given that suffices for database schemes with many relations.

We have implemented a prototype of *ER-Easy*, a user-friendly database scheme design program on a SUN workstation that includes the ideas of Section 2 [31, 32]. ER-Easy allows the user to specify a database scheme by means of *EER* diagrams and converts this scheme into a network database scheme or a relational database scheme that reflects the structure of the underlying universal relation. ER-Easy employs the inference techniques of Sections 2.3 and 2.4 to prevent scheme layouts that misrepresent roles.

Acknowledgements—We thank Steve Pax for many helpful discussions and Youlun Pang and Rakesh Agrawal for helpful comments.

REFERENCES

[1] P.P.-S Chen. The entity-relationship model: toward a unified view of data *ACM Trans. Database Systems* 1(1), 9–36 (1976).
[2] J. D. Ullman. *Principles of Database Systems* 2nd edn. Computer Science Press, Rockville, Maryland (1982).
[3] N. Azar and E. Pichat. *Translation of an Extended Entity-Relationship Model into the Universal Relation with Inclusions Formalism*, pp. 253–268. Elsevier, Amsterdam (1987).
[4] P. P.-S. Chen. An algebra for a directional binary entity-relationship model. *Proc. Int. Conf. Data Engng, IEEE*, pp. 37–40 (1984).
[5] B. Czejdo and D. W. Embley. *An Algebra for an Entity-Relationship Model and its Application to Graphical Query Processing*, pp. 367–374. Plenum Press, New York (1987).
[6] A. Dogac, F. Eyupoglu and E. Arkun. *Vers—A Vector Based Entity Relationship Database Management System*, pp. 323–343. Elsevier, Amsterdam (1987).
[7] A. Sernadas, J. Bubenko Jr and A. Olive (Editors). *An Entity-relationship Query Language*, pp. 19–32. Elsevier, Amsterdam (1985).

[8] R. Elmasri, J. Weeldreyer and A. Hevner. The category concept: an extension to the entity-relationship model. *Database Knowledge Engng* **1**, 75–116 (1985).

[9] V. Markowitz and Y. Raz. An entity-relationship algebra and its semantic description capabilities. *J. Systems Software* **4**, 147–162 (1984).

[10] C. Parent and S. Spaccapietra. An entity-relationship algebra. *Proc. Int. Conf. Data Engng, IEEE*, pp. 500–507 (1984).

[11] K. Subieta and M. Missala. Semantics of query languages for the entity-relationship model. In *Entity-Relationship Approach to Software Engineering*, pp. 197–216 (1987).

[12] P. Ursprung and C. A. Zehnder. *HIQUEL: An Interactive Query Language to Define and Use Hierarchies*, pp. 299–314. Elsevier, Amsterdam (1983).

[13] C. Zaniolo. The database language gem. *Proc. ACM-SIGMOD Int. Conf. Management of Data*, pp. 207–217 (1983).

[14] D. Maier, D. Rozenshtein and J. Stein. Representing roles in universal scheme interfaces. *IEEE Trans. Software Engng* **11**(7), 644–652 (1985).

[15] D. Maier and D. S. Warren. Specifying connections for a universal relation scheme database. In *Proc. ACM-SIGMOD Int. Conf. Management of Data*, pp. 1–7 (1982).

[16] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM* **13**(6), 377–387 (1970).

[17] E. F. Codd. *Data Base Systems*, pp. 65–98. Prentice-Hall, Englewood Cliffs, N. J. (1972).

[18] A. Klug. Equivalence of relational algebra and relational calculus query language having aggregate functions. *J. ACM* **29**(3), 699–717 (1982).

[19] M. M. Zloof. Query-by-example: operations on the transitive closure. Technical Report RC 5526, IBM, Yorktown Hts., New York (1975).

[20] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Proc. 6th ACM Symp. Principles of Programming Languages*, pp. 110–120 (1979).

[21] R. Agrawal. Alpha: an extension of relational algebra to express a class of recursive queries. In *3rd International Conference on Data Engineering*, pp. 580–590 (1987).

[22] H. F. Korth and J. D. Ullman. System/u: a database system based on the universal relation assumption. *XP1 Workshop on Relational Database Theory* (1980).

[23] D. Maier, D. Rozenshtein, S. C. Salveter, J. Stein and D. S. Warren. Toward logical data independence: a relational query language without relations. In *Proc. ACM-SIGMOD Int. Conf. Management of Data*, pp. 51–60 (1983).

[24] E. Babb. Joined normal form: a storage encoding for relational databases. *ACM Trans. Database Systems* **7**(3), 588–614 (1982).

[25] J. Biskup and H. Bruggeman. Universal relation views: a pragmatic approach. In *Proc. 9th Int. Conf. Very Large Databases*, pp. 172–185 (1983).

[26] H. F. Korth, G. Kuper, J. Feigenbaum, A. van Gelder and J. D. Ullman. System/u: a database system based on the universal relation assumption. *ACM TODS* **9**(3), 331–347 (1984).

[27] H. H. Chen, S. M. Kuck, J. Peterson and Y. Sagiv. A user's manual for AURICAL: a universal relation implementation via CODASYL. Technical Report UIUCDCS-R-82-1114, University of Illinois at Urbana-Champaign, Urbana, Ill. (1982).

[28] Z. Q. Zhang and A. O. Mendelzon. *A Graphical Query Language for Entity-Relationship Databases*, pp. 441–448. Elsevier, Amsterdam (1983).

[29] A. Flory and S. T. March. *SCRABBLE: A Local Database Management System*, pp. 271–286. Elsevier, Amsterdam (1987).

[30] A. Flory and S. T. March. The functional dependency model: a unified approach to information systems development. University of Minnesota Working Paper MISRC-WP-86-05 (1985).

[31] R. John and A. Lewe. ER-Easy: a user-friendly graphic ER-diagram editor for interactive database scheme design. Technical Report UIUCDCS-R-87-1338, University of Illinois at Urbana-Champaign, Urbana, Ill. (1987).

[32] M. Najork. Enhanced ER-Easy. Technical Report UIUCDCS-R-88-1464, University of Illinois at Urbana-Champaign (1988).

[33] P. P.-S. Chen. The entity-relationship model: toward a unified view of data. *ACM Trans. Database Systems* **1**(1), 9–36 (1976).

[34] C. Batini, M. Lenzerini and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys* **18**(4), 323–364 (1986).

[35] J. M. Smith and D. Smith. Database abstractions: aggregation and generalization. *ACM Trans. Database Systems* **2**(2), 105–133 (1977).

[36] T. J. Teorey, D. Yang and J. P. Fry. A logical design methodology for relational databases using the extended entity-relationship model. *ACM Computing Surveys* **18**(2), 197–222 (1986).

[37] J. D. Ullman. *Principles of Database Systems*, 2nd Edn. Computer Science Press, Rockville, Maryland (1982).

[38] S. M. Kuck and S. Pax. A relational calculus with transitive closure. Manuscript in preparation.