*Relational Schema Integration: Dealing with Inter-relation Correspondences and Querying Over Component Relations**

Whan-Kyu Whang ETRI, KOREA,
Sharma Chakravarthy University of Florida, and
Shamkant B. Navathe Georgia Institute of Technology

ABSTRACT: The need for accessing independently developed database management systems using a unified conceptual view has been well recognized. View or schema integration is an important component of this problem of designing heterogeneous or federated database management systems. In the proposed approach, we assume that relations that are known to be inter-related are grouped into a cluster for further processing.

In this paper, we address the problems of: i) identification of relationships among relations grouped into a cluster, ii) checking the consistency of specified relationships, and iii) exhaustively deriving new relationships from the ones specified. These activities lead to a proper merging of relations from component databases into global view relations. Specifically, seven types of relationships (*equal, contains, contained-in, overlap, disjoint, properly contains, and properly contained in*) are considered. Merging of component schemas in a federated database environment can benefit

* This work was supported by a grant from U.S. West Corporation.

substantially from the knowledge of the relationship between relations in a cluster. We present an algorithm to check the consistency of asserted relationships and to derive all possible relationships from the given set of known relationships. We extend the relational query language SQL to support set operations on constituent relations so that queries can be expressed on an "integrated schema" per our approach. The set operations in ESQL (extended SQL proposed in this paper) are expressed nonprocedurally in a single statement, unlike conventional SQL which requires blocks of code and set operations separating the blocks.

Index Terms: Heterogeneous database management systems, Extended SQL, Relationship between entities, Inferring new relationships, Schema integration.

1. Introduction

Schema integration, which is referred to as merging of schemas that have been developed independently, arises in two different contexts [BaLN86, ShLa90, Jac85]:

1) View integration/reconciliation (in logical database design): During the design phase, several views of a database are merged to form a conceptual schema for the entire database. Note that the view here refers to an application's view (also termed an external view) of the database. Reconciliation is generally required before a feasible conceptual schema that can support several external views can be arrived at. The view reconciliation problem is formally treated using database logic in [Jac85]. Inconsistencies that arise in this view reconciliation process can be resolved systematically and a sufficient condition for avoiding inconsistencies is also given in [Jac85].
2) Schema integration (in federated databases): Schemas of pre-existing databases are merged to form a global schema that provides a unified or integrated conceptual view of the

underlying databases. The integration process needs to preserve the semantics of individual databases as well. The global schema can then be used as an external interface to access and modify data in one or more constituents of the federation.

The methodology used for the above two variations of schema integration are quite similar although the metrics used are different. For example, in view integration, the choice of the underlying conceptual schema may favor the most frequent user (by incurring the least overhead in the translations between the external and conceptual levels). On the other hand, the choice of the global schema for database integration is likely to be based on preservation of the semantics of individual databases. Another major difference between the two lies in the way user queries and transactions are processed after integration. In view integration, user queries and transactions specified against each external view are mapped to the requests on the conceptual schema. In schema integration, user queries against the global schema are transformed into requests on the underlying constituent database schemas. The results of these requests are then assembled to produce the result of the original query on the global/unified schema.

Our approach for schema integration consists of four steps: 1) data model conversion, 2) clustering of related entities as in [NaGa82], 3) identification of relationships among relations in a cluster, and 4) merging of relations into a unified/global schema. First, schemas from existing (possibly different) data models are transformed into a common data model to facilitate the integration process. Second, identical or similar real world entities represented in different schemas are grouped together to be generalized into a generic concept. Third, the relationships among relations in the same group are identified for defining a global schema. Finally, relations in each cluster are merged into a global schema using the semantics of attributes and relationships obtained from Step 3. The block diagram of the schema integration process is shown in Figure 1.1 which is self-explanatory.

In this paper, we propose solutions to steps 3) and 4) of the schema integration process. We assume that steps 1) and 2) have already been applied. We also propose a query language (an extension of SQL termed ESQL) that includes constructs for querying the federated database elegantly. Simplicity and expressiveness of ESQL are demonstrated and contrasted with SQL.

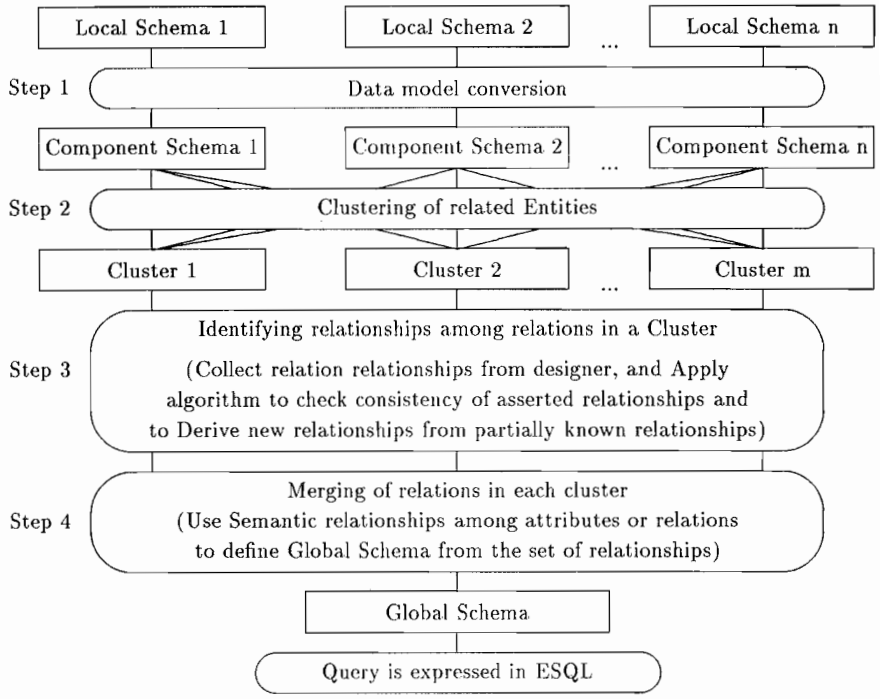

Figure 1: Steps for Creating a Federated Database Schema and Querying it

The remainder of this paper is structured as follows. Section 2 summarizes the schema integration process. Section 3 deals with the seven types of relationships, presents an algorithm to check consistency of the asserted relationships, and to derive new relationships from partially known ones. In Section 4 we introduce an extended relational model to represent a global schema associated with component schemas in a cluster. We also extend SQL by allowing set operations on component relations to query instances of relations related to one another. Section 5 shows an implementation of the algorithm presented in Section 2 using PROLOG. Section 6 presents our conclusions and future work.

2. *The Schema Integration Process*

The *schema integration problem* is as follows. Given a collection of schemas (possibly heterogeneous, developed using different data models), how can one construct a global or unified schema that will

support all of the underlying schemas? The choice (or the construction) of the global schema should not result in a loss of information and should provide the ability to query and/or update underlying databases either individually or collectively. A generalization of this problem, which is not addressed in this paper, is that of dealing with a number of underlying systems not all of which are necessarily databases.

2.1 Data Model Conversion

In many enterprises that require uniform access to independently developed, pre-existing databases, it is the case that constituent database management systems that are being integrated have been developed using different data models. Furthermore, many "legacy systems" in existence today have a variety of files (such as sequential, ISAM, VSAM) which can be incorporated by first specifying them in terms of a relational schema.

Prior to integration, all constituent database schemas need to be converted to a common data model[1]. In this paper, we use the relational model as the common data model[1]. We call the resulting database a *relational federated database*. We use the term *component schema* to refer to a schema that is converted to the relational model from the model used for the local schema. This step is not necessary for a local schema that is already specified using the relational model. In some situations, it may even be necessary to develop the schema of a database-in-use before performing the model conversion. This process is referred to as *reverse engineering*. In fact, there are many instances of databases that have been developed and currently being used for which the schema has not been developed in the first place. Reverse engineering is an important first step that needs to be addressed for realizing a federation or even migration from the current system to a different one.

2.2 Clustering of Relations

In order to integrate component schemas into a unified schema, we need to group the relations from each component schema which represent the same or overlapping real world entity. These individual

1. This is not a general requirement. In [Wha92], Horn clauses have been used as a canonical representation into which relational, hierarchical, and network models are converted.

entities from different component schemas are then merged into a generalized relation. For example, the two relations MAIN_FACULTY and BRANCH_FACULTY in the main campus and branch campus databases, respectively, can be merged into a general concept by the relation, FACULTY. Each such group of relations is called a *cluster* and is integrated independently of other groups. This process simplifies the integration problem.

In this process, determination of relations that belong to a cluster which is based on the semantics of the entities and the requirements of the federated database, is ultimately made by the designer. As component databases are developed at different times by different groups of designers, no assumptions can be made about the choice of names given to entities/relations. Also, if one component schema has a relation SECRETARY and another component schema has a relation ENGINEER, it may be useful to integrate them as an EMPLOYEE relation even though the sets of real world entities represented by them are known to be disjoint. However, this process of grouping cannot be automated completely as it involves interpretation and use of entities in component databases. In general, the semantics implied by the relations are not simply captured by the syntax, but some heuristics can be used to help the designer decide clustering or even automate the process partially. First, relations with the same name are examined to determine whether they indicate the same real world entities. When they have similar names, the data dictionaries including thesaurus can be used to find similarity of concepts. Second, the common key attributes of relations can be used to broadly group the relations. For example, two relations EMPLOYEE and PROJECT having their primary key as Soc_Sec_No and Proj_Name, respectively, cannot be clustered because they do not have a common key. Third, the number of common attributes can also be used as an aid to find the same real world entities. This grouping process, although *ad hoc*, helps in reducing the complexity of the relationship determination problem. In the final analysis, a user assisted tool seems to be a good way of addressing this problem.

2.3 Identification of Relationships

Once the relations from component databases are clustered by the designer, relationships among the relations in a cluster need to be

identified. Possible relationships between any two relations are: EQUAL, CONTAINS, CONTAINED-IN, OVERLAP, DISJOINT, PROPERLY-CONTAINS and PROPERLY-CONTAINED-IN. Determination of the type of relationship that exists between relations in a cluster is essential for the merging of component schemas. Whether the merging process leads to a minimal set of relations depends upon the knowledge of relationships between every pair of relations in the cluster and the heuristics used for merging.

2.4 Merging

After identifying the relationships, the designer needs to indicate the semantic relationships of attributes of the relations in the same cluster in order to merge similar attributes in the global schema [LaNE89, MaEf84, MoBu81]. Consider, for example, attributes GRADE and SCORE taken from two different student databases. GRADE is defined as a character type denoting a letter grade, while SCORE is defined as an integer type representing a value between 1 and 100. Although the names and data types of the two attributes are different, they are semantically equivalent (i.e., they represent the same concept and there exists functions for converting one attribute to the other and vice versa). This equivalence between attributes coming from component databases cannot be determined entirely using the syntax. The designer must ultimately determine the semantic relationships among attributes as rules based on syntax cannot derive such relationships and determine the existence of functions for conversion. Note that there is a difference between merging of attributes and relations. Typically, attribute merging leads to the definition of a function for conversion of values between the domains of the attributes whereas merging of relations requires the use of the appropriate set operators (such as join, union, intersection) to map global queries into queries on component databases. Attribute merging is also discussed as the crucial part of schema integration in models based on classification such as CANDIDE [BeGN89], and is discussed in [ShGN93].

After the semantic relationships are specified for attributes, relations in each cluster are merged obtaining a single federated database schema. We use an extended relational model to define the schema for the federated database in Section 4.1. We use the term *global schema* to refer to a schema that is obtained by integrating component schemas

using the extended relational model. The component schemas themselves are assumed to be in the relational model.

Among the four steps of integration proposed in the introduction, clustering of relations with detection and resolution of conflicts in naming, structure, and domain (steps 1 and 2) are not addressed in this paper. They have been addressed in [BaLe84, LaNE89, MoBu81], and as pointed out in Navathe et al. [NaEL86] this process requires designer intervention as it is subjective and depends on naming conventions used by the designers. In this paper we present an algorithm and techniques for automating part of the merging process (steps 3 and 4). For this purpose, we examine the relationship among relations in a cluster in terms of key attributes in the following section.

3. *Relationships Among the Relations in a Cluster*

The relationship between two relations is determined by the values of key attributes from participating relations. We assume that a common key exists or can be defined between relations R_1 and R_2. In the absence of this (or some other reasonable assumption), there is no easy way to match real world entities represented by the two relations. Depending upon the values of key attributes, we define seven types of relationships (shown below) on their domains. The domain of a relation is the set of tuples in that relation. Recall that some of the known relationships are asserted by the designer. The process of identifying all of the relationships cannot be completely automated for the following reasons: 1) current data models cannot capture real world state information completely, and 2) the semantics or "interpretation" of the schema may differ even when the same data model is used for designing the database depending on the intended use. Observe the similarity of the following definitions with those in [ElNa84] where the Extended ER model was used to define entities and relationships as objects and the domain of an object is the "set of real world entities."

case 1: Identical domains (EQUAL)
R_1 EQUAL R_2 $(R_1 = R_2) \stackrel{def}{=} Dom(R_1) = Dom(R_2)$

case 2: Containing domains (CONTAINS)
R_1 CONTAINS R_2 $(R_1 \text{ c } R_2) \stackrel{def}{=} Dom(R_1) \supseteq Dom(R_2)$

case 3: Contained domains (CONTAINED IN)
R_1 CONTAINED IN R_2 (R_1 ci R_2) $\stackrel{def}{=}$ $Dom(R_1) \subseteq Dom(R_2)$

case 4: Overlapping domains (OVERLAP)
R_1 OVERLAP R_2 (R_1 o R_2) $\stackrel{def}{=}$ $Dom(R_1) \cap Dom(R_2) \neq \emptyset$
$\wedge$ $Dom(R_1) \not\subseteq Dom(R_2)$
$\wedge$ $Dom(R_1) \not\supseteq Dom(R_2)$

case 5: Disjoint domains (DISJOINT)
R_1 DISJOINT R_2 (R_1 d R_2) $\stackrel{def}{=}$ $Dom(R_1) \cap Dom(R_2) = \emptyset$

case 6: properly containing domains (PROPERLY-CONTAINS)
R_1 PROPERLY-CONTAINS R_2 (R_1 pc R_2) $\stackrel{def}{=}$ $Dom(R_1) \supset Dom(R_2)$

case 7: properly contained domains (PROPERLY-CONTAINED-IN)
R_1 PROPERLY-CONTAINED-IN R_2 (R_1 pci R_2) $\stackrel{def}{=}$ $Dom(R_1) \subset Dom(R_2)$

For each pair of relations that belong to the same cluster, a relationship between the two is asserted. Without this relationship information, schema integration cannot be done because the attributes of relations have different semantics depending on the relationships. In general, there are ${}_nC_2 = n*(n-1)/2$ relationships for n relations. Even if the number of relations belonging to the same cluster is not very large, the number of relationships to consider will still be large (for example, for ten relations, the number of relationships to consider will be 45). Hence, if the number of relations to be integrated are large and only some of the assertions are specified by a designer, an algorithm is required to aid the designer to derive new assertions from partially known ones; also there is a need for checking the consistency of assertions that are provided by the designer.

The algorithm presented in this paper is similar to the algorithm developed by Elmasri et al. [ElLN86]. However, the algorithm developed in [ElLN86] is not complete in the sense that only four of the seven types of relationships were considered. The OVERLAP relationship, which is perhaps one of the most commonly occurring relationships, was not considered at all. Furthermore, the algorithm in [ElLN86] stipulates that the application of a transitive rule yields a definite value and not a set of values. The algorithm presented in this paper, in contrast, is complete in the sense that we consider all the seven types of relationships and the final result is more specifically given even in the case when the transitive rule does not produce a

definite value. Our algorithm uses two inputs, namely, a Relationship Assertion (RA) matrix and a Transitive Rule (TR) table. In the RA matrix, the relationship assertions among relations are placed in an n by n matrix, where n is the number of relations belonging to the same cluster. The goal of the algorithm is to fill out the RA matrix with entries for each RA(i,j) indicating possible relationships between relations i and j. Each value RA(i,j) is a subset of possible relationships from the following set:

```
{ =, c, ci, d, o, pc, pci }
```

The entries in the above set correspond to the relationship "equal", "contains", "contained in", "disjoint", "overlap", "properly contains", and "properly contained in" respectively. If the relationship between two relations is not known, then it is represented by the symbol u (for unknown).

Note that "contains" and "contained in" (as well as "properly contained" and "properly contained in") are inverse set relationships; both are included as a convenience to the user who may think of the relationship in one direction only.

Suppose we have three databases containing faculty information of a university. Database 1 (main campus) and database 2 (branch1 campus) both contain the relation FACULTY, while database 3 (branch2 campus) contains three relations, FACULTY, ENGINEERING_FACULTY and ELECTRICAL_ENGINEERING_FACULTY. Their relationships are assumed to be given as follows:

```
(MAIN_FACULTY OVERLAP BRANCH1_FACULTY)
(MAIN_FACULTY DISJOINT BRANCH2_FACULTY)
(BRANCH2_FACULTY CONTAINS BRANCH2_ENG_FACULTY)
(BRANCH2_ENG_FACULTY CONTAINS BRANCH2_ELEC_ENG_FACULTY)
(BRANCH1_FACULTY DISJOINT BRANCH2_ELEC_ENG_FACULTY)
```

These assertions capture inter- as well as intra-schema semantics. The graph representation of these assertions is shown in Figure 3.1. The tabular representation of asserted relationships in the form of an RA matrix is shown in Figure 3.2. In the graph representation, a cluster is a connected graph with relations as nodes and relationship assertions as edges. The label on the thick lines in Figure 3.1 represents an explicitly stated assertion, while the thin lines represents

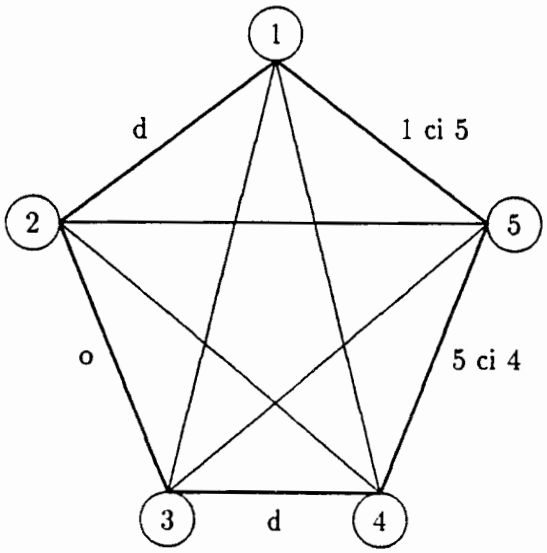

Figure 3.1: Graph Representation of Relationship among Relations

unspecified assertions to be derived. Observe that in Figure 3.1, directionality of the relationship is required only for ci.

The RA matrix is always symmetric; ci and c (as well as pc and pci) represent the same relationship but in different directions.

Transitive Rule (TR) table[2] in Figure 3.3 is used to check consistency of the RA matrix as well as to derive correct relationships from the RA matrix.

Node	1	2	3	4	5
1	=	d	u	u	ci
2	d	=	o	u	u
3	u	o	=	d	u
4	u	u	d	=	c
5	c	u	u	ci	=

Figure 3.2: Relationship Assertion (RA) Matrix of Figure 3.1

2. The relationship unknown denoted by u is purposely omitted from the table as it produces only u as a result of the application of the transitive rule.

E_2 / E_1	=	ci	c	d	o	pci	pc
=	=	ci	c	d	o	pci	pc
ci	ci	ci	ALL	d	{d,o,pci}	pci	ALL
c	c	{=,ci,c,o}	c	{d,o,pc}	{o,pc}	{=,ci,c,o}	pc
d	d	{d,o,pci}	d	ALL	{d,o,pci}	{d,o,pci}	d
o	o	{o,pci}	{d,o,pc}	{d,o,pc}	ALL	{o,pci}	{d,o,pc}
pci	pci	pci	ALL	d	{d,o,pci}	pci	ALL
pc	pc	{=,ci,c,o}	pc	{d,o,pc}	{o,pc}	{=,ci,c,o}	pc

Figure 3.3: Transitive Rule (TR) Table

Let E_1, E_2, and E_3 be relationship assertion symbols between relations R_1 and R_2, R_2, and R_3, and R_1 and R_3, respectively. E_1 and E_2 are used to index rows and columns, respectively on the table shown in Figure 3.3. If E_3 is the symbol for TR(E_1, E_2) in the table, then for any three relations R_1, R_2, and R_3 the following transitive relationship holds:

$$(R_1\ E_1\ R_2)\ \text{and}\ (R_2\ E_2\ R_3) \rightarrow (R_1\ E_3\ R_3)$$

Each of E_1, E_2 can be one of the elements from the set {=,ci,c,d,o,pci,pc} and E_3 is either a single relationship or a subset of relationships from the same set. When an entry in the TR table contains more than one element, it indicates all possible relationships under the corresponding transitive rule. For example, when the relationship E_1 between R_1 and R_2 is ci and the relationship E_2 between R_2 and R_3 is c, then the relationship E_3 between R_1 and R_3 can be either =,ci,c,d,o,pci, or pc; i.e., the set {=,ci,c,d,o,pci,pc} indicated by the entry ALL in TR(ci,c). The algorithm uses this rule table[3] to derive an unspecified assertion (thin edges in Figure 3.1). A derivation for an edge consists of two transitive edges that have only one intermediate node. Thus, the set of derivations for an edge includes all paths of length two. In general, there are n − 2 derivations for any edge, where n is the number of nodes (relations) in a cluster. For example, if there are 10 relations (nodes), then there are 45 edges and for each edge there are 8 cases of transitive rules to consider.

3. The correctness of transitive rules can be easily verified using Venn diagrams and hence is not elaborated further.

3.1 Algorithm

Algorithm 3.1 checks the consistency of the RA matrix specified by the designer and deduces new assertions from the existing ones in the RA matrix.

ALGORITHM 3.1: Algorithm for Checking Consistency of Specified Relationships and Deriving New ones.

INPUT: Relationship Assertion (RA) matrix and Transitive Rule (TR) table.

OUTPUT: Consistency of the RA matrix and new assertions derivable from them.

METHOD: The first portion of the algorithm performs consistency checking for the labeled edges by building transitive paths. If the relationship obtained from the rule in the TR table contradicts with the specified assertion in the RA matrix, an inconsistency message along with the participating relationships and relations is returned to the designer. For the unlabeled edges, the algorithm finds all the transitive paths for each unlabeled edge and checks its consistency. If the intersection of all the derived relationship assertions on the unlabeled edge is empty, it means that there is inconsistency[4] among the specified assertions. If the intersection contains only one element, the assertion is definite (and unique) and is inserted into the RA matrix. This information is used for subsequent steps of processing. This process continues until no more assertions can be derived or any inconsistencies detected. At the last step, the unlabeled edges for which relationships cannot be derived are identified.

```
PROCEDURE Derivation_Of_NewAssertions_And_Consistency_Checking
BEGIN
   LE := all labeled edges in RA matrix;
   NewUE := all unlabeled edges in RA matrix;

   /* check consistency for labeled edges by building
     transitive paths */
```

4. Mutual inconsistency of asserted relationships can also be detected using this algorithm. This is the case when the relationship derived using the transition rule is different from the one specified for an edge.

```
T := all transitive paths in LE;
FOR each transitive path t ∈ T DO
    IF assertion in RA is different from the transitive
      relationship in TR table
       THEN inconsistent_state;

/* check consistency for unlabeled edges and deriving new
  relationships using TR table */
REPEAT
    UE := NewUE;
    FOR each unlabeled edge e_i ∈ UE DO
      BEGIN
        T_i := all transitive paths;
        R_i := ∅;
        FOR each transitive path t ∈ T_i DO
            IF assertion is found in RA matrix
               THEN BEGIN
                       r := transitive relationship in TR
                         table;
                       R_i := R_i ∩ r
                    END
               ELSE R_i := Unknown;
            IF R_i = ∅
               THEN  inconsistent_state
               ELSE IF R_i contains one element
                       THEN BEGIN
                               insert the element in R_i
                                 into RA matrix;
                               NewUE := UE - e_i
                            END;
        END;
  UNTIL UE ≠ NewUE and NewUE ≠ ∅;

  /* for those edges that cannot be derived, a designer
    feedback is required */
  IF R_i contains more than one element or R_i contains
    unknown
     THEN output R_i's;
END
```

3.2 An Example

We shall illustrate the above algorithm using the example introduced earlier in Figures 3.1, 3.2, and 3.3. The list of the labeled edges in

Figure 3.1 is [1-2, 2-3, 3-4, 4-5, 5-1]. There are no transitive paths between the labeled edges having length two. Therefore, we do not need to consider the consistency of relationships in the initial state, but consider the derivation of assertions of unlabeled edges. The initial list of the unlabeled edges is [1-3, 1-4, 2-4, 2-5, 3-5]. For example, the relationship of unlabeled edge 1-3 is obtained as follows:

```
1) edge 1-3

   transitive paths          possible relationships
      d     o
   1-----2-----3             {d, o, pci}
      u     d
   1-----4-----3             {u}
     ci    ci
   1-----5-----3             {u}

   intersection of possible relationships = {d, o, pci}
```

For the edge 1-3, there are three possible transitive paths, 1-2-3, 1-4-3, and 1-5-3. From Figure 3.1, the relationships of edges 1-2, 2-3, 4-3, and 1-5, are initially given as d, o, d, and ci, respectively. The edges 1-4 and 5-3 are unlabeled, therefore it is marked as u denoting unknown. When all the relationships in the edges of transitive paths are identified, the Transitive Rule table in Figure 3.3 is used to derive the relationship of transitive path. By this we mean that if the labels on edges ab and bc are l1 and l2 respectively, the label l3 on the edge ac can be derived as the "transitive relationship" by applying the rule table. For example, for the transitive path 1-2-3 where the relationships of edges 1-2 and 2-3 are d and o, the relationships of edge 1-3 will be one of {d,o,pci} that is found in the TR table in Figure 3.3. If one of the edges is unknown in the transitive path as in 1-4-3 and 1-5-3, the relationship in transitive path results in "unknown." "Unknown" relationship can be taken to be one of the elements, {=,ci,c,d,o,pci,pc}. When all the relationships in transitive paths are identified, the intersection[5] of them will determine the relationship of that edge with the relations corresponding to the nodes. Here the intersection of relationships {d,o,pci}, {u} and {u} results in {d,o,pci}. Derivation of relationship for other edges is shown below.

5. Note that the relationship unknown acts as the set of all possible relationships for the intersection operation.

2) edge 1-4

transitive paths	possible relationships
1 —d— 2 —u— 4	{u}
1 —u— 3 —d— 4	{u}
1 —ci— 5 —ci— 4	{ci}

intersection of possible relationships = {ci}

3) edge 2-4

transitive paths	possible relationships
2 —d— 1 —ci— 4	{d, o, pci}
2 —o— 3 —d— 4	{d, o, pc}
2 —u— 5 —ci— 4	{u}

intersection of possible relationships = {d, o}

4) edge 2-5

transitive paths	possible relationships
2 —d— 1 —ci— 5	{d, o, pci}
2 —o— 3 —u— 5	{u}
2 —u— 4 —c— 5	{u}

intersection of possible relationships = {d, o, pci}

5) edge 3-5

transitive paths	possible relationships
3 —u— 1 —ci— 5	{u}
3 —o— 2 —u— 5	{u}
3 —d— 4 —c— 5	{d}

intersection of possible relationships = {d}

After the first iteration, the relationship of edges 1-4 and 3-5 are unique: {ci} and {d}, respectively. Their relationships are inserted in the RA matrix. These relationships are used to derive relationships of unlabeled edges in the next iteration. The second iteration proceeds with the unlabeled edges, [1-3, 2-4, 2-5], in the same manner as in the first iteration and the results obtained are as follows:

```
For edge 1-3, relationship = {d}
For edge 2-4, relationship = {d,o}
For edge 2-5, relationship = {d,o}
```

After the second iteration, the relationship of the edge 1-3 is known, while that of the edge 2-4 has not changed from the first iteration and that of edge 2-5 is more precisely determined. The third iteration proceeds with the unlabeled edges, [2-4, 2-5]. The third iteration does not yield any relationship having a definite value. Therefore, the algorithm stops here and returns the result to the designer. In conclusion, the relationships obtained from the algorithm are given below and the corresponding RA matrix is depicted in Figure 3.4.

```
(MAIN_FACULTY DISJOINT BRANCH2_ELEC_ENG_FACULTY)
(BRANCH2_ELEC_ENG_FACULTY CONTAINED IN BRANCH2_FACULTY)
(MAIN_FACULTY DISJOINT BRANCH2_ENG_FACULTY)
(BRANCH1_FACULTY DISJOINT or OVERLAP BRANCH2_ENG_FACULTY)
(BRANCH2_FACULTY DISJOINT or OVERLAP BRANCH2_FACULTY)
```

The first three cases give definite values for the relationship, while the last two give two possible relationships.

Node	1	2	3	4	5
1	=	d	d	ci	ci
2	d	=	o	{d,o}	{d,o}
3	d	o	=	d	d
4	c	{d,o}	d	=	c
5	c	{d,o}	d	ci	=

Figure 3-4: Relationship Assertion (RA) Matrix after applying Algorithm 3.1

The main contributions of this algorithm are: i) validating user specified relationships among relations in a federated environment and ii) deriving new relationships from partially known relationships. The transitive rule table provides the inference rules for deriving unspecified relationships; it is not provided by the designer and need not be changed unless new relationships are added. Even when new relationships are added, the algorithm does not change as it is driven by the TR table. Algorithm 3.1 has been implemented in PROLOG. We discuss the implementation in Section 5.

4. *Global Schema and An Extended SQL For Querying*

Earlier work on query languages for federated databases has concentrated on querying against the global schema. For merging component relations, we take the approach of generating a generalization of a relation (entity) from the component relations (entities) in the cluster. Earlier work, however, does not consider the relationship between occurrences of an entity type and occurrences of its subentity types (e.g., information about "intersection" occurrences between two overlapped subentity types or "difference" occurrences by subtracting a particular subentity type from its entity type). Suppose we integrate the databases of state universities described earlier and assume that faculty are allowed to work on more than one campus. From the federated database, the Regents may want to know the names of faculty working in more than one campus, for example; or the names of faculty who are working in 3 specific campuses. To answer these queries, it is necessary to provide set operations on the component relations, which are related to one another in terms of relationships (derived in the previous section) among them.

Although conventional SQL allows set operations on the relations that are union-compatible, it requires several subqueries and a procedural specification to relate those subqueries. In contrast, the set operations proposed in ESQL are expressed nonprocedurally in a single statement and are described below.

In the following two subsections, we propose a mechanism for specifying integrated schemas over relations and ESQL for supporting set operations on component relations, respectively.

4.1 The Global Relational View

As described in Section 3, there are seven possible relationships among relations in a cluster. Once these relationships are known, relations belonging to the same cluster can be integrated into a generalized relation. The generalized relation plays the role of the "global relation" using its "component" relations. We call a generalized relation a *global view relation* and relation constituents of a global view relation as *component relations*. A global view relation can be a component relation at a higher level of abstraction. The attributes that are common (semantically, not just syntactically) to component relations would be the attributes of a global view relation. The names of component relations are preserved in the global view relation. By introducing the concept of a global view relation into the relational model, we are able to perform set operations on the component relations that are named in the global view relation. The global view relation is a metarelation in a sense that it encompasses the relations that participate in that global view relation. However, it is reduced to a traditional relation unless the names of its component relations are explicitly specified.

The names of the global view relation and component relations can be used to denote tuple variables in the FROM clause of ESQL. The general FROM clause syntax can be defined as follows. In this notation, [] denotes one or zero occurrences and symbols enclosed in single quotes denote literals.

```
<Global view relation> [ 'AS' <Component
relation> | <FunctionName>
(' <Relation> ',' <Component relation> ')']
<Relation>: <Global view relation> |
<Component relation>
<FunctionName>: UNION | DIFF | INTS
```

The "AS" construct is used for a global view relation to provide various functions of its component relations. Unless stated explicitly, the global view relation implies the union of its component relations.

As an example, consider the two component relational schemas shown in Figure 4.1 and Figure 4.2. They represent information related to a university database, either on the main campus or on a branch campus. For simplicity, type declarations are omitted. For FACULTY relations in both databases, the first one describes main campus faculty, who have a yearly salary and have an office on the main

```
MAIN_FACULTY(Name, Salary, Office#)
MAIN_OFFICE(Office#, Phone)
MAIN_UNDERGRADUATE(SS#, UndergradName, GPA, Class, Major)
MAIN_GRADUATE(SS#, GradName, GPA, DegreeProgram, Major)
MAIN_ADVISOR(Name, SS#)
MAIN_ENROLLMENT(SS#, CourseName, Grade)
MAIN_COURSE(CourseName, Section#, Time)
```

Figure 4.1: Component Schema at Site 1

```
BRANCH_FACULTY(Name, Salary, Phone)
BRANCH_UNDERGRADUATE(SS#, UndergradName, GPA, Class)
BRANCH_GRADUATE(SS#, GradName, GPA)
BRANCH_ENROLLMENT(SS#, CoureseName, Grade)
BRANCH_COURSE(CourseName, Section#, Time)
```

Figure 4.2: Component Schema at Site 2

```
FACULTY(Name, Salary, Phone, [Main_faculty, Branch_faculty])
UNDERGRADUATE(SS#, UndergradName, GPA, Class,
              [Main_undergraduate, Branch_undergraduate])
GRADUATE(SS#, GradName, GPA, [Main_graduate, Branch_graduate])
STUDENT(SS#, StudName, GPA, [Undergraduate, Graduate])
ENROLLMENT(SS#, CourseName, Grade,
           [Main_enrollment, Branch_enrollment])
COURSE(CourseName, Section#, Time, [Main_course, Branch_course]
```

Figure 4.3: Global Schema

campus. The second one describes branch faculty, who have a monthly salary, but do not have an office. The same person can be both a main campus faculty and a branch campus faculty. The global schema shown in Figure 4.3 provides an integrated view of faculty, in which MAIN_FACULTY and BRANCH_FACULTY are seen as component relations of a generalization hierarchy having FACULTY as a global view relation. The relation FACULTY has three single valued attributes: Name, Phone, and Salary. The information about offices of faculty is assumed to be irrelevant for the global view. The relations MAIN_UNDERGRADUATE and BRANCH_UNDERGRADUATE, and MAIN_GRADUATE and BRANCH_GRADUATE are integrated into the global view relations UNDERGRADUATE and GRADUATE, respectively. The relations UNDERGRADUATE and GRADUATE are integrated into the global view relation STUDENT.

The global schema is shown in Figure 4.3. A list (shown in square brackets) specifies the names of the component relations that participate in the global view relation. This is termed the "relation-name" attribute of the global view relation. This schema representation is similar to GEM [Zani83]. The arithmetic comparison operations, such as $=, \neq, <, \leq, >, \geq$ are not applicable to the "relation-name" attribute, while set operations such as union, intersection, and difference are allowed on this attribute.

There are two options for the visibility of integrated and component schemas in a heterogeneous database depending upon whether component schemas are accessible to the user or not. One option allows the user to access a global schema as well as component schemas. By allowing the user to access component schemas, the user can query attributes of component schemas that are not integrated in the global schema (e.g., `Office#` in Figure 4.1). However, when we join two relations in both global and component schemas, sophisticated query processing is required. This option is not recommended if all the attributes of component schemas are represented in the global schema. The other option allows the user to access only the attributes of the global schema. This option is recommended for the novice users who want to avoid the complexity of many similar attributes in global and component schemas. Whether component schemas should be visible to the user or not is ultimately a matter of policy.

4.2 ESQL as a Query Language

ESQL (Extended SQL) is designed to be a generalization of SQL. Whenever the relation-name attribute in the global view relation is not used, the global view relation is identical to a conventional relation and the syntax of the query is reduced to that of SQL. For example, the following query that retrieves those faculty whose salary is over $50,000 uses the FACULTY as a normal relation.

```
SELECT   f.name
FROM     faculty f
WHERE    f.salary > 50,000
```

Figure 4.4: Find those faculty whose salary is over $50,000.

The global view relation FACULTY is the union of its component relations. Thus the same query can also be generated using component relations as follows:

```
SELECT    f.name
FROM      f IS faculty AS UNION(main_faculty, branch_faculty)
WHERE     f.salary > 50,000
```

Figure 4.5: Same as Figure 4.4

The "IS" construct is used to denote f as a tuple variable for faculty. In relational calculus the above query is expressed as shown below. Note that for those faculty who are both at the main and branch campuses, the annual salary is computed by adding the salary as main_faculty to the salary as branch_faculty computed from monthly salary.

Relational calculus equivalent of the ESQL query is shown in Figure 4.5:

```
{t.name | (t ∈ main_faculty ∧ (∄ u) (u ∈
  branch_faculty
                                        ∧ (u.name =
                                          t.name))
                                ∧ (t.salary > 50,000))
          ∧ (t ∈ branch_faculty ∧ (∄ u) (u ∈
            main_faculty
                                        ∧ (u.name =
                                          t.name))
                                ∧ (t.salary*12 >
                                  50,000))
          ∧ (∃ u) (∃ v) (u ∈ main_faculty ∧ v ∈
            branch_faculty
                        ∧ (u.name = v.name)
                        ∧ (t.name = u.name)
                        ∧ (u.salary + v.salary*12 >
                          50,000)) }
```

The equivalent SQL query is given as follows:

```
SELECT    m.name
FROM      main_faculty m
WHERE     NOT EXISTS (SELECT *
                      FROM branch_faculty b
                      WHERE m.name = b.name)
```

```
AND     m.salary > 50,000
UNION
SELECT  b.name
FROM    branch_faculty b
WHERE   NOT EXISTS  (SELECT *
                     FROM main_faculty m
                     WHERE b.name = m.name)
AND     b.salary*12 > 50,000
UNION
SELECT  m.name
FROM    main_faculty m
WHERE   EXISTS (SELECT *
                FROM branch_faculty b
                WHERE m.name = b.name
                AND m.salary + b.salary*12 >
                  50,000)
```

As we can observe from the above example, the query shown in Figure 4.5 when expressed in SQL requires several subqueries and the user has to know the details of the integration. However, in ESQL, the query is a single statement and expresses the intent in a succinct manner. When a query is invoked, the integration represented internally is mapped to operations of component relations. Query processing and optimization of ESQL queries and the translation of ESQL queries into queries on component relations is discussed in [WhNC91 , WhCN93 , Wha92]. Note that a complete knowledge of component relations is not required to express queries in ESQL; the relation-name field captures sufficient information to help formulate a query without having to know the details.

If we wish to retrieve the salary of the main_faculty whose name is "Smith" in the global view, we can write the query in ESQL as follows:

```
SELECT  f.salary
FROM    f IS faculty AS main_faculty
WHERE   f.name="Smith"
```

Figure 4.6: Find the salary of the `main_faculty` whose name is "Smith" in the global view.

The above query can also be expressed in the following way without an explicit declaration of tuple variable:

```
SELECT  salary
FROM    faculty AS main_faculty
WHERE   name="Smith"
```

Figure 4.7: Same as Figure 4.6

The query "List the names of faculty who work at the main and branch campuses and earn more than $50,000 a year" can be expressed as:

```
SELECT  f.name
FROM    f IS faculty AS INTS(main_faculty, branch_faculty)
WHERE   f.salary > 50,000
```

Figure 4.8: Find those faculty who work at both the main campus and the branch campus, and earn more than $50,000 a year.

In relational calculus the above query is defined as follows:

$$\{t.\text{name} \mid (t \in \text{main\_faculty} \wedge (\exists\ v)\ (v \in \text{branch\_faculty} \wedge (t.\text{name} = v.\text{name}) \wedge (t.\text{salary} + v.\text{salary}*12 > 50,000)))\}$$

For another example, to query the faculty who works only at the main campus one can write:

```
SELECT  f.name
FROM    f IS faculty AS DIFF(main_faculty, branch_faculty)
```

Figure 4.9: Find the faculty who work only at the main campus.

We can define the query in relational calculus as follows:

$$\{t.\text{name} \mid (t \in \text{main\_faculty} \wedge (\nexists\ u)\ (u \in \text{branch\_faculty} \wedge (u.\text{name} = t.\text{name})) \wedge (t.\text{salary} > 50,000))\}$$

Again, if the above queries were to be expressed in SQL, it will require several subqueries.

If we want to know what campus faculty "Smith" belongs to, the query can be written as follows:

```
SELECT  faculty*
FROM    faculty f
WHERE   f.name = "Smith"
```

Figure 4.10: Find the campus in which faculty "Smith" works.

The * operator specifies how the global view relation is composed from its component relations in the global schema. By allowing the component relations in the global schema, we can access the attributes of component schema that cannot be integrated in the global schema (e.g., Office# in Figure 4.1). However, even though both the attributes Salary in MAIN_FACULTY (Figure 4.1) and FACULTY (Figure 4.3) are accessible to the user, the semantics is different. For example, if we want to retrieve salary of main_faculty whose name is "Smith" in the component schema, it will be:

```
SELECT     salary
FROM       main_faculty
WHERE      name = "Smith"
```

Figure 4.11: Find the salary of the main_faculty whose name is "Smith" in the component schema.

If "Smith" happens to work both at the main and branch campuses, Figure 4.11 returns the salary received only at the main campus, while Figure 4.6 returns the salary computed at both campuses.

4.2.1 Aggregate Functions and Grouping

Aggregate functions are important in federated databases, because one might frequently perform aggregate operations on each component relation participating in the global view relation. Consider an integrated database of state universities described earlier. From the federated database, the Board of Regents may want to know the total number of graduate students or the average GPA of each state university. To answer these queries it is necessary to group the global view relation into component relations and apply aggregate functions to each component relation. In SQL, the "GROUP BY" clause is used to group the tuples

that have the same value for some collection of attributes. In addition to the facility provided by conventional SQL, the "GROUP BY" clause in ESQL is used to group the component relations in the global view relation and functions such as AVG, SUM, COUNT, MAX, and MIN can be applied to each component relation independently. In ESQL, the "GROUP BY" clause is used as shown below:

```
GROUP BY COMPONENT
```

By using the keyword COMPONENT, the global view relation is partitioned into component relations which are specified in the relation-name attribute of the global view relation. As in SQL the grouping attributes appear in the SELECT clause, the keyword COMPONENT must also appear in the SELECT clause. For example, the query for finding the total number of graduate students and computing the average of GPA in each of the state university in Florida, can be formulated as follows:

```
SELECT COMPONENT, COUNT(*), AVG(GPA)
FROM       graduate
GROUP BY   COMPONENT
```

Figure 4.12: Find the total number of graduate students and the average GPA for graduate students in each state university

If the global view relation GRADUATE is composed of the graduate students from the component relations University_of_Florida, Florida_State_University, University_of_South_Florida, etc, the answer to the query will be in the format shown below:

COMPONENT(graduate)	COUNT(*)	AVG(GPA)
University_of_Florida	. . .	. . .
Florida_State_University	. . .	. . .
University_of_South_Florida	. . .	. . .
. . .	. . .	. . .

5. *Implementation*

The Algorithm 3.1 (described in Section 3.1) for checking the consistency of relationships and deriving new ones as well as ESQL to SQL

translation has been implemented in PROLOG. In the following we show the implementation of the Algorithm 3.1 given in Section 3.1. In PROLOG, one typically distinguishes between facts and rules. Facts that are considered in the algorithm are component relations in a cluster, relationships among component relations, and the transitive rule table. The representation of relations in a cluster, their relationships, and the transition rule table in PROLOG are described below:

(a) Relations in the same cluster are asserted as follows:

```
relations([main_f, branch1_f, branch2_f, branch2_eng_f,
  branch2_ele_eng_f]).
```

The predicate "relations" contains the name of the relation that belongs to the same cluster.

(b) Relationship between a pair of relations is asserted as follows:

```
relationship(main_f, branch1_f, o).
relationship(main_f, branch2_f, d).
relationship(branch1_f, branch2_ele_eng_f, d).
relationship(branch2_ele_eng_f, branch2_eng_f, ci).
relationship(branch2_eng_f, branch2_f, ci).
```

The predicate "relationship" contains a pair of relations in the first and second arguments, and the relationship between them in the third argument. For example, `relationship(main_f, branch_f, o)` means that the relationship value between the pair of relations MAIN_FACULTY and BRANCH_FACULTY is "overlap."

(c) Before representing the Transitive Rule table shown in Figure 3.4 using Horn clauses, let us consider again the meaning of the table. If for three relations R_1, R_2 and R_3, E_1, E_2 and E_3 are relationships between R_1 and R_2, R_2 and R_3, and R_1 and R_3, respectively, then the following transitive relationship holds:

$$(R_1\ E_1\ R_2)\ \text{and}\ (R_2\ E_2\ R_3) \rightarrow (R_1\ E_3\ R_3)$$

Each of E_1, E_2, and E_3 can be one of the following elements, {=, c, ci, d, o, pc, pci}. The predicate "`transitive_rule`" represents the relationship among E_1, E_2, and E_3 as follows:

```
transitive_rule(E1, E2, E3).
```

For example, transitive_rule(c,d,[d,o,pc]) means that for the three relations R_1, R_2, and R_3, if the relationships between R_1 and R_2, and R_2 and R_3, are c (CONTAINS) and d (DISJOINT), respectively, then

the relationship between R_1 and R_3 should be one of the elements, {d,o,pc}. The Transitive Rule table in Figure 3.4 is asserted as follows:

```
transitive_rule(e, e, [e]).
transitive_rule(e, ci, [ci]).
transitive_rule(e, c, [c]).
transitive_rule(e, d, [d]).
transitive_rule(e, o, [o]).
transitive_rule(e, pci, [pci]).
transitive_rule(e, pc, [pc]).

transitive_rule(ci, e, [ci]).
transitive_rule(ci, ci, [ci]).
transitive_rule(ci, c, [e,ci,c,d,o,pci,pc]).
transitive_rule(ci, d, [d]).
transitive_rule(ci, o, [d,o,pci]).
transitive_rule(ci, pci, [pci]).
transitive_rule(ci, pc, [e,ci,c,d,o,pci,pc]).

transitive_rule(c, e, [c]).
transitive_rule(c, ci, [e,ci,c,o]).
transitive_rule(c, c, [c]).
transitive_rule(c, d, [d,o,pc]).
transitive_rule(c, o, [o,pc]).
transitive_rule(c, pci, [e,ci,c,o]).
transitive_rule(c, pc, [pc]).

transitive_rule(d, e, [d]).
transitive_rule(d, ci, [d,o,pci]).
transitive_rule(d, c, [d]).
transitive_rule(d, d, [e,ci,c,d,o,pci,pc]).
transitive_rule(d, o, [d,o,pci]).
transitive_rule(d, pci, [d,o,pci]).
transitive_rule(d, pc, [d]).

transitive_rule(o, e, [o]).
transitive_rule(o, ci, [o,pci]).
transitive_rule(o, c, [d,o,pc]).
transitive_rule(o, d, [d,o,pc]).
transitive_rule(o, o, [e,ci,c,d,o,pci,pc]).
transitive_rule(o, pci, [o,pci]).
transitive_rule(o, pc, [d,o,pc]).

transitive_rule(pci, e, [pci]).
transitive_rule(pci, ci, [pci]).
```

```
transitive_rule(pci, c, [e,ci,c,d,o,pci,pc]).
transitive_rule(pci, d, [d]).
transitive_rule(pci, o, [d,o,pci]).
transitive_rule(pci, pci, [pci]).
transitive_rule(pci, pc, [e,ci,c,d,o,pci,pc]).

transitive_rule(pc, e, [pc]).
transitive_rule(pc, ci, [e,ci,c,o]).
transitive_rule(pc, c, [pc]).
transitive_rule(pc, d, [d,o,pc]).
transitive_rule(pc, o, [o,pc]).
transitive_rule(pc, pci, [e,ci,c,o]).
transitive_rule(pc, pc, [pc]).
```

With the "transitive–rule" predicate, we can now represent relationship by transitive path in the form of a Horn clause as follows:

```
relationship_by_transitive_path(R1,R2,R3,E3) :-
     relationship(R1,R2,E1),
     relationship(R2,R3,E2),
     transitive_rule(E1,E2,E3).
```

Clauses of (a) and (b) constitute the input to the algorithm and are asserted by the system dynamically, whereas clauses of (c) state the static information that is used to derive relationships using transitive rules and therefore is asserted once.

With the above inputs, the algorithm is implemented in PROLOG. The top-level clauses of PROLOG are shown below. First it reads relations in a cluster and relationships between a pair of relations. Then it checks the consistency of asserted relationships. If the asserted relationships are consistent, then new relationships are derived from the asserted relationships; otherwise, the process stops and returns the inconsistent relationship to the designer. The built-in predicate "asserta" is used to accommodate dynamically changed relationships during derivation of new relationships. Finally, the built-in predicate "retract" is used to remove from the dynamic databases all clauses whose head matches the relationship.

```
top :-
     read_data,
     find_pairs_asserted(AssertedList),
     find_pairs_not_asserted(NotAssertedList),
     check_consistency_of_asserted_equiv(AssertedList),
```

```
        derive_new_equiv_from_asserted_equiv(NotAssertedList),
        retract_all(pairs(_)),
        retract_all(relationship(_,_,_)).

check_consistency_of_asserted_equiv([]).
check_consistency_of_asserted_equiv([Pair|L])  :-
        Pair  =  [R1,R2],
        relationship(R1,R2,Eq),
        get_transitive_path_from_pair(Pair,TransitivePath),
        check_equiv_from_transitive_path(Eq,TransitivePath),
        check_consistency_of_asserted_equiv(L).

check_equiv_from_transitive_path(Eq,[]).
check_equiv_from_transitive_path(Eq,[TransitivePath|L])  :-
        TransitivePath  =  [R1,R2,R3],  !,
        relationship_by_transitive_path(R1,R2,R3,E),
        (member(Eq,E)
        ;
        writeln('consistency error')),
        check_equiv_from_transitive_path(Eq,L).
check_equiv_from_transitive_path(Eq,[TransitivePath|L])  :-
        check_equiv_from_transitive_path(Eq,L).

derive_new_equiv_from_asserted_equiv(PairList):-
        asserta(pairs(PairList)),
        derive_new_relationship(PairList),
        pairs(NewPairList),
        ((PairList  =  NewPairList),  !
        ;
        derive_new_equiv_from_asserted_equiv(NewPairList)).
derive_new_relationship([]).
derive_new_relationship([Pair|L])  :-
          get_transitive_path_from_pair(Pair,TransitivePath),
          intersection_of_possible_equiv(TransitivePath,Eq),
          Pair  =  [R1,R2],
          writeln(relationship(R1,R2,Eq)),
          (length(Eq,1),  !,  Eq  =  [SingleEq],
          asserta(relationship(R1,R2,SingleEq)),
          pairs(PairsList),  delete(Pair,PairsList,NewPairList),
          asserta(pairs(NewPairList)),
          derive_new_relationship(L)
          ;
          derive_new_relationship(L)).
```

PROLOG implementation enables us to non-procedurally specify the relations in a cluster, relationship among relations in a cluster (the RA matrix), the TR table, and the transitive relationship rule. Changes to the TR table (when new relationships are added or deleted) only change the assertions in (c). Problem specific inputs are provided in (a) and (b).

6. *Conclusions*

We have presented an algorithm to check the consistency of asserted relationships among component relations in a cluster. The algorithm also derives new ones from partially known relationships. Our work extends the work of [ElLN86] by considering more types of relationships and by giving results explicitly as a set when the derived relationship is not unique. The algorithm described in this paper is also implemented in PROLOG (actually KB-Prolog [BoDMP89]), which is suitable for this purpose because the inputs (i.e., relationships and transitive rule table) are asserted as facts and new relationships are derived using rules. The approach presented here can be used as a component of a knowledge discovery [Pia91] system that facilitates the user to uncover relationships for use in different ways.

We have extended the relational model to represent a global view relation from component relations. With simple extensions to the relational query language SQL, ESQL can query the information on the component relations using a set of operators as part of the query. Set comparison operator and "GROUP BY" clauses are applicable to the component relations, while set operations (i.e., union, difference, and intersection) are applicable to the tuples of the component relations. For details on query processing and optimization of ESQL queries refer to [Wha92 , WhCN93].

References

[BaLe84] Batini, C., M. Lenzerini, "Methodology for Data Schema Integration in the Entity-Relationship Model," *IEEE Transactions on Software Engineering,* Vol. 10, No. 6, pp. 650–663, November 1984.

[BaLN86] Batini, C., M. Lenzerini, and S.B. Navathe, "A Comparative Analysis of Methodologies for Database Schema Integration," *ACM Computing Surveys,* Vol. 18, No. 4, pp. 323–364, December 1986.

[BeGN89] Beck, H. W., S. K. Gala, and S.B. Navathe, "Classification as a query processing technique in the CANDIDE semantic data model", in Proc. 5th IEEE Data Engg. Conf., Los Angeles, CA, Feb. 1989, pp. 572–581.

[BoDMP89] Bocca, J., M. Dahmen, G. Macartney, and P. Pearson, "KB-Prolog User Guide", ECRC Munich, 1989.

[BrOT86] Breitbart, Y., P.L. Olson., and G.R. Thompson, "Database Integration in a Distributed Heterogeneous Database System," In *Proceedings of the 2nd International Conference on Data Engineering,* pp. 301–310, 1986.

[CaVi83] Casanova, M., and M. Vidal, "Towards a Sound View Integration Methodology," In *Proceedings of the 2nd ACM SIGACT/SIGMOD Conference on Principles of Database Systems* (Atlanta, Ga.), ACM, New York, pp. 36–47, March 1983.

[ElLN86] Elmasri, R., J. Larson, and S.B. Navathe, "Schema Integration Algorithms for Federated Databases and Logical Database Design," Tech. Rep. No. CSC-86-9: 8212, Honeywell Corporate Systems Development Division, Camden, Minn.

[ElNa84] Elmasri, R., and S.B. Navathe, "Object Integration in Logical Database Design," In *Proceedings of the 1st International Conference on Data Engineering,* pp. 418–425, 1984.

[ElNa89] Elmasri, R., and S.B. Navathe, Fundamentals of Database Systems, The Benjamin/Cummings Publishing Company, Inc, 1989.

[Jac85] Jacobs, B. E., Applied Database Logic: Fundamental Issues (Volume I), Prentice-Hall, Inc., Englewood Cliffs, 1985.

[LaNE89] Larson, P., S.B. Navathe, and R. Elmasri, "A Theory of Attribute Equivalence in Databases with Application to Schema Integration," *IEEE Transactions on Software Engineering,* Vol. 15. No. 4, pp. 449–463, April 1989.

[MaEf84] Mannino, M.V., and W. Effelsberg, "Matching Techniques in Global Schema Design," In *Proceedings of the 1st International Conference on Data Engineering,* pp. 418–425, 1984.

[Pia91] Piatesky-Shapiro, G., "Knowledge Discovery in Databases", (Ed.) *Proc. of 1991 AAAI Workshop,* Anaheim, July 1991.

[MoBu81] Motro, A., P. Buneman, "Constructing Superviews," In *Proceedings of the ACM SIGMOD International Conference on Management of Data,* pp. 54–64, May 1981.

[Motr87] Motro, A., "Superviews: A Virtual Integration of Multiple Databases," *IEEE Transactions on Software Eng.,* Vol. 13, No. 7, pp. 785–798, July 1987.

[NaEL86] Navathe, S.B., R. Elmasri, and J.A. Larson, "Integrating User Views in Database Design," *IEEE Computer,* pp. 50–62, January 1986.

[NaGa82] Navathe, S.B., and S.G. Gadgil, "A Methodology for View Integration in Logical Database Design," In *Proceedings of the 8th International Conference on Very Large Data Bases,* pp. 142–164, 1982.

[SSGN91] Savasere, A., A.P. Sheth, S. Gala, S.B. Navathe, and H. Marcus, "On Applying Classification to Schema Integration," In the *First International Workshop on Interoperability in Mutidatabase Systems,* Kyoto, Japan, pp. 258–261, 1991.

[ShLa90] Sheth, A.P., and J.A. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys,* Vol. 22, No. 3, pp. 183–236, September 1990.

[ShGN93] Sheth, A.P., S. K. Gala, and S.B. Navathe, "On Automatic Reasoning for Schema Integration", International Journal of Intelligent and Cooperative Information Systems (IJICIS), 1993, to appear.

[Wha92] Whang, W.K., "A Logic-Based Approach to Federated Databases", Ph.D thesis, Electrical Engineering Department, University of Florida, Gainesville, January 1992.

[WhNC91] Whang, W.K., S.B. Navathe, and S. Chakravarthy, "Logic-Based Approach to Realizing a Federated Information System", In *Proc. of the First International Workshop on Interoperability in Multidatabase Systems,* Kyoto, 1991.

[WhCN93] Whang, W.K., S. Chakravarthy, and S.B. Navathe, "Query Processing in Federated Databases using Logic-Based approach," in preparation, 1993.

[Zani83] Zaniolo, C., "The Database Language GEM," In *Proceedings of the ACM SIGMOD International Conference on Management Data,* pp. 207–218, 1983.