1988

Query Processing on the Entity-Relationship Graph Based Relational Database Systems.

Hung-pin Chen
Louisiana State University and Agricultural & Mechanical College

Order Number 8819930

Query processing on the entity-relationship graph based relational database systems

Chen, Hung-pin, Ph.D.

The Louisiana State University and Agricultural and Mechanical Col., 1988

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

QUERY PROCESSING ON THE ENTITY-RELATIONSHIP GRAPH BASED RELATIONAL DATABASE SYSTEMS

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of the
Doctor of Philosophy

in
The Department of Computer Science

Hung-pin Chen
B.S., Tatung Institute of Technology, Taipei, Taiwan, 1975
M.S., National Cheng-Kung University, Tainan, Taiwan, 1977
May 1988

ACKNOWLEDGEMENTS

Several people have provided assistance and encouragement during my doctoral work. It is impossible to acknowledge everyone involved. Especially, I would like to express my appreciation of the support and guidance given by my adviser, Dr. Peter P. Chen, Murphy J. Foster Chair Professor of Computer Science. Studying under Dr. Chen has been a distinct privilege, both with formal coursework as well as informal learning situation. His professionalism and unbiased attitude have a great impact on me.

My committee members Dr. John A. Brewer III, Dr. Donald Kraft, Dr. Sitharama Iyengar, and Dr. Leslie Jones have provided many valuable discussions and suggestions. I thank them for their input and support.

Finally, I would like to express special thanks to my wife Shu-Hwa, for her understanding and encouragement during the difficult times of the entire doctoral research.

Table of Contents

ABSTRACT

An *ERG* (Entity-Relationship Graph) can be used to provide a semantic structure to a relational database system. An *ERG* is defined by *local regions*. A *local region* contains two nodes of entity types and a node of relationship type. The semantic constraints of the database represented by the *ERG* (Entity-Relationship Graph) can be used to enforce the global integrity of the database system. A query is mapped onto the *ERG* to obtain an *ERQG* (Entity-Relationship Query Graph). This mapping can be specified by the user by navigating the database or automatically allocated by the system via a universal relation interface. The *ERQG* representation of a query can be semantically decomposed into a sequence of Local Regions. These Local Regins can then be processed according to their order in the query. The ER-semijoin operation is introduced to process this sequence of Local Regions. Using this approach, architectures of database systems are proposed - two-phase interface and one-phase interface. An implementation of a user interface is also discussed.

LIST OF SYMBOLS AND ABBREVIATIONS

$\bowtie$ - natural join symbol.

$\div$ - division symbol.

π - projection symbol.

$\cap$ - intersection.

$\cup$ - union.

$=$ - equality symbol.

$\neq$ - not equal symbol.

$\neg$ - negation symbol.

$\rightarrow$ - implication symbol.

$\wedge$ - "and" symbol.

$\vee$ - "or" symbol.

$\subset$ - contained property.

$\subseteq$ - contained.

$\in$ - member of.

$\forall$ the quantifier "for all".

$\exists$ - the quantifier "there exist".

ϕ - the empty set.

CHAPTER 1
INTRODUCTION

1. ERM and Extended-ERM

The Entity Relationship Model (ERM) was introduced as a semantic model for the *enterprise view* of data [Chen1976]. An enterprise view allows a database designer to view the whole enterprise. The most important benefit of introducing ERM as an enterprise model on database design is that an ERM used as an enterprise schema is more stable and easier to understand than the user schema.

As introduced by Chen, an ERM contains entity types, relationship types, and attributes. An entity type may have an *Existency Dependency* or *ID Dependency* with the other entity types, and such an entity type is called a weak entity type. The existent dependency is that the existence of an entity type depends on the existence of another entity type; the ID dependency is that an entity type cannot be uniquely identified by its own attributes and has to be identified by its relationship with other entity types. The semantics of an ERM can always be represented in the relationship types which connect the entity types. A cardinality is the description of the association between an entity type and a relationship type. The cardinality, which represents connection between a pair of entity types to a relationship type, is one of the fundamental semantics of an ERM, denoted as 1:1, 1:m, m:1 or m:n.

An extended semantic representation of an ERM, which is obtained from modifying the ERM, always depends on the application of an ERM to represent a real world. Thus in order to represent more sophisticated information in the real world, several different types of extended ERM which are related to different applications of information in the real world has been proposed. [Sen1981, Neum1986, Webr1981].

2. A Relational Database System Based on ERM

The semantic structure of an ERM can be converted to implementational models such as network data models, hierarchical data models, and relational data models. Conversion of the structure of an ERM to a hierarchical model or a network model can destroy the semantics carried in the ERM. The mapping of an ERM to the relational data model has the benefit that either an entity type or a relationship type can represent a physical relation. Therefore, each entity type or relationship type of an ERM can be directly mapped into the physical level. [Hawr1984]. Another benefit of the mapping of a relational data model from an ERM is that its entity type and relationship type are related to different semantics. Thus a relational database system based on ERM can be viewed as two levels: conceptual and physical. Since there is no difference between the representation of an entity type and relationship type in the physical level, we may have a unique way of handling entity types and relationship types in the physical level. The semantic difference among entity type, weak entity type, and relationship type can then be built in the conceptual level. Due to the different properties in representing the real world, an entity type or a relationship type may have a semantic linkage to another entity type or relationship type. With distinguishable semantic linkages between entity types and relationship types, the semantic structure of a database represented in ERM can be grouped into *semantic regions*. Such a *semantic region* is discussed in chapter 2.

The four schema approach for database architecture was proposed by Prabuddha [Hase1981, Sen1981]. The traditional three-schema architecture of a database contains external schema, internal schema, and conceptual schema. In the four schema approach, the conceptual schema is decomposed into enterprise schema and canonical schema such that the architecture of database contains external schema, internal schema, enterprise schema, and canonical schema. The enterprise schema is the ERM (Entity-Relationship Model); and the canonical schema is regarded as the data

structure of the enterprise schema. The utility of enterprise schema may enhance the overall conceptual framework of the database system.

A database and knowledge base integrity control method and constraints validation is presented by Nguyen and Qian [Nguy1986, Wied1986]. Nguyen introduced the prototypes and database samples for the control of semantic integrity. We will introduce the semantic control of a relational database system that uses ERM instead of prototypes and database samples. In our application, a relational database system can be viewed as a knowledge based system that contains the knowledge of the conceptual level and the knowledge of the physical relations of the database systems. That is, after the mapping of an ERM into a relational database, the semantics of the database system are preserved as part of the knowledge of a user-friendly interface or knowledge based system. The relational database system whose physical relations are mapped from an ERG, and whose semantics are represented by an ERG in the conceptual level is called RDKER (Relational Database with Knowledge of Entity Relationship Graph).

3. Overview on a Universal Relation

Though the relational data model proposed by Codd can help the user without the necessity of navigating the physical database, it can not remove the need of the user to navigate the logical database. In order to help the user navigating the logical database, many user-friendly interface models such as a natural language interface, a semantic query language [Poon1978, Doug1985], and a universal relation interface are proposed. A universal relation model is always based on a relational data model which may help the user without the need to navigate the logical database. [Brad1985, Davi1984, Kent1983, Ullm1983]. In other words, the purpose of a universal relation is to provide the user a simplified scheme of database. This simplified universal relation model can help the user to make a query without the necessity of understanding

the underlining structure of the relational database. Since the user does not need to know the conceptual structure of the database, all of the attributes in the scheme have to be globally defined. Thus the general universal schema assumption is that all of the attributes in the universal relation should be uniquely and globally defined.

To construct a universal relation interface, there are two different approaches. A universal relation model which treats the whole relational database system as a universal relation or several universal relations is always called a *pure* universal relation approach. That is, a universal relation which uses multivalue dependency and maximal objects as proposed by the Ullman [Davi1984], is a *pure* universal relation approach. Thus a universal relation system, such as system/U [Henr1984], which is based on the theory of maximal objects, multivalue dependency, and chase manipulation is a *pure* universal relation approach. In this approach, the maximal object is defined as the largest set of objects in which the user is willing to navigate [Davi1984, Alfr1979].

Instead of a maximal object, an extension join was proposed by Sagiv for the allocation of the accessing paths of a query on a universal relation interface [Sagi1983]. In Sagiv's approach to the universal relation scheme, the concepts of multivalue dependency and maximal objects are not employed. In the execution of an extension join, the access paths of a user's query is automatically allocated by the aid of functional dependency among attributes. Besides the extension join, Sagiv also defines the physical representation phase of the relational database system as a representative instance.

There is a new trend to the universal relation that does not use the maximal object or extension join. Such an approach to the universal relation uses a semantic model, the Entity-Relationship Model (ERM). [Brad1985]. In Brady's approach to the universal relation, the outer join is exerted as the processing operator for a query.

Instead of the outer join, an efficient operation -ER-semijoin based on the ERM will be introduced.

4. ERM for Database Design and Query Processing

Since the Entity-Relationship Model (ERM) was introduced by Chen, it has become a widely accepted semantic model in database design [Hawr1985, Fry1982]. Besides being a good semantic model for the designing of database systems, An Entity-Relationship Diagram (ERD) can also be used as an access model which may help users navigate database systems [Zhan1983, Schu1986, Poon1978].

We will introduce the application of an ERM as the semantic structure of a relational database system. In order to use an ERM as the structure of a relational database system, more constraints are required on an ERM. An extended ERM represented by an ERG which may represent the semantic structure of a relational database system will be investigated. The research contains the following series of phases:

(1). The definition of semantic regions of ERG (Entity-Relationship Graph) and their constraints on the global consistency of the database: The updating of a database can be categorized as modification, insertion, and deletion. The application of an ERG to the maintenance of the integrity for the updating of a database will be discussed in this chapter.

(2). Explore efficient query tree processing based on ERG: acyclic subgraphs of an ERQG can be decomposed into a set of local regions which can be processed by an ER-semijoin. An ER-semijoin is an efficient operator that can be processed in the conceptual level with the semantics of ERM.

(3). Define an ERQG (Entity Relationship Query Graph) and tree structures of subgraphs of an ERQG on a relational database system: The semantic structure of a relational database system can be defined as an ERG (Entity Relationship

Graph). A query graph can be obtained by mapping the query onto this ERG such that the query graph contains a subgraph of the ERG and the relational operators of a query on the ERG. Besides helping the user navigate the database, an ERG can also be applied to the automatic allocation of access paths of a query on a universal relation interface and the conversion of cyclic subgraphs of an ERQG to tree structures.

The traditional relational database system always use functional dependency between the attributes to obtain the integrity and consistency on the processing of user's query (update and retrieval). In a RDKER (Relational Database System with Knowledge of ERG), the ERG is treated as part of the knowledge of a database system and which is in the conceptual level. With the knowledge of the semantic structure of a relational database system, the information integrity and data consistency of the relational database system on the respondence of user's query can be enforced through the constraint of the semantics of ERG.

Based on the query processing on a *local Region*, the operation of join operation on access paths of user's query can be efficiently modified by using ER-semijoin. ER-semijoin, an efficient operation technique for the processing of the access path of user's query, is based on the built-in knowledge of the ERG in the query system of databases. With this knowledge of the ERG in the query system, a user's query about the information in the databases can be decomposed into access paths which contain only Entity nodes and relationship nodes. Then, by exerting ER-semijoin operation on these entity nodes and relationship nodes, the processing of a query can avoid all redundant operation on Entity nodes of physical instances. Nevertheless, ER-semijoin operation will also reduce the unnecessary operation of attributes in the related relations.

Being different from traditional join operators, such as natural join, equijoin,

semijoin, and outer join, the ER-semijoin is an intelligent operator for obtaining the function of joining two relations. The operation of ER-semijoin, based on the relations in the conjunct local regions, which is much more efficient than the traditional joining operators that are implemented on relationship nodes of local regions one relation after another.

(4). Introduce an ERG approach to the universal relation interface: A universal relation Interface based on the semantic structure of an ERG is discussed.

A new universal relation approach that is based on the *ERG* and which uses ER-semijoin for the query processing will be proposed. In this approach to the universal relation, the *ERG* (Entity-Relationship Graph) is used as the structure of a relational database system. For a query, an *ERG* approach has several advantages. The first advantage of the *ERG* approach is that an query graph can be represented in an directed graph. The second advantage of the *ERG* approach is that the relational operators of the query can be mapped into the *ERG*. Then, such an *ERG* can be employed on the allocation of the access paths via a universal relation interface. That is, an *ERQG* on the universal relation interface which is based on *ERG* can be obtained. Finally, we may employ ER-semijoin for the query processing of the *ERQG*.

(5). Develop an integral relational database based on ERG: The ERG, ERQG, ER-semijoin, and universal relation interface can be applied either to One-Phase database systems or Two-Phase database systems. The architecture of the construction of these systems is discussed.

The organization of these phases is illustrated in the Fig. 1.

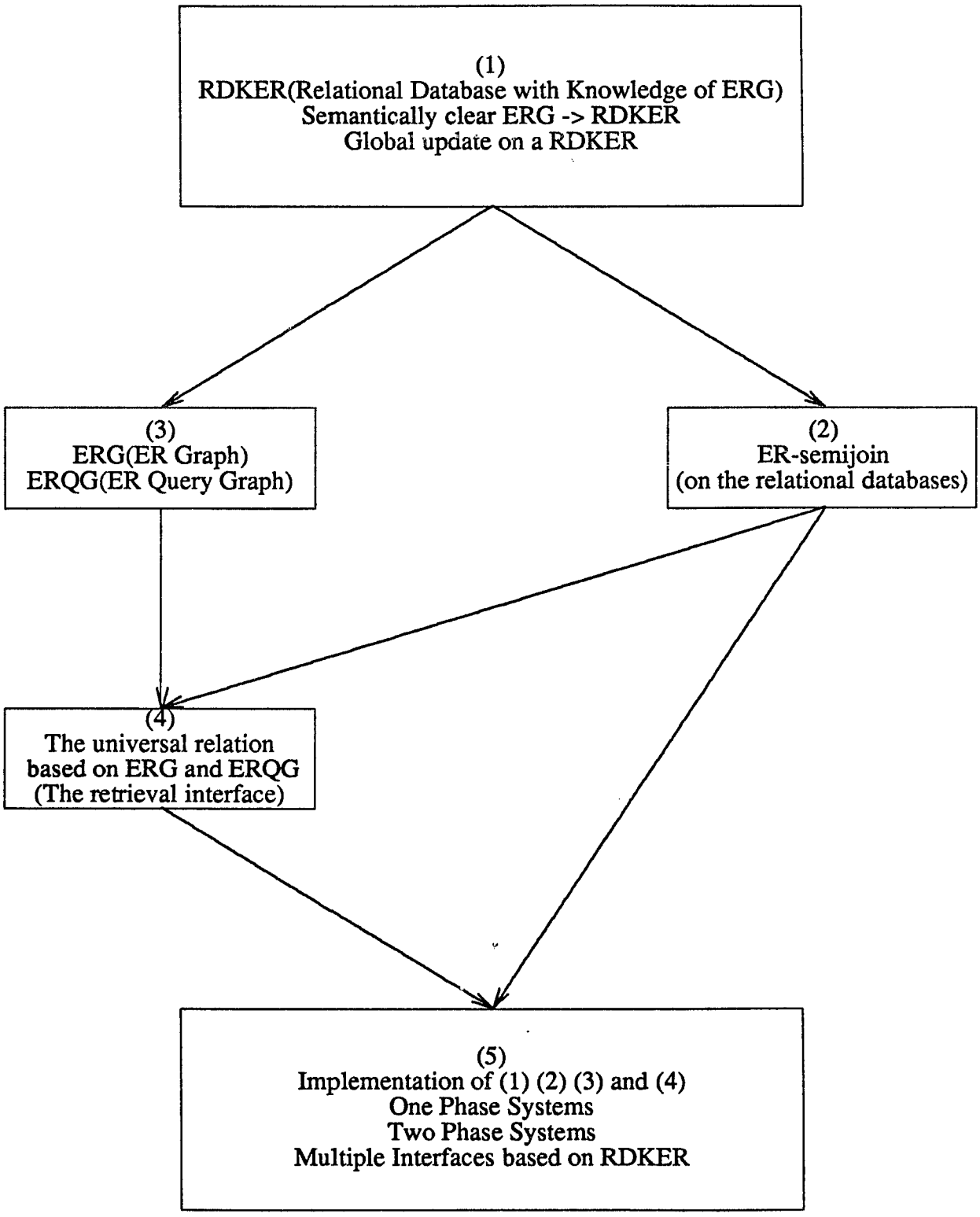


Fig. 1 The Organization of the phases of the research.

An *ERG* (Entity-Relationship Model) may be used as an intermediate level for the query processing of Multi-Model DBMS [Daya1983, Good1983, Morg1983]. A Heterogeneous Database interface between global schema and local schema using *ERG* as a conceptual intermediate is proposed by Katz and Goodman [Good1983]. Hai and Mattew also employed *ERG* as a intermediate interface among multimodel database systems [Daya1983].

The semantic structure of a well designed relational database system can be represented in an *ERG*. This representation has the advantage that a *NJ query* can be mapped onto the *ERG* to obtain a subgraph of the *ERG*. A natural join query is a query which simply computes the natural join of relations on a relational schema. An *ERQG* (ER-Query Graph) represents the natural join query based on the ERG. For a cyclic *ERQG*, the cyclic subgraphs of the restriction part of the *ERQG* can always be converted to tree structures. Finally, ER-semijoin can be employed to process the subgraphs represented in the tree structures.

The representation of an *ERG* as the semantic structure of a relational database has many advantages. The first advantage is that an query on an *ERG* can be represented by an *ERQG* which can be processed more efficiently than a traditional query graph on a relational database. That is, we may use ER-semijoin to process an *ERQG*, which is an efficient query processing technique that we may skip the processing on some entity nodes in the access paths of a query. Such an optimization query processing technique of ER-semijoin can be applied to all other intermediate interfaces built on top of the RDKER (Relation Database with Knowledge of *ERG*). Besides, we may build a universal relation interface by using *ERG* without exerting FD (functional dependency) for the allocation of access paths of a query on a universal relation interface. Nevertheless, for a query of updating (insertion, deletion, modification), we may use local constraints defined on the local regions of the *ERG* to maintain the integrity of the database [Chen1987].

The architectures of the construction of database interfaces based on the *ERG* and *ERQG* can be categorized into two-phase interface and one-phase interface. A two-phase interface of relational data model using *ERG* and *ERQG* contains more than two separate phases - the user-friendly interface and the underlying database system, i.e. to process a query, the transaction of the information between the interface and the database systems is necessary. A one-phase interface of relational data model using *ERG* and *ERQG* contains only one phase, i.e. the user-friendly interface is part of the database system.

CHAPTER 2
THE ENTITY-RELATIONSHIP GRAPH AND ITS CONSTRAINTS ON THE GLOBAL CONSISTENCY OF DATABASE

1. ENTITY-RELATIONSHIP GRAPH AND LOCAL REGIONS

The *ERM* is a semantic model which views the real world in terms of entity types and relationship types between entity types. An entity is an object which exists in the real world and can be distinctly identified. Entities can be classified into different entity types. A relationships describes the association between two entities or among several entities. Relationships can be classified into different relationship types. The properties of the entities and relationships can be expressed in terms of attribute-value pairs. An *ERD* is introduced as the diagrammatic technique to describe the *ERM*. In an *ERD*, an entity type is represented by the rectangular box and a relationship is represented by the diamond box. For example, in Fig. 1, PERSON and BOOK are entity types, BORROW and WRITE are relationship types. The labels on of the arcs *N* and *M* are cardinalities which indicate that the relationship types is many to many [Chen1977].

To implement an *ERD* with extended representation in the diagram, an extended *ERD* is introduced. In this extended *ERD*, an entity type or a relationship type is represented as a node; an entity node is a node of entity type and a relationship node is a node of relationship type; an arc is the connection between an entity node and a relationship node. In this application, a relationship node of an "exist dependency" or "id dependency" and the relationship node of "ISA" type is defined as the binary relationship.

In an ERD, if the existence of an entity node depends on the existence of another entity node then this entity node is called a weak entity node.

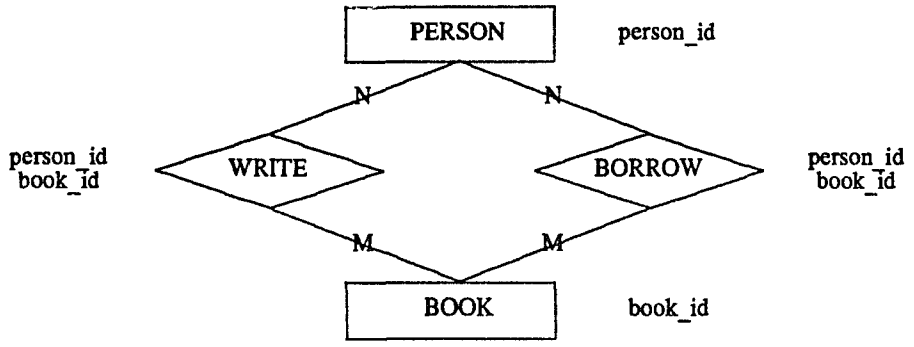


Fig. 1 The Entity-Relationship Diagram of the relational database LIBRARY.

DEFINITION 1 A *local region* is a subgraph of an *ERD*. A local region contains two nodes of entity types and a node of relationship type, two arcs such that each arc connects the node of the relationship type and one of the entity types, and two labels, denoted as $[E_i, R_j, E_k, ARC_{ij}, ARC_{jk}, C_{ij}, C_{jk}, G]$, where E_i and E_k represent entity nodes or weak entity nodes, R_j represents a relationship node; ARC_{ij} and ARC_{jk} are arcs with end nodes E_i, R_j and R_j, E_k respectively; C_{ij} and C_{jk} represent the cardinalities of the relationships and are the labels of the local region; G describes the type of the local region, which will be discussed separately.

In the physical representation level of a local region, the constraints among the values of the key attributes of nodes is called a local constraint. The local constraint of local regions is defined separately according to the different types of semantic representation of local regions.

We call a local region $[E_i, R_j, E_k, ARC_{ij}, ARC_{jk}, C_{ij}, C_{jk}, G]$ with order from E_i to R_j and from R_j to E_k a directed local region. In a directed local region $[E_i, R_j, E_k, ARC_{ij}, ARC_{jk}, C_{ij}, C_{jk}, G]$, the node E_i is called the tail and the node E_k is called the head.

A local region can be categorized according to its semantic representation and local constraint into four types - dependency local region (D), specialization local region (S), role-relationship local region (R), common local region (C).

DEFINITION 2 The *dependency local region* $[E_i, R_j, E_k, ARC_{ij}, ARC_{jk}, C_{ij}, C_{jk}, D]$ is a directed local region with tail E_i and head E_k, in which the node E_k has "existence dependency" or "id dependency" on the node E_i, and which satisfies the local constraint

$$\pi_{PK_k} r(E_k) = \pi_{PK_k} r(R_j)$$

$$\pi_{PK_i} r(R_j) \subseteq \pi_{PK_i} r(E_i)$$

where PK_i, PK_k are the key attributes of the nodes of the local region E_i, E_k respectively; $r(E_i)$, $r(R_j)$, $r(E_k)$ are physical instances which represent the E_i, R_j, and E_k in the physical level respectively.

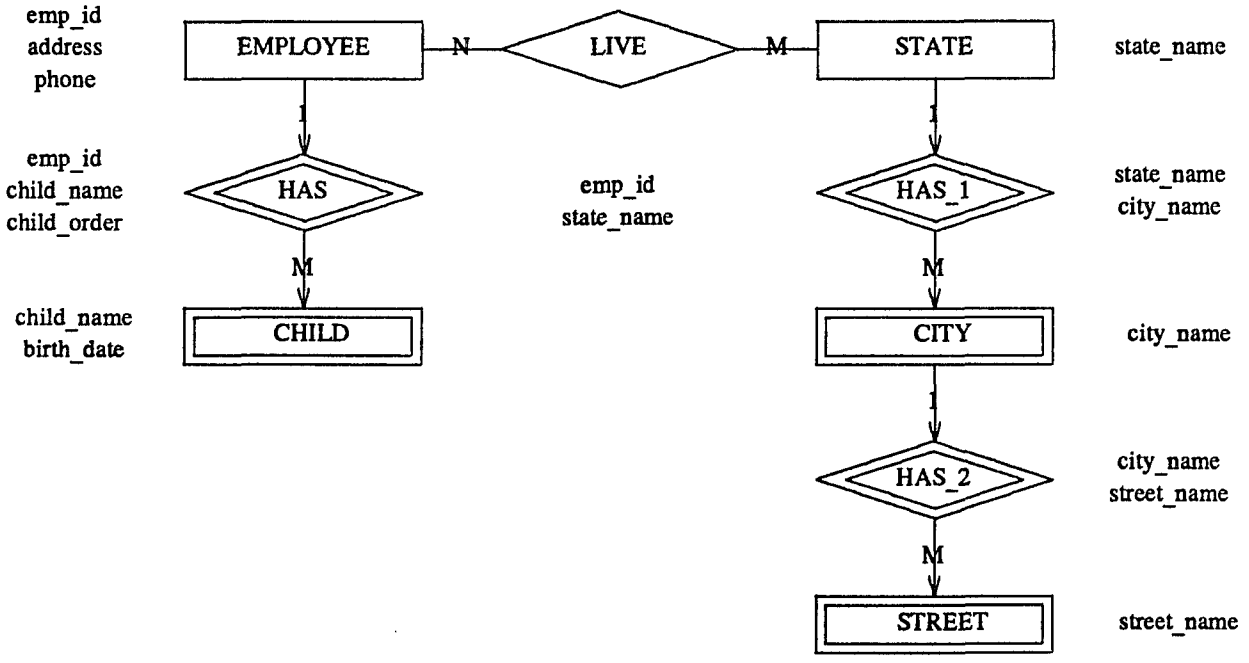


Fig. 2 The ERD of the relational database EMPLOYEE.

The head of a dependency local region represents a weak node of the tail. According to the local constraint of the dependency local region, the transitive property of weak nodes exists. In other words, the head of a dependency local region can be the weak node of the tail of another dependency local region iff the tail of the first local region is the weak node of the second local region. For two dependency local

regions $[E_i, R_j, E_k, ARC_{ij}, ARC_{jk}, C_{ij}, C_{jk}, D]$ and $[E_k, R_l, E_m, ARC_{kl}, ARC_{lm}, C_{kl}, C_{lm}, D]$, E_k is a weak node of the node E_i and E_m is a weak node of E_k, From the transitive property E_m is a weak node of E_i. In the connected local regions, we assume that an entity node of a dependency local region can not be a weak node of its weak nodes.

Example 1: The *ERD* as shown in fig. 2 has dependency local regions [*EMPLOYEE*, *HAS*, *CHILD*, ARC_{E_H}, ARC_{H_C}, 1, *M*, *D*], [*STATE*, *HAS_1*, *CITY*, ARC_{S_H1}, ARC_{H1_C}, 1, *M*, *D*], and [*CITY*, *HAS_2*, *STREET*, ARC_{C_H2}, ARC_{H2_STREET}, 1 *M*, *D*]. In these local regions, the node *CHILD* is a weak node of the node *EMPLOYEE*, the node *CITY* is a weak node of the node *STATE*, the node *STREET* is a weak node of the node *CITY* and a weak node of the node *STATE*.

In the local regions with common entity nodes, a node is called the content of another node if the collection of the values of key attributes on this node is the subset of the collection of that on another node.

DEFINITION 3 The *specialization local region* $[E_i, ISA, E_k, ARC_{ij}, ARC_{jk}, C_{ij}, C_{jk}, S]$ is a directed local region with tail E_i and head E_k, in which the node ISA describes the specialization relationship such that the node E_k is the content of the node E_i, and which satisfies the local constraint

$$\pi_{PK_i} r(E_k) \subseteq \pi_{PK_i} r(E_i)$$

The head of a specialization local region represents the object which is a content of the tail. The head is called the specialization node of the tail in a specialization local region. According to the local constraint of the specialization local region, the transitive property of content exists. In other words, the head of a specialization local region can be the specialization node of the tail of another specialization local region iff the tail of the first local region is the head of the second local region. We assume that in the local regions with common entity nodes, a node can not be the specializa-

tion node of its specialization node.

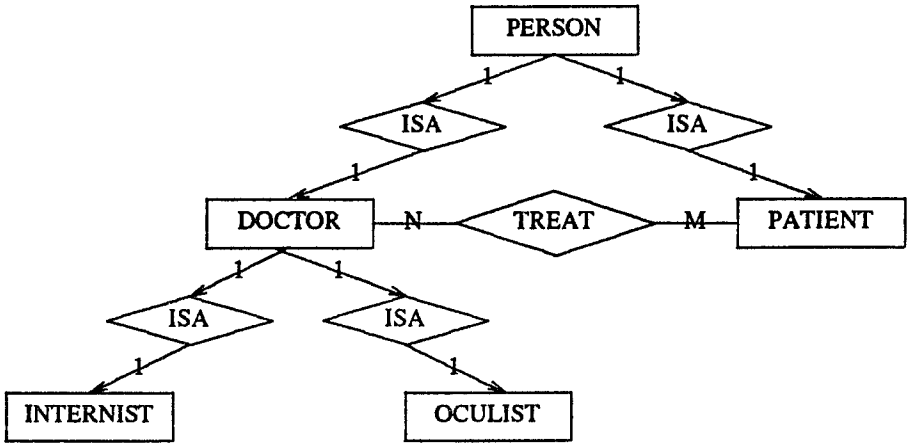


Fig. 3 The ERD of a relational database PERSON.

Example 2: The *ERD* as shown in fig. 3 has specialization local regions [*PERSON*, *ISA*, *DOCTOR*, ARC_{PERSON_ISA}, ARC_{ISA_DOCTOR}, 1, 1, *S*], [*PRESON*, *ISA*, *PATIENT*, ARC_{PERSON_ISA}, $ARC_{ISA\_PATIENT}$, 1, 1, *S*], [*DOCTOR*, *ISA*, *INTERNIST*, ARC_{DOCTOR_ISA}, ARC_{ISA_INT}, 1, 1, *S*], [*DOCTOR*, *ISA*, *OCULIST*, ARC_{DOCTOR_ISA}, ARC_{ISA_OCU}, 1, 1, *S*]. In these local regions, the node *DOCTOR* and the node *PATIENT* are the contents of the node *PERSON*,the node *INTERNIST* and the node *OCULIST* are the contents of the node *DOCTOR* and they are also the contents of the node *PERSON*.

The local constraint of a local region which is not a directed local region is

$$\pi_{PK_i} r(R_j) \subseteq \pi_{PK_i} r(E_i)$$

$$\pi_{PK_k} r(R_j) \subseteq \pi_{PK_k} r(E_k)$$

DEFINITION 4 The *role-relationship local region* $[E_i, R_j, E_k, ARC_{ij}, ARC_{jk}, C_{ij}, C_{jk}, R]$ is a local region in which the nodes E_i, E_k are heads of another specialization local regions and they are contents of the same node.

Example 3: The *ERD* as shown in fig. 3 has a role-relationship local region [*DOCTOR*, *TREAT*, *PATIENT*, ARC_{DOCTOR_TREAT}, $ARC_{TREAT\_PATIENT}$, *N*, *M*, *R*]. In this local regions, the

node *DOCTOR* and the node *PATIENT* are the contents of the node *PERSON*.

DEFINITION 5 The *common local region* $[E_i, R_j, E_k, ARC_{ij}, ARC_{jk}, C_{ij}, C_{jk}, R]$ is a local region which is neither a directed local region nor a role-relationship local region.

Example 4: The *ERD* as shown in fig. 2 has a common local region [*EMPLOY*, *LIVE*, *STATE*, ARC_{EMP_LIVE}, ARC_{LIVE_STATE}, N, M, C]. In this local region, the node *EMPLOYEE* and the node *STATE* are not the head of any directed local region.

The local region in an *ERD* can be either a directed subgraph or an undirected subgraph. The specialization local region and dependency local region are directed subgraphs. The local regions other than specialization local region and dependency local region are undirected subgraphs.

DEFINITION 6 An *ERG* (Entity-Relationship Graph) is an *ERD* in which if a pair of entity nodes are adjacent to a relationship node, then these entity nodes and their adjacent relationship node can be represented by one of the local region as defined above.

In an *ERG* nodes of a local region can be the tails of the other local regions. If there is a tail of a dependency local region which is not a head of any dependency local region, then we may construct a hierarchical region starting from this node.

DEFINITION 7 A *hierarchical region* H_R of an ERG is a structure which is the subgraph of an *ERG* such that

(1) there is a special node called the start node which is the tail of a dependency local region and which is not a head of any dependency local region,

(2) all of the dependency local regions that contain the start node are contained in the region,

(3) the dependency local regions whose tails are in the hierarchical region are contained in the region.

Example 5: The *ERG* as shown in fig. 2 has two hierarchical regions. The hierarchical region with root *EMPLOY* is {[*EMPLOYEE*, *HAS*, *CHILD*, ARC_{E_H}, ARC_{H_C}, 1, *M*, *D*]}; the hierarchical region with root *STATE* is {[*STATE*, *HAS*_1, *CITY*, ARC_{S_H1}, ARC_{H1_C}, 1, *M*, *D*], [*CITY*, *HAS*_2, *STREET*, ARC_{C_H2}, ARC_{H2_STREET}, 1 *M*, *D*]}.

If there is a tail of a specialization local region which is not a head of any specialization local region, then we may construct an inheritance region starting from this node.

A *directed inheritance structure* $T_{R'}$ of an ERG is the structure which is a subgraph of an *ERG* such that

(1) there is a special node called the start node which is the tail of a specialization local region and which is not a head of any specialization local region,

(2) all of the specialization local regions that contain the start node are contained in the structure,

(3) the specialization local regions whose tails are in the directed inheritance structure are contained in the directed inheritance structure.

DEFINITION 8 An *inheritance region* $I_{R'}$ of an *ERG* is a subgraph of an *ERG* and which contains (i) the directed inheritance structure $T_{R'}$ starting from the start node *R'* and (ii) the role-relationship local regions whose two end nodes of entity types are in the directed inheritance structure.

Example 6: The *ERG* as shown in fig. 3 represents an inheritance region with root *PERSON* as { [*PERSON*, *ISA*, *DOCTOR*, ARC_{PERSON_ISA}, ARC_{ISA_DOCTOR}, 1, 1, *S*], [*PRESON*, *ISA*, *PATIENT*, ARC_{PERSON_ISA}, $ARC_{ISA\_PATIENT}$, 1, 1, *S*], [*DOCTOR*, *ISA*, *INTERNIST*, ARC_{DOCTOR_ISA}, ARC_{ISA_INT}, 1, 1, *S*], [*DOCTOR*, *ISA*, *OCULIST*, ARC_{DOCTOR_ISA}, ARC_{ISA_OCU}, 1, 1, *S*], [*DOCTOR*, *TREAT*, *PATIENT*, ARC_{DOCTOR_TREAT}, $ARC_{TREAT\_PATIENT}$, *N*, *M*, *R*]}.

For two end nodes of an arc, one end node is called the *adjacent node* of the other. The adjacent nodes of an entity node are relationship nodes and the adjacent node of a relationship node are entity nodes. A *surrounding region* of a node which is not a head or a relationship node of a directed local region consists of (i) a central node which is represented by this node, (ii) the adjacent nodes of the central node, which are also not a head or a relationship node of a directed local region, (iii) the arcs connect the central node and its adjacent nodes. A surrounding region of an entity node is called an *entity surrounding region.* A surrounding region of a relationship node is called a *relationship surrounding region.* We assume that in a surrounding region of an *ERG*, the key attributes is uniquely defined. In other words, in a surrounding region, the same key attributes in the relationship nodes is defined by only one entity node.

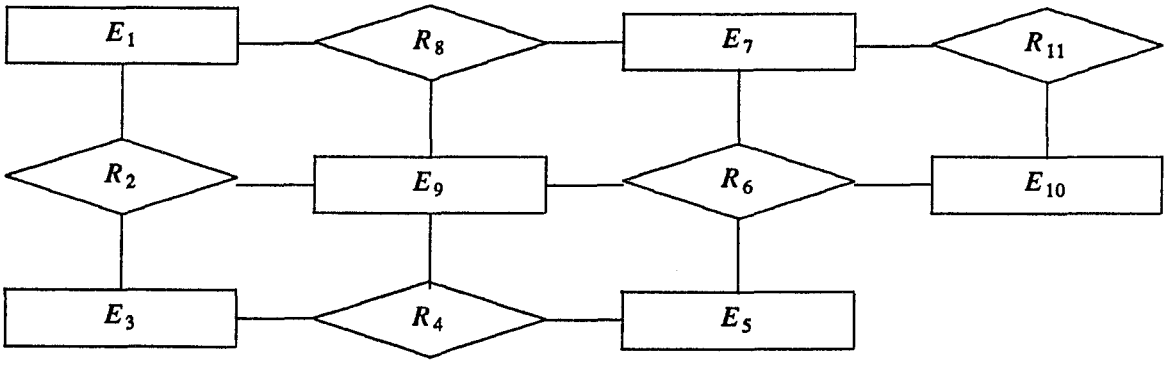


Fig. 4 An ERG of a RDKER.

Example 7: The semantic model of a relational database is represented by the ERG in fig. 4. The entity surrounding region of the entity node E_7 is $\{E_7, R_8, R_6, R_{11}\}$. The relationship surrounding region of the the relationship node R_6 is $\{R_6, E_7, E_9, E_5, E_{10}\}$.

A relationship surrounding region with a central node of n-ary relationship can be represented by n(n-1)/2 local regions.

Example 8: The relationship surrounding region [*WORK*, *PERSON*, *PROJECT*, *DEPT*] of an *ERG* with a 3-ary central node of relationship type *WORK* can be represented by 3(3-1)/2 local regions as [*PERSON*, *WORK*, *PROJECT*, *N*, *M*, *C*] [*DEPT*, *WORK*, *PERSON*, 1, *M*, *C*] [*PROJECT*, *WORK*, *DEPT*, *N*, *M*, *C*].

2. UPDATING PROPAGATION STRUCTURE FOR DATA CONSISTENCY IN THE ERG

A query of updating is categorized as insertion, deletion, and modification. To maintain the integrity of a database for updating a node in an *ERG*, some of the adjacent nodes of this node have to be updated. Thus the processing of updating has to be propagated from a node to its adjacent nodes which have to be updated. Such a propagation of updating from a node to its adjacent node is called *updating propagation.* Similarly, the updating of each adjacent node of a node may require that some nodes adjacent to each adjacent node to be updated. The updating of a node can be iteratively propagated until the integrity of the database is maintained.

In a query of updating, the user specifies the attribute values of a node in an *ERG* to be updated. This specified node in a query of updating is called the start node of updating with respect to the query. The structure which represents the propagation sequence for the updating of a start node is called the *updating propagation structure* of a start node. In the construction of a updating propagation structure, we assume that the head of a specialization local region can not be the head of a dependency local region.

For a query of updating with a start node, a top down *updating propagation structure* of the start node which is in a hierarchical region is a structure such that:

(i) The nodes of the dependency local regions which contains the start node as their tails or as their relationship nodes can be contained in the structure. In these dependency local regions, if the start node is a relationship node, then the heads

of these local regions are contained in the structure as the children of the start node; else all of the relationship nodes of these local regions are contained in the structure as the children of the start node and the head of these local regions are contained in the structure as the children of their relationship nodes. If a head has more than one adjacent relationshiop node in the structure, then duplicates the head such that each head in the structure has only one parent node.

(ii) The nodes of the dependency local regions which is never used in the construction of the structure and whose tails are in the structure can be contained in the structure. The relationship nodes of the selected dependency local regions are contained in the structure as the children of their tails, then the heads of these local regions are contained in the structure as the children of their adjacent relationship nodes in the structure. If a head has more than one adjacent relationship node in the structure, then duplicates the head such that each head in the structure has only one parent node.

(iii) The relationship nodes of the common local regions which have their nodes in the structure are contained in the structure as the children of their adjacent nodes in the structure.

For a query of updating with a start node, a top down *updating propagation structure* of the start node which is in a inheritance region is a structure such that:

(i) The nodes of the specialization local regions which contains the start node as their tails or as their relationship nodes can be contained in the structure. In these specialization local regions that contain the start node, if the start node is a relationship type, then all of the heads of these local regions are contained in the structure as the children of the start node; else all of the relationship nodes of these local regions are contained in the structure as the children of the start node and the heads of these local regions are contained in the structure as the children

of their relationship nodes. If a head has more than one adjacent relationshiop node in the structure, then duplicates the head such that each head in the structure has only one parent node.

(ii) The nodes of the specialization local regions which is never used in the construction of the structure and whose tails are in the structure can be contained in the structure. The relationship nodes of the selected specialization local regions are contained in the structure as the children of their tails, then the heads of these local regions are contained in the structure as the children of their adjacent relationship nodes in the structure. If a head has more than one adjacent relationship node in the structure, then duplicates the head such that each head in the structure has only one parent node.

(iii) The relationship nodes of the common local regions which have their nodes in the structure are contained in the structure as the children of their adjacent nodes in the structure.

(iv) the relationship nodes of the role-relationship local regions whose two entity nodes are in the structure are duplicated and contained in the structure as the children of its adjacent entity nodes.

For a query of updating with a start node, a bottom up *updating propagation structure* of the start node which in a hierarchical regions is a structure such that:

(i) The nodes of the dependency local regions which contain the start node as their heads or as their relationship nodes can be contained in the structure. In these dependency local regions, if the start node is a relationship node, then all of the tails of these local regions are contained in the structure as the children of the start node; else all of the relationship nodes of these local regions are contained in the structure as the children of the start node and the tails of these local regions are contained in the structure as the children of their relationship nodes.

If a tail has more than one adjacent relationship node in the structure, then duplicates the tail such that each tail in the structure has only one parent node.

(ii) The nodes of the dependency local regions which is never used in the construction of the structure and whose heads are in the structure are contained in the structure. The relationship nodes of the selected dependency local regions are contained in the structure as the children of their heads, then the tails of these local regions are contained in the structure as the children of their adjacent relationship node in the structure. If a tail has more than one adjacent relationship node in the structure, then duplicates the tail such that each tail in the structure has only one parent node.

For a query of updating with a start node, a bottom up *updating propagation structure* of the start node which is in an inheritance region is a structure such that:

(i) case I: the start node is a relationship node of a role-relationship local region

In a role-relationship local region, the key attributes are specified with the role. From the specified role of the start node, the entity node in the local region which has the specified role can be identified. The entity node in the role-relationship local region which has the same specified role with that of the start node is contained in the structure as the child of the start node;

case II: the start node is in a specialization local region

The nodes of the specialization local regions which contains the start node as their heads or as their relationship nodes may be contained in the structure. In these specialization local regions, if the start node is a relationship type, then all of the tail of these local regions are contained in the structure as the children of the start node; else all of the relationship nodes of these local regions are contained in the structure as the children of the start node and the tails of these local regions are contained in the structure as the children of their relationship nodes.

If a tail has more than one adjacent relationship node in the structure, then duplicates the tail such that each tail in the structure has only one parent node.

(ii) The nodes of the specialization local regions which is never used in the construction of the structure and whose tails are in the structure are contained in the structure. The relationship nodes of the selected specialization local regions are contained in the structure as the children of their heads, then the tails of these local regions are contained in the structure as the children of their adjacent relationship nodes in the structure. If a tail has more than one adjacent relationship node in the structure, then duplicates the tail such that each head in the structure has only one parent node.

The possibility for a top down updating propagation structure of a start node in an inheritance or hierarchical region to be a cycle is that there are more than one local regions with their nodes in the structure having the same head. Similarly, the possibility for a bottom up updating propagation structure of a start node in an inheritance or hierarchical region to be a cycle is that there are more than one local regions with their nodes in the structure having the same tail. According to rules of the creation of a updating propagation structure, if there are more than two local regions which have the same head for the top down propagation structure or the same tail for the bottom up propagation structure then the common node is duplicated and contained in the structure. Since there is not any node in the structure has more than two parent, these updating propagation structures are tree structures.

In an inheritance region of an *ERG*, a node which is not the start node of the region can be the start node of a hierarchical region. Likewise, a node which is not the start node of a hierarchical region can be the start node of an inheritance region. In this case, the updating propagation structure of a start node in an inheritance region may propagate to hierarchical regions; and the updating propagation structure of a

start node in a hierarchial region may propagate to inheritance regions. For the convenience, we may construct the updating propagation structure according to the updating propagating rules of a start node in an inheritance region or in a hierachical region and extend the nodes in the structure as the start nodes of other hierarchical regions or inheritance regions respectively if it is available. Thus the updating propagation structure of a start node in an inheritance region or a hierachical region can be constructed iteratively form the updating propagating rules of a start node discussed above.

A *surrounding updating propagation structure* of a start node in a surrounding region is a tree structure such that

(i) The nodes in the surrounding region of the start node in the *ERG* are contained in the structure. These nodes adjacent to the start node in the *ERG* are the children of the start node in the tree.

(ii) Each node in the tree may have children. The nodes in the surrounding region of a node in the tree, which contain the key attributes of the central node and that are not the parent of its central node in the tree, are contained in the structure as the children of their central node in the tree. If a child has more than one parent in the tree, then it is duplicated such that each child in the tree has only one parent.

The surrounding updating propagation structure of a relationship node in a surrounding region is called the *relationship surrounding updating propagation structure*; the updating propagation structure of an entity node in a surrounding region is called the *entity surrounding updating propagation structure*.

These updating propagation structures defined above are tree structures. The height of a updating propagation structure for updating is the longest path from the root to a leaf. In a updating propagation structure, the *degree of updating propagation structure* is defined as the height of the tree.

THEOREM 1 The degree of an *entity surrounding updating propagation structure* is 1.

Proof: In a surrounding region, the key attributes of an entity node is uniquely defined. In other words, the key attributes of the relationship nodes in this surrounding region can not represent the key attributes of their adjacent entity types other than the root. The updating propagation of the key attributes starts from the root of entity node and ends at its adjacent relationship nodes. Thus the height of the such a updating propagation tree is 1.

THEOREM 2 The degree of a *relationship surrounding updating propagation structure* is 2.

Proof: In a surrounding region, the key attributes of a relationship node should be defined by its adjacent entity types. It is obvious that the height of updating propagation from the relationship node to its adjacent entity nodes is 1. The theorem 1 prove that the updating propagation of an entity node is 1. The updating propagation of the relationship node is the height of the updating propagation from the root to its adjacent entity nodes and the height of the updating propagation of its adjacent entity nodes. Thus the degree of updating propagation of such a relationship is 2.

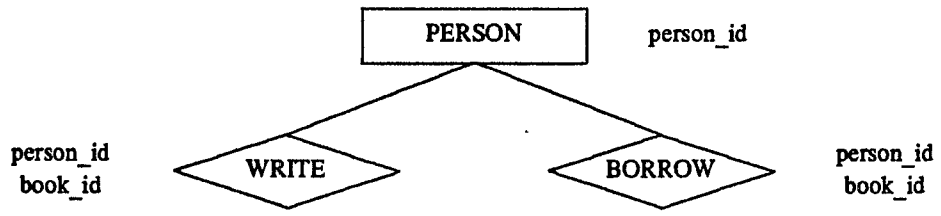


Fig. 5.a Updating propagation structure of entity node PERSON.

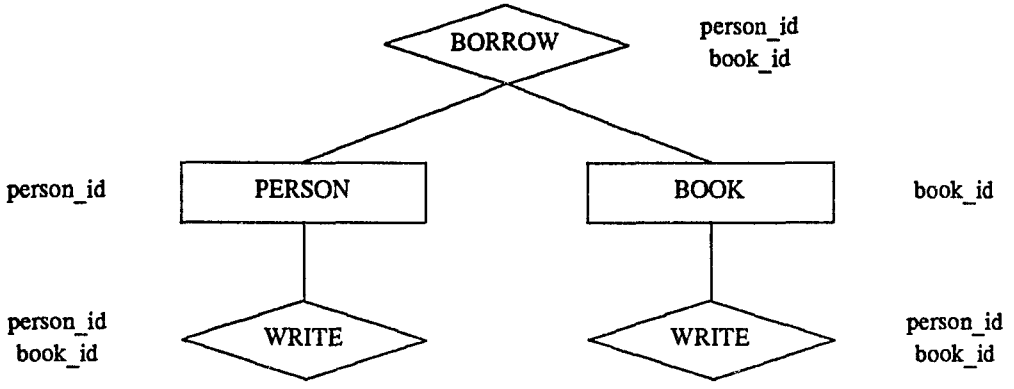


Fig. 5.b Updating propagation structure of BORROW.

Example 9: An *ERG* of database *LIBRARY* is shown in fig.1. The updating propagation tree of the entity node PERSON is shown in fig. 5.a which represents the updating propagation of the primary key person_id of the node PERSON. The updating propagation tree of the relationship node BORROW consists of two subtrees, one subtree has the primary key person_id and the other subtree has the primary key book_id is shown in fig. 5.b . The degree of updating propagation of entity node PERSON is the height of the tree of fig. 5.a and the degree of updating propagation of relationship BORROW is the height of the tree of fig. 5.b. That is, the degree of updating propagation of the entity node PERSON is 1 and the degree of updating propagation of the relationship node WRITE is 2.

3. DATA CONSISTENCY OF A RELATIONAL DATABASE BASED ON THE ERG

The advantage of using semantic level to control the integrity of a relational database during the updating (modification, insertion, deletion) is that the constraints among the key values of the physical instances can be defined in the semantic level based on the local regions. A physical instance is a table in the physical level which is represented by an entity node or relationship node in the semantic level. Thus with

the knowledge of local regions in an *ERG*, the integrity control of a relational database based on the *ERG* can be achieved.

To update (modify, insert, delete) a physical instance of an entity node or a relationship node in an *ERG*, null value constraints in the physical level should be studied. A null value is a special value which is used to represent an unknown or inapplicable value. In most database systems, some of the information may contain null value, for example, the attributes of an entity node PERSON is PERSON[person_id, phone, address, ...]. If a new person does not have a phone number yet, the null value of the phone number should be allowed. But if the person_id is null, the system will have a problem in the allocation of the tuple. Thus, the proper allowance of null values will make the system more flexible and user-friendly. We assume that the projection on a column of a physical instance does not contain null value.

The *full null constraint* on a physical instance of a RDKER is that there is no null value contained in the primary keys. The *partial null constraint* on a physical instance of a RDKER is that for each tuple at least one of the values of the key attributes is not null.

Since an entity node represents the distinguishable entities of a database and its primary keys are uniquely defined on each entity as an identifies, the null value in the key attributes of an entity is a contradiction that an entity can be identified by primary keys. Thus the null value is prohibited in the physical instance of an entity node in an *ERG*

The null constraint on the relationship nodes depends on the design strategy of the database system [Date1983]. For example, the local region [PERSON, BUY, BOOK] contains two entity nodes PERSON, BOOK and a relationship node BUY. In the physical level, the primary keys of PERSON and BOOK should not be null. However the null value in the key attributes of BUY may be partially allowed. The policy

of this database design is that the department may allow the employees to buy books without receipt when the price of the book is under twenty dollars. In this case, the physical instance of the relationship may have some people who bought a book without containing the information about the book. The null value of "book_id" on the physical instance of the relationship node is allowed in this example.

For an n-ary relationship other than a binary relationship, the partial null constraint is always necessary. For example, let *RS* be a relationship surrounding region and $RS = [WORK, PERSON, PROJECT, DEPARTMENT]$, where *WORK* is a relationship node surrounding by the entity nodes *PERSON*, *PROJECT*, and *DEPARTMENT*. Then, the person who works in a department and who does not work in a project yet should have a null value on the primary keys of the physical instance *PROJECT*.

A query on a database is categorized as retrieval and updating. The updating can further be categorized into modification, insertion, and deletion. For a query of updating, the correctness and accuracy in the database can be controlled in the semantic level. The integrity control of a relational database based on the local regions of the *ERG* can be categorized according to the information of a node to be updated as

(1) the attributes to be updated are key attributes of a node in an inheritance region,

(2) the attributes to be updated are key attributes of a node in a hierarchical region,

(3) the attributes to be updated are key attributes of a node which is neither in a hierarchical region nor in an inheritance region,

(4) the attributes to be updated are nonkey attributes.

The updating of the nonkey attribute values of a node will not affect the integrity of a database. Thus the updating of nonkey attributes will not be discussed in this chapter.

The local constraints of the local regions in an *ERG* can be applied directly on the integrity checking of a relational database based on the *ERG*. The integrity checking of

a database by applying the local constraints of all local regions in an *ERG* is called the *static integrity checking* based on the local regions. The integrity checking of a database which uses local constraints in updating propagation structures of a node to be updated is called the *dynamic integrity checking* based on the local regions.

The integrity checking in a local region contains the following contents:

(i) Each physical instance of a local region has to be at least in the third normal form.

(ii) The key values of the physical instance of the relationship node should satisfy the cardinality constraints defined in the local region. That is, in the physical instance of a relationship node, the key values of the entity nodes have to be checked whether they violate the cardinality specified in the local region.

(iii) The physical instances in a local region should satisfy the local constraint on the local region.

The normal form of (i) have been widely discussed [Date1983, Hawr1984], and cardinality constraints of (ii) is discussed by Lenzerini [Sant1983. We will discuss the using of the updating propagation tree and the local constraints of the local regions of the nodes in the tree for the updating of the database in the following sections. As discussed by the DATE that the updating of a relational database always depend on the policy of the database design [Date1983], the detail implementation of these updating propagation trees is depend on the policy of the database design. The design policy of a database during update is not discussed here.

Example 10: The *ERG* as shown in fig. 2 has local regions [*EMPLOYEE*, *HAS*, *CHILD*, ARC_{E_H}, ARC_{H_C}, 1, *M*, *D*], [*STATE*, *HAS_*1, *CITY*, ARC_{S_H1}, ARC_{H1_C}, 1, *M*, *D*], [*CITY*, *HAS_*2, *STREET*, ARC_{C_H2}, ARC_{H2_STREET}, 1 *M*, *D*]}, and [*EMPLOYEE*, *LIVE*, *STATE*, ARC_{E_L}, ARC_{L_S}, N, *M*, *C*]. The static integrity checking of this database can be processed as (i) checking whether the physical instances of *EMPLOYEE*, *HAS*, *CHILD*, *STATE*, *LIVE*,

HAS_1, *CITY*, *HAS_2*, and *STREET* are in the third normal form or not, (ii) checking the cardinality restriction on the physical instances of *HAS*, *HAS_1*, and *HAS_2* for 1:m relationship; *LIVE* for n:m relationship, and (iii) checking the local constraints of the physical instances of these local regions respectively.

The dynamic integrity checking in an *ERG* is more complex than the static integrity checking based on the local regions. In the following sections, we will discuss the dynamic integrity checking based on local regions of an *ERG* for the updating of a database. In a updating propagation tree, if an entity node is duplicated, then the counter for the number of duplication have to be set up on this node. In the processing of updating by using the updating propagation structure, if an entity node is duplicated, then the updating of the subtree with this node as start node can be propagated only if all the duplicated nodes of this entity node in the tree is updated.

3.1. Deletion

The updating propagation for the deletion of key values of nodes in inheritance region, hierarchical region, and surrounding region can be represented by the top down inheritance updating propagation structure, top down hierarchical updating propagation structure, and surrounding updating propagation respectively. A updating propagation tree specifies the sequence of the deletion of the physical instances of the nodes in the tree when key values of the start node are deleted. The deletion start from the start node of the structure. If a node in the structure contains the information to be deleted, then the processing of deletion should be propagated from this node to all the children of this node; else the processing of deletion should not be propagated to any child of this node.

Deletion of Key Values in an Inheritance Region

An inheritance region may contain two types of local regions - role-relationship local regions and specialization local regions. A role relationship local region is an undirected local region such that the deletion of the tuples of the relationship node of this local region will not affect the integrity of the other nodes in the *ERG*.

To delete the key attributes of a node in a specialization local region of an inheritance region, a top down inheritance updating propagation structure which has this node as start node can be created. The top down inheritance updating propagation structure describes the processing order which specifies the sequence of the nodes in the inheritance region and the nodes connected to the inheritance region to be deleted.

Example 11: The database *PERSON* as shown in fig. 3 represents an inheritance region with start node *PERSON*. To deleted a tuple from the physical instance *PERSON*, some tuples of the physical instance in the top down inheritance transition with start node *PERSON* have to be deleted. This updating propagation tree contains entity nodes *PERSON*, *DOCTOR*, *PATIENT*, *INTERNIST*, *OCULIST*, and *TREAT*, where the node *TREAT* is duplicated and connected to the nodes *DOCTOR* and *PATIENT*. In the physical instances of the entity nodes *DOCTOR*, *PATIENT*, *INTERNIST*, *OCULIST*, the tuples whose key values contain the key values of the tuple deleted in *PERSON* have to be deleted. To delete the physical instance *TREAT*, the deletion has to be processed on both the key attributes with role *DOCTOR* and the key attributes with role *PATIENT* That is, the tuples in the physical instance of *TREAT* have to be deleted if (i) the key values of the role *DOCTOR* which contains the key values of the deleted tuple in *PERSON*, or (ii) the key values of the role *DOCTOR* which contains the key values of the deleted tuple in *PERSON*.

Deletion of Key Values in a Hierarchical Region

To delete the key attributes of a node in a dependency local region of a hierarchi-

cal region, a top down hierarchical updating propagation structure which has this node as start node can be created. The top down hierarchical updating propagation structure describes the processing order which specifies the sequence of the nodes in the hierarchical region and the node connected to the hierarchical region to be deleted. The processing of the deletion should obey the local constraint of the dependency local region. If the start node of a top down hierarchical updating propagation structure is a entity node and that is not the start node of the hierarchical region in which this node is contained in, then the local constraints between the relationship nodes of the dependency local regions which have this node as head and this node have to be checked.

Example 12: In the database as shown in fig. 2, a top down hierarchical updating propagation structure with root *STATE* can be represented by the list with order of nodes as *STATE*, *HAS_1*, *CITY*, *HAS_2*, *STREET*. The sequence for the processing of the checking of local constraints in this structure are :

(1) $\pi_{state\_name} r(R_{HAS\_1}) \subseteq \pi_{state\_name} r(E_{STATE})$;

(2) $\pi_{city\_name} r(E_{CITY}) = \pi_{city\_name} r(R_{HAS\_1})$;

(3) $\pi_{city\_name} r(R_{HAS\_2}) \subseteq \pi_{city\_name} r(E_{CITY})$;

(4) $\pi_{street\_name} r(E_{STREET}) = \pi_{street\_name} r(R_{HAS\_2})$.

If the tuples of *STATE* are deleted, then the tuples in the *HAS_1* which violate the local constraint (1) are deleted; then the tuples in the *CITY* which violate the constraint (2) are deleted; then the tuples in the *HAS_2* which violate the constraint (3) are deleted; then the tuples in the *STREET* which violate the constraint (4) are deleted.

Deletion of Key Values in a Surrounding Region

We assume that the deletion of the key values of the relationship node in an undirected local region does not affect the integrity of any other node in the *ERG*. To

delete the key values of an entity node in a surrounding local region, an *entity surrounding updating propagation structure* which has this node as root can be created.

If the root of an entity surrounding updating propagation is also the roots of inheritance regions or hierarchical regions, then the top down inheritance updating propagation structures or top down inheritance updating propagation structures are created as the subtree of the root. Such a updating propagation structure describes the processing order which specifies the sequence of the nodes in the *ERG* to be deleted.

Example 13: In the database *LIBRARY* as shown in fig. 1, the updating propagation structure of *PSRSON* is shown in fig. 5.a. To delete a tuple in the physical instance *PERSON*, the tuples of the relations *WRITE* and *BORROW*, which contain the key values of the tuple deleted in the relation *PERSON* have to be deleted.

3.2. Insertion

The updating propagation for the insertion of key attributes of nodes in inheritance region and hierarchical region can be represented by the bottom up inheritance updating propagation structure and bottom up hierarchical updating propagation structure. The bottom up inheritance updating propagation structure describes the processing order which specifies the sequence of the nodes to be inserted.

Insertion of Key Values in an Inheritance Region

To insert the key attributes of a node of an inheritance region, a bottom up inheritance updating propagation structure which has this node as root can be created.

Example 14: The database *PERSON* as shown in fig. 3 represents an inheritance region. To insert a tuple into the physical instance of the node *OCULIST*, a bottom up inheritance updating propagation structure with root *OCULIST* can be constructed. In this

example, the processing sequence can be represented by a list of nodes as *OCULIST*, *DOCTOR*, *PERSON*. To apply the local constraint of a specialization local region into this structure, the sequence of local constraints checking in this structure are

$$\pi_{person\_id}\, r(E_{OCULIST}) \subseteq \pi_{person\_id}\, r(E_{DOCTOR});$$

$$\pi_{person\_id}\, r(E_{DOCTOR}) \subseteq \pi_{person\_id}\, r(E_{PERSON}).$$

The first constraint specifies the key values of the new tuple of the *OCULIST* should be defined in the *DOCTOR*, and the second constraint specifies that the same key values should also be defined in the *PERSON*.

Insertion of Key Values in a Hierarchical Region

To insert the key attributes of a node in a dependency local region of an hierarchical region, a *bottom up updating propagation structure of hierarchical regions* which has this node as root can be created.

Example 15: The database *EMPLOYEE* as shown in fig. 2 contains two hierarchical regions. To insert a tuple into the physical instance of the node *CITY*, a bottom up hierarchical updating propagation structure with root *CITY* can be constructed. In this example, the processing sequence can be represented by a list of nodes as *CITY*, *HAS*, *STATE*. By applying the local constraint of dependency local region into this sequence, the constraints in this structure are

$$\pi_{city\_name}\, r(E_{CITY}) = \pi_{city\_name}\, r(R_{HAS\_1});$$

$$\pi_{state\_name}\, r(R_{HAS\_1}) \subseteq \pi_{state\_name}\, r(E_{STATE}).$$

The first constraint specifies the key values of the new tuple of the *CITY* should be defined in the *HAS*_1, and the second constraint specifies that the key values of the state_name in the tuples containing the inserted key should be defined in the *STATE*.

Insertion of Key Values in a Surrounding Region

The insertion of tuples of an entity node which is not in a directed local region will not affect the integrity of the other nodes in the *ERG*. To insert the key attributes to a relationship node, the entity nodes which is in the relationship surrounding region of the relationship node and which contain any attribute to be inserted have to be checked. If there is any entity node which have a new key attributes to be inserted and which is the root of an inheritance region or a hierarchical region, then bottom up updating propagation tree of an inheritance region and hierarchical region with this node as root have to be created as the subtree of this node.

3.3. Modification

A trivial case of the modification is that the user want to modify the key values of a relationship node without the modification of the values of the entity nodes connected to the relationship node. In this case, if the new values are defined in the entity node which contain the key attributes and which is adjacent to the relationship node, then the modification can be processed; else the modification does not allowed.

Modification of Key Values in an Inheritance Region

The modification of key values of a start node in an inheritance region can be categorized as

(i) modification of all nodes in the inheritance region and related nodes which have to be modified for the modification of these nodes,

(ii) modification of nodes in the top down updating propagating structure of the start node.

The policy of modification can be specified by the database designer. The user may have to select the option of the modification before the processing of the modification.

The modification of a node in an inheritance region may start from any node in the region, and a top down inheritance updating propagation structure with this node as root can be created. If a node to be modified is not a start node of an inheritance region, then before the processing of the modification in the updating propagation tree the local constraint between this node and the tails of the specialization local regions which have this node as head have to be checked. If the new tuples of the root node of the top down hierarchical transition tree does not violate the integrity of the database, then the modification can be propagated in the updating propagation tree start from the root. For any node in the tree contains the information to be modified, then the processing of modification should be propagated from this node to all the children of this node; else the processing of modification should not be propagated to any child of this node.

Example 16: The physical instances of *PERSON*, *DOCTOR*, *INTERNIST*, and *OCULIST* contain key values [$<p_1>$, $<p_2>$, $<p_3>$, $<p_4>$, $<p_5>$], [$<p_1>$, $<p_3>$, $<p_4>$, $<p_5>$], [$<p_1>$,$<p_5>$], [$<p_1>$, $<p_3>$, $<p_5>$] respectively. To modify the key value p_1 of *DOCTOR* to p_6 by option (i), these nodes in the inheritance region contain the value of p_1 can be modified to p_6. To modify the key value p_1 of *DOCTOR* to p_2 by option (ii), the nodes *DOCTOR*, *INTERNIST* and *OCULIST* which contain the value of p_1 have be modified to p_2.

Modification of Key Values in a Hierarchical Region

The modification of the key values of an entity node of a dependency local region can be extended to the relationship nodes of the directed local region which have this entity node as tail or the undirected local regions which contain this entity node.

The modification of the values of key attributes of a relationship node in a dependency local region can be processed as

(i) if the key attributes to be modified are the key attributes of the tail, then the modification is a trivial case;

(ii) if the key attributes to be modified are the key attributes of the head, the new key values can be either contained in the values of the key attributes of the head or not. For the former case, the processing of the modification does not have to be extended to the subtree with this relationship node as root. For the later case, the modification have to be extended to the modification of the head on the same key attributes.

Example 17: The nodes *EMPLOYEE*, *HAS*, *CHILD* represent nodes of a hierarchical region in fig.2. The physical instances of these nodes *EMPLOYEE*, *HAS*, *CHILD* contain key values [$<e_1>$, $<e_2>$,$<e_3>$, $<e_4>$], [$<e_1, c_1>$, $<e_1, c_2>$, $<e_2, c_3>$], and [$c_1>$, $<c_2>$, $<c_3>$] respectively. To modify key values of $<e_1, c_1>$, to $<e_4, c_1>$ is a trivial case. To modify key values of $<e_1, c_1>$, to $<e_1, c_4>$, the key value c_1 of *CHILD* has to be modified to c_4.

Modification of Key Values in a Surrounding Region

The modification of the key values of a relationship node in a surrounding region does not affect its adjacent entity nodes. For the modification of the key values of an entity node in a surrounding region, the relationship nodes in the surrounding region of this entity node have to be modified. Since the degree of surrounding updating propagation of an entity node is 1, the modification of an entity node will only propagate to the adjacent nodes of this entity node. If this entity node is a start node of an inheritance region or a start node of a hierarchical region, the modification should be

extended to the inheritance region or the hierarchical region as discussed in the previous sections.

CHAPTER 3
ER-SEMIJOIN OPERATION ON LOCAL REGIONS OF A QUERY ON AN ENTITY-RELATIONSHIP GRAPH

1. ER-SEMIJOIN

Goodman propose a *NJ query* (Natural Join query) which simply computes the natural join of relations on a relational schema [Chu1981, Shmu1982]. Berstein's and Goodman's works are based on the logic structure of a relational database defined by a relational schema. In this chapter, we study a NJ query based on the logical structure of a relational database represented by the acyclic subgraph of an *ERG*. Such a NJ query whose navigation paths on the conceptual level can be represented by an acyclic subgarph of *ERG* and which can be computed by the natural join operation. For the convenience, we use query to represent natural join query on the acyclic subgraph of an *ERG*.

A physical instance of a database is defined as the representation of an entity node or a relationship node in the physical level. The collection of physical instances of a database (D) is defined as a representative instance, denoted as Rep(D) [Sagi1983]. The access paths of a query on a RDKER may be decomposed into a sequence of entity nodes and relationship nodes such that for each entity node or relationship node there is one and only one physical instance in corresponding to it. Then, for a query on an *ERG*, we can exert natural join operator on these physical instances in sequence to compute the query.

We observe that those queries which have long access paths can be implemented more efficiently by skipping the unnecessary joining operation on the entity nodes in a local region. Nevertheless, the joining operation on physical instances in some local regions of a query can be reduced to the primary keys. The following example will

illustrate the operation of ER-semijoin on a local region of a query.

DEFINITION 1 : In a local region of a query, the **object entity node** is the entity node in the local region which contains the attributes whose domain is restricted. In the physical instance of an object entity node, the tuples whose values of restricted attributes are in the restricted domain are called object tuples. The collection of the key values of the object tuples of an object entity node is called **object list.**

DEFINITION 2 : In a local region of a query, if one of the entity nodes is an object entity node, then the other entity node is a **target entity node** with respect to the object entity node. In the physical instance of the relationship node, the tuples whose attribute values of the object entity node is in the object list is called the target tuples. A **target list** is the collection of the key values of the target entity node of the target list.

DEFINITION 3 : Let L_i be a local region with two entity nodes E_{i-1} and E_{i+1}, with E_{i+1} being the object entity node and E_{i-1} being the target entity node; and a relationship node R_i. The operation of acquiring the target list from instance of relationship R_i and the object list of E_{i+1} is called ER-semijoin. We may illustrate the utility of ER-semijoin as follows:

(i). Object list of E_{i+1} is O_{i+1}, where $O_{i+1} = \{[a_{(i+1)1}, \ldots, a_{(i+1)j}, \ldots, a_{(i+1)k}] \mid k \geq 1;$ $[a_{(i+1)1}, \ldots, a_{(i+1)k}]$ is the list of the value on the primary keys of $E_{(i+1)}\}$

(ii). Target list of E_{i-1} is T_{i-1}, where $T_{i-1} = \{[a_{(i-1)1}, \ldots, a_{(i-1)j}, \ldots, a_{(i-1)n}] \mid (n \geq 1) \wedge ([a_{(i-1)1}, \ldots, a_{(i-1)n}] \in \pi_{PK_{(i-1)}}(R_i))$; $[a_{(i-1)1}, \ldots, a_{(i-1)n}]$ is the list of the value on the primary key of $E_{(i-1)}\}$; where $PK_{(i-1)}$ is the primary keys of the entity node E_{i-1}.

(iii) $S = \{r_m \mid (r_m \in R_i) \wedge (\pi_{PK_{i+1}}(r_m) \in O_{i+1})\}$, where $PK_{(i+1)}$ is the primary keys of the

entity node E_{i+1}

(iv) $T_{i-1} = \{t_m \mid (\exists r_j\ ((r_j \in S) \wedge (\pi_{PK_{(i-1)}}(r_j) = t_m))) \wedge ((\forall\ t_n\ )\ (t_n \in T_{i-1})\ (m \neq n) \rightarrow (t_m \neq t_n))\}$

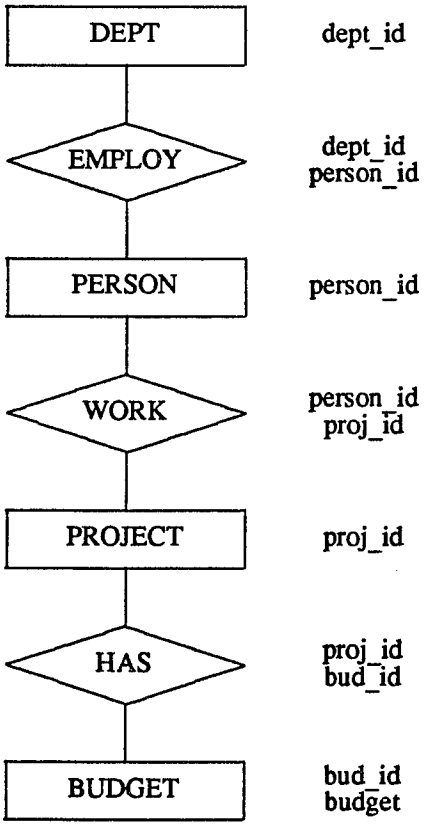


Fig. 1 The ERG of the relational database *DEPARTMENT*.

Example 1 : Let DEPT, EMPLOY, PERSON, WORK, PROJECT, HAS, BUDGET be the entity nodes and relationship nodes of the ERG *DEPARTMENT* as shown in fig. 1. The query "Find the *dept_id* of the *DEPTs* that *EMPLOY* the *PERSONs* who *WORK* in the *PROJECT* which *HAS BUDGET* with budget = 1500000.00 ", has long navigation path that can be represented by local regions as {[*DEPT*, *EMPLOY*, *PERSON*], [*PERSON*, *WORK*, *PROJECT*], [*PROJECT*, *HAS*, *BUDGET*]}. The utility of ER-semijoin on this query can be processed according to the following procedures:

(i). Selecting and projecting on the primary key of the entity node *BUDGET* for *bud_id* = 1500000.00

(ii). Employing ER-semijoin on the local region [*PROJECT*, *HAS*, *BUDGET*].

(iii). Employing ER-semijoin on the local region [*PERSON*, *WORK*, *PROJECT*].

(iv). Employing ER-semijoin on the local region [*DEPT*, *EMPLOY*, *PERSON*].

The procedures for the processing of ER-semijoin on a local region $[E_1, R, E_2]$ can be illustrated as:

(i). If E_2 is conjunctive to any local region that is processed before $[E_1, R, E_2]$, a set of values on the primary key of E_2 can be obtained by executing ER-semijoin on that conjunctive local region.

If the attributes of E_2 is restricted in the query, then selecting the primary key of E_2 from the tuples whose restricted attributes is in the restricted domain.

(ii). Collecting the value of the primary keys of E_1 from the tuples of R such that each tuple whose attributes of the primary key E_2 is in the list obtained from step (i).

Example 2 : For the same database as illustrated in example 1, the physical instance of the local region [*Project*, *Has*, *Budget*] are represented in Table 1. In this local region, the object entity node is *BUDGET* and the object list for "budget=1500000.00" is {[b000011], [b000013]}. The target entity node of this local region is *PROJECT* and the target list is {[p00111], [p00222]}.

Table 1 Physical Instances of the Database *DEPARTMENT*.

DEPT	
dept_id	location
d_001	Tower
d_002	Basement
d_003	A_building

EMPLOY	
dept_id	person_id
d_001	555555550
d_001	555555551
d_002	555555552
d_002	555555553
d_003	555555554
d_003	555555555

PERSON	
person_id	phone
555555550	3434445
555555551	3434446
555555552	3555555
555555553	4555555
555555554	5555555

PROJECT	
proj_id	date
p00111	5/6/83
p00211	6/7/83
p00222	8/7/83
p00257	5/3/84

HAS	
proj_id	bud_id
p00111	b000011
p00211	b000012
p00222	b000013

BUDGET	
bud_id	budget
b000011	1500000.00
b000012	1040000.00
b000013	1500000.00
b000014	2000000.00
b000015	1800000.00

WORK	
person_id	proj_id
555555550	p00111
555555551	p00111
555555552	p00211
555555553	p00211
555555554	p00222

ER-semijoin can be implemented on the RDKER (Relational Database with Knowledge of ERG) by using logical programming on a user's interface. The following rules illustrate an example for the implementation of ER-semijoin by using the logic programming.

```
er_semi_join(Ent1,Rel,Ent2,Att1,Att2) →
entity_node(Ent1,K11,key),
entity_node(Ent2,K21,key), abstract_name(K11,K1),
abstract_name(K21,K2), rel_res_1(Rel,Att_set,L),
var_list(L,R2), append([Rel],R2,R1),
R=..R1, att_order(Att_set,K1,O1,1),
att_order(Att_set,K2,O2,1),
findall(Attri,in_key_set(R,O1,O2,Attri,Att1),Att2).
```

2. EQUIVALENT OPERATION OF QUERY IN ER-SEMIJOIN

The physical instances of a local region must be consistent to the **local constraint** of a local region. The local constraint of a local region is that the projection of the primary keys of an entity node on the relationship node is the subset of the projection of those primary keys on the entity node which has the same primary keys.

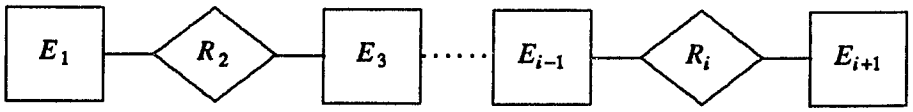


Fig. 2 An ERG of a database.

For a local region L_i, $L_i = [E_{i-1}, R_i, E_{i+1}]$ as shown in fig. 2, the physical representation of the entity nodes and the relationship of L_i can be denoted as :

$$E_{i-1} = r(a_{(i-1)1}, \ldots, a_{(i-1)k}, a_{(i-1)l}, \ldots); k \geq 1, l \geq 0 \quad (1.0)$$

$$E_{i+1} = r(a_{(i+1)1}, \ldots, a_{(i+1)g}, a_{(i+1)m}, \ldots); g \geq 1, m \geq 0 \quad (2.0)$$

$$R_i = r(a_{(i-1)1}, \ldots, a_{(i-1)k}, a_{(i+1)1}, \ldots, a_{(i+1)g}, a_{i1}, \ldots, a_{in}, \ldots); n \geq 0 \quad (3.0)$$

where $a_{(i-1)1}, ..., a_{(i-1)k}$, $a_{(i+1)1}, ..., a_{(i+1)k}$ and $a_{i1}, ..., a_{in}$ are the keys of $E_{(i-1)}$ $E_{(i+1)}$, and R_i respectively. Then the physical instances of L_i obeys the constraint

$$\pi_{a_{(i+1)k}}(R_i) \subseteq \pi_{a_{(i+1)k}}(E_{i+1}) \tag{4.0}$$

$$\pi_{a_{(i-1)k}}(R_i) \subseteq \pi_{a_{(i+1)k}}(E_{i-1}) \tag{5.0}$$

In a query, the attributes whose values are to be retrieved are called **target attributes**; the attributes whose domain are restricted are called **restricted attributes**.

Example 3 : For the database represented by the ERG in fig. 2, let a_{jp} be the pth attribute of the jth relation (an entity node or a relationship node). The following query is expressed on a universal relation as :

retrieve a_{1m} , a_{1n}

where $a_{(i+1)q} \neq 3000.00$

This query has access path $[E_1, R_2, ..., E_{i+1}]$; the attributes a_{1m} , a_{1n} are target attributes and the attribute $a_{(i+1)q}$ is a restricted attribute.

THEOREM 1 : For two conjunt local regions $[E_{i-1}, R_i, E_{i+1}]$ $[E_{i+1}, R_{i+2}, E_{i+3}]$ of a query, the target list T is the attributes of E_{i-1} to be retrieved and the object list O specifies a list of the key values of E_{i+3}. Then, the following two procedures on the query processing of these local regions are equivalent :

(i). First decompose this segment of the query into two local regions, then implement ER-semijoin on these two local region sequentially from E_{i+3} to E_{i-1}.

(ii). First apply the natural join operator to all entity nodes and relationship nodes in these two local regions and select tuples whose value of restricted attributes is in the restricted domain, then project target attributes on these tuples.

Proof: By procedure (ii), the processing of the query can be represented as

$$T = \pi_{Ta}(\delta((E_{i-1} \bowtie R_i \bowtie E_{i+1}) \bowtie (E_{i+1} \bowtie R_{i+2} \bowtie E_{i+3}))) \tag{6.0}$$

where δ is the operation to select the tuples whose key values of E_{i+3} is in the O.

The implementation of natural join on these two contiguous local regions is equivalent to the following equation :

$$(E_{i-1} \bowtie R_i \bowtie E_{i+1}) \bowtie (E_{i+1} \bowtie R_{i+2} \bowtie E_{i+3}) = (E_{i-1} \bowtie R_i \bowtie E_{i+1} \bowtie R_{i+2} \bowtie E_{i+3})$$

From the local constraint of Eq. 4 and 5, the entity node E_{i+1} can be omitted since there is no effect on the query. That is, the above joining operation on two local region can be further reduced to: $(E_{i-1} \bowtie R_i \bowtie R_{i+2} \bowtie E_{i+3})$

Thus, Eq. 6 can be reduced to

$$T = \pi_{Ta}(\delta(E_{i-1} \bowtie R_i \bowtie R_{i+2} \bowtie E_{i+3}) \qquad (7.0)$$

By employing the optimizing processing techinique [Ullm1982, Ullm1983], the Eq.7 can be optimized as (1) select the tuples of R_{i+2} whose key values of E_{i+3} are in O then project on the key values of PK_{i+1}; (1) select the tuples of R_i whose key values of E_{i+1} are in the list obtained from step (1), then project on the key values of PK_{i-1}. The optimizing operation is the interpretation of ER-semijoin processing of procdure (i). Thus, (i) and (ii) are the equivalent processing procedures for the query.

Example 4 : As in example 2, we may employ the natural join operator on the relations PROJECT, HAS, and BUDGET. By using natural join on all of the relations of the local region [PROJECT, HAS, BUDGET], the intermediate relation is shown in table 2. By selecting the tuples from this intermediate relation with "budget = 1500000.00" and projecting on "proj_id", the result is equivalent to the implementation of ER-semijoin on the same local region.

Table 2 The relation of $(PROJECT \bowtie (HAS \bowtie BUDGET))$.

proj_id	date	bud_id	budget
p00111	5/6/83	b000011	1500000
p00211	6/7/83	b000012	1040000
p00222	8/7/83	b000013	1500000

DEFINITION 4 : Let D be the representative instance of a database. For a local region L of D, the full reduction of L relative to D, denoted as FR(L,D), is π_L ($\bowtie D$). In

other words, FR(L,D) is the portion of L that take part in the join with all other relations in D.

DEFINITION 5 : The full semantic reduction of a local region L is the full reduction of the local region to its relationship node R denoted as FSR(R,L), is $\pi_R(\bowtie L)$. In other words, FSR(R,L) is the portion of R that takes part in the join with the other two entity nodes in the local region.

By the local constraint of a local region, the full semantic reduction of a local region with a binary relationship is equivalent to the physical instance of a relationship node.

THEOREM 2 : In an query on an ERG, if a local region which does not contain target attribute and restricted attribute then the full reduction of this local region can be reduced to its full semantic reduction.

Proof: It is obvious that the relationship node of the local region contains the keys of its adjacent entity nodes. From equations 8.0 and 9.0, the projection on one of the keys of the relationship node is the subset of the projection of that key on the entity node adjacent to this relationship node. Then, by mathching the key values of the tuples of relationship node to the key values of entity nodes, the joining of the relationship node and its adjacent entity nodes in the local region is equivalent to adding the values of nonkey attributes of a matched tuple of the entity nodes into each corresponding tuple of the relationship node. Since the local region does not contain the target attribute, the nonkey attributes of the entity nodes in this local region have nothing to do with the information to be retrived. Thus we can reduce the full reduction of such a local region to its full semantic reduction.

A RDKER (Relational Database System with Knowledge of Entity-Relationship Model) is a relational database whose concept view can be represented by a semantic structure of an ERG(Entity-Relationship Graph) [Chen1987]. Each RDKER has a

logical view which is represented as a semantic Entity-Relationship Model and a physical view that is represented by a set of physical relations(tables). For a RDKER, there is a function of "one-to-one and onto" mapping between its semantic model and physical relations. That is, for an entity node or a relationship node in an ERG, there is a bijective function which maps it to a physical instance.

Example 5 : The physical representation of the database *DEPARTMENT*, whose physical relations [DEPT, EMPLOY, PERSON, WORK, PROJECT, HAS, BUDGET] represented in table 1, has a semantic view represented by the ERG in fig. 1. It is obvious that for each entity node or relationship node in this ERG, there is a "one-to-one and onto" function which maps this entity node or relationship node to the representative instance of table 1.

3. PHYSICAL REPRESENTATION OF A LOCAL REGION

As discussed in section 2, a RDKER which is not a single entity node should be able to be decomposed into a set of local regions.

DEFINITION 6 : For a RDKER, each relation of an entity node or a relationship node of the ERG can be represented as $r_i([A_j]_i,[PK_m]_i,N_i,B_i,BP_i)$, where

r_i : Physical instance of an entity node or relationship node of ERG.

$[A_j]_i$: A set of attributes of r_i.

$[PK_m]_i$: A set of primary keys of r_i.

N_i : Cardinality or number of tuples (rows) of the physical instance r_i.

B_i : Bytes of a tuple of physical space of r_i

BP_i : Bytes of physical space of each primary key of relation r_i.

In a RDKER, the representation of an entity node or a relationship node as $r_i([A_j]_i,[PK_m]_i,N_i,B_i,BP_i)$ is called the **physical denotation** of an entity node or a relationship node of the ERG in a RDKER. An entity node or a relationship of the ERG

in a RDKER has one and only one physical instance corresponding to it. The physical denotation of this entity node or relationship node carries the information of the physical instance of an entity node or relationship node.

Example 6 : For the relation *BUDGET* as in Table 1, the attribute *bud_id* is defined as 8 bytes and the attribute *budget* is defined as 10 bytes. The physical denotation of *BUDGET* is *Budget* ([*bud_id*, *budget*], [*bud_id*], 5, 18, 8).

DEFINITION 7 : The physical denotation of a local region $[E_{i-1},R_i,E_{i+1}]$ of an ERG in a RDKER can be represented as $[r_{i-1}([A_{j_1}]_{i-1}, [PK_{m_1}]_{i-1},N_{i-1},B_{i-1},BP_{i-1}), r_i([A_{j_2}]_i,[PK_{m_2}]_i, N_i, B_i, BP_i),r_{i+1}([A_{j_3}]_{i+1},[PK_{m_3}]_{i+1},N_{i+1},_{i+1},BP_{i+1})]$, where $[PK_{m1}]_{i-1},[PK_{m3}]_{i+1} \subseteq [PK_{m2}]_i$. Let

$\rho(r_i)_{(PK)_{i-1}}$: the value set of $(PK)_{i-1}$ which is obtained from projecting the primary keys of entity node E_{i-1} on the physical instance of relationship node R_i.

$\rho(r_{i-1})_{(PK)_{i-1}}$: the value set of $(PK)_{i-1}$ which is obtained from projecting the primary keys of entity node E_{i-1} on the physical instance of entity node E_{i-1}.

$\rho(r_i)_{(PK)_{i+1}}$: the value set of $(PK)_{i+1}$ which is obtained from projecting the primary keys of entity node E_{i+1} on the physical instance of relationship node R_i.

$\rho(r_{i+1})_{(PK)_{i+1}}$: the value set of $(PK)_{i+1}$ which is obtained from projecting the primary keys of entity node E_{i+1} on the physical instance of entity node E_{i+1}.

The proportion of occurrences of the primary keys of E_{i-1} on R_i, denoted as $PKO_{i,(i-1)}$ is $PKO_{i,(i-1)} = \frac{| \rho(r_i)_{(PK)_{i-1}} |}{| \rho(r_{i-1})_{(PK)_{i-1}} |}$. The proportion of occurrences of the primary key of E_{i+1} on R_i, denoted as $PKO_{i,(i+1)}$, is $PKO_{i,(i+1)} = \frac{| \rho(r_i)_{(PK)_{i+1}} |}{| \rho(r_{i+1})_{(PK)_{i+1}} |}$.

Example 7 : The local region [*BUDGET*, *HAS*, *PROJECT*] is a local region of Table 1. For the relation *PROJECT*, the attribute *proj_id* is defined as 8 bytes and the attribute *date* is defined as 12 bytes; and for the relation *BUDGET*, the attributes is defined as in

example 6. Then the physical denotation of this local region is [*Budget* ([*bud_id*, *budget*], [*bud_id*], 5, 18, 8), *Has* ([*bud_id*, *proj_id*], [*bud_id*, *proj_id*], 3, 16, 16), *Budget* ([*proj_id*, *date*], [*proj_id*], 4, 20, 8)].

LEMMA 1 : The $PKO_{i,(i-1)}$ and $PKO_{i,(i+1)}$ of a local region $[E_{i-1},R_i,E_{i+1}]$ have the property that $0 \le PKO_{i,(i-1)},PKO_{i,(i+1)} \le 1$.

The $PKO_{i,(i-1)}$ and $PKO_{i,(i+1)}$ will be helpful in estimating the occurrences of the values of the primary keys of the entity node in the physical instance of the relationship node. The maximal possibility of occurrences is that when all values of the primary keys in the physical instance of an entity node are occurring in the physical instance of the relationship, which means that the $PKO_{l,j}$ of an entity node E_j in the physical instance of a relationship node R_l is one. The zero value of $PKO_{l,j}$ means that the projection of primary key of E_j on the physical instance of R_l is null.

In the physical instances of a local region $[E_{i-1}, R_i, E_{i+1}]$ of the ERG in a RDKER, the frequency of occurrences of E_{i-1} on R_i, denoted as $((FR)_{i,(i-1)})_j$, is the frequency of the jth value of the primary keys of entity node E_i in the relationship node R_i. The average frequency of occurrences of E_{i-1} on R_i is the average of the frequency of occurrences of the entity node E_i on the relationship node, denoted as $((FR)_{i,(i-1)})_{avg}$

With the proportion of occurrences and the average frequency of an entity node on a relationship node in a local region, we may estimate the space complexity of the intermediate table for the operation of natural join in a local region. In the same way, the space complexity of ER-semijoin can also be obtained.

LEMMA 2 : In the physical instances of a local region $[E_{i-1},R_i,E_{i+1}]$, the average frequency of entity node E_{i-1} on relationship node R_i is $((FR)_{i,(i-1)})_{avg} = \frac{\Sigma_j((FR)_{i,(i-1)})_j}{N_{i-1}}$ and the average frequency of E_{i+1} on R_i is $((FR)_{i,(i+1)})_{avg} = \frac{\Sigma_j((FR)_{i,(i-1)})_j}{N_{i+1}}$.

Example 8 : As in example 7, $((FR)_{PROJECT,HAS})_{avg} = \Sigma_j((FR)_{PROJECT,HAS})_j \ / \ N_{PROJECT} = (1*1+1*1+1*1+1*0)/4 = 3/4 = 0.75$.

LEMMA 3 : The local region $[E_{i-1},R_i,E_{i+1}]$ which has the physical denotation as represented in Lemma 2, then space complexity $Sp_{\bowtie}$ of operating natural join on the local region $(E_{i-1} \bowtie (R_i \bowtie E_{i+1}))$ is :

$$Sp_{\bowtie} = N_i * (B_i + B_{i+1} - BP_{i+1}) + Max((N_i * (B_{i-1} + B_i + B_{i+1} - BP_{i-1} - BP_{i+1}) + N_{i-1} * B_{i-1}), (N_{i+1} * B_{i+1} + N_i * B_i))$$

where Max is the function of the maximal space complexity, and

Case I : which has cardinality $M:N$ between the target entity node E_{i-1} and the object entity node E_{i+1}, has the property that

$$N_i = ((FR)_{i,(i-1)})_{avg} * N_{i-1}$$
$$= ((FR)_{i,(i+1)})_{avg} * N_{i+1}$$

Case II : which has cardinality 1:M between E_{i-1} and E_{i+1} has the properties that

$$N_i = ((FR)_{i,(i+1)})_{avg} * N_{i+1}$$
$$= PKO_{i,(i-1)} * N_{i-1}$$

Case III : which has cardinality 1:1 between E_{i-1} and E_{i+1}, has the following properties :

(1). $N_i = PKO_{i,(i-1)} * N_{i-1}$
$= PKO_{i,(i+1)} * N_{i+1}$

(2). $N_i \leq N_{i-1}, N_{i+1}$

Proof :

Case I : (1): From Definition 8 and Lemma 2, the order of the physical instance of the relationship node R_i is N_i. The order of the values of the primary keys of E_{i-1} in the relationship R_i is $N_i = ((FR)_{i,(i-1)})_{avg} * N_{i-1}$; and the order of the values of the primary keys of E_{i+1} in the relationship R_i is $N_i = ((FR)_{i,(i+1)})_{avg} * N_{i+1}$. Since the entity node or relationship node of a RDKER is at least in 3NF [Chen1987], the order of each

column of R_i should be equal. Thus $N_i = ((FR)_{i,(i-1)})_{avg} * N_{i-1}$ and $N_i = ((FR)_{i,(i+1)})_{avg} * N_{i+1}$.

(2): The local constraint of a local regions requires that

$$\pi_{a_{(i+1)k}}(R_i) \subseteq \pi_{a_{(i+1)k}}(E_{i+1}) \tag{4.0}$$

$$\pi_{a_{(i-1)k}}(R_i) \subseteq \pi_{a_{(i+1)k}}(E_{i-1}) \tag{5.0}$$

Then from the local constraint of a local region, the order of the intermediate table by the operation of natural join on a local region should be equal to N_i. Since the processing of the operation $(E_{i-1} \bowtie (R_i \bowtie E_{i+1}))$ is processed in two steps as $R_i \bowtie E_{i+1}$ and $E_{i-1} \bowtie R_i \bowtie E_{i+1}$, the space complexity is the maximal space of these two steps. That is,

$Sp_{\bowtie} = Max(N_i * (B_i + B_{i+1} - BP_{i+1}) + (N_i * (B_{i-1} + B_i + B_{i+1} - BP_{i-1} - BP_{i+1}) + N_{i-1} * B_{i-1}),$

$(N_i * (B_i + B_{i+1} - BP_{i+1}) + N_{i+1} * B_{i+1} + N_i * B_i)).$ $= N_i * (B_i + B_{i+1} - BP_{i+1}) +$

$Max((N_i * (B_{i-1} + B_i + B_{i+1} - BP_{i-1} - BP_{i+1}) + N_{i-1} * B_{i-1}), (N_{i+1} * B_{i+1} + N_i * B_i)).$

Case II :

(1). A local region with the 1:M relationship requires that in the physical instance of relationship R_i, the projection of the primary keys of E_{i-1} on R_i may have multiple occurrences. But by projecting the primary keys of E_{i+1} on the physical instance of R_i, each value must be unique. That is, the average of frequency of the primary keys of E_{i+1} on the relationship node R_i is equal to one.

(2). The estimation of the space complexity can be obtained by the same reason as Case I.

Case III : (1). For the same reason as in the Case II, the frequency of occurrence of E_{i-1} and E_{i+1} should be equal to one. That is, $((FR)_{i,(i-1)}) = ((FR)_{i,(i+1)}) = 1$.

(2). The estimation of the space complexity can be obtained by the same reasoning as in (2).

(3). As $N_i = PKO_{i,(i-1)} * ((FR)_{i,(i-1)}) * N_{i-1} * B_{i-1} = PKO_{i,(i+1)} * ((FR)_{i,(i+1)}) * N_{i+1} * B_{i+1}$, by substituting the frequency of occurrence into the equation of N_i, we get $N_i = PKO_{i,(i-1)} * N_{i-1} = PKO_{i,(i+1)} * N_{i+1}$. From Lemma 1, for $0 \le PKO_{i,(i-1)}, PKO_{i,(i+1)} \le 1$,

substituting into the previous equation, we get the result $N_i \leq N_{i-1}, N_{i+1}$.

Example 9 : Let the local region [*BUDGET*, *HAS*, *PROJECT*] illustrated in example 8 be a 1:1 relationship.

(1). $PKO_{PTOJECT,HAS} = (1 + 1 + 1)/4 = 0.75$

$PKO_{BUDGET,HAS} = (1 + 1 + 1)/5 = 0.6$

$N_{HAS} = PKO_{PROJECT,HAS} * N_{PROJECT} = 0.75 * 4$

$= PKO_{BUDGET,HAS} * N_{BUDGET} = 0.6 * 5 = 3$

(2). $N_{HAS} * (B_{HAS} + B_{BUDGET} - BP_{BUDGET}) = 3*(18 + 16 - 8) = 78$

$N_{HAS} * (B_{HAS} + B_{BUDGET} + B_{PROJECT} - BP_{BUDGET} - BP_{PROJECT}) + N_{HAS} * B_{HAS}$

$= 3*(18 + 16 + 20 - 8 - 8) + 4*20 = 3*38 + 80 = 194$

$N_{BUDGET} * B_{BUDGET} + N_{HAS} * B_{HAS} = 5*18 + 3*16 = 138.$

Thus $Max((N_{HAS} * (B_{HAS} + B_{BUDGET} + B_{PROJECT} - BP_{BUDGET} - BP_{PROJECT}) + N_{HAS} * B_{HAS}),$

$(N_{BUDGET} * B_{BUDGET} + N_{HAS} * B_{HAS})) = Max(138, 194) = 194.$

So, $Sp_{\bowtie} = 78 + 194 = 272.$

The ER-semijoin processing is based on a local region which represents the semantic unit of an ERG in a RDKER (Relational Database with Knowledge of ERG). The local regions of a query are obtained by decomposing a semantic structured query. The procedures for the processing of a query via ER-semijoin are :

(1). Parsing the syntax and checking the local constraint of the query.

(2). Converting a query into a semantic structured query — ER-query graph.

(3). Converting an ER-query graph into an ER-query tree.

(4). Decomposing an ER-query tree into a local region with a semantic order (the conjunctive sequence of the local regions of the query which may represent the access paths of the query).

(5). Processing local regions at inverted semantic order by employing ER-semijoin.

Steps 1-4 constitutes the query preprocessor based on a RDKER. These can be treated as higher level steps as compared to step 5 which processes the physical instances. In other words, the main processing schema which manipulates the operation of physical instances is the ER-semijoin. As the overhead of query preprocessing is much lower than the processing of the physical instances of a database, the time complexity and space complexity of the query processing of a query on a RDKER can be reduced to the time complexity and space complexity of ER-semijoin processing on the physical instances.

4. EFFICIENCY OF THE OPERATION OF ER-SEMIJOIN

The efficiency of time complexity and space complexity of exerting ER-semijoin on the access paths of a query can be obtained by comparing the time complexity and space complexity of employing the operator of natural join on the same access paths.

In a local region, if an object list is empty, then the order of the object list is zero. With zero order of the object list, the order of the target list must be equal to zero. The zero result of the empty target list can be directly obtained from the operation of ER-semijoin on the local region.

LEMMA 4 : In a local region with an empty object list, the empty target list will be obtained by the application of ER-semijoin on the physical instances of the local region.

Proof : The implementation of ER-semijoin on an empty object list of a local region $[E_{i-1},R_i,E_{i+1}]$, of which E_{i-1} is the target entity node and E_{i+1} is the object entity node, can be done according to Definition 3 as :

(i). For an empty object list, $O_{i+1} = \varnothing$.

(ii). Substituting $(O_{i+1} = \varnothing)$ into $S_i = \{ r_i \mid (r_i \subseteq R_i) \wedge (r_i \subseteq O_{i+1})\}$

$\Rightarrow S_i = \{r_i \ (r_i \subseteq R_i) \wedge (r_i \subseteq \varnothing)\}$

$\Rightarrow S_i = \varnothing$.

(iii). Substituting $(S_i = \varnothing)$ into $Q_i = \{t_m \mid (t_m \subseteq S_i) \wedge (m \neq n \Rightarrow t_m \neq t_n)\}$, we get $Q_i = \varnothing$.

(iv). Target list of E_{i-1}, $T_{i-1} = \pi_{k_{(i-1)}}(Q_i) = \pi_{k_{(i-1)}}(\varnothing) = \varnothing$, where $k_{(i-1)}$ is the key of E_{i-1}.

Thus, the operation of ER-semijoin on a local region will yield an empty target list.

The ER-semijoin can be applied recursively to the set of local regions of a query. In a recursive implementation of ER-semijoin on the local region of a query, the target list of a local region becomes the object list of its conjunctive local region which is to be processed next. Thus, the current target list will only be related to the next relationship node. That is, the operation of ER-semijoin in a set of local regions, the target entity Type of the current local region, can always be skipped.

The **coefficient of the spatial full semantic reduction** of a local region L_i, denoted as η_{L_i}, is the ratio of the space complexity obtained by operating ER-semijoin on the local region to the space complexity acquired by using natural join on the same local region. That is, $\eta_{L_i} = \frac{Sp_{ER}}{Sp_{\bowtie}}$.

LEMMA 5 : The local region $[E_{i-1},R_i,E_{i+1}]$ has physical denotation $[r_{i-1}([A_{j1}]_{i-1}, [PK_{m1}]_{i-1},N_{i-1},B_{i-1},BP_{i-1}),r_i([A_{j2}]_i,[PK_{m2}]_i,N_i,B_i,BP_i),r_{i+1}([A_{j3}]_{i+1},[PK_{m3}]_{i+1},N_{i+1},B_{i+1},BP_{i+1})]$, order of object list CN_o, and order of target list CN_t. Then the space complexity of implementing ER-semijoin in this local region is

$$Sp_{ER} = (BP_{i+1} * CN_o) + (B_i * N_i) + (BP_{i-1} * CN_t)$$

Proof : The space complexity Sp of the operation of ER-semijoin on the local region $[E_{i-1},R_i,E_{i+1}]$ is the maximal space complexity for the implementation of ER-semijoin on this local region. In case of the operation of ER-semijoin on a local region, the maximal space complexity is equal to the summation of space complexity of object list, target list, and physical instance of relationship node. Since the space complexity

of object list is $CN_o * BP_{i+1}$ and the space complexity of target list is $CN_t * BP_{i-1}$, we get

$Sp = (BP_{i+1} * CN_o) + (B_i * N_i) + (BP_{i-1} * CN_t)$

LEMMA 6 : The range of the coefficient of the full reduction η_L is $0 \le \eta_{L_i} \le 1$.

Proof : The space complexity of employing ER-semijoin on local region with physical denotation $[r_{i-1}([A_{j1}]_{i-1}, [PK_{m1}]_{i-1}, N_{i-1}, B_{i-1}, BP_{i-1}), r_i([A_{j2}]_i, [PK_{m2}]_i, N_i, B_i, BP_i), r_{i+1}([A_{j3}]_{i+1}, [PK_{m3}]_{i+1}, N_{i+1}, B_{i+1}, BP_{i+1})]$ has the worst case space complexity and best case complexity as :

(i) From Lemma 5 we get :

$$Sp_{ER} = (BP_{i+1} * CN_o) + (B_i * N_i) + (BP_{i-1} * CN_t)$$
$$\le (BP_{i+1} * PKO_{i(i+1)} * N_{i+1}) + (B_i * N_i) + (BP_{i-1} * PKO_{i(i-1)} * N_{i-1}). \quad (8.0)$$

and from Lemma 3 $Sp_{\bowtie} \le N_i * (B_{i-1} + B_i + B_{i+1} - BP_{i-1} - BP_{i+1}) + N_{i-1} * B_{i-1} + N_{i+1} * B_{i+1}$.

Thus $Sp_{\bowtie} \le N_i * B_{i-1} + N_i * B_i + N_{i+1} * B_{i+1} - N_i * BP_{i-1} - N_i * BP_{i+1} + N_{(i-1)} * B_{(i-1)} + N_{(i+1)} * B_{(i+1)}$. So, $Sp_{\bowtie} - Sp_{ER} = N_i * B_{i-1} + N_i * B_{i+1} - N_i * BP_{i-1} - N_i * BP_{i+1} - BP_{i-1} * PKO_{i(i-1)} * N_{i-1} - BP_{i+1} * PKO_{i(i+1)} * N_{i+1}$. For $PKO_{i(i-1)}, PKO_{i(i+1)} \le 1$, the derivation of $Sp_{\bowtie} - Sp_{ER} \ge 0$ is trivial. The result is equivalent to $Sp_{\bowtie} \ge Sp_{ER}$. Then

$$\eta_{L_i} = \frac{Sp_{ER}}{Sp_{\bowtie}} \le 1$$

(ii). The best case of the space complexity of the operation of ER-semijoin on a corresponding physical instance of a local region is when the object list approaches empty. From the Lemma 6 when the object list approaches empty, the space occupied by the target list approaches zero. Substituting $CN_o = 0$ and $CN_t = 0$ into Eq. 8, we get $Sp_{ER} = 0 + B_i * N_i + 0$. Since the ER-semijoin is an intelligent operator on a local region, when the object list is empty, the operation of ER-semijoin can skip the join operation to get an empty object list. Thus $B_i * N_i \approx 0$. So, the best case of space complexity approaches zero.

Example 10 : ER-semijoin is applied on the same local region [*BUDGET*, *HAS*, *PROJECT*] as illustrated in example 9. From Table 1, the object list of this local region is [$b000011$, $b000012$, $b000013$, $b000014$, $b000015$], and $CN_o = 5$.

$B_{HAS} * N_{HAS} = 3 * 16 = 48$.

By employing ER-semijoin on this local region, the target list is [$p00111$, $p00211$, $p00222$]. $\Rightarrow CN_t = 3$.

$\Rightarrow Sp_{ER} = CN_t * BP_{PROJECT} + N_{HAS} * B_{HAS} + CN_o * BP_{BUDGET} = 5*8 + 3*16 + 3*8 = 72$

$\Rightarrow \eta = \frac{Sp_{ER}}{SP_{\bowtie}} = \frac{72}{272} = 0.2647.$

Sammy proposed a matrix representation model to estimate the storage cost on a distributed database system [Rior1976]. Substituting a single node into Sammy's model, the monthly cost of the storage of a single file on a single node can be obtained as $G = b * l$, where G is the cost, b is the average cost per bit and l is the average length in bits of file. By converting the cost to CPU time and the starage to retrieval we get $T = C * b * l$, where T is the time complexity of retrieval and C is the constant. For the big memory space complexity, the overhead for the optimization of block accessing is necessary. Thus, the equation is modified as $T = C * b^{\nu} * l$, where ν is the exponetial coeffienct and $\nu \geq 1$. Considering the time complexity of searching during the operation of join, the model of correlation equation which estimates the time complexity from the space complexity can be obtained.

Let $[E_{i-1}, R_i, E_{i+1}]$ be a local region in a RDKER, and $[r_{i-1}([A_{j1}]_{i-1}, [PK_{m1}]_{i-1}, N_{i-1}, B_{i-1}, BP_{i-1}), r_i([A_{j2}]_i, [PK_{m2}]_i, N_i, B_i, BP_i), r_{i+1}([A_{j3}]_{i+1}, [PK_{m3}]_{i+1}, N_{i+1}, B_{i+1}, BP_{i+1})]$ be the physical denotation of this local region. Assuming the order of operating natural join on the local region is $E_{i-1} \bowtie (R_i \bowtie E_{i+1}) = E_{i-1} \bowtie R_{temp}$. Then the correlated equation is the expression of the time complexity which correlated to the space complexity on $[E_{i-1}, R_i, E_{i+1}]$, denoted as $T_{\bowtie}$, and $T_{\bowtie} = C_m(B_m * N_m)^{\nu} + C_n(B_n * N_n)^{\nu} + T_s(R_i, E_{i+1}) + T_s(R_{temp}, E_{i-1})$, where C_m and C_n is the coefficient of space complexity for

$(R_i \bowtie E_{i+1})$ and $(R_{temp} \bowtie E_{i-1})$, respectively; v is the correlational coefficient of space complexity and time complexity; N_m and N_n is the order of the intermediate relation; $T_s(R_{temp}, E_{i-1})$ and $T_s(R_i, E_{i+1})$ are the time complexity of searching the matching of the primary keys between relations (R_{temp}, E_{i-1}) and (R_i, E_{i+1}), respectively.

THEOREM 3 : The time complexity of employing natural join on the local region $[E_{i-1}, R_i, E_{i+1}]$ whose physical instances are unsorted is :

(i). Time complexity of intermediate relation R_{temp}, where $R_{temp} = (R_i \bowtie E_{i+1})$ is

$$C_0(N_i(B_i + B_{i+1} - BP_{i+1}))^v + (\frac{N_{i+1}+1}{2}) * N_i,$$

where the physical denotation of R_{temp} is $[r_{temp}, ([A_{j2}]_i \cup [A_{j3}]_{i+1}), [PK_{m1}]_{i-1}, N_i, B_i + B_{i+1} - BP_i, BP_{i-1})$,

(ii) Time complexity of $(R_{temp} \bowtie E_{i-1})$ is $T_{\bowtie} = C_1(N_i(B_{i-1} + B_i + B_{i+1} - BP_{i-1} - BP_{i+1}))^v + (\frac{N_{i-1}+1}{2}) * N_i$.

(iii). Time complexity of using natural join on local region $[E_{i-1}, R_i, E_{i+1}]$ which has order of operation $(E_{i-1} \bowtie (R_i \bowtie E_{i+1}))$ is :

$$T_{\bowtie}(E_{i-1}, R_i, E_{i+1}) = (C_0(N_i(B_i + B_{i+1} - BP_{i+1}))^v + (\frac{N_{i+1}+1}{2}) * N_i) + (C_1(N_i(B_{i-1} + B_i + B_{i+1} - BP_{i-1} - BP_{i+1}))^v + (\frac{N_{i-1}+1}{2}) * N_i). \quad (9.0)$$

Proof :

(i). The intermediate relation R_{temp} is created during the processing of natural join operator on R_i and E_{i+1}. The attributes of R_{temp} is the union of attributes of physical instance of R_i and the attributes of the physical instances of E_{i+1}, that is, the set of attributes of R_i is $[A_{j2}]_i \cup [A_{j3}]_{i+1}$. After the operation of natural join of R_i with E_{i+1}, R_i should join with E_{i-1}. Thus the primary keys of intermediate relation of R_i can be reduced to $[PK_{m1}]_{i-1}$. As a local region of a RDKER has the property of local constraint as illustrated in the Eq. 3.0 and Eq. 4.0 :

$$\pi_{a_{(i+1)k}}(R_i) \subseteq \pi_{a_{(i+1)k}}(E_{i+1}) \tag{4.0}$$

$$\pi_{a_{(i-1)k}}(R_i) \subseteq \pi_{a_{(i-1)k}}(E_{i-1}). \tag{5.0}$$

The operation of natural join operator on R_i and E_{i+1}, has equivalence with natural join expression as

$$R_i \bowtie E_{i+1} = \pi_{[A_{j2}]_i \cup [A_{j2}]} \sigma_{R_i.a_i=E_{i+1}.a_1 \wedge ,...,R_i.a_j=E_{i+1}.a_j}(R \times S), \text{ where } a_1 ... a_j \subseteq PK_{i+1}. \tag{9.0}$$

The physical instances of a local region have the property of semantic integrity. Thus for each value of the primary keys in the physical instance of the relationship node, there is exactly one value in the column of the physical instance of the related entity node corresponding to it. That is, in the physical instances, the mapping from the primary key of the relationship node to the same attributes of its adjacent entity node is a bijective function. Then, the order of the intermediate relation after the process $\sigma_{R_i.a_i=E_{i+1}.a_1 \wedge ,...,R_i.a_j=E_{i+1}.a_j}(R \times S)$ is equal to the order of R_i. After the operation of the cartesian product on R and S, the length of each tuple of relation obtained from $R_i \times E_{i+1}$ is $B_i + B_{i+1}$. Since the basic difference between natural join and equal join is that in the natural join processing one of the duplicated columns of the intermediate relation are deleted, the duplicated primary key PK_{i+1} of the primary key of E_{i+1} is deleted after the operation of projection. Thus, with the subtraction of the duplicated primary key, the length of the each tuple in intermediate relation is $B_i + B_{i+1} - BP_{i+1}$. For the operation of natural join on the physical instances of $[R_i, E_{i+1}]$, the search algorithm is:

```
procedure search(value1,value2);
begin
   For each value1 of PK_{i+1} in relationship_node
      do find value2 in entity_node
      where the PK_{i+1} of value2=value1;
end.
```

For each value of PK_{i+1} in the physical instance of R_i, the best case to find the value is 1, and the worst case is N_{i+1}. So, the average time complexity of searching the primary key in E_{i+1} is $\frac{N_{i+1}+1}{2}$. Thus $T_{\bowtie}$ = Time complexity of searching + time complexity of the operation of join. That is

$$T_{\bowtie} = C_0(N_i(B_i + B_{i+1} - BP_{i+1}))^{\nu} * N_i + (\frac{N_{i+1}+1}{2}) * N_i.$$

$$\Rightarrow T_{\bowtie} = C_0(N_i(B_i + B_{i+1} - BP_{i+1}))^{\nu} + (\frac{N_{i+1}+1}{2})) * N_i.$$

(ii). The time complexity of $T_{\bowtie}(R_{temp} \bowtie E_{i-1})$ can be obtained by the same procedures as described in the step(i). By substituting the physical denotation of R_{temp} into R_i and E_{i-1} into E_{i+1}, via the same induction processes, we may get $T_{\bowtie}$ as:

$$T_{\bowtie} = C_1(N_i(B_{i-1} + B_i + B_{i+1} - BP_{i+1} - BP_{i-1}))^{\nu} + (\frac{N_{i-1}+1}{2}) * N_i$$

where C_1 is the coefficient of space complexity of the operation of natural join on R_{temp} and E_{i-1}; ν is the correlation coefficient of space complexity as defined in the Definition 10.

The correlation coefficient is the coefficient via which the space complexity and time complexity of the operation of join are correlated. If the time complexity of processing join is proportional to the space complexity, then the value of ν is equal to 1.

THEOREM 4 : The time complexity of employing natural join on the local region $[E_{i-1}, R_i, E_{i+1}]$ whose physical instances are sorted is :

(i). Time complexity of intermediate relation R_{temp}, where time complexity of the creating intermediate relation of $R_{temp} = R_{i-1} \bowtie E_{i+1}$ is $C_0(N_i(B_i + B_{i+1} - BP_{i+1}))^{\nu} + (N_i + N_{i+1})$

where the physical denotation of R_{temp} is $[r_{temp}, ([A_{j2}]_i \cup [A_{j3}]_{i+1}, [PK_{m1}]_{i-1}, N_i, B_i + B_{i+1} - BP_{i,BP_{i-1}})$,

(ii) Time complexity of $(R_{temp} \bowtie E_{i-1})$ is $T_{\bowtie} = C_1(N_i(B_{i-1} + B_i + B_{i+1} - BP_{i-1} - BP_{i+1}))^v * N_i + (N_i + N_{i-1})$.

(iii). Time complexity of operating natural join on the local region L_i where $L_i = [E_{i-1}, R_i, E_{i+1}]$ which has the order of operation $(E_{i-1} \bowtie (R_i \bowtie E_{i+1}))$ is :

$$T_{\bowtie}(E_{i-1}, R_i, E_{i+1}) = (C_0(N_i(B_i + B_{i+1} - BP_{i+1}))^v + (N_i + N_{i+1})) + (C_1(N_i(B_{i-1} + B_i + B_{i+1} - BP_{i+1} - BP_{i-1}))^v + (N_i + N_{i-1})). \quad (11.0)$$

Proof : The physical instances of a relation in local region can be stored in a sorted order of its primary key. For an indexed relation, the pointer can be set up to the indexed attributes of the relations. Then the joining operation can be implemented more efficiently during the searching of the tuples in the different relations that are to be joined. That is, for two sorted relations with order N_1 and N_2, the time complexity of searching primary keys in both indexed relations is $N_1 + N_2$. Thus the time complexity of searching primary keys for the natural join operation on the physical instances of R_i, E_{i+1} is $N_i + N_{i+1}$. Substituting the time complexity of sorted relations in (i) of Theorem 1, we obtain the result as : $T_{\bowtie}(R_i, E_{i+1}) = C_0(N_i(B_i + B_{i+1} - BP_{i+1}))^v + (N_i + N_{i+1})$. By the same reason, substituting the time complexity of natural join operation on the sorted relations, we may get the result is, as in (ii). Then for the natural join operation on the sorted files of a local region with the processing order as $(E_{i-1} \bowtie (R_i \bowtie E_{i+1}))$, the time complexity is :

$$\begin{aligned} T_{\bowtie}(E_{i-1}, R_i, E_{i+1}) &= T_{\bowtie}(E_{i-1}, R_{tem}) + T_{\bowtie}(R_i, E_{i+1}). \\ &= (C_0(N_i(B_i + B_{i+1} - BP_{i+1}))^v + (N_i + N_{i+1})) \\ &\quad + (C_1(N_i(B_{i-1} + B_i + B_{i+1} - BP_{i+1} - BP_{i-1}))^v + (N_i + N_{i-1})). \end{aligned}$$

THEOREM 5 : local region $L_i = [E_{i-1}, R_i, E_{i+1}]$ which is a local region in the access paths of a RDKER. In this local region, the the object entity node is E_{i+1} which has object list O_{i+1} with order CN_o, and the target entity node is E_{i-1} which has target list T_{i-1} with order CN_t. The time complexity of the operation of ER-semijoin on this local region of unsorted files is

$$T_{ER} = C_o * (BP_{i-1} * CN_t + BP_{i+1} * CN_o + N_i * B_i)^{v} + (N_i * \frac{CN_o + 1}{2}) \quad (12)$$

Proof: Form the definition 3, the processing of ER-semijoin is

(1). $S = \{r_m | (r_m \subseteq R_i) \wedge (\pi_{PK_{i+1}}(r_m) \subseteq O_{i+1})\}$

(2). $T_{i-1} = \{t_m \mid (\forall t_m)((\exists r_j \subseteq S) \wedge (\pi_{PK_{i-1}}(r_j) = t_m)) \wedge ((\forall t_m)((\exists t_n)(t_m \in T_{i-1})(t_n \in T_{i-1})(m \neq n) \rightarrow (t_m \neq t_n)))\}$ For the processing of ER-semijoin on a local region, the intermediate file to be created is the target list T_{i-1}, and the working memory for the processing of searching, selection, and projection is the working memory of R_i and the object list O_{i+1}. Then the space complexity of implementing ER-semijoin on the local region is $T_{sp} = C_o * (BP_{i-1} * CN_t + BP_{i+1} * CN_o + N_i * B_i))$; and the average time complexity of the processing (2) is $(N_i * \frac{CN_o + 1}{2})$. Thus by substituting these terms into the correlation equation of time complexity, the result of T_{ER} is

$$T_{ER} = C_o * (BP_{i-1} * CN_t + BP_{i+1} * CN_o + N_i * B_i))^{v} + (N_i * \frac{CN_o + 1}{2})$$

THEOREM 6 : local region $L_i = [E_{i-1}, R_i, E_{i+1}]$ which is a local region in the access paths of a RDKER with physical denotation $[r_{i-1}([A_{j1}]_{i-1}, [PK_{m1}]_{i-1}, N_{i-1}, B_{i-1}, BP_{i-1}), r_i([A_{j2}]_i, [PK_{m2}]_i, N_i, B_i, BP_i), r_{i+1}([A_{j3}]_{i+1}, [PK_{m3}]_{i+1}, N_{i+1}, B_{i+1}, BP_{i+1})]$. $[PK_{i-1}]$. In this local region, the object entity node is E_{i+1} which has object list O_{i+1} with order CN_o, and the target entity node is E_{i-1} which has target list T_{i-1} with order CN_t. The time complexity of the operation of ER-semijoin on this local region of sorted files is

$$T_{ER} = C_o * (BP_{i-1} * CN_t + BP_{i+1} * CN_o + B_i * N_i)^{v} + (N_i + CN_o) \quad (15.0)$$

Proof : The only difference of operating ER-semijoin operation on the sorted files is the time complexity of searching. That is, we substitute the time complexity of searching $N_i + CN_o$ into the Eq. 13, the time complexity of implementing ER-semijoin on a local region is $T_{ER} = C_o * (BP_{i-1} * CN_t + BP_{i+1} * CN_o + B_i * N_i)^{v} + (N_i + CN_o)$.

Example 11: The correlational coefficient of the correlation equation is :

$$\nu=\frac{log\,(T_{\bowtie}-Ts)-C_c}{\log(C_o * t)} \tag{16.0}$$

where $T_{\bowtie}$ is the time complexity of operating natural join on the local region; T_s is the time complexity of searching the value set of the primary key in the value set of the primary key of object entity node; C_c and C_o are constants.

Proof : As the space complexity is a linear function of time, let $Sp_{\bowtie}=C_o+C_1$.

for t=0, the space of working memory is 0, i.e. $Sp_{\bowtie}(B)|_{t=0}=C_1=0$.

$\Rightarrow C_1=0, \Rightarrow Sp_{\bowtie}(B)=C_o * t$

from the correlation equation, we may get $T_{\bowtie}=C_2(C_o * t+C_1)^{\nu}+T_s$.

$\Rightarrow (T_{\bowtie}-T_s)=C_2(C_o * t+C_1)^{\nu}$.

$\Rightarrow log\,(T_{\bowtie}-T_s)=logC_2+\nu * log\,(C_o * t+C_1)$.

$\Rightarrow log\,(T_{\bowtie}-T_s)=logC_2+\nu * log\,(C_o * t)$.

$\Rightarrow log\,(T_{\bowtie}-T_s)=C_c+\nu * log\,(C_o * t)$, where $C_c=logC_2$.

$\Rightarrow log\,(T_{\bowtie}-T_s)-logC_2=\nu * log\,(C_o * t)$.

$\Rightarrow \nu=\frac{log\,(T_{\bowtie}-T_s)-C_c}{logC_o+logt}$.

In the physical instances of a RDKER, if the files are indexed, the time complexity of searching is constant. That is, in Eq. 14.0, Ts = constant, and Ts << $T_{\bowtie}$. Then, Eq. 14.0 can be simplified as $\nu=\frac{logT_{\bowtie}-C'_c}{logt}$, where $C'_c=C_c-\nu logC_o$.

THEOREM 7 Let the coefficient of temporal full semantic reduction of a local region be υ then $0\le\upsilon\le1$.

Proof: For the unsorted files the theorem can be derived from the Eq. 9 and 12; for the sorted files the theorem can be derived from Eq. 11 and 13.

CHAPTER 4
ENTITY-RELATIONSHIP QUERY GRAPH PROCESSING ON THE RELATIONAL DATABASE SYSTEMS

1. ENTITY-RELATIONSHIP GRAPH AND ENTITY-RELATIONSHIP QUERY GRAPH

An *ERG* can be represented either by a single entity node or by local regions. For an *ERG* which is not a single entity node, it should be able to be represented as a set of local regions [Chen1987b]. The "local region" is the semantic unit of a semantically clear *ERM*. Each local region in an *ERM* contains a pair of entity types and a relationship type that connects these entity types. Such a semantically clear *ERM* which represents the semantic structure of a relational database can be further defined by an implementation model represented as an *ERG* (Entity-Relationship Graph). By using an *ERG* as the semantic structure of a *RDKER*, the accessing direction of a query on the database can be represented in the subgraph which is obtained by mapping a query onto an *ERG*. Then by adding the relational operators of the query to the subgraph, an *ERQG* can be obtained. In other words, a query can be represented by an *ERQG* (ER-query Graph) which consists of a subgraph of *ERG* and the relational operators of a query. Besides on a global accessing interface of a database (e.g. on a universal relation interface), a query can be processed with the aid of the *ERG* according to the following steps : (i) Allocating of subnodes (attributes) (ii) Obtaining an *ERQG* from subnodes and *ERG* (iii) Processing of a query based on the *ERQG*. These subnodes, main-nodes, and the allocation of subnode and main-nodes of an *ERG* will be discussed in this section.

1.1. DEFINITIONS OF ERG AND ERQG

Since an *ERG* is an implementation model which is derived from an *ERM*, it inherits the semantics of an *ERM* defined on a relational database. Like an *ERM*, it may have many different abstract levels [Chen1983], and likewise an *ERG* may contain several abstract levels. In this chapter, only the basic level of the implementation model of an *ERG* will be discussed. A **basic level** of an *ERG* is designed such that each entity node or relationship node of the graph are connected to the attribute nodes of that node. For our convenience, we use *ERG* to represent the graph of the basic level of a *RDKER*.

An *ERG* is defined as an extended *ERD* [Chen1987b]. In an *ERG*, each pair of entity nodes and the relationship nodes that connects them is defined as a local region. In the previous chapter, we concern only entity nodes and relationship nodes and the arcs connects these nodes. Now, we want to discuss attributes nodes connects to entity nodes and relationship nodes. For this purpose, the entity nodes and relationship nodes of an *ERG* are called main-nodes and the arcs connects them are called main-arcs.

In an *ERG*, there are two types of nodes: main-node and subnode and there are two types of arcs: main-arc and sub-arc. A *main-node* is an entity node or a relationship node and a *subnode* is a node which represents an attribute of an entity node or a relationship node. A *main arc* is the arc in the *ERG* whose two end nodes are main-nodes; a *sub-arc* is an arc in the *ERG* that one of the end nodes of the arc is a main-node and the other end node is the subnode. An *unit graph U* of an *ERG* which is not a single entity type is the graph representation of a local region. The *unit graph* $U = [e_i, r_j, e_k, ARC_{ij}, ARC_{jk}, \{a_{i1}, \ldots, a_{il}\}, \{a_{j1}, \ldots, a_{jm}\}, \{a_{k1}, \ldots, a_{kn}\}, \{d_{i1}, \ldots, d_{il}\}, \{d_{j1}, \ldots, d_{jm}\}, \{d_{k1}, \ldots, d_{kn}\}, C_i, C_k]$ of a local region L is a labeled graph which consists of three main-nodes, two main arcs, three sets of subnodes and three sets of sub-

arcs; where e_i and e_k are the main-node of entity type, r_j is the main-node of relationship type; $\{a_{p1}, \ldots, a_{pq}\}$ is a set of q subnodes of attributes of the main-node e_p or r_p; $\{d_{t1}, \ldots, d_{ts}\}$ is a set of s sub-arcs and both of the two end nodes of each sub-arc d_{ts} are the main-node e_t or r_t and the subnode a_{ts}; ARC_{ij} and ARC_{jk} are the main-arcs with two end nodes $<e_i, r_j>$ and $<e_k, r_j>$ respectively; C_i and C_k are the cardinality of e_i and e_k that are labeled on the arc ARC_{ij} and ARC_{jk} separately, and $C_i, C_k \in \{1, M, N\}$.

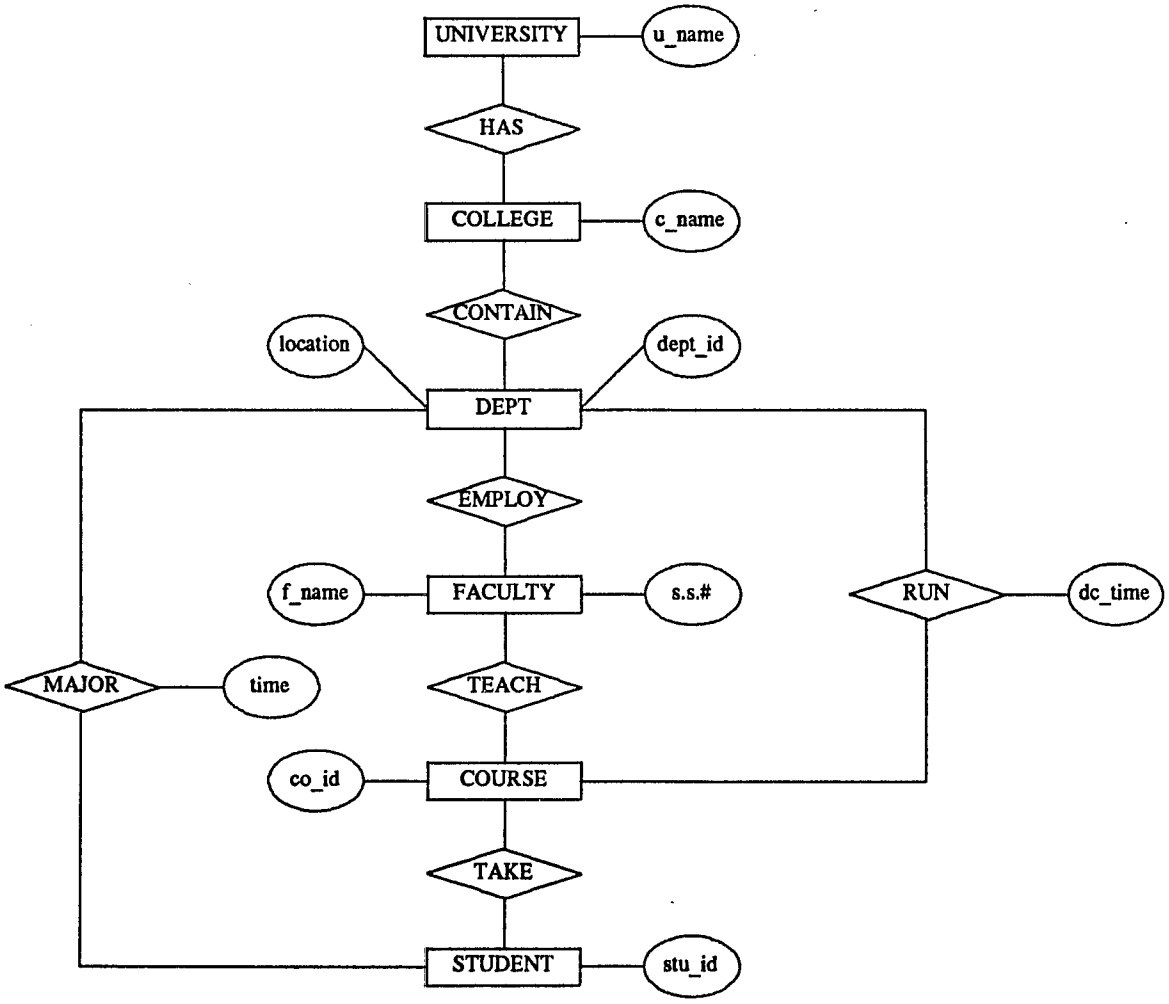


Fig. 1 The *ERG* of the relational database UNIVERSITY.

Example 1 : Fig. 1 is an *ERG* for the relational database system UNIVERSITY. The figure shows two types of main-nodes : entity type based main-node and relationship

type based main-nodes. The main-nodes {UNIVERSITY, COLLEGE, DEPT, FACULTY, COURSE, STUDENT} are main-nodes of entity types. Those {HAS, CONTAIN, EMPLOY, TEACH, TAKE, RUN, MAJOR} are the main-nodes of relationship types. The subnodes are {u_name, c_name, location, dept_id, f_name, s.s#, time, co_id, stu_id, dc_time}. In the *ERG* of the UNIVERSITY, the nodes of ellipse are the subnodes; the nodes of rectangle are the entity type based main-node; the nodes of the rhombus are the relationship type base main-node.

Two local regions are said to be conjunct if they have the common main-nodes. The conjunction of two local region L_1 and L_2 are the common main-nodes of these local regions, denoted as $L_1 \copyright L_2$, where © is the operator of conjunction. For example, as shown in fig. 1, [DEPT, EMPLOY, FACULTY] © [DEPT, RUN, COURSE] = {DEPT}; [UNIVERSITY, HAS, COLLEGE] © [COURSE, TAKE, STUDENT] = ∅.

A *NJ query* (Natural Join Query) on a relational schema may calculate the natural join of all relations in derived database [Chu1981, Shmu1982, Shmu1981]. As discussed by Goodman, queries which may be represented by tree schemas are easier to be processed than those cyclic queries. An acyclic query can be either converted to the tree schema and processed by the semijoin or directly processed by joining the relations in the acyclic graph into a new relation [Kamb1985b, Shmu1981, Shmu1982].

In a RDKER, the logic structure of the relational database is represented by an *ERG*. A *NJ query* on a relational database based on an *ERG* can be represented by a subgraph of *ERG*. For a NJ query with acyclic structure, ER-semijoin can be employed to processed it. The processing of acyclic subgraphs of a NJ query will be studied in this chapter. A NJ query based on an *ERG* can be viewed as a relation with natural join operator between relations represented as entity node or relationship node.

DEFINITION 1 A *unit query graph UQ* is a unit graph with the relational operators on the main-nodes such that $UQ_i = [e_i, r_j, e_k, ARC_{ij}, ARC_{jk}, \{a_{i1}, \ldots, a_{il}\}, \{a_{j1}, \ldots, a_{jm}\}, \{a_{k1}, \ldots, a_{kn}\}, \{d_{i1}, \ldots, d_{il}\}, \{d_{j1}, \ldots, d_{jm}\}, \{d_{k1}, \ldots, d_{kn}\}, C_i, C_k, L_i, L_j, L_k$; where e_i and e_k are the main-node of entity type, r_j is the main-node of relationship type; $\{a_{p1}, \ldots, a_{pq}\}$ is a set of q subnodes of attributes of the main-node e_p or r_p; $\{d_{t1}, \ldots, d_{ts}\}$ is a set of s sub-arcs and both of the two end nodes of each sub-arc d_{ts} are the main-node e_t or r_t and the subnode a_{ts}; ARC_{ij} and ARC_{jk} are the main-arcs with two end nodes $<e_i, r_j>$ and $<e_k, r_j>$ respectively; C_i and C_k are the cardinality of e_i and e_k that are labeled on the arc ARC_{ij} and ARC_{jk} separately, and $C_i, C_k \in \{1, M, N\}$; L_i, L_j, L_k are relational operators on the main-nodes i, j, and k respectively and $L_i, L_j, L_k \in \{\wedge, \vee, \neg, \times, -, \Theta\}$ which are operators of relational algebra that represent *and* (intersection), *or* (union), *negation, cartesian product, difference*, and *exclusive or* respectively.

A query may have sub-queries which contain two derived relations which are represented by NJ queries In other words, by representing a NJ query as a subgraph of *ERG*, a query may have subgraphs such that a subgraph of a query may contain two subgraphs with relational operators between these two subgraphs.

DEFINITION 2 A NJERQG (Natural Join Entity-Relationship Query Graph) is a query graph (i) which contains a connected graph such that the natural join operator is to be employed on all the main-nodes of the graph, (ii) which can be represented by a sequence of unit query graphs.

Query processing of a *NJ query* and equijoin query can be processed by semijoin [Kamb1985b, Shmu1981, Shmu1982]. In this chapter, we extended the function of the query processing such that queries which can be represented by *NJ queries* with relational operators between them can be processed by the optimizing technique discussed in this chapter. We also assume that the relational database is based on an *ERG*

[Chen1987b]. Those queries which satisfy these limitations can be represented by an *ERQG* which is defined in the following definition.

DEFINITION 3 An ERQG (Entity-Relationship Query Graph) of a query on a *RDKER* is a query graph such that it can be represented as

(1) A NJERQG (Natural Join Entity-Relationship Query Graph), or

(2) A query graph which contains a relational operator between two NJERQGs, or

(3) A nested query graph which contains a relational operator between two nested query graphs (a query graph which contains a relational operator between two NJERQGs is a nested query graph).

Thus, an *ERQG* can be represented as a query graph based on an *ERG* which contains a sequence of unit query graphs and a precedence order between unit query graphs such that (i) the innerest pair of parenthesis specifies a subgraph of natural join (ii) the relational operator between two subgraphs specifies the relational operator between two subgraphs.

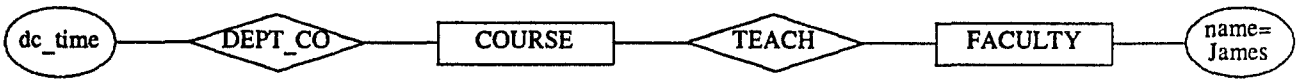


Fig. 2.a

Fig. 2.b

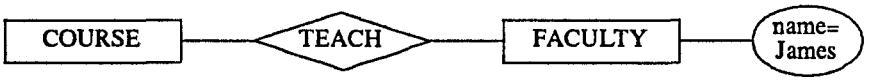


Fig. 2.c

Fig.2 An *ERQG* on the *ERG* UNIVERSITY.

The trivial case of an *ERG* contains only a main-node. We now discuss the *ERG* and *ERQG*, which is not a trivial case and that is composed of more than one main-node. A *walk* of an *ERQG* is a finite nonempty sequence $W = v_0 d_{01} v_1 d_{12} v_2 \cdots d_{(n-1)n} v_n$, where v_i is a main-node of *ERQG* and d_{ij} is a main-arc of *ERQG* with two main-node v_i and v_j. If all of the arcs of the walk W are distinct, then this walk is a trail. For a trail W of *ERQG*, if all of the nodes of W are distinct, W is called a *path* [Thul1981].

Two main-nodes v_1 and v_2 in an *ERG* are said to be *connected* if there is at least a path from v_1 to v_2 or from v_2 to v_1. The set of main-nodes in the path which connects two main-nodes is called *connecting nodes*. For example, the path $v_1\ d_{1a}\ v_a\ d_{ab}\ v_b\ d_{bc}\ v_c\ d_{c2}\ v_2$ connects the main-nodes v_1 and v_2, the set $\{v_1\ v_a\ v_b\ v_c\ v_2\}$ is the connecting nodes for the main-nodes v_1 and v_2. If all pairs of nodes of a graph are connected, then such a graph is called a connected graph. An *ERG* is a connected graph. That is, each pair of nodes in an *ERG* should be connected. A cycle of an *ERG* is a path whose start

node and end node is the same.

A query interface on a relational database can be categorized into two types as CLNI (conceptual level navigating interface) and URI (universal relation interface). For a query on a CLNI interface, the user has to navigate the conceptual level of the relation database; for a query on a URI, the user does not need to navigate the conceptual level of the relational database. Most of the query language on a relational database such as QUEL and SQL are conceptual level navigating interface; while the query language designed on the System/U is a universal relation interface. The allocation of a *ERQG* from a query on a relational database is based on the type of the query interface.

In an *ERQG*, the subnodes whose information are to be retrieved are called *retrieval subnodes*, and the main-nodes that adjacent to the retrieval subnodes are called *retrieval main-nodes*; the subnodes which restrict the domain of the retrieval subnodes are called *restricted subnodes*, and the main-nodes that are adjacent to the restricted subnodes are called the *restricted main-nodes*.

In a CLNI, the *target part* of an *ERQG* (*TERQG*) contains retrieval subnodes and the main-nodes which are in the navigating path of the query; the *restriction part* of an *ERQG* (*RERQG*) contains the restricted subnodes and the main-nodes of the navigating paths that do not contain the arcs of the target part.

Example 2 : A query of SQL on the relational database *UNIVERSITY* of fig. 1 is :

SELECT c_name, dept_id, location

FROM COLLEGE, CONTAIN, DEPT

WHERE COLLEGE.c_name = CONTAIN.c_name

AND CONTAIN.dept_id=DEPT.dept_id

The target part of this NJ query is specified in the query and which is represented as COLLEGE, CONTAIN, DEPT. The restriction part of this query is empty.

Example 3 : The *ERQG* of the query "Find the dc_time of the courses that are taught by the faculty with name = 'James' " on UNIVERSITY of fig.1 is represented in the fig. 2.a. This *ERQG* can be decomposed into a target part represented in the fig. 2.b and a restriction part represented in the fig. 2.c.

1.2. AUTOMATIC ALLOCATION OF ERQG ON A UNIVERSAL RELATION

By using ERG as the semantic structure of a relational database, the *ERQG* can be allocated on a universal relation. The traditional universal relation approach always use the functional dependency to obtain the access paths of a query on a universal relation [Sagi1983, Ullm1983]. The allocation of an ERQG on a universal relation by the ERG approach does not have to use the functional dependency between attribute.

Any subode of the *ERQG* that represents the target attribute of a query is a **target subnode**. In an *ERQG*, if a subnode specifies the domain of the value of an attribute in a query, then this subnode is defined as a **restricted subnode**; the main-node that directly connected to the restricted subnode is the restricted main-node.

Example 3 : The following query Q is based on a Universal Relation interface on the relational database UNIVERSITY represented in fig. 1 :

Q: retrieve u_name, c_name
where s.s.# = '45678912345'

In the query Q, the subnodes are u_name, c_name, and ss.#. where u_name and c_name are target subnodes and s.s# is a restricted subnode. Thus the main-nodes UNIVERSITY and COLLEGE are target nodes; the main-node FACULTY is a restricted main-node.

DEFINITION 4 : Let TN_Q be the set of target subnodes of a *ERQG* on an universal relation based on an *ERG* such that $TN_Q = \{a_1, \ldots, a_k\}$. The target part of the *ERQG* *(TERQG)* contains TN_Q and the target main-nodes, where the target main-nodes of the

query is equivalent to $T_e \cup T_r \cup T_p$, in which T_e, T_r, and T_p are defined as follows :

(1). $T_e = \{e_i \mid (\exists a_j)((a_j \in TN_Q) \wedge (a_j \in \{a_{i1}, \ldots, a_{im}\}))$; where $\{a_{i1}, \ldots, a_{im}\}$ is the set of subnodes of the entity type main-node e_i }.

(2). $T_r = \{r_l \mid (\exists a_q)((a_q \in TN_Q) \wedge (a_q \in \{a_{l1}, \ldots, a_{ln}\}))$; where $\{a_{l1}, \ldots, a_{ln}\}$ is the set of subnodes of the relationship type main-node r_l }.

(3). T_P which contains all nodes in $\{P_i \mid (P_i = \{v_s, \ldots, v_f, \ldots, v_n\})$, where $(v_s, v_e \in (T_e \cup T_r))$ and P_i representing connecting nodes from v_s to v_e }.

DEFINITION 5 : Let RN_Q be the set of restricted subnodes of an *ERQG* on a universal relation based on an *ERG* such that $RN_Q = \{a_1, \ldots, a_q\}$. The restriction part of the *ERQG* (*RERQG*) contains RN_Q and the restriction main-nodes, where the restriction main-nodes of the query is equivalent to $R_e \cup R_r \cup R(v_e)$, in which R_e, R_r, and $R(v_e)$ are defined as follows :

(1). $R_e = \{e_s \mid (\exists a_i)((a_i \in RN_Q) \wedge (a_i \in \{a_{s1}, \ldots, a_{su}\}))$; where $\{a_{s1}, \ldots, a_{su}\}$ is the set of subnodes of the entity type main-node e_s }

(2). $R_r = \{r_t \mid (\exists a_w)((a_w \in RN_Q) \wedge (a_w \in \{a_{t1}, \ldots, a_{tv}\}))$; where $\{a_{t1}, \ldots, a_{tv}\}$ is the set of subnodes of the relationship type main-node r_t }.

(3). $R(v_e) = \{P(v_e)_i \mid (P(v_e)_i = \{v_s, v_g, ..., v_e\})$, where $((v_s \in (R_e \cup R_r)) \wedge (v_e \in (T_e \cup T_r \cup T_P)) \wedge (v_s \notin (T_e \cup T_r \cup T_p))) \wedge (\forall v_h (v_h \in (v_s, v_g, ..., v_e)(v_h \neq v_e) \rightarrow (v_h \notin (T_e \cup T_r \cup T_p))))$; P_i is the connecting nodes from v_s to v_e }.

For any nonempty subgraph of a *RERQG*, the intersection of this subgraph with the *TERQG* is a nonempty set of main-nodes which is a subset of $T_e \cup T_r \cup T_p$. The **degree** of a node of an *ERQG* is given by the number of main-arcs that are incident at that node. For the convenience of query processing, we assign the direction of the processing order to an *RERQG*. A path of a *RERQG* is a directed graph from the restricted

main-node to the target main-node. The **in-degree** of a node of *RERQG* is the number of main-arcs that have this node as head. The **out-degree** of a node of *RERQG* is the number of main-arcs that have this node as tail.

The *ERG* F is called a subgraph of the *ERG* G denoted as $F \subseteq G$, if every node of F (main-node or subnode) is also a main-node of G and every arc of F (main-arc or subarc) is also an arc of G. An *RERQG* can be decomposed into a set of connected subgraphs, such that each subgraph contains just one restricted main-node. Thus a *RERQG* is the union of its subgraphs *SRERQG* (subgraph of restriction part of ER-query Graph) that may be denoted as $RERQG = \cup_i SRERQG_i$. A *RERQG* does not have to be a connected graph, but each $SRERQG_i$ of *RERQG* should be a connected graph.

Then, for a *SRERQG*, we define the **start node** of the the *SRERQG* as the restricted main-node in the *SRERQG*; The **end node** of a main-node in a *SRERQG* is defined as a the target main-node of the *SRERQG*. From (3) of definition 5 we know that for each *SREQG*, the only main-node in the target part is the end node of the *SRERQG*. The sequence of query processing of a *SRERQG* starts from the start node to the end node of the *SRERQG*.

The **ring sum** of two query graphs $ERQG_1$ and $ERQG_2$ denoted as $ERQG_1 \oplus ERQG_2$, which does not have any isolated node and consists of only those arcs which are either in $ERQG_1$ or in $ERQG_2$ but not in both of them.

THEOREM 1 : Let $\oplus$ be the operator of ring sum, $ERQG_q$ be an ER-query Graph, and $TERQG_q$ and $RERQG_q$ be the subgraphs of target part and restriction part of $ERQG_q$ respectively. Then $(TERQG_q \oplus RERQG_q) = (TERQG_q \cup RERQG_q)$

Proof: Let $TERQG_q = \{<l_{TERQG_t}, L_{TERQG_t}>_1, \ldots, <l_{TERQG_t}, L_{TERQG_t}>_s\}$ and $RERQG_q = \{<l_{RERQG_t}, L_{RERQG_t}>_1, \ldots, <l_{RERQG_t}, L_{RERQG_t}>_t\}$. From (3) of Definition 3, we get $(\forall l_{RERQG_t})$ $(l_{RERQG_t} \notin TERQG_q)$

$\Rightarrow (TERQG_q \oplus RERQG_q) = (TERQG_q \cup RERQG_q)$.

By representing an *ERQG* as *TERQG* $\cup$ *RERQG*, the query of these *ERQG* can be processed in two steps as : (i) processing the *RERQG* (ii) processing the *TERQG*. And the query processing of an *ERQG* should starts from the start nodes of *SRERQG*s and end at end nodes.

2. STRUCTURE OF ERQG

The processing of an acyclic query graph has more advantage than the processing of a cyclic query graph, thus the conversion of a cyclic query graph to the tree structure is widely studied [Epst1982, Shmu1981]. For the convenience of the query processing, we assigned the direction of processing order to the *ERQG*. As an *ERQG* represents a subgraph of an *ERG*, if the subgraph of the *ERG* represented by an *ERQG* which has a path that starts from a main-node and ends at the same main-node then the *ERQG* is a cyclic query graph. The representation of a query by an *ERQG* is different from that by a query graph in a traditional relational database. An *ERQG* is represented by a set of local regions and it can always be processed by the ER-semijoin based on the *ERG*, which is not applicable on the traditional relational database.

An *ERQG* can be either a cyclic graph or an acyclic graph. To process an acyclic graph efficiently, the query graph is always decomposed into a set of local regions so that we may use optimizing operator - ER-semijoin to process it. For a cyclic graph whose restriction part is an acyclic graph, ER-semijoin can also be used to process this restriction part. In this section we will discuss the structure of an *ERQG* based on the structure of the target part and the restriction part of the *ERQG*.

In the query processing of a *ERQG*, a subnode which is not a restricted subnode, target subnode, or a prime (key attribute) is called a redundant subnode . In other words, during the query processing of an *ERQG*, we may neglect these redundant sub-

nodes. A main-node of entity type of an *ERQG* which is not a restricted main-node and such that it can also be neglected in the query processing is called a redundant main-node.

Thus an *ERQG* with nonempty *RERQG* can be processed in the order as (i) convert the cyclic subgraphs of the restriction part to the tree structures (if it is available) (ii) process of the restriction part (ii) process of the target part. For the processing of a restriction part which contains n disjunct subgraphs, each subgraph can be processed independently as in steps (i) and (ii). In the following sections, the categories of a *ERQG* and the conversion of subgraphs of restriction part to a tree structure is discussed.

2.1. CATEGORIES OF ERQG

Since several relational query interfaces may be built on the top of a relational database system [Li1984], the user may access a database system through any interface built on the system. The representation of a query by an *ERQG* helps the processing of the query on this system in a unique way, i.e. the processing of a query can be reduced to the processing of an *ERQG* on any interface of a database system with multiple interfaces.

A cyclic subgraph of a *RERQG* may always be converted to a tree structure and which can be decomposed into a set of local regions. Thus for a subgraph of a *RERQG* which is either an acyclic graph or one that can be converted to an acyclic graph, we may use ER-semijoin to process this subgraph [Chen1987a].

As discussed in the previous section, an *ERQG* which is based on an *ERG* can be decomposed into of *TERQG* and of *RERQG*, these *TERQG* and *RERQG* can be either a cyclic graph or an acyclic graph. Thus an *ERQG* can be grouped as : *cyclic ERQG* and *acyclic ERQG*. According to the structures of the *TERQG* and *RERQG*, an *ERQG* can

further be grouped into the following five types:

Type I : Linear structure of an *ERQG* : both *TERQG* and *RERQG* of an *ERQG* are linear structure.

Example 4 : The main-nodes of an *ERG* is shown in fig. 3, let a_{i_p} be the pth subnode of the main-node r_i. The query Q_1 is expressed in the query language for the universal relation as :

Q_1: retrieve a_{n_1} a_{n_2}

where a_{v_1} = *'VALUE ...'*.

The *ERQG* representation of the query Q_1 is a linear structure.

Type II : Tree structure of an *ERQG* : The structure of an *ERQG* is an acyclic graph.

Example 5 : The main-nodes of an *ERG* is shown in fig. 3, let a_{x_p} and a_{j_q} be the pth sub-node of main-node r_x and the qth subnode of r_j respectively. The query Q_2 is expressed in the query language for the universal relation as :

Q_2: retrieve a_{x_1}, a_{x_2}, a_{x_3}

where a_{n_4} = (*'VALUE* $\cdots$ *'*) $\vee$ (a_{v_2} = *'VALUE* $\cdots$ *'*).

Both the *TERQG* and *RERQG* of the *ERQG* of Q_2 are tree structures.

The *ERQG*s of Type I and Type II are acyclic graphs.

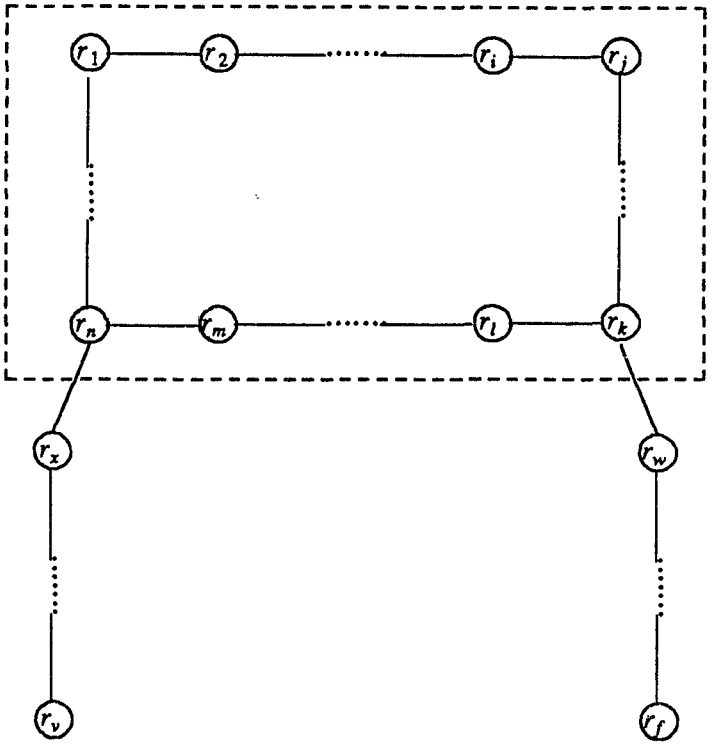


Fig. 3 An cyclic *ERQG* where only *RERQG* or *TERQG* is cyclic.

Type III : Only *RERQG* is a cyclic graph.

Example 6 : The main-nodes of an *ERG* is shown in fig. 3. Let a_{x_p}, a_{k_q}, and a_{j_g} be the pth subnode of main-node r_x, the qth subnode of main-node r_k, and the gth subnode of r_j respectively. The query Q_3 is expressed in the query language for the universal relation as :

Q_3: retrieve a_{x_1}, a_{x_2}

where $(a_{k_1} = 'VALUE \cdots ') \wedge (a_{j_1} = 'VALUE \cdots ')$.

The *ERQG* of the query Q_3 shows that only the *RERQG* of this *ERQG* is a cyclic graph.

Type IV : Only *TERQG* is a cyclic graph.

Example 7 : The main-nodes of an *ERG* is shown in fig. 3. Let a_{n_p}, a_{v_q}, and a_{j_g} be the pth subnode of main-node r_n, the qth subnode of main-node r_v, and the gth subnode of

r_j respectively. The query Q_4 is expressed in the query language for the universal relation as :

Q_4: retrieve a_{j_1}, a_{n_2}

where $(a_{v_1} = 'VALUE \cdots ') \wedge (a_{v_2} = 'VALUE \cdots ')$.

For the *ERQG* of this query, the *TERQG* of this *ERQG* is a cyclic graph while the *RERQG* of the *ERQG* is an acyclic graph.

Type V : Both *TERQG* and *RERQG* are cyclic graphs.

Example 8 : The main-nodes of an *ERG* is shown in fig. 3. Let a_{n_p}, a_{k_q}, a_{i_w}, and a_{f_g} be the pth subnode of main-node r_n, the qth subnode of main-node r_k, the wth subnode of main-node r_i, and the gth subnode of r_f respectively. The query graph Q_4 is expressed in the query language of the universal relation as :

Q_5: retrieve a_{i_1}, a_{k_2}

where $(a_{f_1} = 'VALUE \cdots ') \vee (a_{n_1} = 'VALUE \cdots ') \vee (a_{i_2} = 'VALUE \cdots ')$.

In Q_5, both the *TERQG* and the *RERQG* of the *ERQG* are cyclic graphs.

2.2. BRANCHING AND MERGING ON A RERQG

Comparing a cyclic query graph with a query tree, the query processing on the query tree has more advantages than the query processing on the cyclic graph: (1) The query graph can be decomposed into segments of subgraph, then we can employ semi-join [Yu1984], ER-semijoin [Chen1987a] and etc. on the physical instances of the subgraph. For a cyclic graph, we have to use join operator on the representative instances of the cyclic graph, such a processing procedure takes more time and more physical space than the the query processing on the decomposed segments (2) In a distributed database system, the decomposition of a cyclic query into segments can reduce the size of the transactions among distributed nodes [Kamb1985b].

DEFINITION 6 : Two queries are said to be **semantically equivalent** if both of them will produce the same result under the processing rules that are applied on their physical instances. Two ERQGs are said to be **equivalent** if both of them represent two semantically equivalent queries.

A main-node whose out-degree is greater than or equal to two is called a **branching node** (as fig. 4.a). A main-node whose in-degree is greater than or equal to two is called a **merging node** (as fig. 4.b). A main-node can be a merging node for a set of arcs and be a branching node for another set of arcs (as fig. 4.c). The loop of an arc on a main-node is not allowed, i.e., a main-node cannot be a merging node and a branching node of an arc.

A **branching arc** is an arc that has a branching node as its tail, the head of this branching arc is the **branching head**; a **merging arc** is an arc that has a merging node as its head, the tail of this merging arc is the **merging tail.** The merging of two merging arcs of a merging node is the unification of the semantics of two merging arc.

The **order of a branching node** is the out-degree of that branching node, and the **order of a merging node** is the in-degree of that merging node. For a branching node with n branching arcs, the order of this node is n, and this node is a **n-ary** branching node. In the same way, for a merging node with n merging arcs, the order of this node is an **n-ary** merging node. An n-ary branching node may have another role as an m-ary merging node. For example, the node B_1 in fig. 4.c is a branching node with order 2 and a merging node with order 2.

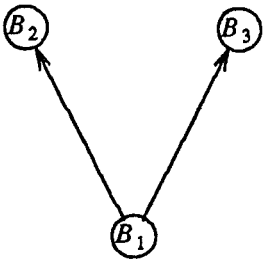


Fig. 4.a A branching node of *RERQG*.

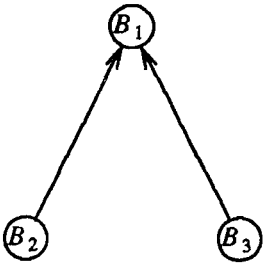


Fig. 4.b A merging node of *RERQG*.

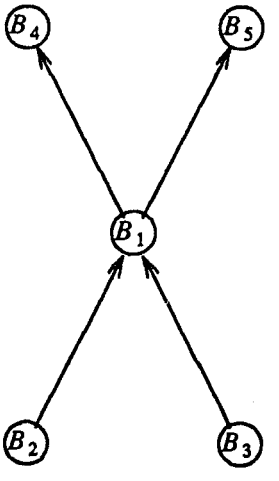


Fig. 4.c A node which is a merging node and a branching node of *RERQG*.

Fig. 4 A subgraph of a *RERQG* with branching node and merging node.

2.3. MERGING ARCS ON A MERGING NODE

Two merging arcs with same head may be merged to a new arc. The head of this

new arc is the original merging head and the tail of this new arc is obtained by applying the relational operators on the physical instances of these two merging tails.

DEFINITION 7 : For arcs ARC_{ph} and ARC_{qh} (where p, q are the tails of merging arc ARC_{ph} and ARC_{qh} respectively; h is the head of these two merging arcs and h is an entity main-node), the operators which represent the merging of these two arcs are defined as follows : denoted as

(1). Union (logical OR) : $(ARC_{ph} \cup ARC_{qh} \rightarrow ARC_{gh}) \wedge (r_p \cup r_q \rightarrow r_g)$; where ARC_{gh} is the new arc with tail g; r_p and r_q are the physical instances that contain only values of key attributes which are common key attribute of the main-node p and q; r_g represent the derived relation [Date1983] obtained from processing the operator on r_p and r_q. This derived relation corresponds to a new node that is the tail of the arc ARC_g. The union operator ($\cup$) in the *ERQG* is mapped to the physical instances on which the operator ($\cup$) of the relational algebra can be applied.

(2). Intersection (Logical AND) : $(ARC_{ph} \cap ARC_{qh} \rightarrow ARC_{gh}) \wedge (r_p \cap r_q \rightarrow r_g)$; where ARC_{ph}, ARC_{qh}, ARC_{gh}, r_g, r_p,r_q, p, q, and g are the same with (1). The intersection operator ($\cap$) in the *ERQG* should be mapped to the physical instances on which the operator ($\cap$) of the relational algebra can be applied.

(3). Difference : $(ARC_{ph} - ARC_{qh} \rightarrow ARC_{gh}) \wedge (r_p - r_q \rightarrow r_g)$ where ARC_{ph}, ARC_{qh}, ARC_{gh}, r_g, r_p,r_q, p, q, and g are the same with (1); The difference operator (–) in the *ERQG* should be mapped to the physical instances on which the operator (–) of the relational algebra can be applied.

(4). Cartesian Product : $(ARC_{ph} \times ARC_{qh} \rightarrow ARC_{gh}) \wedge (r_p \times r_q \rightarrow r_g)$. where ARC_{ph}, ARC_{qh}, ARC_{gh}, r_g, p, q, and g are the same with (1); r_p and r_q are the physical instances of the main-node p and q respectively. The cartesian operator ($\times$) in the *ERQG* should be mapped to the physical instances on which the operator ($\times$) of the relational algebra can be applied.

(5). Join $(ARC_{ph}$ join $ARC_{qh} \rightarrow ARC_{gh}) \wedge (r_p \bowtie r_q \rightarrow r_g)$; where ARC_{ph}, ARC_{qh}, ARC_{gh}, r_g, p, q, and g are the same with (1); r_p and r_q are the physical instances of the main-node p and q respectively. The join operator ($\bowtie$) in the *ERQG* should be mapped to the physical instances on which the operator ($\bowtie$) of the relational algebra can be applied.

(6). Negation $(\neg\ ARC_{ph} \rightarrow ARC_{gh}) \wedge (\neg\ r_p \rightarrow r_g)$; where ARC_{ph}, ARC_{gh}, r_q, p, and g are the same with (1). r_p is the physical instances of the main-node p. The negation operator ($\neg$) in the *ERQG* should be mapped to the physical instances on which the operator ($\neg$) of the relational algebra can be applied.

(7). Exclusive-OR $(ARC_{ph} \ominus ARC_{qh} \rightarrow ARC_{gh}) \wedge (r_p \ominus r_q \rightarrow r_g)$; where ARC_{ph}, ARC_{qh}, ARC_{gh}, r_g, r_p,r_q, p, q, and g are the same with (1). The Exclusive-OR operator ($\ominus$) in the *ERQG* should be mapped to the physical instances on which the operator ($\ominus$) of the relational algebra can be applied. The operation of this operator on the physical instances can be represented by an equation which is relevant to the union, intersection, and difference operator as : $(r_p \ominus r_q) = (r_p \cup r_q) - (r_p \cap r_q)$.

(8). Division $(ARC_{ph} \div ARC_{qh} \rightarrow ARC_{gh}) \wedge (r_p \div r_q \rightarrow r_g)$. where ARC_{ph}, ARC_{qh}, ARC_{gh}, r_g, p, q, and g are the same with (1); r_p and r_q are the physical instances of the main-node p and q respectively. The division operator ($\div$) in the *ERQG* should be mapped to the physical instances on which the division operator ($\div$) of the relational algebra can be applied.

Example 9 : For the relational database system *UNIVERSITY* as represented in *ERG* of the fig. 1, the following queries have the same access paths but different operator representation on the merging node *DEPT* :

(i). Find the dept_id of the *DEPT* that *RUN* the *COURSE* and that *EMPLOY* the *FACULTY* who *TEACH* the same *COURSE* with course_id = 'csc4354'. This query has the relational operator representation "$\cap$" on the merging node *DEPT*.

(ii). Find the dept_id of the *DEPT* that *RUN* the *COURSE* or that *EMPLOY* the *FACULTY* who *TEACH* the same *COURSE* with course_id = 'csc4354'. This query has the relational operator representation "$\cup$" on the merging node *DEPT*.

(iii). Find the dept_id of the *DEPT* that *RUN* the *COURSE* and that does not *EMPLOY* the *FACULTY* who *TEACH* the same *COURSE* with course_id = 'csc4354'. This query has the relational operator representation "$-$" on the merging node *DEPT*.

The merging of arcs on a merging node depends on the relational operator representation of that merging node. The application of these operators (*Union*, *Intersection*, *Difference*, *Cartesian product*, *Join*, *Exclusive-OR*, *Division*) to merge the arcs of a merging node of an *ERQG* is equivalent to the implementation of these operators on the corresponding physical instances of the merging tails of these arcs.

2.4. DECOMPOSITION OF BRANCHING ARC AND BRANCHING NODE

A branching node with order n can be decomposed into n separate arcs with separate tails. The decomposition of a branching node depends on the semantic representation of that node. The decomposition of a branching node can be categorized as the decomposition of the branching node of entity type and the decomposition of the branching node of the relationship type.

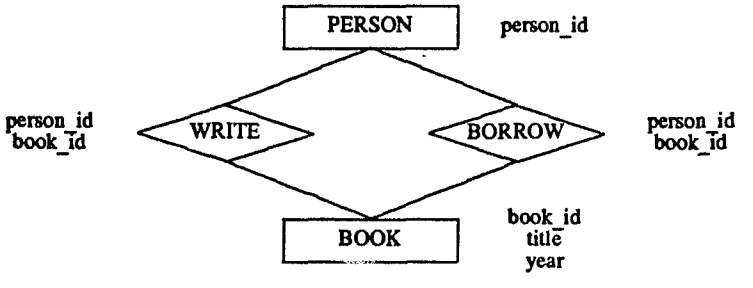


Fig. 5 The *ERG* of the *RDKER LIBRARY*.

The decomposition of a branching node or the breaking up of a branching arc

will convert a cyclic query graph into a tree structure [Shmu1981]. Many different techniques for the conversion of a cyclic query graph to a query tree on a relational database system are proposed [Epst1982, Shmu1981, Kamb1985a, Shmu1982]. The conversion of cyclic subgraphs of an *ERQG* to tree structures in a *RDKER* is different from the conversion of a cyclic query graph to a query tree in a traditional relational database system. The difference between a cyclic *ERQG* of a *RDKER* and a cyclic query graph of a traditional relational database system is illustrated by the Example 10.

The conversion of cyclic graph to a query tree in the traditional relational database system does not have to consider the semantic representation of relations. That is, a traditional relational database system always uses the data definition on the attributes and the functional dependency among attributes to recognize a relation and the relationship between relations. In a *RDKER* (Relational Database with Knowledge of ERG), we use the semantic representation of a relation in a local region to recognize a relation.

Example 10 : In the relational database as represented in the fig. 5, the relationship types BORROW(person_id, book_id) and WRITE(person_id, book_id) contain the same set of attributes. In this case, we can not distinguish these two relationship types by just looking into the data definition of the set of the attributes of these two relationship types. Concerning to the semantic representation in the *ERG*, these two relationship types have different semantics which represent WRITE and BORROW separately.

A relational database system, which is based on the semantics of its *ERG*, is different from traditional relational database system. Thus the technique for the mapping of cyclic subgraphs of restriction part of an *ERQG* to tree structures is different from the mapping techniques that is applicable to the traditional relational database system

[Epst1982, Shmu1981].

2.4.1. AN ENTITY-TYPE BASED BRANCHING NODE

For a cyclic query graph, the decomposition of a branching node will convert a cyclic subgraph to a tree structure [Epst1982]. As discussed in the previous section that a cyclic *ERQG* of a *RDKER* has a semantic representation, thus the decomposition of a branching node of subgraphs an *ERQG* to a tree structure is equivalent to the semantic decomposition of that branching node.

An **old arc** of a branching node is a branching arc before the decomposition rule is applied. A **new arc** of a decomposed branching node represents a new arc obtained from the decomposition of a branching arc on that branching node.

An n-ary branching node can be decomposed into n arcs by the following **decomposition rule** :

(i). Decomposition of an n-ary branching node of a relationship type:

An n-ary branching node of a relationship type can be decomposed into *n* arcs. The head of these new arcs are the original heads of the old arcs; the tail of each new arc is obtained by projecting the key attributes of the head on the physical instance of the branching node.

(ii). Decomposition of an n-ary branching node of an entity type:

An n-ary branching node of an entity type can be decomposed into *n* arcs. The head of these new arcs are the original heads of the arcs; the tails of each new arc is obtained by projecting the primary keys of the branching node on the physical instance of that branching node.

Example 11 : The database UNIVERSITY is shown in fig. 5. On this database, the query " find person_id, where year > 1980 " can be mapped onto the database such that it can be represented by an *ERQG* as in fig. 5. Since the main-node BOOK is a

branching node with order 2, this query graph can be decomposed into *ERQG* as shown in fig. 6 according to the decomposition rule.

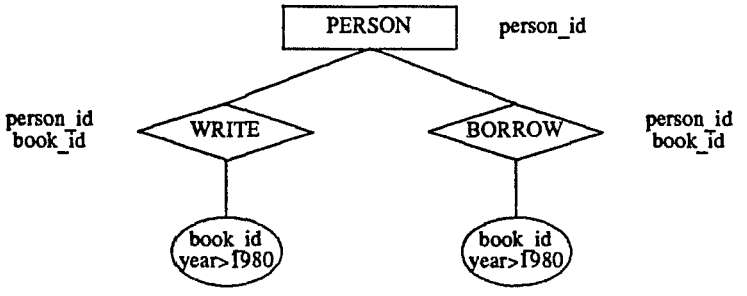


Fig. 6 The decomposition of the branching node *BOOK*.

From the local constraint of a local region, and the decomposition rule on a branching node of an entity type, the following two procedures for the processing on a branching node are equivalent:

(1). Directly join the physical instances of old arcs of branching node by steps as:

(i) join the physical instances of the end nodes of each old arc on the branching node, (ii) project on the key attributes of the head of the arc on this intermediate relation separately.

(2). The branching node can be either a entity node or a relationship node. The query processing on a branching node is depend on whether the branching node is a entity node or a relationship node as:

a. For a branching node of relationship type, (i) decompose the arcs of a branching node into a set of new arcs by the decomposition rule, then (ii) implement the join operation on the end nodes of each new arc, finally project the primary key of each head node.

b. For a branching node of the entity type, (i) decompose the arcs of a branching node into a set of new arcs according to the decomposition rule, then (ii) implement the join operation on the end nodes.

A cyclic subgraph of a *RERQG* that has single branching node can be decomposed to a tree structure by processing the decomposition rule on the branching node. For a cyclic subgraph which has more than one branching node, it can be converted to several "single branching node contained" cyclic subgraphs by duplicating the cyclic subgraph to each branching node.

Let n_i and n_j be two branching nodes in a cyclic subgraph of *RERQG*. If there is a path p_1 from n_i to n_j and a path p_2 from n_j to n_i such that. $p_1 = n_i v_{i0} n_0 v_{01} n_1 \cdots n_j$ and $p_2 = n_j v_{jq} n_q v_{q(q-1)} n_{q-1} \ldots n_i$, then, the paths p_1 and p_2, are called the **compound path.**

An acyclic subgraphs of an *ERQG* can be decomposed to a sequence of local regions that we may employ ER-semijoin to processed it. In case of a cyclic subgraph of a restriction part have multiple branching nodes and merging nodes, the processing of such a cyclic subgraph by converting it to an acyclic structure may be not efficient. For a subgraph of a *RERQG* which is either a cyclic graph with multiple branching nodes and merging nodes or a compound path, all of the main-nodes in the graph can be aggregated to an **aggregation node** which is obtained by joining all the entity nodes and relationship nodes in the subgraph.

DEFINITION 8 : Let G be a subgraph of an *ERQG* which contains a set of main-nodes of relationship types $\{r_1, \ldots, r_i, \ldots, r_w\}$, a set of restricted main-nodes of entity types $\{e_{r1}, \ldots, e_{rj}, \ldots, e_{rp}\}$, and a set of non restricted main-nodes of entity types $\{e_{n1}, \ldots, e_{nk}, \ldots, e_{nq}\}$. Then the **aggregation node** M of this subgraph G is $M = (\bowtie_i r_i) \bowtie (\bowtie_j e_{rj}) \bowtie (\bowtie_{nk} e_{nk})$, where $\bowtie_i r_i = r_1 \bowtie \ldots, r_i, \cdots, \bowtie r_w$; $\bowtie_j e_{rj} = e_{r1} \bowtie \ldots, e_{rj}, \ldots, \bowtie e_{rp}$; $\bowtie_{nk} e_{nk} = e_{n1} \bowtie \ldots, e_{nk}, \ldots \bowtie e_{nq}$.

THEOREM 2 : Let G be a subgraph of an *ERQG* defined as in Definition 8 and *PK* be the primary keys of a branching main-node in G. Then,

$$\pi_{PK} M = \pi_{PK} (\bowtie_i r_i) \bowtie (\bowtie_j e_{rj}) \bowtie (\bowtie_{nk} e_{nk})$$

$= \pi_{PK}(\bowtie_i r_i) \bowtie (\bowtie_j e_{rj})$.

Proof : A subgraph of an *ERQG* should be able to be decomposed into a set of unit graphs. For a "non-restricted main-node" of an entity node in a local region, its semantics can be reduced to the relationship node of the local region [Chen1987b]. In other words, ER-semijoin can be extended to the creation of an aggregation node of any subgraph of an *ERQG*. Since main-nodes of entity type, which is not a restricted main-node may be skipped during the ER-semijoin operation, the Theorem is proved.

Concerning to the *ERQG* (Entity-Relationship Query Graph) representation of a query, the *TERQG* (Target part of *ERQG*) is an undirected subgraph of an *ERQG* and the *RERQG* (Restriction part of *ERQG*) is a directed subgraph of an *ERQG*. To process of the undirected subgraph of a *TERQG*, an aggregation node processing technique may be applied. Entity-Relationship Query Graph) it can be processed as an aggregation node.

CHAPTER 5
AN ERG APPROACH TO THE UNIVERSAL RELATION

1. BASIC ASSUMPTIONS ON THE UNIVERSAL RELATION OF SEMANTIC APPROACH

In order to implement the universal relation by the utility of the *ERG* proper assumptions on this universal relation scheme are necessary. The general assumption of the universal relation is that each attribute in the scheme is globally and uniquely defined.

To obtain the access paths of a query on a universal relation interface, by using the semantics of an *ERG*, certain assumptions on the *ERG* are required. After the general assumptions of a universal relation based on an *ERG* the assumption which make the preprocessing of the query on the conceptual level feasible is made.

1.1. GLOBAL AND UNIQUE ROLES OF ATTRIBUTES

The purpose of the universal relation interface on a database system is to relieve the user from the work of navigating the conceptual level of a relational database system. In other words, on a universal relation interface, the user may directly query on the attributes of a database and he do not have to know the structure of the conceptual level. To allow the user to do the query directly on the attributes of a database the first assumption made on a universal relation scheme is that there exists a set of attributes in a universal relation [Ullm1983]. Since the user does not have to know the conceptual level on a universal relation scheme, the application of an attribute to more than one representation is not allowed. This restriction on a universal relation is based on the second universal relation assumption. The second assumption on a universal relation scheme is that all the attributes in a universal relation scheme are uniquely

defined.

The first assumption of the universal relation asks that all the attributes of a universal relation should be globally defined. The second assumption of the universal scheme is to enforce an attribute should not have more than one representation in a database.

1.2. ASSUMPTION ON THE PHYSICAL LEVEL OF DATABASE SYSTEMS

The **representative instances** $Rep(P)$ of a database D is a collection of all of the physical instances (physical relations) of entity types and relationship types of a database [Sagi1983]. A physical instance r_i is a set of tuples which represents the true value of a relation in the physical database. A physical instance of entity type is a physical instance whose representation in the semantic level is an entity type; a physical instance of relationship type is a physical instance whose representation in the semantic level is a relationship type. For a database D with physical instances $r_1, r_2, ..., r_k$, the representative instances of this database is denoted as $Rep(D) = \{r_1, r_2, \cdots, r_k\}$.

Though the concept of exerting representative instance to represent the physical level of a database is similar to that proposed by Sagiv, the approach to a universal relation using semantic model is quite different from Sagiv's approach to the universal relation. In Sagiv's approach the functional dependency is employed to obtain the access paths of a query; in the semantic approach, a query is mapped onto an *ERG* of a database to obtain a set of access paths represented as an *ERQG*.

A query can be categorized into two types : *Update* and *Retrieval*. The *Update* can be further grouped into *Modification*, *Insertion*, and *Deletion*. The global updating of a database system is discussed in chapter 1.

For the global retrieval of a database, a query is processed by a query processor in the

conceptual level. That is, a query on a universal relation interface can be mapped onto an *ERG* to obtain an *ERQG*. Then, the preprocessing of a query decomposition and optimization can be implemented in the conceptual level on an *ERQG*. To obtain proper access paths on the universal relation interface, we employ semantic model rather than functional dependency as basis for the allocation of access paths of a query. In the next section, the assumptions made on an *ERG* will make the preprocessing of a query on the semantic level feasible.

1.3. ASSUMPTION ON THE SEMANTIC MODEL

To employ the semantic model as the structure of a relational database system, a relational database should be defined by a semantic clear *ERG*. [Chen1987a].

The second assumption on an *ERG* is that for an entity type or a relationship type in an *ERG* there is a unique physical instance corresponding to it, and for a physical instance of a representative instance, there is a unique entity type or relationship type in the semantic model to represent it. That is, a relational database system is well designed on a semantically clear *ERG* and there is a bijective function between entity nodes or relationship nodes of the semantic level and the physical instances of the representative instances.

LEMMA 1 Let r_1 and r_2 be the physical instances of relations R_1 and R_2 on a relation scheme, and $\{PK_1\}$, $\{PK_1, PK_2,..., PK_m\}$ are the sets of primary key of relations R_1 and R_2 respectively. Then, $\Pi_{PK_1}(r_2) \subseteq \Pi_{PK_1}(r_1) \Rightarrow (\Pi_{R_2}(r_1 \bowtie r_2) = r_2) \wedge (\Pi_{R_2}(r_1 \bowtie r_2) \bowtie \Pi_{R_1}(r_1 \bowtie r_2) = r_1 \bowtie r_2)$.

Proof : Let $r_1 \bowtie r_2 = \{\cup_i^m V_{n,i}\}$ and $r_2 = \{\cup_j^n V_{o,j}\}$; where $V_{n,i}$ is a tuple of the physical instance $r_1 \bowtie r_2$ and $\{\cup_i^m V_{n,i}\}$ is the set of m tuples of the physical instance $r_1 \bowtie r_2$; $V_{o,j}$ is a tuple of the physical instance of r_2 and $\{\cup_j^n V_{o,j}\}$ is the set of n tuples of the physical instance of r_2. The implementation of the natural join operator on r_1 and r_2, the

property $\Pi_{R_2}(r_1 \bowtie r_2) \subseteq r_2$ should be true. Thus to prove $\Pi_{R_2}(r_1 \bowtie r_2) = r_2$ we have to prove $r_2 \subseteq \Pi_{R_2}(r_1 \bowtie r_2)$. Since $\Pi_{PK_1}(r_2) \subseteq \Pi_{PK_1}(r_1)$, for each $V_{o,j}$ of r_2, there exist a tuple of r_1 whose projection on PK_1 is equal to the projection of $V_{o,j}$ on PK_1. Thus for each $V_{o,j}$ in r_2, there exist a tuple $V_{n,i}$ corresponding to it. That is, $r_2 \subseteq (r_1 \bowtie r_2)$. The equality $(r_1 \bowtie r_2) = r_2$ is proved. The next step we want to prove $\Pi_{R_2}(r_1 \bowtie r_2) \bowtie \Pi_{R_1}(r_1 \bowtie r_2) = r_1 \bowtie r_2$. As $(r_1 \bowtie r_2) = r_2$, we get the equal cardinality of r_2 and $r_1 \bowtie r_2$, that is $m = n$. In other words, the cardinality of $\Pi_{R_1}(r_1 \bowtie r_2)$ is equal to the cardinality of $r_1 \bowtie r_2$ and the cardinality of r_2. Thus for $\Pi_{PK_1}(\Pi_{R_1}(r_1 \bowtie r_2)) = \Pi_{PK_1}(r_2)$, $r_2 \bowtie r_1 = r_2 \bowtie \Pi_{R_1}(r_1 \bowtie r_2) = \Pi_{R_2}(r_1 \bowtie r_2) \bowtie \Pi_{R_1}(r_1 \bowtie r_2)$.

In an *ERG*, a unit graph represents a local region which contains a pair of entity types and a relationship that connects to these entity types. A local region $[E_{i-1}, R_i, E_{i+1}]$ may have the property of the local integrity iff [Chen1987a]

$$\pi_{a_{(i+1)k}}(R_i) \subseteq \pi_{a_{(i+1)k}}(E_{i+1})$$
$$\pi_{a_{(i-1)k}}(R_i) \subseteq \pi_{a_{(i+1)k}}(E_{i-1}). \qquad (1)$$

Since a local region has the property of the local integrity, the physical instances of a local region has the property of the join dependency which is proved in the Lemma 2.

LEMMA 2 Let a relation scheme R whose semantic structure be represented by an *ERG*, and $R = \{L_i \mid L_i = [E_{i-1}, R_i\ E_{i+1}]\}$. Then, $\forall(L_i)\ ((L_i \in R)\ ,\ (L_i = [E_p, R_q, E_s] \Rightarrow r_L = r_{E_p} \bowtie r_{R_q} \bowtie r_{E_s} = \Pi_{E_p}(r_L) \bowtie \Pi_{R_q}(r_L) \bowtie \Pi_{E_s}(r_L))$; where r_{E_p}, r_{R_q}, and r_{E_s} are the physical instances of E_p, R_q, and E_s respectively.

Proof : Let PK_p be the set of the primary key of E_p and PK_q be the set of the primary key of E_s. For the integration rule of a local region of an *ERG*, the physical instances of local region L has the property of $(\Pi_{PK_p}(r_{R_q}) \subseteq \Pi_{PK_p}(r_{E_p})) \wedge (\Pi_{PK_s}(r_{R_q}) \subseteq \Pi_{PK_s}(r_{E_s}))$. From the LEMMA 1, $(\Pi_{R_q}(r_{E_p} \bowtie r_{R_q}) = r_{R_q}) \wedge (\Pi_{E_p}(r_{E_p} \bowtie r_{R_q}) \bowtie \Pi_{R_q}(r_{E_p} \bowtie r_{R_q})) = r_{E_p} \bowtie r_{R_q}$, and $(\Pi_{R_q}(r_{E_s} \bowtie r_{R_q}) = r_{R_q}) \wedge (\Pi_{E_s}(r_{E_s} \bowtie r_{R_q}) \bowtie \Pi_{R_q}(r_{E_s} \bowtie r_{R_q}) = r_{E_s} \bowtie r_{R_q})$.

Thus $\Pi_{E_p}(r_{E_p} \bowtie r_{R_q}) \bowtie \Pi_{R_q}(r_{E_p} \bowtie r_{R_q}) \bowtie r_{R_q}$, and $\Pi_{E_s}(r_{E_s} \bowtie r_{R_q}) \bowtie \Pi_{R_q}(r_{E_s} \bowtie r_{R_q}) = (r_{E_p} \bowtie r_{R_q}) \bowtie (r_{E_s} \bowtie r_{R_q}) = r_{E_p} \bowtie r_{E_s} \bowtie r_{R_q} = r_L$. From the integration rule, we amy also get $\Pi_{E_s}(r_{E_s} \bowtie r_{R_q}) = \Pi_{E_s}(r_{E_s} \bowtie r_{R_q} \bowtie E_p)$, $\Pi_{E_p}(r_{E_p} \bowtie r_{R_q}) = \Pi_{E_q}(r_{E_s} \bowtie r_{R_q} \bowtie E_p)$, and $\Pi_{R_q}(r_{E_p} \bowtie r_{R_q}) = \Pi_{R_q}(r_{E_s} \bowtie r_{R_q}) = \Pi_{R_q}(r_{E_s} \bowtie r_{R_q} \bowtie E_p)$. Then $r_L = r_{E_p} \bowtie r_{R_q} \bowtie r_{E_s} = \Pi_{E_p}(r_L) \bowtie \Pi_{R_q}(r_L) \bowtie \Pi_{E_s}(r_L)$; where r_{E_p}, r_{R_q}, and r_{E_s} are the physical instances of E_p, R_q, and E_s respectively.

From Lemma 2, the join dependency property for the physical instances of a local region in an *ERG* is proved. Such a join dependency property of the physical instances of a local region can be extended to the *ERG*.

THEOREM 1 Let R_{ER} be a relation scheme whose structure can be represented in an *ERG* as $R_{ER} = \{L_1, ..., L_i\}$ and $R_{ER} = \{E_1, \ldots, E_i, \ldots, E_m, R_1, \ldots, R_j, \ldots, R_n\}$; where E_i and R_j are entity type or relationship type respectively. And let r be the physical instance such that $r = \bowtie_{i,j}^{m,n}(r_{E_i}(r) \bowtie r_{R_j}(r))$; $E_i, R_j \in R_{ER}$. Then, $r = \bowtie_{i,j}^{m,n}(\Pi_{E_i}(r) \bowtie \Pi_{R_j}(r)) = \Pi_{E_1}(r) \bowtie \Pi_{E_2}(r) \ldots \bowtie \Pi_{E_m}(r) \bowtie \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \ldots \bowtie \Pi_{R_n}(r)$.

Proof :

(i) For an *ERG* with single entity type, the proof of the Theorem is trivial type; for an *ERG* with only one local region, the Theorem is proved by the LEMMA 2.

(ii). For $R_{ER} = \{L_1, ..., L_k\}$, assuming $r(k) = \bowtie_{i,j}^{m,n}(\Pi_{E_i}(r) \bowtie \Pi_{R_j}(r)) = \Pi_{E_1}(r) \bowtie \Pi_{E_2}(r) \ldots \bowtie \Pi_{E_m}(r) \bowtie \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \ldots \bowtie \Pi_{R_n}(r)$.

(iii). For $R_{ER} = \{L_1, ..., L_k, L_{k+1}\}$, let $L_{k+1} = [E_g, R_h, E_l]$.

Case 1. If $(E_g \in R_{ER}) \wedge (R_h \in R_{ER}) \wedge (E_l \in R_{ER})$, Then $r(k+1) = r(k) = \bowtie_{i,j}^{m,n}(\Pi_{E_i}(r) \bowtie \Pi_{R_j}(r)) = \Pi_{E_1}(r) \bowtie \Pi_{E_2}(r) \ldots \bowtie \Pi_{E_m}(r) \bowtie \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \ldots \bowtie \Pi_{R_n}(r)$. Then, by the inductive hypothesis, the Theorem is proved.

Case 2. If $(E_g \in R_{ER}) \wedge (R_h \notin R_{ER}) \wedge (E_l \in R_{ER})$, Then $r(k+1) = r(k) \bowtie r_{R_h} = (\bowtie_{i,j}^{m,n}(\Pi_{E_i}(r) \bowtie \Pi_{R_j}(r))) \bowtie r_{R_h} = ((\Pi_{E_1} r(k) \bowtie r_{R_h}) (\bowtie \Pi_{E_2} r(k) \bowtie r_{R_h}) ,..., \bowtie (\Pi_{E_m} r(k) \bowtie r_{R_h}) \bowtie (\Pi_{R_1} r(k) \bowtie r_{R_h}) \bowtie (\Pi_{R_2} r(k) \bowtie r_{R_h}),..., \bowtie (\Pi_{R_n} r(k) \bowtie r_{R_h})) \bowtie r_{R_{h(l)}} = (\Pi_{E_1} r(k+1) \bowtie \Pi_{E_2} r(k+1),...,$

$\bowtie \Pi_{E_m} r(k+1)) \bowtie \Pi_{R_1} r(k+1) \bowtie \Pi_{R_2} r(k+1)),..., \bowtie \Pi_{R_n} r(k)) \bowtie r_{R_{h(k)}} = (\Pi_{E_1} r(k+1) \bowtie \Pi_{E_2} r(k+1),...,$ $\bowtie \Pi_{E_m} r(k+1)) \bowtie \Pi_{R_1} r(k+1) \bowtie \Pi_{R_2} r(k+1)),..., \bowtie \Pi_{R_n} r(k)) \bowtie \Pi_{R_{r(k+1)}}$ Then, by inductive hypothesis, the Theorem is proved. Case 3. If $(E_g \in R_{ER}) \wedge (R_h \notin R_{ER}) \wedge (E_l \notin R_{ER})$, Then $r(k+1) = r(k) \bowtie r_{R_h} \bowtie r_{R_l}$. From case 2, $r(k+1) = r(k) \bowtie r_{R_h} (\bowtie_{i,j}^{m,n} (\Pi_{E_i}(r) \bowtie \Pi_{R_j}(r)))$ $\bowtie r_{R_h} = (\Pi_{E_1}(r(k+1)) \bowtie \Pi_{E_2}(r(k+1)) \ ... \ \bowtie \Pi_{E_m}(r(k+1)) \bowtie \Pi_{R_1}(r(k+1)) \bowtie \Pi_{R_2}(r(k+1)) \ ...$ $\bowtie \Pi_{R_n}(r(k+1))) \bowtie \Pi_{R_{h(r(k+1))}}$ By viewing $r(k) \bowtie r_{R_h}$ as an intermediate physical instance of an entity type Then, by Lemma 2, we may get $r(k+1) = r(k) \bowtie r_{R_h} \bowtie r_{R_l} = (\bowtie_{i,j}^{m,n}$ $(\Pi_{E_i}(r(k)) \bowtie \Pi_{R_j}(r(k)))) \bowtie r_{R_h} \bowtie r_{R_l} = (\Pi_{E_1}(r(k+1)) \bowtie \Pi_{E_2}(r(k+1)) \ ... \ \bowtie \Pi_{E_m}(r(k+1)) \bowtie$ $\Pi_{R_1}(r(k+1)) \bowtie \Pi_{R_2}(r(k+1)) \ ... \ \bowtie \Pi_{R_n}(r(k+1))) \bowtie \Pi_{R_{h(r(k+1))}} \bowtie \Pi R_l(r(k+1))$. Then, by inductive hypothesis,

Case 4. If $(E_g \notin R_{ER}) \wedge (R_h \in R_{ER}) \wedge (E_l \notin R_{ER})$, the Theorem can be proved with the same reason as Case 3.

Thus the Theorem is proved.

Ullman proved that there exists a relation for a universal set of attributes of a universal relation if and only if this relation satisfies the join dependency on the physical instances of the universal relation. [Ullm1982]. From Theorem 1, for a relational database whose semantic structure can be represented by a semantically clear *ERG*, the join dependency is always true. That is, the assumption that there exists a universal set of attributes on a universal relation is always true.

2. QUERY DECOMPOSITION AND ITS ASSOCIATED ACCESS REGIONS

The universal relation assumption based on an *ERG* requires that for a query on a universal relation, there is one semantic extended region corresponding to it. By representing a semantic structure of a relational database in an *ERG*, a query on the relational database can be represented by an *ERQG* which contains a subgraph of an *ERG* with logics of the query. An *ERQG* can be grouped into two types. That is, each valid query can be grouped into a target part and a restriction part. The target part

pertains to the attributes of the information to be retrieved. Thus for a valid query, the target part should not be empty. The restriction part restricts the domain of the attributes to be retrieved. An empty restriction part always means that the domain of the target part is not restricted.

2.1. THE SEMANTIC EXTENDED REGION OF A QUERY ON A UNIVERSAL RELATION

A query on a universal relation based on the semantics of an *ERG* yields **lossless information** if the access paths of the query cover all of the possible attributes of the query. For two different main_nodes in an *ERG*, there may exist more than one path connecting them. The lossless information to represent the relationship between two main_nodes in an *ERG* is obtained by the union of all the paths that connect them. Thus we define the subgraph of two separate main_nodes in a query as the set of paths that connect them. [Chen1987b].

The target part and restriction part of a natural join query of a universal relation is defined in the previous chapter.

DEFINITION 1 A minimal semantic extended target part of a query is the minimal set of the target part of the query which represents lossless information.

We define the target part of the qeury to cover all the loseless information. By deleting the redundant main-nodes as in the Example 1, the target part of a query can be represented by a minimal semantic extended target part.

Example 1: The query Q on the database *COMPANY* of fig. 1 is :

Q retrieve :dept_id, person_id

where part_id = "part00001532".

To represent lossless information of Q, the target part can be [*DEPT*, *EMPLOY*, *PERSON*] or [*EMPLOY*, *PERSON*]. The minimal extended target part is

[*EMPLOY*, *PERSON*].

DEFINITION 2 A minimal semantic extended restriction part of a query is the minimal set of the restriction part of the query which may specify the domain of lossless information.

The minimal semantic extended restriction part of a query can only be obtained after the allocation of the minimal semantic extended target part. We define the restriction part of the qeury to cover all the loseless information and which do not cover the redundant paths, thus it is a minimal semantic extended restriction part.

Example 2: The query Q in example 1 has two equivalent semantic extended restriction parts represented as [*PERSON*, [*WORK ON*, *MANAGE*], *PROJECT*, *USE*, *PART*] and [*PERSON*, [*WORK ON*, *MANAGE*], *PROJECT*, *USE*]. The minimal semantic extended restriction part is [*PERSON*, [*WORK ON*, *MANAGE*], *PROJECT*, *USE*].

DEFINITION 3 A semantic extended region *SR* of a query Q is the union of minimal semantic extended target part and minimal semantic extended restriction part of Q. That is $SR = T_{MIN} \cup R_{MIN}$, where T_{MIN} is the minimal semantic extended target part of Q, and R_{MIN} is the minimal semantic extended restriction part of Q.

The minimal semantic extended target part (T_{MIN}) of a query contains the minimal set of local regions and includes all of the attributes to be retrieved such that $T_{MIN} = \{L_i \mid (L_i \subseteq SR) \wedge ((\forall j)\ ((j \neq i) \wedge (L_i \neq L_j)) \rightarrow (\exists m)\ ((m \neq i) \wedge (L_i \neq L_m) \wedge (L_m \cap L_i \neq \varnothing))\} \wedge T_{MIN} \neq \varnothing$. The minimal extended restriction part of the query contains the minimal set of local regions that connect the target part and the attributes whose domain is specified in the query such that $R_{MIN} = \{L_i \mid L_i \subseteq SR\}$. $SR = (T_{MIN} \cup R_{MIN}) \wedge (R_{MIN} \neq \varnothing \rightarrow (TG_{MIN} \oplus RG_{MIN} = T_{MIN} \cup R_{MIN}))$; where TG is the graph representation of the target part, RG is the graph representation of the restriction part, and $\oplus$ is the ring sum operation on the *ERQG* [Chen1987b].

For a valid query, the semantic extended region can be decomposed into a query part and a restriction part. The necessary condition for a valid query is that the target part of the query must not be empty. Then, each query can be decomposed into two sets of local regions. One set of local regions is the set of local regions of the target part, the other set of local regions is the set of the local regions of the restriction part. For a nonempty restriction part, the intersection of the restriction part and the target part should not be empty. By representing a query by an *ERQG*, ($TG_{MIN} \oplus RG_{MIN} = T_{MIN} \cup R_{MIN}$) defines that the intersection of the target part and the restriction part of a query contains a set of main-nodes such that each isolated sub-graph of the restriction part does not have two main-nodes in this set.

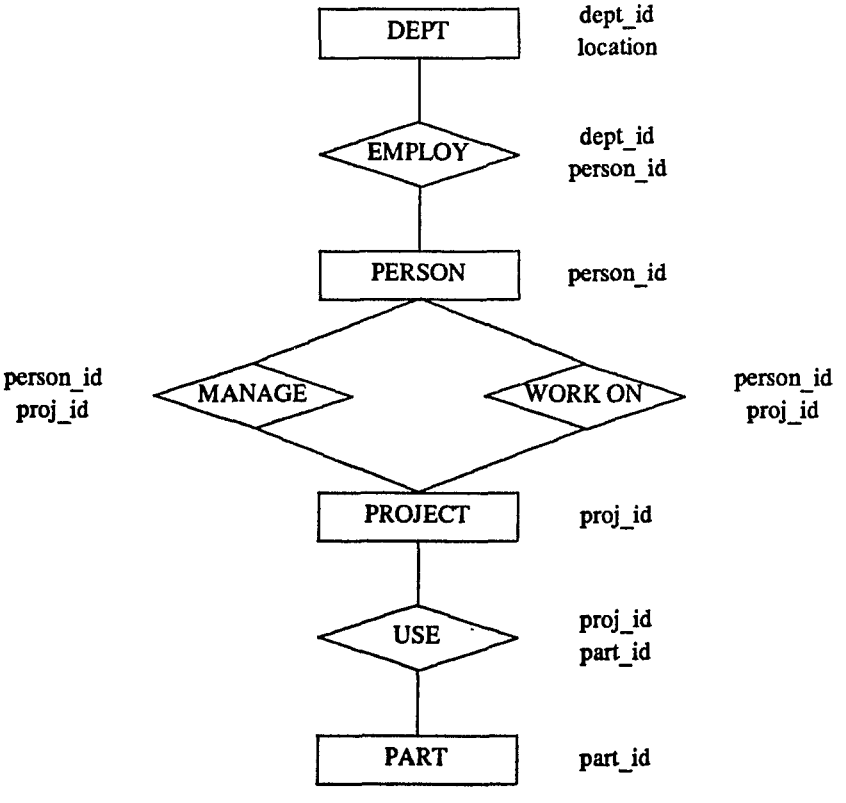


Fig. 1 The Entity-Relationship Graph of the relational database *COMPANY*.

Example 3 : Consider a relational database *COMPANY* whose semantic model is represented in the Entity-Relationship Model as in fig. 1. The query "Find the dept_id

of the *DEPT* that *EMPLOY*s the *PERSON* who *WORK*s *ON* or *MANAGE*s the *PROJECT* that *USE*s the *PART* with part_id ='part000001532'." specifies that the target entity part is *DEPT* and the restricting object is part_id which is the attribute of the entity type *PART*. The query specifies the semantics regions which extend from the local region [*DEPT*, *EMPLOY*, *PERSON*] to the local region [*PROJECT*, *USE*, *PART*]. The semantic extended region of the query includes the local regions [*PERSON*, *WORK ON*, *PROJECT*] and [*PERSON*, *MANAGE*, *PROJECT*], which connect [*DEPT*, *EMPLOY*, *PERSON*] and [*PROJECT*, *USE*, *PART*]. The same query can be represented in a query of the universal relation as "retrieve dept_id where part_id = part000001532' ". These two equivalent queries have the same extended region. From example 1 and example 2, the minimal semantic extended region of these two queries is [*EMPLOY*, *PERSON*, [*WORK ON*, *MANAGE*], *PROJECT*, *USE*].

The allocation of the semantic extended region of a query on a universal relation interface is equivalent to the searching of the access paths to represent the query. Such semantic extended region must contain the lossless information of the query.

Example 4: For the query in example 3, there are two paths between main_nodes *DEPT* and *PART* : [*DEPT*, *EMPLOY*, *PERSON*, *WORK ON*, *PROJECT*, *USE*, *PART*] and [*DEPT*, *EMPLOY*, *PERSON*, *MANAGE*, *PROJECT*, *USE*, *PART*]. The relationship between these two access paths may be one of the following : ***Union***, ***Intersection***, ***Difference***, ***Cartesian Product***, ***Join***, ***Negation***, ***Exclusive-OR***, ***Division***. To represent a lossless query, we take the union of these two access paths and get the result as in Example 3.

For the query based on the universal relation as in the second query of example 3, the allocation of the semantic extended region of the query is necessary. In general, the semantic extended region of a query is not unique. But for a query, there is a unique minimal semantic extended region corresponding to it. In the next section, we will discuss the uniqueness property of the minimal semantic extended region.

2.2. UNIQUE PROPERTY OF A QUERY AND ITS ERQG ON A UNIVERSAL RELATION

For the interpretation of a query of a universal relation approach on an *ERG*, the allocation of different access paths may yield different results. Thus we will prove that for each query on a universal relation interface, there is a unique set of access paths that can be allocated.

DEFINITION 4 Let R be a *RKER* with an *ERQG* G. For any two main_nodes M_1 and M_2 in G, P is a set of access paths between M_1 and M_2 such that $P = \{P_1, ..., P_i, ..., P_n\}$. Let I_l be a relation which is obtained by joining all of the physical instances of the path P_i, $P_i \subseteq$ P. Then, G is a **non-redundant ER** iff $(\forall P_i)(\forall P_j)$ $((P_i \subseteq P)(P_j \subseteq P)$ $(P_i \neq P_j)(P_i - P_j \neq \varnothing)$ $(P_j - P_i \neq \varnothing) \rightarrow (I_i \neq I_j))$

LEMMA 3 The minimal semantic extended target part of a query on a universal relation of a non-redundant ER is unique.

Proof : Assuming there are two different target parts T_1 and T_2 for a query Q, that is $T_1 \neq T_2$.

(i). If $T_1 \subset T_2$, then T_2 is not a minimal semantic extended target part, the result is contradictory to the assumption that T_2 is the minimal semantic extended target part.

(ii). If $T_2 \subset T_1$, then T_1 is not a minimal semantic extended target part, the result is contradictory to the assumption that T_1 is the minimal semantic extended target part.

(iii). If $(T_2 \not\subset T_1) \wedge (T_1 \not\subset T_2)$, then either T_1 or T_2 is redundant. Then, the assumption is contradictory to the fact that the relational database is non-redundant ER.

Thus the minimal semantic extended target part of a query on a universal relation of a non-redundant ER is unique.

LEMMA 4 The minimal semantic extended restriction part of a query on a universal relation of a non-redundant ER is unique.

Proof : Assuming there are two different restriction parts R_1 and R_2 for a query Q, that is $R_1 \neq R_2$.

(i). If $R_1 \subset R_2$, then R_2 is not a minimal semantic extended restriction part, the result is contradictory to the assumption.

(ii). If $R_2 \subset R_1$, then R_1 is not a minimal semantic extended restriction part the result is contradictory to the assumption.

(iii). If $(R_2 \not\subset R_1) \wedge (R_1 \not\subset R_2)$, then either R_1 or R_2 is redundant. Then, the assumption is contradictory to the fact that the relational database is a non-redundant ER.

Then, contradicting results are obtained from the assumption that there are two different restriction parts in a valid query. Thus the uniqueness property of a minimal semantic extended restriction part of a query on a universal relation of a non-redundant ER is proved.

THEOREM 2 The minimal semantic extended region of a query on a universal relation of a non-redundant ER is unique.

Proof : The semantic extended region of a query on a universal relation of a non-redundant ER is the union of its target part and the restriction part of the query. Let TG be the target part subgraph of the query and RG be the restriction part subgraph of the query, and G be the query graph of the query. Then, $TG \oplus RG = TG \cup RG$. Thus the uniqueness property of a semantic extended region can be obtained from the uniqueness properties of its target part and restriction part.

For a valid query, there exists just one semantic extended region to represent it. A minimal semantic extended region can be further decomposed into a semantic extended target part and semantic extended restriction part. The user may navigate the relational database on any of the relational query interfaces such as SQL, QUEL,

QBE, and etc. In a universal relation interface, the system may automatically allocate a minimal semantic extended region on the *ERG* of a database system. The implementation rules for the allocation of the semantic extended region of a query on a universal relation are discussed in the next section.

3. PROPAGATION OF ACCESS PATHS VIA RELATIONSHIP

In a *RDKER* (A Relational Database with Knowledge of *ERG*), the *ERG* represents the semantic structure of the database. In the universal relation model on a *RDKER*, the relationship in a relationship-based surrounding region plays the role of connecting the access paths. The first step of the allocation of the access paths of a query on a universal relation is to obtain the restriction main nodes and the target main nodes of the query [Chen1987b]. The second step of the allocation of access paths is to allocate the paths that connect from restriction main nodes to the target main nodes. This technique of the allocation of the access paths of a query on a universal relation is quite different from that used by the Sagiv [Sagi1983].

A relationship based surrounding region contains the identification or the primary keys of the entity types that are in the region. Such a "connected entity types identification including" property of a relationship type makes the allocation of the access paths between target main nodes and restriction main nodes available.

Example 5 : A relationship-based surrounding region is $\{R_o, E_1, E_2, E_3\}$, where the primary keys of the entity types of E_1, E_2, and E_3 are $\{PK_{11}, PK_{12}, ..., PK_{19}\}$, $\{PK_{21}, PK_{22}, ..., PK_{29}\}$, and $\{PK_{31}, PK_{32}, ..., PK_{39}\}$ respectively. As the primary keys of an entity type or a relationship type may have more than one attribute, the use of identification representation of the entity types makes the allocation of access paths more convenient during the process of searching. The representation of this relationship-based surrounding region in the primary keys of the entity type is :

relationship-surround_region(R#1, PK_1, PK_2, PK_3)

$E(pk_{11}, pk_{12}, ..., pk_{19}, ...)$

$E(pk_{21}, pk_{22}, ..., pk_{29}, ...)$

$E(pk_{31}, pk_{32}, ..., pk_{39}, ...)$, where $PK_1 = [pk_{11}, pk_{12}, ..., pk_{19}, ...]$,

$PK_2 = [pk_{21}, pk_{22}, ..., pk_{29}, ...]$, and $PK_3 = [pk_{31}, pk_{32}, ..., pk_{39}, ...]$.

The representation of this relationship-based surrounding region by the internal identification of the entity types in the system is :

relationship-surround_region(R#1, E_ID_1, E_ID_2, E_ID_3)

$E(E\_ID_1, ...)$

$E(E\_ID_2, ...)$

$E(E\_ID_3, ...)$.

Thus the representation of an entity type or a relationship type by the internal identification representation of the system will simplify the work of allocating the access paths of two main nodes.

For two main-nodes X and Y and an n-ary relationship based surrounding region that connects X any Y, a local region which contains this n-ary relationship and which connects X and Y is called a **projected local region**. A projected local region is defined as the projection of a local region on a relationship-based surrounding region. A projected local region is the full semantic reducer of an n-ary relationship based surrounding region. That is, for an n-ary relationship in an access path of a query, only the projected local region of this n-ary relationship-based surrounding region have to be used for the processing of a query.

The searching of the connecting paths of a target main-node and a restriction main node in an *ERG* can be treated as the searching of the sets of unit graphs (the graph representation of a local region) that connects these two main nodes. The searching algorithm can be illustrated as :

a(X,Y) -> relationship_surrounding_region(R#1, X, Y, Z, ...);

where X, Y, and Z are the identification of the entity and R is the relationship that which has the relationship-based surrounding region of $\{R, X, Y, Z, \cdots\}$.

```
paths(X,Y,L1) -> findall(L,go(X,Y,L),L1).
go(Start,Dest,Route) -> go0(Start,Dest,[],R),rev(R,Route).
go0(X,X,T,[X|T]).
go0(Place,Y,T,R) -> legalnode(Place,T,Next), go0(Next,Y,[Place|T],R).
legalnode(X,Trail,Y) -> (a(X,Y) ; a(Y,X)),not(member(Y,Trail)).
a(X,Y) -> relationship_surrounding_region(R#1, X, Y, Z, ...);
```

THEOREM 3 Let a and b be two main-nodes in an *ERG*, there exists at least a path $P = u_1, ..., u_i, ..., u_k$ such that P connects a and b, where u_i is a unit graph in the *ERG*.

Proof : If the *ERG* is a single entity type, then $a = b$, the Lemma is proved.

For an *ERG* which is not a single entity type, let u_1 and u_2 be the unit graphs that have a and b as their main-nodes respectively. Then, if $u_a \cap u_b \neq \varnothing$, then the Theorem is proved. Now, we want to prove when $u_a \cap u_b = \varnothing$, there exist a path which connects u_a and u_b. Let $G = \{u_i \mid u_i$ is the unit graph of the *ERG*$\}$. From the Theorem 1 of the "*ERQG* and ERQT on a Relational Database System" [Chen1987b], $G = (\{u_1, ..., u_i\})$, and $(\forall u_i \in G)\ ((\forall u_j \in G)(u_i \cap u_j = \varnothing) \rightarrow (\exists u_k \in G)(u_k \neq u_i) \wedge (u_k \cap u_i \notin \varnothing))$. Let $u_a = \{u_1\}$ and $u_b = \{u_2\}$. Since $u_1 \cap u_2 = \varnothing$, then $(\exists u_p \in G\ (u_p \cap \{u_1\} \neq \varnothing))$. Let $u'_1 = u_1 \cup u_p$. If $u'_1 \cap u_2 = \varnothing$, then the Theorem is proved. If $u'_1 \cap u_2 \neq \varnothing$, by the same reason we can add a unit graph into u'_1 to test the intersection of this new graph and u_2. The worst case for the allocation of the connecting path from a to b is that we get $u'' = G - u_2 - u'_1$ and $u'_1 \cap u_2 \neq \varnothing$. Since G is a connected graph, then $(u_1 \cup u'') \cap u_2 \neq \varnothing$, thus the Theorem is proved.

Thus for any two main-nodes in the *ERG*, we can find a set of directed access paths starting from one main-node and ending at the other main-node. Let two main-

nodes be specified as a target main-node and a restriction main-node, then we can get a set of paths such that each path starts from the restriction node and ends at the target main node. In an *ERG*, the rule for the propagation of access paths from a restriction main node to a target main node is called the **access propagation rule.**

THEOREM 4 An *ERQG* contains m target main-nodes and n restriction nodes. If $m, n \neq 0$, then the maximum sets of paths that connects the target main nodes and restriction main-nodes is $m \times n$.

Proof : From Theorem 3 that for each pair of a target node and a restriction node, there is a set of paths that connect them. Thus for each target main-node there are n sets of access paths that connect it to the restriction main-nodes. Since there are n restriction nodes, we can find $m \times n$ pairs of access paths connecting these target main-nodes and restriction main-nodes.

The access paths that connects the target main-nodes and the restriction main-nodes of a query can always be reduced according to the decomposition rule and merging rule of an *ERQG* [Chen1987b]. Any access path of an *ERQG* which contains more than two target main-nodes is called a **redundant access path.** To process an *ERQG* efficiently, the redundant access can be skipped during the allocation of access paths of an *ERQG*. The set of access paths of a target main-node and a restriction main-node of an *ERQG* can be merged by a merging algorithm. The following merging algorithm illustrates an example to merge those access paths that have the same end nodes and that have common portions :

(i). Merging two paths into a combined path: e.g. [1,2,3,d,f,g,p,q,r] and [1,2,3,v,p,q,r] will be merged into [1,2,3,[[f,g],[v]],p,q,r] :

```
merge(A,B,C) -> merge_head(A,B,H,T1,T2), merge_tail(T1,T2,H1,H2,T),
rev(T,Ts), append(H,[[H1|[H2]]],S), append(S,Ts,C).
```

(ii) Merging paths with the same beginning segments:

merge_head([X|Y1],[X|Y2],[X|H0],P,Q) -> merge_head(Y1,Y2,H0,P,Q).

merge_head(U,V,[],U,V).

(iii). Merging paths with the same ending segments of arcs :

merge_tail(T1,T2,P,Q,T) -> abstract_tail(S,T1,H1),

abstract_tail(S,T2,H2),!,merge_tail(H1,H2,P,Q,TT),T=[S|TT].

merge_tail(X,Y,X,Y,[]).

The Theorem 3 proved that for a target main-node and a restriction main-node there exists a set of access paths that connect them. An access path that connects two main-nodes can be represented by a set of unit graphs [Chen1987b]. The Theorem 4 proved that for a query contains m target main-nodes and n restriction main-nodes, there are $m \times n$ sets of access paths that connect them. In other words, the allocation of the access paths of a query on universal relation interface is to find the sets of access paths that connect the restriction main_nodes and the target main_nodes. To reduce the redundant access paths, the optimizing rule can be built in the searching algorithm. Thus with these basis, an optimizing rule for the allocation of the sets of access paths of the *ERQG* of a query on a universal relation is shown in the following procedures:

(1). Allocate the target subnodes and the restriction subnodes of a query on the *ERG*.

(2). Link each subnode to the main-node such that the main-node and the subnode are the end nodes of a subarc.

(3). Group the main-nodes into a set of the target main-nodes and a set of the restriction main-nodes.

(4). Allocate the target part of the *ERQG* of the query by the application of propagation rule on the target main-nodes collected from step(3). region.

(5). Construct the undirected subgraphs from the restriction main_nodes to the target main-nodes of the target part.

(6). Merge the access paths according to the merging algorithm.

(7). Lable the direction of subgraphs from the restriction node to the target node.

(8). Check the in_degree and out_degree of each main-nodes in the access paths such that the cyclic subgraphs of the *ERQG* can be allocated.

(9). Convert the cyclic subgraph of the restriction part of the *ERQG* to the tree structure.

(10). Employ ER-semijoin to process the query represented in the *ERQG* from the step (9).

A minimal semantic extended *ERQG* represents a well optimized semantic representation of an *ERQG*. Since the time complexity of the minimization of the access paths of a query is much less than the time complexity of the processing of the query, the allocation of a minimal query is one of the important optimizing technique for the processing of a query on a *RDKER*.

In the following example, the allocation of the *ERQG* of a query on a universal relation is illustrated.

Example 6 : The query Q is a query on a universal relation interface for the relational database UNIVERSITY shown in fig. 2.

Q: retrieve c_name, u_name
where s.s# = '456789123'

The following procedures illustrate the procedures to obtain the *ERQG* of the query Q from the *ERG* of UNIVERSITY.

(1). The first step to obtain the *ERQG* of the Q is to collect the set of the target attributes and the restriction attributes. From the query, we get the set of target attributes {c_name, u_name} and the set of restriction attributes {s.s#}. By mapping these sets of target attributes and restriction attributes onto *ERG*, we get the sets

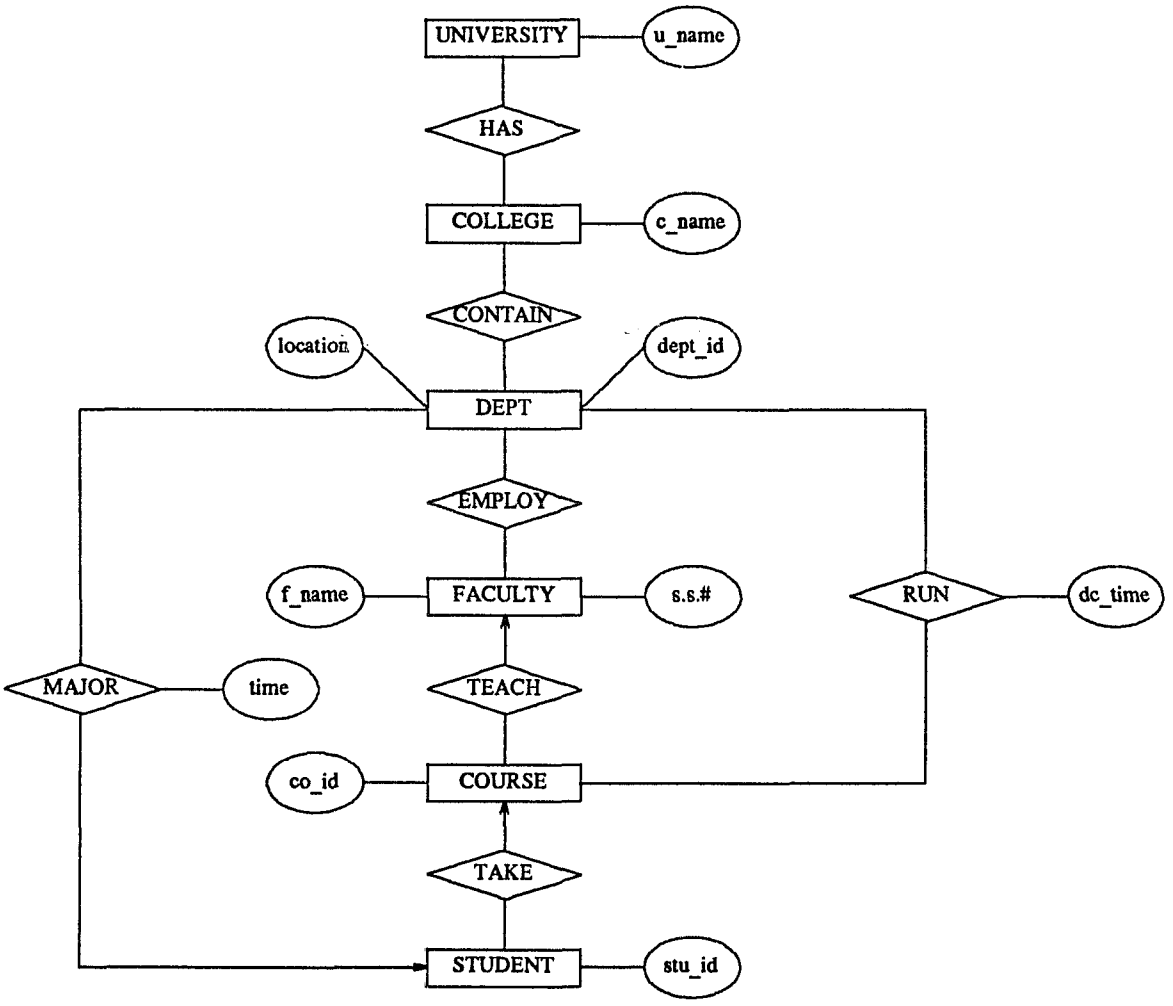


Fig. 2 The *ERG* of the relational database UNIVERSITY.

of target subnodes and restriction subnodes respectively. Then, (i) by linking the u_name to the main-node through subarc $d_{u,U}$, the target main-node UNIVERSITY can be obtained; (ii) by linking the c_name to the main_node through the subarc $d_{c,C}$, the target main-node COLLEGE can be obtained; (iii) by linking the s.s# to the main-node through the subarc $d_{s,F}$, the restriction main-node FACULTY can be obtained.

(2). The target main-nodes are grouped into the set as {UNIVERSITY, COLLEGE}; and the restriction main-nodes are grouped into the set as {FACULTY}.

(3). By propagating the linkage of the target part via the relationship-base surrounding region, we get a new set of target main-nodes as {UNIVERSITY, HAS, COLLEGE} and we also get a set of main-arcs as $\{ARC_{U,H}, ARC_{H,C}\}$.

(4). For these three pairs of main-nodes paths < UNIVERSITY, FACULTY>, <HAS, FACULTY>, and <COLLEGE, FACULTY>, allocate the access paths that connect them. That is, each set of access paths is obtained by applying the searching algorithm on a target main-node and a restriction main-node. The sets of access paths of <UNIVERSITY, FACULTY> and <COLLEGE, FACULTY> are redundant paths. Thus only the set of access paths of <COLLEGE, FACULTY> is not redundant. Thus we get the paths as {[*COLLEGE* $ARC_{C,C}$ *CONTAIN* $ARC_{D,EE}$ *MPLOY* $ARC_{E,F}$ *FACULTY*], [*COLLEGE* $ARC_{C,C}$ *CONTAIN DEPT* $ARC_{D,R}$ *RUN* $ARC_{R,C}$ *COURSE* $ARC_{T,C}$ *TEACH* $ARC_{F,T}$ *FACULTY*], [*COLLEGE* $ARC_{C,C}$ *CONTAIN* $ARC_{C,D}$ *DEPT* $ARC_{M,D}$ *MAJOR* $ARC_{M,S}$ *STUDENT* $ARC_{T,S}$ *TAKE* $ARC_{C,T}$ *TEACH* $ARC_{F,T}$ *FACULTY*]}.

(5). Merging access paths : The set of paths between *COLLEGE* and *FACULTY* can be merged into the following path :

{*COLLEGE* $ARC_{C,C}$ *CONTAIN* $ARC_{C,D}$ *DEPT* [[$ARC_{D,EE}$ *MPLOY FACULTY*] ∨ [[[$ARC_{D,R}$ *RUN* $ARC_{R,C}$] ∨ [$ARC_{M,D}$ *MAJOR* $ARC_{M,S}$ *STUDENT* $ARC_{T,S}$ *TAKE*]] $ARC_{T,C}$ *TEACH* $ARC_{F,T}$]] *FACULTY*}.

(6). The direction of the paths is from the restriction main-node to the target main-node. Thus the main-arcs of the query is directed by the direction from the *FACULTY* (restriction main-node) to the *COLLEGE* (target main-node).

(7). The out-degree of the *COURSE* (main-node) is 2 and the in-degree of the *DEPT* is 2. Thus *COURSE* is a branching node with two branching arcs $ARC_{R,C}$ and $ARC_{T,C}$; *DEPT* is a merging node with two merging arcs $ARC_{M,D}$ and $ARC_{D,E}$.

CHAPTER 6
ERG BASED RELATIONAL DATABASE SYSTEM

1. ONE-PHASE AND TWO-PHASE DATABASE SYSTEMS OF THE RDKER

An *ERG* can be used either on a one-phase interface or on a two-phase interface of a relational database as the semantic structure of the database, which will be discussed in this section. A database system which uses *ERG* as the structure of the one-phase interface of the relational database system is shown in the Fig. 1.

1.1. ONE-PHASE DATABASE SYSTEMS OF THE RDKER

A one-phase interface of a relational database based on the *ERG* (*OPRER*) is composed of five components, namely as user-friendly interface, query mapping interface, query tree converting interface, query processing interface, and database storage. These interfaces have different functions on the processing of a query as:

(1). USER-FRIENDLY INTERFACE : On a one-phase system, multiple interfaces can be used in a database system. That is, more than one query interfaces of the relational database can be applied to an *OPRER*. In such a *OPRER* those query language such as SQL, QBE, QUEL, universal relation Interface, ERG-approached quasi-natural language, etc. are available on the same database system. Then the user may select any query language of the *OPRER* he likes to process a query.

(2). QUERY MAPPING INTERFACE : In a *RDKER*, the semantic structure of the database is represented by an *ERG* and the internal structure of a query on a *OPRER* of the *RDKER* is represented by an *ERQG* [Chen1987a]. To obtain an *ERQG* of a query on a *OPRER*, the query has to be mapped onto the *ERG* to get a subgraph of *ERG*. Then by combining the logics of the query into the *ERG*, a

undirected *ERQG* can be obtained. Finally, by labeling the direction of the query from each restriction main-nodes to the target part of the query, a directed *ERQG* of a query on a *OPRER* can be obtained.

(3). QUERY TREE CONVERSION INTERFACE : A cyclic subgraphs of an *ERQG* can be converted to a semantic equivalent tree structure [Chen1987a].

(4). QUERY PROCESSING : A subgraph of the restriction part of an *ERQG* represented by a tree structure can be further decomposed into a set of local regions. Then, for each local region of an *ERQG*, ER-semijoin can be used to process it. For a complex cyclic subgraph, the aggregation node processing technique can be employed [Chen1987a].

(5). DATABASE STORAGE : Although various interfaces may be applied to the user, the representation of the physical level under all interfaces is the same. That is, the user may select any interface of the system to process a query on the database system.

A database system with multiple interfaces was demonstrated by Li in the construction of the ILEX [Li1984]. Li employed "logic" as the internal structure of a query on the ILEX. While in our research, we use an *ERG* to represent the structure of a database and an *ERQG* to represent the internal structure of a query. One of the advantage of the *ERG* is that it can be used to allocate the access paths of a query on a universal relation interface. The universal relation interface was not introduced in the ILEX. Thus in this research, we use the universal relation to demonstrate the feasibility of using *ERG* as the structure of a relational database, allocation of access paths of a query in a global interface, and the processing of an *ERQG* by the ER-semijoin.

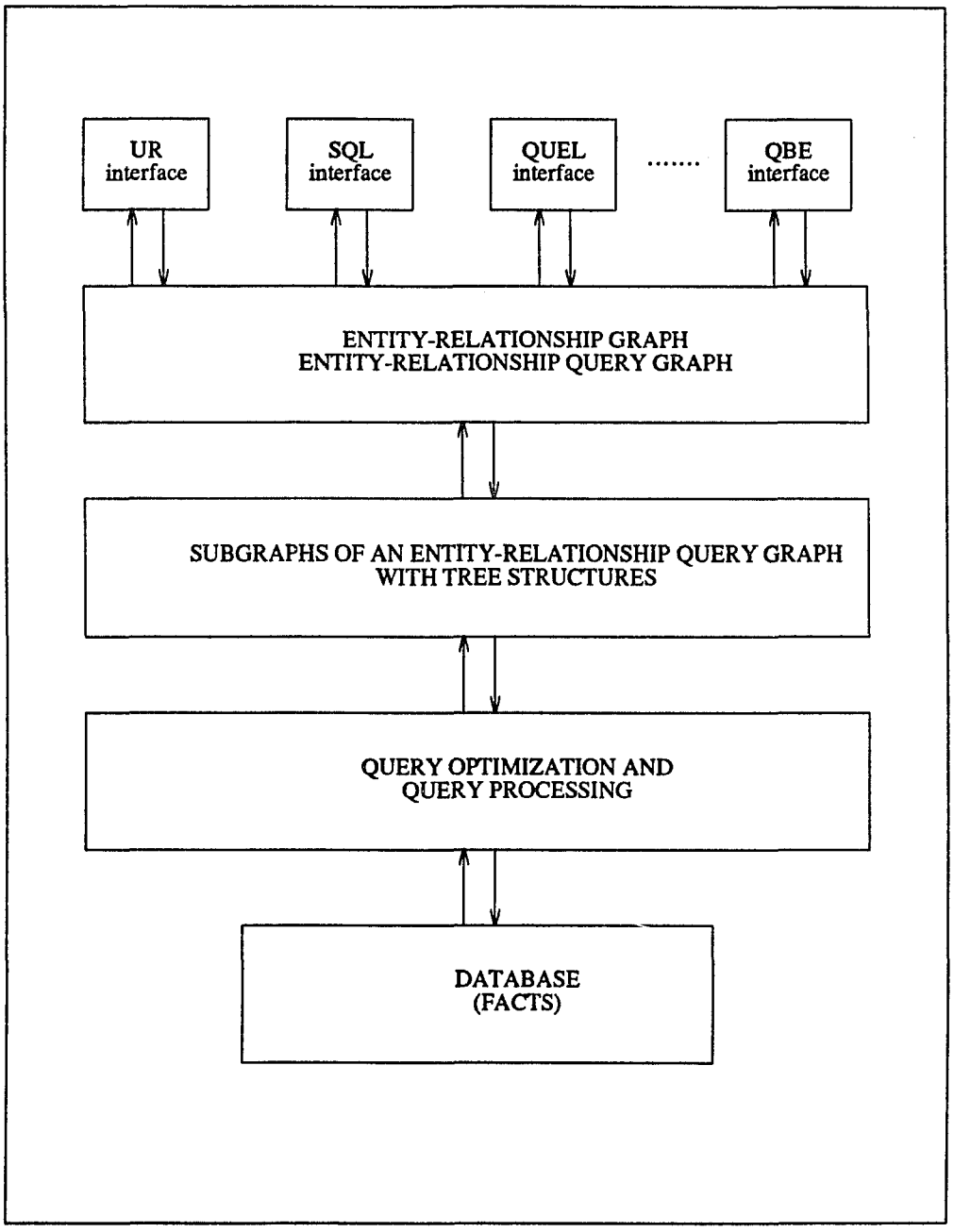


Fig. 1 The architecture of a multiple user interfaces of an *OPRER*.

1.2. TWO-PHASE INTERFACE OF RELATIONAL DATABSE SYSTEM BASED ON ERG

The *TPRER* (two-phase interface of the relational database based on the *ERG*) contains four different phases for the processing of a query : a user-friendly query interface or a two-phase query conversion interface, *ERQG* allocation interface, query mapping interface, and the underlying database systems. The query mapping interface and the entity relationship query tree conversion interface of the *TPRER* have the same functions as that of the *OPRER*. Fig. 2 shows the architecture of a *TPRER*. These *ERQG* processing phases of a *TPRER* are as follows :

(1). USER-FREINDLY INTERFACE OR QUERY TRANSFORMATION INTERFACE: A *TPRER* can be either applied to the building of a user-freindly query interface on top of a database system or the transforming of a query from one underlying database system to another.

A query on the user-friendly interface of a *TPRER* can be mapped onto the *ERG* of the underlying system to obatain an *ERQG*. There are two advantages for the construction of an *ERG* based query interface on top of a database system. The first advantage of the construction of a high level user-friendly interface is that its query language can be easily understood and handled by the user. The second advantage is that by representing a query by with *ERQG*, the query can be implemented according to the operation logic of the ER-semijoin. The proceessing of a query by the ER-semijoin is more efficient than the processing of the query by the traditional joining processing techniques [Chen1987b].

A *TPRER* can also be used as a query language conversion interface. In a *TPRER* a query language from one of the underlying database sytems can be mapped to an *ERQG*. Then, the *ERQG* can be mapped onto the query langauge of another underlying database system.

(2). ERQG REPRESENTATION PHASE : A query can be mapped onto the *ERG* of a underlying database system to obtained an *ERQG*. Most of the cyclic subgraphs of an *ERQG* can be converted to their semantically equivalent trees according to the *ERG* semantic structure of the underlying database system. Example 1 illustrates the intelligent query decomposition and mapping procedures of this phase.

(3). QUERY MAPPING : In this phase, a query represented by an *ERQG* is mapped to the query language of a underlying database system. If the segment of the restriction part of an *ERQG* can be decomposed into a unit graph, then ER-semijoin operation can be employed for the mapping from the *ERQG* to the query language of a underlying database system in the *TPRER* [Chen1987b]. Otherwise a acyclic subgraphs of an *ERQG* can be mapped according to the aggregation node processing technique [Chen1987a].

(4) THE UNDERLYING DATABASE SYSTEMS : From the query mapping phase, a query represented by an *ERQG* is mapped to the executable query language of a underlying database system in the *TPRER*. Then, this executable language is transmitted to the underlying database system for the processing of the query. Finally, the information obtained from the underlying database system can be transmitted back to the user.

In a *TPRER*, a user-friendly query interface can be built on top of a database system for the novel users. Nevertheless, the *TPRER* can also be applied as query mapping interface to the database system which contains distributed nodes.

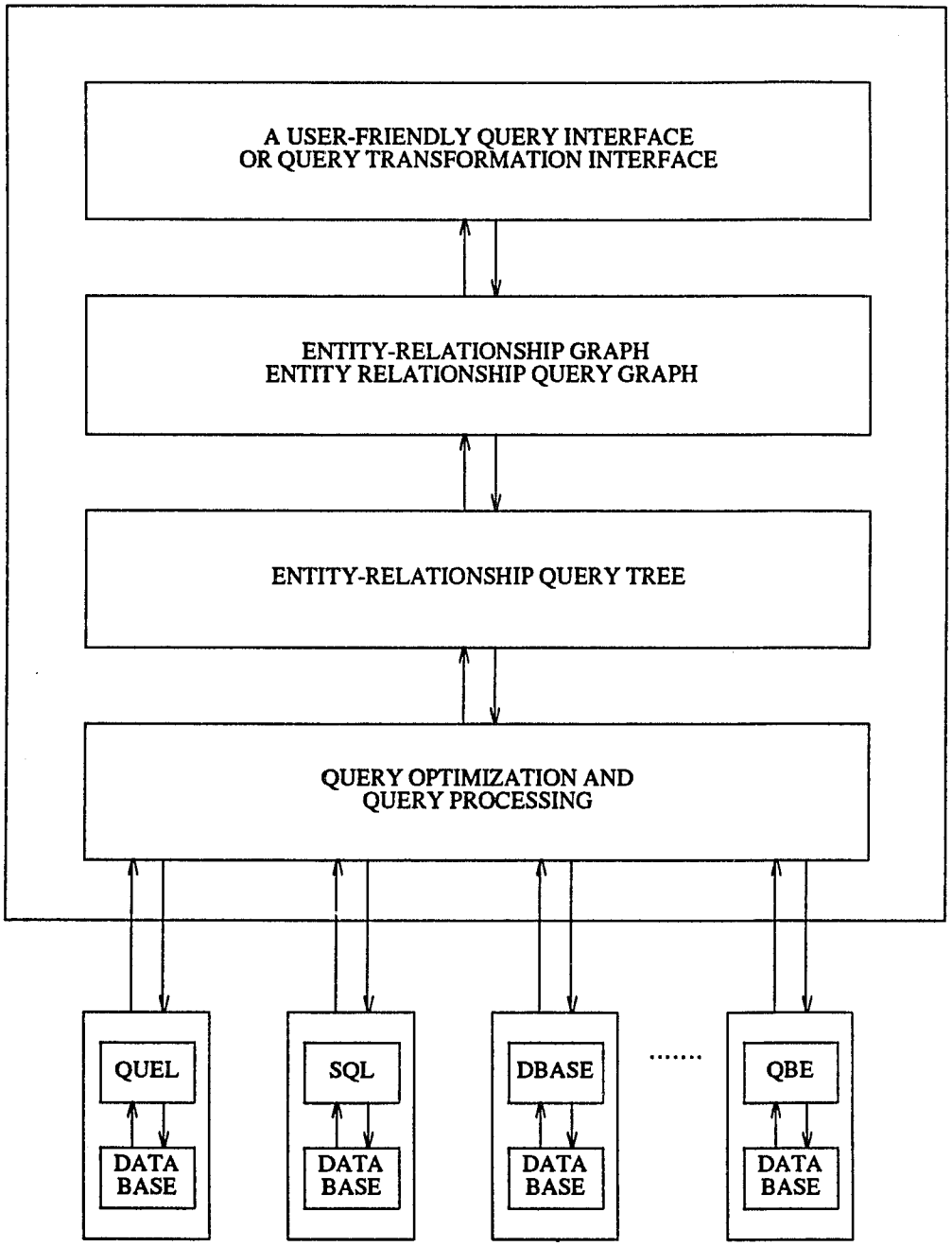


Fig. 2 The architecture of a two-phase interface based of the relational database based on the ERG.

Example 1: The database *BOOK* shown in fig. 3 contains the relations [*PERSON*, *BORROW*, *BOOK*, *PUBLISH*, *COMPANY*], where *PERSON*, *BORROW*, and *COMPANY* are entity nodes; *BORROW* and *PUBLISH* are relationship nodes. A quasi-natural language interface based on the *ERG* is built on top of this database. The query : Find the <*phone*> of the <*PERSON*> who <*BORROW*> the book <*PUBLISHED*> by the <*COMPANY*> with name = '*CROWN CO.*' This query including two local regions : [*PERSON*, *BORROW*, *BOOK*] and [*BOOK*, *PUBLISH*, *COMPANY*]. Thus the mapping of the query to the underlying database system can be processed according to the ER-semijoin operation as follows :

(1). Project on the primary key *company_id* of the tuples of the *COMPANY* which satisfy the condition *company_id* ='*CROWN CO.*'

(2). Select and project on the key *book_id* of the tuples of the *PUBLISH* such that the value of the *company_id* of these tuples are in the collected list from the step (1).

(3). Select and project on the key *person_id* of the tuples of the *BORROW* such that the value of the *book_id* of these tuples are in the collected list from the step (2).

(4). Select the tuples whose primary key *person_id* is in the list collected from the step (3).

(5). Project on the *phone* obtained from (4).

(6). Display the result.

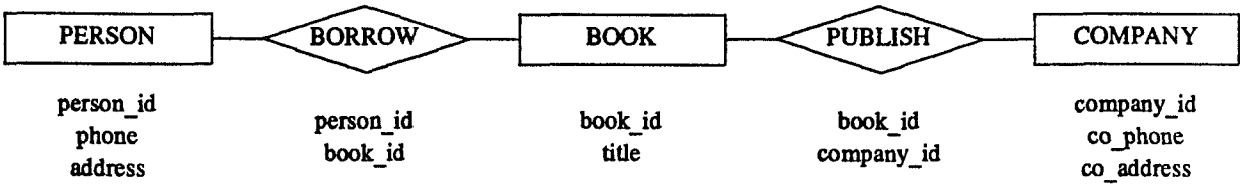


Fig. 3 The *ERM* of the database *BOOK*.

By using the traditional operation of the joining operators to process the query,

the procedures for the processing of the query of Example 1 are : (i) creating an intermediate table *TEMP* that contains only of the key of the *COMPANY* with name = *'CROWN CO'*, (ii) joining the relations of *PERSON*, *BORROW*, *BOOK*, *PUBLISH*, and *TEMP*, (iii) projecting on the attribute *person_id*. By mapping the query onto the database system according to the decomposition procedures of Example 1, the ER-semijoin operation can also be interpretered as a semantics based optimizing operator. The comparison of the efficiency of the time complexity and the space complexity for the execution of a query by using ER-semijoin with that by using tradition join operator is discussed in the Chapter 3.

The user-friendly interface of the *TPRER* can be extended to the multiple interfaces as that discussed in *OPRER*. For example, in a user-friendly interface with multiple interfaces built on top of the system-R, the user may select SQL, QBE, or another interface in this user-friendly interface to process a query. Then, the query will be represented by an *ERQG* and converted to the QUEL. Finally, the query represented in QUEL will be transmitted to the underlying system to process the query. Similarly, a user may also select QUEL , QBE, universal relation interface, etc. to process a query in a multiple interface built on top of the DB-2. In such cases, the *ERG* and *ERQG* are used as the structure of a database and the structure of a query respectively for the executing and optimizing of a query in an *TPRER*.

2. AN ERG BASED DATABASE SYSTEM

An *ERG* based database system with a universal relation interface is demonstrated by using logical programming [Kost1985, Kehl1985]. In this experimental system, we demonstrate the design of a relational database based on the *ERG*, the allocation of the access paths of a query by the aid of the *ERG*, and the processing of the query by using ER-semijoin. In this section, we sketch the basic definition of an *ERG* and the static constraints of attributes in a relational database system. Besides the

semantic constraints of the *ERG*, the user can also define constraints on attributes. Since there are various different static and dynamic constraints with different applications, we will only introduce the basic static constraints in this section.

DDL FOR THE PHYSICAL INSTANCES OF ENTITY NODE AND RELATIONSHIP NODE

In the physical level of a relational database system, the representations of an entity node and the relationship node are the same. That is, we do not have to distinguish an entity node from a relationship node in the physical level. The recognization of an entity node and a relationship node can be done in the semantic level. With the semantic structure represented by an *ERG*, the semantic difference of the entity nodes and the relationship nodes of a relational database base can be processed in the semantic level. To represent the semantic structure of a relational database in the *ERG*, the DDL to define an *ERG* on the database system is necessary. In this section, we introduce the DDL of the *ERG* such that an *ERG* can be used as the structure of a relational database system. In an *ERM*, the semantic unit is represented by a local region. The representation of a local region in the *ERG* is a unit graph [Chen1987a]. The unit graph of an *ERG* is defined as $U_j = [e_i, r_j, e_k, ARC_{ij}, ARC_{jk}, \{a_{il}\}, \{a_{jm}\}, \{a_{kn}\}, \{d_{il}\}, \{d_{jm}\}, \{d_{kn}\}, C_i, C_k]$, which contains the following properties:

(1). ENTITY NODE : entity_node ($[entity\_name]_i$, $[attribute]_{ij}$, $\{key, no\text{-}key\}$, $[specification]_{ij}$, ..., etc.),

where $[attribute]_{ij}$ is the jth attribute of the entity node $[entity\_name]_i$, and this attribute can either be a prime (denoted as *key*) or a non-prime (denoted as *no_key*). The $[specification]_{ij}$ is the specification of the type of this attribute of an entity node.

(2). RELATIONSHIP NODE : relationship_node ($[relationship\_name]_i$, $[attribute]_{ij}$, $\{key, no\_key\}$, $[specification]_{ij}$, ..., etc.),

where [*attribute*]$_{ij}$ is the jth attribute of the relationship node [*relationship_name*]$_i$, and this attribute can either be a prime (denoted as *key*) or a non-prime (denoted as *no_key*). The [*specification*]$_{ij}$ is the specification of the type of this attribute of a relationship node.

(3). ENTITY_RELATIONSHIP : entity_relationship ([*relationship_name*]$_i$, [*entity_name*]$_j$, {*W*,*E*}, {*role*,*no_role*}, ...,),

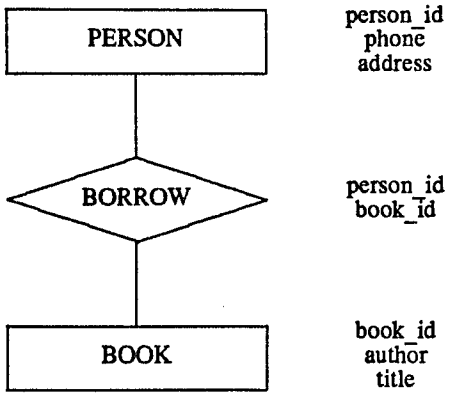
where [*entity_name*]$_j$ is the entity node that connects to the relationship [*relationship_name*]$_i$. The [*entity_name*]$_j$ must either be an entity node *E* or be a weak entity node *W*; such an entity-relationship set may either be a role type (IS-A) or a non-role type (not an IS-A).

(4). LOCAL_REGION (UNIT GRAPH) : local_region ([*relationship_name*]$_i$, [*entity_name*]$_{i-1}$, [*entity_name*]$_{i+1}$, [*card*]$_{i-1}$, [*card*]$_{i+1}$),

where [*relationship_name*]$_i$ is the relationship which connects the entity nodes [*entity_name*]$_{i-1}$ and entity node [*entity_name*]$_{i+1}$; [*card*]$_{i-1}$ and [*card*]$_{i+1}$ are the cardinalities of the local region [PERSON, BORROW, BOOK].

The predicates of (1) and (2) defines the physical instances of an entity node and a relationship node. The first term of such a predicates represents the identification of an entity node or a relationship node; the second term of the predicate defines an attribute in the relation; the third term of the predicate declares whether the attribute is a primary key or not; then, the specification of the type of the attribute may be added. In the Example 2, these DDL of a unit graph of an *ERG* is illustrated.

Example 2: A relational data model *LIBRARY* is represented by the *ERM* as



The person_id is the primary key of the relation PERSON, book_id is the primary key of the relation BOOK, and both of person_id and book_id are the keys of the BORROW. Then, the database *LIBRARY* can be define as follow :

entity_node (person, person_id, key, c-9).

entity_node (person, phone, no_key, i-10).

entity_node (person, address, no_key, c-32).

relationship_node (borrow, person_id, key, c-9).

relationship_node (borrow, book_id, key, c-12).

entity_node (book, book_id, key, c-12).

entity_node (book, author, c-30).

entity_node (book, title, c-20).

The data types c-12, i-10, and etc. will be discussed in the next section.

The ENTITY-RELATIONSHIP defines the connection of an entity node and a relationship node in the *ERG*. The connection of two entity nodes to a relationship node of the *ERG* is expressed by two of these predicates. An n-ary relationship can be represented by *n* predicates of ENTITY-RELATIONSHIPs whose first item is the specific relationship node. The representation of a local region is the projection of two entity nodes and a relationship node on an n-ary relationship node.

Example 3: The representation of the database *LIBRARY* of Example 1 contains the fol-

lowing semantic structures :

entity_relationship (borrow, person, *E*, no_role, ...).

entity_relationship (borrow, book, *E*, no_role, ...).

local_region (borrow, book, person, n, n).

These two predicates of 'entity_relationship' represent two partial graphs that represent a unit graph of a local region. An entity node can either be a weak entity node *W* or an entity node *E*. In this Example both of the *PERSON* and *BOOK* are entity nodes denoted as *E*. Both of *PERSON* and *BORROW* are non-role types (IS-A type) denoted as "no_role". The cardinality of 'n, n' in the predicate local_region means there are n:m relationship between the entity node *PERSON* and *BOOK*.

The cardinality '1, 1' , '1, n' or 'n, 1' specify the 1:1 and 1:n relationship between two entity nodes respectively. The cardinality amongst multiple attributes (more than two attributes) can also be defined according to the application.

The connection of an entity nodes and a relationship nodes in the *local-region* specifies a main-arcs of the *ERG* that has two main_nodes(entity nodes and relationship nodes). While the connection of an attribute and an entity nodes or a relationship_node specifies a sub_arc that has a main-node (entity node or relationship node) and a subnode (attribute). The information defined above sketch the skeleton of an *ERG*. That is, the information of the sets of the unit graphs of an *ERG* $ERG_o = \{U_i | U_j = [e_i, r_j, e_k, ARC_{ij}, ARC_{jk}, \{a_{il}\}, \{a_{jm}\}, \{a_{kn}\}, \{d_{il}\}, \{d_{jm}\}, \{d_{kn}\}, C_i, C_k]$ can be represented by data definition of an *ERG*.

The DDL of the unit graph is the basic definition of an *ERG*. Since the *ERM* can be extended to an *extended-ERM* according to the implementation of the real world, more predicates can be added to the *ERG* to represent an *extended_ERM*. For example, in the representation of a semantically clear *ERM*, the semantic regions such as *hierarchical region* and *inheritance region* of an *ERG* can be represented as follows

[Chen1987c]:

(1). HIERARCHICAL REGION: hierarchical ([*entiy_name*]$_i$, [*relationship_name*]$_j$, ..., [*entity_name*]$_m$, [*relationship_name*]$_n$). This predicate defines a list of entity nodes and relationship nodes in the hierarchical region.

(2). INHERITANCE REGION: inheritance ([*entiy_name*]$_f$, [*relationship_name*]$_g$, ..., [*entity_name*]$_k$, [*relationship_name*]$_l$). This predicate defines a list of entity nodes and relationship nodes in the inheritance region.

2.1. DATA TYPES FOR ATTRIBUTES OF RELATIONS

The first character of an attribute must be the character between the alphabet 'a' to 'Z'and the length of an attributes must under 32. The hyphen is used as the recognizer for type definition of the name of an attribute. Thus the alphabet of the name of attribute may be any character except the character "-". There are three basic types which are defined in the Table 1.

type of attributes	representation	maximum number
character	-c-000	3 digits
integer	-n-000	3 digits
floating point	-n-000-000	3-3 digits

Table 1. DDL of the type of attributes.

For example: The entity node *person* contains three attributes: *address*, *social security number*, and *salary*.

(1). The attribute *address*, a list of characters with 32 character, is represented as address-c-32.

(2). The attribute *security number*, an integer with 9 digits, is represented as security_number-n-9.

(3). The attribute *salary*, a floating point with 2 decimal points and 9 digits, is represented as salary-n-9-2.

The data definitions of these attributes are:

(person address-c-32 no-key)

(person security_number-n-9 key)

(person salary-n-9-2 no-key).

Besides these basic data definition on the attributes, the user may define data definitions according to the application.

2.2. STATIC CONSTRAINTS FOR USER'S INTERFACE

In the *URERG* (Universal Relation based on the *ERG*), the semantic constraints of the *ERG* can be automatically enforced to maintain the semantic integrity of the database. The processing of a query of updating (modification, insertion, deletion) for the global access of attributes is discussed in the chapter 5. Besides the semantic constraint of the *ERG*, a *URERG* do allow the user to define the domain of an attribute. Since in a universal relation each attribute is uniquely defined in the database system, the declaration of entity node or relationship node to which an attribute belongs to is unnecessary. In a relational database system where universal relation assumption is not applied, the declaration of the attributes in an entity node or a relationship node is needed.

The static constraints of a database is desirable in a relational database system[Nanc1983]. In a *URERG*, the user is allowed to specify the static constraint on the attributes. For example, the maximum salary of an employee of a company is defined as 100,000.00. If the input salary is higher than the maximum value, the system may friendly notify the user. The predicate of constraint is declared as

CONSTRAINT (entity / relationship, attribute, type specification).

The CONSTRAINT is the predicate for the specification of the static constraints. The *type* is the declaration of the type of constraint. For example, user may use MAX to specify the maximum value of salary of employees in a company. The different types of constraints on URERG is summarized in the table 2.

SPECIFICATION	*TYPE*	*SPECIFICATIONLIST*
maximal value	MAX	digital value
minimum value	MIN	digital value
range of value	RAV	minimal value & maximal value
range of character	RAC	lower character & upper character

Table 2. Constraints of the attributes which users may specified.

Example 4: The information of a company is represented as [*EMPLOYEE*, *WORK*, *DEPARTMENT*], where the entity node *EMPLOYEE* has attributes person_id, social security number, salary, address, age, and etc. Constraints on these attributes may be optionally specified as:

CONSTRAINT (employee salary RAV 400 20000)

CONSTRAINT (employee age MAX 55)

Since the dynamic constraints is always depend on the application, we will not discuss the dynamic constraints in this chapter.

3. A UNIVERSAL RELATION QUERY LANGUAGE BASED ON ERG

A universal relation interface based on the *ERG* is quite different from the traditional approach to the universal relation. The traditional universal relation approach always uses the functional dependency amongst attributes to allocate the access paths of a query. These access paths are represented as maximal object [Ullm1983], or

extension join [Sagi1983]. For the *ERG* approach to the universal relation, the *ERG* is used as the structure of the database and it also is employed as the tool to allocate the access paths of a query on the universal relation interface. The theoretical survey of the universal relation of the *ERG* approach are discussed in chapter 5. The universal relation introduced by Ullman and Sagiv is just for the retrieval of information from the representative instances. For the *ERG* approach to the universal relation, besides the retrieval, the updating (modification, deletion, and insertion) of the representative instances can be achieved by the enforcement of the semantic constraints of the database represented by an *ERG*.

The benefit of using *ERG* as the theoretic basis of universal relation is that the semantic integrity among physical instances of the relational database system can be easily obtained, that is, the semantic integrity of updating the universal relation with semantic structure represented by an *ERG* can be achieved within a constant semantic transition level of the attributes that are to be updated [Chen1987c]. Thus in the *ERG* approach to the universal relation, a query may be categorized into two types as retrieval and updating. The syntax of the retrieval of the universal relation by the *ERG* approach is similar to the system/U [Henr1984]. But the traditional universal relation like system/U does not have the function of updating (insertion, deletion, modification).

3.1. QUERY LANGUAGE OF THE UNIVERSAL RELATION BASED ON ERG

On the universal relation based on the *ERG* the user is able to do a query which may either be the query of retrieval or of updating (modification, insertion, deletion). In this section we discuss the syntax of the query language on the universal relation based on the ERG.

The Syntax of the Universal Relation Based on the ERG :

A query on the universal relation based on the *ERG* contains target part and the restriction part. The target part of a query consists of a list of attributes to be retrieved; and the restriction parts of the query contains the predicates that define the domain of the attributes to be retrieved.

The context free grammar of the universal relation query language by the *ERG* approach is illustrated as follows :

<QUERY>::= <UPDATE> | <RETRIEVAL>.

<UPDATE>::= <MODIFICATION> | <INSERTION> | <DELETION> | <APPEND-ING>.

<RETRIEVAL>::= <TARGET PART> <RESTRICTION PART>.

<TARGET PART>::= [RETRIEVE] <AGGREGATE> <ATTRIBUTE_LIST>.

<ATTRIBUTE_LIST>::= [ATTRIBUTE] <ATTRIBUTE_LIST> | [ATTRIBUTE].

<RESTRICTION_PART>::= <AGGREGATE> <RESTRICTION_LIST>.

<RESTRICTION_LIST>::=

ε | (<RESTRICTION_LIST>) | <ATT_SET> <RESTRICTION_LIST> | <RESTRICTION_LIST> <LOGIC PAIR>.

<ATT_SET>::= [ATTRIBUTE] <LG> [ATTRIBUTE] | [ATTRIBUTE].

<AGGREGATE>::= ε | COUNT | SUM | AVG | MAX | MIN.

<LG>::= ∧ | ∨ | − | + | × | ¬ | Θ.

<MODIFICATION>::=

[MODIFY] x [FROM] a [TO] b | [MODIFY] R [FROM] (a_1,..., a_i) [TO] (b_1, ..., b_i) | [MODIFY] x ; [TUPLE] [FROM] (a_1, ..., a_i) [TO] (b_1, ..., b_i).

<DELETION>::=

[DELETE] t, x op a | [DELETE] x op a | [DELETE] r ; (a_1, ..., a_i).

<INSERTION>::=

[INSERT] <INSERT_APPEND>.

<APPENDING>::=

[APPEND] <INSERT_APPEND>.

<INSERT_APPEND>::=

x a | x (x') a (a') r (a_1, ..., a_i).

op ::= = | > | < | ≥ | ≤.

The query language of system/U contains only the function of retrieval. Brosda proposed the theory of global consistency of modification, insertion, and deletion through the Universal Relation by using the chase manipulation and extension join. In the *URERG*, the control of the semantic integrity of a query of Update (modification, insertion, and deletion) is obtained by the semantic constraints of the structure of the relational database system represented in the *ERG*. Through the semantic control based on the structure of the database represented by an *ERG*, the query (Retrieval and Update) can be processed without losing the semantic integrity of the database.

In a *URERG*, a query language is categorized as either Retrieval or Update (Modification, Insertion, and Deletion). The syntax of the query of retrieval on the universal relation based on the *ERG* is the same as that of other universal relations. That is, each statement of retrieval consists of two lists. The first list is a list of target attributes to be retrieved. The second list is composed of a set of predicates that limit the domain of the target attributes to be retrieved.

The syntax for the Update of the *URERG* are as follows:

(1). MODIFICATION : The modification of a relational database system on a universal relation interface can be categorized into two types - modification of attributes of tuples and modification of tuples. In other words, a user can either modify the attributes of tuples or the tuples of a relation of a relational database

system without understanding the conceptual level of the database. A query of modification can be processed according to the functions as follows:

(i). Modifying the value of an attribute from the value a to value b : [MODIFY] x [FROM] a [TO] b.

(ii). Modifying of a tuple of a relation from a set of values to another set of values : [MODIFY] R [FROM] (a_1,...,values a_i) [TO] (b_1, ..., b_i).

(iii). Modification of a set of values of attributes to another set of values. [MODIFY] x ; [TUPLE] [FROM] (a_1, ..., a_i) [TO] (b_1, ..., b_i).

(2). DELETION : The function of deletion can be categorized as the deletion of the specified attributes of a relation or the deletion of the tuples of a relation.

(i). Deleting the column of attribute x with value a : [DELETE] x op a; the values in the column of attribute x which satisfy the condition will be deleted.

(ii) Deleting the tuple of a relation of which contains a specified set of values : [DELETE] r ; (a_1, ..., a_i).

(3). Inserting and Appending : The only difference of the function of appending and inserting is that the function of appending inserts the new tuple to the end of the file of the relation while the inserting can be processed before or at the tuple which satisfies the condition :

(i). Inserting or appending value to a primary key : [INSERT / APPEND] x a ; the value a of attribute x is going to be inserted or appended.

(ii). Inserting or appending value a' to the attribute x' : [INSERTION / APPEND-ING] x (x') a (a'); where x' and x are in the same relation and the x is the primary key with value a.

(iii). Inserting of a set of value to a relation : [INSERTION / APPENDING] r (a_1, ..., a_i) ; where r is a relation and (a_1, ..., a_i) is the tuple to be inserted or

appended.

The universal relation based on the *ERG* allows the user to update the physical representation of the database either through the universal relation representation (attributes) or through the relations of the conceptual level. The purpose of the permission of the update through conceptual level is that it may apply a convenient interface to the user.

CHAPTER 7
DISCUSSIONS AND CONCLUSIONS

A RDKER (Relational Database with Knowledge of ERG) can be viewed on two levels: the conceptual level and the physical level. The physical level of a RDKER represents physical relations of entity nodes and relationship nodes. In the physical level, there is no specific difference between entity nodes and relationship nodes. Thus, both entity nodes or relationship nodes of an ERG are treated as physical relations in the physical level. The conceptual level of a RDKER contains the semantic structure of the physical level represented in the ERG. That is, for each entity node or relationship node in the conceptual level, there is one and only one relation in the physical level corresponding to it.

The semantic structure of a relational database can be widely applied in the query optimization, dynamic constraints for updating (modification, insertion, deletion) attributes, allocation of access paths on the Universal Relation interface, and conversion of cyclic sub-query graphs to tree structures. To employ an ERM as the semantic structure of a relational database system, more restrictions on the ERM are needed. Thus, we define a semantically clear ERG such that it may be applied as the structure of a relational database.

An ERG that is not a single entity node should be able to be represented by a finite set of local regions. A local region contains a pair of entity nodes and a relationship node that connects these entity nodes. Owing to the different semantic properties of entity nodes and relationship nodes of an ERG, the ERG can be further grouped into a set of different regions. These regions of an ERG may help consistency and integrity control of a database.

By defining a database as an *ERG*, the integrity control for the updating of a data-

base can be obtained by using the semantic structure of the *ERG*. The local constraints of local regions of an *ERG* can be either applied to the static integrity checking or dynamic integrity checking of a database. By using the updating propagation structure of a nodes in the *ERG*, the dynamic integrity control for the updating of that node can be achieved.

In a RDKER, a query can be mapped onto the ERG to obtain a semantic structured query. A semantic structured query can be decomposed into a set of semantic units represented by local regions. Then, ER-semijoin can be employed to process the query based on these local regions.

The semantics of a local region can always be reduced to its relationship. Thus, to process two conjunctive local regions by using the ER-semijoin, the operation can always be reduced to two relationship nodes. While, to process these two local regions by the joining operation, five relations (three entity nodes and two relationship nodes) have to be operated.

The comparison of employing the ER-semijoin with joining operation on the processing of a local region is done using the coefficients of full semantic reduction. The coefficient of full semantic reduction of a local region is categorized as spatial coefficient and temporal coefficient of full semantic reduction. The range of spatial and temporal coefficient of full semantic reduction of a local region is between zero and one. That means for the best case, the ratio of space complexity by using ER-semijoin to the ratio of that by using traditional join operator approaches zero. The worst case is that the space complexity of using ER-semijoin on a local region are equal to that exerting joining operation on that local region.

An *ERG* contains main-nodes and subnodes. A main-node represent a node of entity type or relationship type; and a subnode that represent an attribute of a main-node. For each subnode there is one and only one subarc which connects it to a

main-node. On a universal relation a global set of attributes is assumed such that the user's view is a set of subnodes. For a query on a universal relation based on a *RDKER* the access paths of a query can be allocated by mapping it to the *ERG*.

The *ERG* in a *RDKER* can be decomposed into a set of unit graphs, and a query on this *RDKER* may be mapped on to the *ERG* to acquire a semantically structured graph represented by a subgraph of the *ERG*. Then an *ERQG* can be obtained by adding the relational operators of the query to this semantically structured *ERG*.

The query processing on an *ERQG* is independent of the user-friendly interface. It is more stable to represent the structure of a query in an *ERQG* than to represent it in the structure of a specific user-friendly interface. That is, we can use several interfaces on the top of a relational database system for the user. A query may be processed on different interfaces with the same *ERQG* representation. For a cyclic subgraph of an *ERQG*, it can be converted to a tree structure such that we may use the efficient operator - ER-semijoin to process this acyclic graph. For a complex cyclic subgraph of an *ERQG* or the *TERQG* (Target part of Entity-Relationship Query Graph) it can be processed as an aggregation node.

On a universal relation interface of a relational database, the user does not have to navigate the conceptual level. The universal relation proposed by Ullman and Sagiv uses function dependency amongst attributes to allocate the access paths of a query. We investigate a universal relation approach to a relational database by using semantic structure of a relational database for the processing of a query on a universal relation interface.

The semantic structure of a relational database can be applied to the allocation of the access paths and to the query decomposition of a query of a universal relation. On this universal relation based on an *ERG*, the non-redundant ER assumption is made on the *ERG*. The non-redundant ER assumption on an *ERG* ensures that for each pair of

main-nodes in the *ERG*, there will be no redundant path which has duplicated semantics. Thus for a query on a universal relation of a non-redundant ER, a unique minimal semantic extended region on the semantic level can be allocated. The minimal semantic extended region of a query has a corresponding minimal *ERQG*. For a query on a universal relation interface, there exists one and only one *ERQG* in corresponding to it. Then for an *ERQG*, we may use ER-semijoin operation to process it.

A one-phase multiple interface system and a two-phase interface system that are based on the structure of a relational database represented by an *ERG* are proposed. For a one-phase multiple interface system, multiple relational query interfaces can be built in the system. Thus the user may select any query interface in the system he likes. The internal representation of a query on any interface of the system is represented by an *ERQG*.

A two-phase interface system is to construct a user-friendly interface on top of a relational database system. That is, we may construct a natural language interface or universal relation interface on top of any other relational database system. A query on such a user-friendly interface is represented by an *ERQG*. The ER-semijoin can be used as an optimizing operation for the conversion of a query represented by an *ERQG* to the query language of the underlying relational database system.

A universal relation interface can be built either on the two-phase interface system or on the one-phase interface system. The universal relation interface based on the *ERG* of the relational database has advantage that the static constraints can be enforced by the semantic structure of *ERG*. The query interface of the universal relation based on the semantic structure contains the function of updating (modification, insertion, deletion) and retrieval, while the query on the system/U has only the function of retrieval.

References

[Alfr1979] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman, "Efficient Optimization of a Class of Relational Expressions," *ACM TODS*, vol. 4(4), pp. 435-454, 1979.

[Berg1985] H. L. Berghel, "Simplified Integration of Prolog with RDBMS," *Database*, p. 3, Spring 1985.

[Brad1985] L. I. Brady, "An Universal Relation Assumption Based on Entity-Relationship and Relationships," *Proceedings the 4th International Conference on ER-Approach*, pp. 208-215, IEEE Computer Society, Oct. 1985.

[Chen1981] P. Atzeni and Peter.P. Chen, "Completeness of Query Languages for the ERM," *ER Approach to Information Modeling and Analysis*, pp. 109-122, North-Holland, 1981.

[Chen1976] Peter P. Chen, "The Entity-Relationship Model: Toward a Unified View of Data," *ACM Transaction on Computer Science*, vol. 1, pp. 9-36, January 1976.

[Chen1977] Peter. P. Chen, "The Entity-Relationship Approach to Logical Database Design," *QED Information Science*, 1977.

[Chen1983] Peter P. Chen, "English Sentence Structure and Entity-Relationship Diagrams," *Information Science*, vol. 29, May 1983.

[Chu1981] P. A. Berstein and D. W. Chu, "Using Semijoin to Solve Relational Queries," *J. ACM*, vol. 28(1), pp. 25-40, January 1981.

[Date1983] C. J. Date andddison-Wesley, *An Introduction to Database System*, 1983.

[Davi1984] David Maier, Jeffrey D. Ullman, and Moshe Y. Vardi, "On the Foundations of the Universal Relation Model," *ACM transactions on Databse*

Systems, vol. 9(2), pp. 283-308, June 1984.

[Daya1983] Hai Yann Hwang and Umeshwar Dayal, "Using the Entity-Relationship Model for Implementing Multi-Model Database Systems," *ER Approach to Information Modeling and Analysis*, pp. 235-256, North-Holland, 1983.

[Daya1981] Mohamed G. Ggouda and Umeshwar Dayal, "Optimal Semijoin Schedules for Query Processing," *ACM -SIGMOD international Conference on Management of Data*, pp. 164-173, ACM, Ann Arbor, Michigan, Aapril 29, 1981.

[Dona1986] James Donahue, "Whiteboards: A Graphical Database Tool," *ACM Transactions on Office Information Systems*, vol. 4(1), pp. 24-41, January 1986.

[Doug1985] Douglas M. Campbell, David W. Embley, and Bogdan Czeido, "A Relational Complete Query Language for an Entity-Relationship Model," *Proceedings the 4th International Conference on Entity-Relationship Approach*, pp. 90-97, IEEE Computer Society, Chicago, Oct. 28, 1985.

[Epst1982] Robert S Epstein, "Query Processing Techniques for Distributed, Relational Data Base Systems," *UMI Research Press*, 1982.

[Erol1985] A. Dogac and Peter. P. Chen and N. Erol, "The Design and Implementation of an Integrity System for the Relational DBMS RAP," *Proceeding the 4th International Conference on ER Approach*, pp. 295-302, IEEE Computer Society, Oct. 1985.

[Fry1982] W. Teorey and Palmer E. Fry, *Design of Database Structures*, p. 27, Prentice Hall, 1982.

[Gilb1986] L. Bic and J. P. Gilbert, "Learning from AI : New Trends in Database Technology.," *Computer*, pp. 44-54, Mrach 1986.

[Good1983] R. H. Katz and N. Goodman, "View Processing in MULTIBASE, A Heterogeneous Database System," *ER Approach to Information Modeling and Analysis*, pp. 257-278, North-Holland, 1983.

[Hase1981] Prabuddha De and William D. Haseman, "Four-Schema Approach: An Extended Model for Database Architecture," *Information System*, vol. 6(2), pp. 117-121, 1981.

[Hawr1984] I. T. Hawryszkiewycz, *Database Analysis and Design*, p. 423, SRA, 1984.

[Hawr1985] I. T. Hawryszkiewycz, "A Computer Aid for ER Modeling," *Proceedings the 4th International Conference on ER-Approach*, pp. 64-69, IEEE Computer Society, Oct. 1985.

[Henr1984] Henry F. Korth, Gabriel M. Kuper, "System/U: A Database System Based on the Universal Relation," *ACM Transactions on Database Systems*, vol. 9(3), pp. 331-347, September 1984.

[Houg1986] Raymond C. Houghton, "Designing User Interfaces: A key to System Success," *Information Systems Management*, vol. 3(3), pp. 56-62, Summer 1986.

[Ibar1984] Toshihide Ibaraki, "On the Optimal nesting Order for Computing N-Relational Joins," *ACM TODS*, vol. 9(3), pp. 482-502, September 1984.

[Kamb1985a]

Yahiko Kambayashi, "Processing Cyclic Queries," *Query Processing in Database Systems*, pp. 62-67, springer-berlag, 1985.

[Kamb1985b]

Yahiko Kambayshi, "Processing Cyclic Queries," *Query Processing in Database System*, Springer-Verlag, 1985.

[Kapl1984] S. Jerrold Kaplan, "Designing a Portable Natural Language Database Query System," *ACM TODS*, vol. 9(1), pp. 1-19, March 1984.

[Kehl1985] Richard Fikes and Tom Kehler, "The Role of Frame-Based Representation in Reasoning," *Communication of ACM*, vol. 28(9), pp. 904-920, September 1985.

[Kent1983] William Kent, "The Universal Relation Revisited," *ACM Transactions on Database Systems*, vol. 8(4), pp. 644-648, December 1983.

[Kim1985] Won Kim, "Global Optimization of Relational Queries: A first Step," *Query Processing in Database Systems*, pp. 206-216, Springer-Berlag, 1985.

[Kost1985] Alexis Koster, "Prolog Applications for Database Design with the Information Center," *Proceedings of 1985 ACM Computer Science Conference*, pp. 12-14, March 1985.

[Li1984] Deyi Li, *A Prolog Database System,* John Willey & Sons Inc., 1984.

[Maie1983] David Maier, *The Theory of Relational Databases,* p. 93, Computer Science Press, 1983.

[Malo1981] D. R. Mcgregor and J. R. Malone, "The Fact Database" An Entity-Basedm System Using Inference," *ER-Approach to Information Model and Analysis*, pp. 537-562, North-Holland, 1981.

[Mell1984] W. F. Clocksin and C. S. Mellish, *Programming in Prolog,* 2nd ed., Springer Verlag, 1984.

[Mend1984] Alberto O. Mendelzon, "Database and Their Tableaux," *ACM TODS*, vol. 9(2), pp. 264-282, June 1984.

[Morg1983] Mattew Morgenstern, "A Unifying Approach for Conceptual Schema to Support Multiple Data Models," *ER Approach to Information Modelinh*

and Analysis, pp. 279-298, North-Holland, 1983.

[Motr1986] Aamihai Motro, "BAROQUE: A Browser for Relational Databases," *ACM Transactions on Office Information Systems*, vol. 4(2), pp. 164-181, April 1986.

[Nanc1983] Yves Tabourier and Dominique Nanci, "Integrity Constraints in the Entity-Relationship Model," *ER Approach to Information Modeling and Analysis*, pp. 73-108, North-Holland, 1983.

[Neum1986]U. W. Lipeck and K. Neumann, "Modeling and Manipulating Objects in Geoscientific Databases," *5th International Conference on ER-Approach*, pp. 105-126, ER Institute, November, 1986.

[Nguy1986] G. T. Nguyen, "Object Prototypes and Database Samples for Expert Database Systems," *Proceedings of the First International Conference on Expert Systems*, pp. 3-14, 1986.

[Poon1978] George Poonen, "CLEAR: A Conceptual Language for Entities and Relationships," *International Conference on the Management of Data*, Italy, 1978.

[Rior1976] Sammy Mahmoud and J. S. Riordon, "Optimal Allocation of Resources in Distributed Information Networks," *ACM TODS*, vol. 1(1), pp. 66-78, March 1976.

[Sagi1981a] Yehoshua Sagiv, "Can We Use the Universal Relation Without Using Nulls?," *ACM -SIGMOD international Conference on Management of Data*, pp. 108-120, ACM, Ann Arbor, Michigan, Aapril 29, 1981.

[Sagi1983] Yohoshua Sagiv, "A Characterization of Globally Consistent Database and Their Correct Access Paths," *ACM TODS*, vol. 8, pp. 266-286, June 1983.

[Sagi1981b] Yehoshua C. Sagiv, *Optimization of Queries in Relational Databases,* UMI Research Press, 1981.

[Sahn1982] Ellis Horowitz and Sartaj Sahni, *Fundamental of Data Structures,* Computer Science Press, 1982.

[Saka1983] Hirotaka Sakai, "Entity-Relationship Approach to Logical Database Design," *ER-Approach to Software Engineering*, pp. 155-187, North-Holland, 1983.

[Sant1983] M. Lenzerini and G. Santucci, "Cardinality Constraints in the ERM," *ER Approach to Software Engineering*, pp. 529-549, North-Holland, 1983.

[Sant1985] M. Lenzerini and G. Santucci, "SERM : Semantic Entity-Relationship Model," *Proceedings the 4th International Conference on ER Approach,* IEEE Computer Approach, Oct. 1985.

[Schu1986] Gary Schuldt, "ER-Based Access Model," *5th International Conference on ER-Approach*, pp. 161-180, ER Institute, November, 1986.

[Sen1981] Prabuddha De and Arun Sen, "An Extended Entity Relationship Model with Multi Level External Views," *ER-Approach to Information Modeling and Analysis*, pp. 455-467, North-Holland, October, 1981.

[Shmu1982] Nathan Goodman and Oded Shmueli, "Tree Queries : A Simple Class of Relational Queries," *ACM Transactions on Database Systems*, vol. 7(4), pp. 653-677, December 1982.

[Shmu1981] Oded Shmueli, "The Fundamental Role of Tree Schemas in Relational Query Processing," *Harvard University,* August 1981.

[Shor1984] Bruce G. Buchanan and Edward H. Shortliffe, *Rule Based Expert Systems,* Addison-Wesley, 1984.

[Thul1981] Swamy Thulasiraman, *Graphs, Networks, and Algorithms,* Wiley Intersci-

ence Publication, 1981.

[Tibe1985] M. Schkolnick and P. Tiberio, "Estimating the Cost of Update in a Relational Database," *ACM TODS*, vol. 10(2), pp. 163-179, June 1985.

[Toby1986] Toby J. Teorey, Dongqing Yang, James P. Fry, "A Logical Design Methodology for Relational Databases Using Entity-Relationship Model," *ACM Computing Surveys*, vol. 18(2), pp. 197-222, June 1986.

[Tsic1983] Simon Gibbs and Dinysis Tsichritzis, "A Data Modeling Approach for Office Information Systems," *ACM Transactions on Office Information Systems*, vol. 1(4), pp. 299-319, October 1983.

[Ullm1983a]David Maier and Jeffrey D. Ullman, "Maximal Objects and the Semantics of Universal Relation Databases," *ACM TODS*, vol. 8(1), pp. 1-14, ACM, March 1983.

[Ullm1982] Jeffrey D. Ullman, *Principles of Database Systems,* Pitman, 1982.

[Ullm1983b]Jeffrey D. Ullman, "On Kent's "Consequences of Assuming an Universal Relation"," *ACM Transactions on Database Systems*, vol. 8(4), pp. 637-643, December 1983.

[Vass1985] Matthias Jarke and Yannis Vassiliou, "A Framework for Choosing a Database Query Language," *ACM Computing Surveys*, vol. 17(3), pp. 313-340, September 1985.

[Voss1985] Volkert Brosda and Gottfried Vossen, "Updating a Relational Databse through a Universal Schema Interface," *ACM Conference*, p. 66, 1985.

[Wall1984] T. F. Wallance, *Communicating with Database in Natural Language,* p. 142, Ellis Horwood Ltd., 1984.

[Webr1981]Neil W. Webre, "An Extended ERM and Its Use on a Defense Project," *ER Approach to Information Modeling and Analysis*, pp. 173-194,

North-Holland, 1981.

[Wied1986] Xiaolei Qian and Gio Wiederhold, "Knowledge-based Integrity Constraint Validation," *The International PRE-VLDB Symposium of 1986*, pp. 292-304, August, 1986.

[Wong1976]Joseph Wong, "Decomposition-A Strategy for Query processing," *ACM TODS*, vol. 1(3), pp. 223-241, 1976.

[Yann1985] Yannis Vassiliou, Jim Clifford, and Matthias Jarke, "Database Access Requirements," *Query Processing in Database Systems*, pp. 156-170, Springer-Verlag, 1985.

[Yone1984] Hiroshi Maruyama and Akinori Yonezawa, "A Prolog-Based Natural Language Front-End System," *New Generation Computing*, vol. 2, pp. 91-99, Ohmshal and Springer, 1984.

[Yu1984] Clement Yu, "Distribute Database Query Processing Topics," *Topics in Information Systems*, pp. 48-62, Springer-Verlag, 1984.

[Yu1985] Clement Yu, "Distributed Database Processing," *Query Processing in Database System*, pp. 48-61, springer-Verlag, 1985.

[Zhan1983] Zhi-Qian Zhang, "A Graphical Query Language for Entity-Relationship Databases," *ER Approaches to Software Engineering*, pp. 441-464, North-Holland, 1983.

APPENDIX

```
/*****************************************************************/
 A relational database system is created by using C-prolog
 on Vax/Unix. In this system, the following functions are
 demonstrated :
 (1). Entity-Relationship Graph defined by the Local Regions.
 (2). ER-semijoin based on the ERG.
 (3). A simple data input interface.
 (4). A universal relation interface.
/*****************************************************************/


/****************************** db ******************************/
/*CONCATENATION OF TWO LISTS WITHOUT DUPLICATE ELEMENT */
cat_list([],X,X).
cat_list([A|B],C,Z):-member(A,C),!,cat_list(B,C,Z).
cat_list([A|B],C,[A|Z]):-!,cat_list(B,C,Z).

/* HEAD AND TAIL OF LIST */
tail([X|Y],Y).
head([X|Y],X).

/*LIST INTERSECTION IS USED TO TEST WHETHER TWO ATTRIBUTE LIST OF
  TWO DIFFERENT RELATIONS HAVE THE COMMON ATTRIBUTE.(INTERSECTION
  IS NOT EMPTY) */
intersection([],B,[]).
intersection([X|Y],B,[X|C]):-member(X,B),!,intersection(Y,B,C).
intersection([X|Y],B,C):-intersection(Y,B,C).

/*SEARCH WHETHER AN ATTRIBUTE IS IN AN LIST */
search_attribute(A,[A|C]).
search_attribute(A,[B|C]):-search_attribute(A,C).

/* The following rules copy one file to another */
file_copy(F1,F2):- see(F1),tell(F2),
                   repeat,read(X),write(X),nl,
                   (X=end_of_file,!,told(F2),seen(F1));
                   (write(X),fail).

 /* THE RULE COLLECT A SUBSET OF A LIST WHICH DOES NOT CONTAINED
    IN ANOTHER LIST */
not_sublist([],B,[]).
not_sublist([X|Y],B,[X|C]):-not(member(X,B)),!,not_sublist(Y,B,C).
not_sublist([X|Y],B,C):- not_sublist(Y,B,C).

  /*FINDALL : COLLECT ALL POSSIBLE VALUES IN A LIST */
findall(X,G,_) :-
```

```
        asserta(found(mark)),
        call(G),
        asserta(found(X)),
        fail.
findall(_,_,L) :- collect_found([],M),!,L=M.
collect_found(S,L):-getnext(X),!,collect_found([X|S],L).
collect_found(L,L).
getnext(X) :-retract(found(X)),!, X == mark.

/*APPEND*/
append([],X,X).
append([A|B],C,[A|D]):-append(B,C,D).

/*MEMBER*/
member(X,[X|_]).
member(X,[_|Y]):-member(X,Y).

/*NEGATION*/
no(P):-call(P),!,fail.
no(P).

 /* REVERSING A LIST */
rev([],[]).
rev([H|T],L) :- rev(T,Z),append(Z,[H],L).

 /* GET THE LAST ELEMENT OF A LIST */
last(X,[X]).
last(X,[_|Y]):-last(X,Y).

 /* SEPARATE A LIST INTO LAST AND REMAINING LIST */
abstract_tail(X,[X],[]).
abstract_tail(X,[Y|Z],[Y|T]):- abstract_tail(X,Z,T).

 /* MERGE TWO PATHS INTO A COMBINED PATH: e.g. [1,2,3,d,f,g,p,q,r]
   AND [1,2,3,v,p,q,r] WILL BE MERGED INTO [1,2,3,[[f,g],[v]],p,q,r] */
merge(A,B,C):- merge_head(A,B,H,T1,T2),merge_tail(T1,T2,H1,H2,T),
          rev(T,Ts),append(H,[[H1|[H2]]],S),append(S,Ts,C).

 /* MERGER THE SAME HEADS OF TWO LISTS */
merge_head([X|Y1],[X|Y2],[X|H0],P,Q):- merge_head(Y1,Y2,H0,P,Q).
merge_head(U,V,[],U,V).

 /* MERGE THE SAME TAILS OF TWO LISTS */
merge_tail(T1,T2,P,Q,T):- abstract_tail(S,T1,H1),
                  abstract_tail(S,T2,H2),
                  !,merge_tail(H1,H2,P,Q,TT),T=[S|TT].
merge_tail(X,Y,X,Y,[]).

 /* TAKE AN SPECIFIC ELEMENT AWAY FROM A LIST              */
```

```
exclude(S,[],[]).
exclude(S,[S|T],G):- exclude(S,T,G).
exclude(S,[H|T],[H|T1]):- exclude(S,T,T1).
/******************************** db1 ***********************************/

/* INPUT STRING WHICH IS ENDED BY "RETURN" */
reading_1(Z):- tab(2),get0(X),check_end(X,Z).
check_end(10,[]).
check_end(27,[]).
check_end(X,[W|W1]):- readword(X,W,X1),W ==[],restsent(X1,W1),!.
reading([W|W1]):-tab(2),get0(X),readword(X,W,X1),W ==[],restsent(X1,W1),!.
reading([]):-!.
restsent(10,[]):-!.    /*The rest of the sentence is empty */
restsent(X,[W2|W1]):-readword(X,W2,X1), W2 ==[], restsent(X1,W1).
restsent(_,[]):-!.
readword(C,W,C2):-in_sp_word(C,NewC),!,get0(C2),name(W,[NewC]).
readword(C,W,C2):-in_sp_word_1(C,NewC),get0(C3),chk_name(W,NewC,C3,C2).
chk_name(A,60,62,C2):- !,name(A,[60,62]),get0(C2).
chk_name(A,B,C,C2):- C == 61,!,name(A,[B]),C2=C.
chk_name(A,B,C,C2):- !,name(A,[B,C]),get0(C2).
readword(C,W,C2):-in_sp_word_1(C,NewC),!,get0(C1),ck_restword(C,C1,Cs,C2,W),
                  name(W,[NewC|Cs]).
ck_restword(C,61,Cs,C2,W):- get0(C2),name(W,[C,61]),write(here0).
ck_restword(60,62,Cs,C2,W):- get0(C2),name(W,[60,62]).
ck_restword(C,C1,Cs,C2,W):- restword(C1,Cs,C2),write(oi),put(C),put(C1),
                   name(W,[NewC|Cs]).
readword(C,W,C2):-in_word(C,NewC),!,get0(C1),restword(C1,Cs,C2),
                  name(W,[NewC|Cs]).
readword(10,[],_):-!.   /*The first letter is "return" */
readword(_,W,C2):-get0(C1),readword(C1,W,C2).
restword(C,[],C2):- (in_sp_word(C,NewC);in_sp_word_1(C,NewC)),!,C2=C.
restword(C,[NewC|Cs],C2):-in_word(C,NewC),!,get0(C1),restword(C1,Cs,C2).
restword(C,[],C).


/* THE COMMON CHARACTERS FOR ALL LANGUAGES */
in_word(X,Y) :- language(L),in_w(X,Y,L).
in_word(X,Y) :- in_w(X,Y,all).
in_sp_word(X,Y):- language(L),in_spw(X,Y,L).
in_sp_word_1(X,Y):- language(L),in_spw_1(X,Y,L).

/* LANGUAGE s_u IS DEFINED AS SIMPLE BASED ON UNIVERSAL RELATION
   INTERFACE*/
language(s_u).
/* These characters can appear within a word in any language */
in_spw(40,40,_).              /*  _          */
in_spw(41,41,_).
in_spw(61,61,_).
```

```
/* > < */
in_spw_1(60,60,s_u).
in_spw_1(62,62,s_u).

/* These characters can appear within a word in ahy language */
in_w(95,95,_).              /*  _          */
in_w(45,45,_).              /*  -          */

in_w(C,C,_) :- C > 96, C < 123.  /* lower case a..z */
in_w(C,C,_) :- C > 64, C < 91.   /* upper case A..Z */
in_w(C,C,_) :- C > 45, C < 58.   /* . / 0 1 2---9   */
in_w(C,C,_) :- C > 38, C < 44.   /*   ' ( ) * +     */
in_w(C,C,_) :- C > 59, C < 63.   /*   < = >         */

/* s_u (SIMPLE BASED ON UNIVERSAL RELATION INTERFACE */
in_w(58,95,s_u).            /*  : => _       */

/* These characters can appear within a word in any language */
in_w(126,126,p).
in_w(35,35,p).
in_w(27,27,p).
in_w(63,63,p).
in_w(58,95,p).
in_w(C,C,p) :- C > 64, C < 91.
/* These characters can appear within a word in ML */
in_w(37,37,m).
in_w(58,58,m).
in_w(91,123,m).
in_w(44,38,m).
in_w(C,D,m) :-C > 64, C < 91, D is C +32.
/* These characters can appear a word in EL  */
in_w(44,38,e).
in_w(58,58,e).
in_w(C,D,e) :-C>64, C < 91, D is C +32.
 /***************************** db2 *********************************/
base:- all_att,att_of_entity(At_e), asserta(att_of_entity(At_e)),
  att_of_relationship(At_r), asserta(att_of_relationship(At_r)),
  att_of_entity_1(At_e_1), asserta(att_of_entity_1(At_e_1)),
  att_of_relationship_1(At_r_1),
  asserta(att_of_relationship_1(At_r_1)),att_of_all(Att_a),
  asserta(att_of_all(Att_a)),att_of_all_1(At_a_1),
  asserta(att_of_all_1(At_a_1)),
  e_r_set(L1,L2),
  nl,nl,nl,nl,nl,nl,nl,
  write('*************************************************************'),
  nl,write('** ENTITY RELATIONSHIP MODEL BASED RELATIONAL
  DATABASE SYSTEM ***'),
  nl,write('*************************************************************')
  ,nl,nl,keyword.
```

```
keyword:- base_info1,read_more(A,B),!
        head(B,C),name(C,D),head(D,G),
        lower(G,Z),!,check(Z).
read_more(W,K):- reading(A),A == [],!,K=A.
read_more(W,K):- read_more(Q,K).
lower(X,Y):- X < 96,!,Y is X+32.
lower(X,Y):- Y is X.
check(113):- query_help(A).
check(117):- write('Undo; Please select keyword again '),nl,!,keyword.
check(105):- !,call_inputdata,!,keyword.
check(101):- halt.
check(114):- !,write('Execution Begin'),nl,write('To be finished'),
          nl,!,keyword.
check(_):- write('Please select correctly'),!,keyword.
call_inputdata:- input_data.
call_inputdata:-!.
base_info1:- write(' Input Data:i/I   Query:q/Q   Undo:u/U   Run:r/R'),
          write('    Exit:e/E'),nl,write(' =====>').

 /*The following rules testing the input string */
print_test(W):-nl,write('String of input is:'),!,print_screen(W).
print_screen([]):-!.
print_screen([W1|W2]):-W1 ==[],!,write(W1),write(' '),print_screen(W2).
 /***************************** db3 *****************************/

entity_relationship(order,offer,card1,no_role,'ord_id-c-8').
entity_relationship(customer,offer,cardn,no_role,'cust_id-c-8').
entity_relationship(product,place_on,cardn,no_role,'prod_id-c-6').
entity_relationship(order,place_on,cardn,no_role,'ord_id-c-8').
relationship_type(offer,'quantity-n-7',no_key).
entity_type(order,'ord_id-c-8',key).
entity_type(order,'ord_date-c-8',no_key).
entity_type(order,'shi_date-c-8',no_key).
entity_type(customer,'cust_id-c-8',key).
entity_type(product,'prod_id-c-6',key).
basic_rel(offer,'quantity',no_key).
basic_entity(order,'ord_id',key).
basic_entity(order,'ord_date',no_key).
basic_entity(order,'shi_date',no_key).
basic_entity(customer,'cust_id',key).
basic_entity(product,'prod_id',key).
 /********************************* db4 *********************************/

 /* INTERFACE FOR INPUT DATA OF DATABASES. THE INFORMATION IS
   STORED AS FACT OF RELATION */
input_data:- display_all_rel(Z),nl,
          write('  Please input the relation name===> '),!,reading(Relation),
          abstracting(Relation,Y),rel_exi(Y,Z).
abstracting([],[]):- !.
```

```
abstracting(A,B):- head(A,B).
rel_exi(A,B):- member(A,B),!,data_input(A).
rel_exi(A,B):- nl,write('Relation '),write(A),write(' does not exist'),nl,
        write(' select relation name again ? (y/n) '),reading(P),
        str_test(P),!,input_data.

str_test(P):- P==[].
str_test(P):- !,head(P,Q),name(Q,R),head(R,T),lower(T,S),(S ==78,
        S ==110).
 /* DISPLAY THE ATTRIBUTES OF A RELATION; ASK USER WHETHER TO INPUT
   DATA OR NOT;*/
data_input(Rel):- display_rel(Rel,Attlist),nl,write( 'input data ? (y/n) '),
          length(Attlist,N),!,
          reading(X),str_test(X),
          !,input_tuple(Rel,N).

 /* INPUT A TUPLE OF A RELATION. USER MAY INPUT DATA RECURSIVELY*/
input_tuple(Rel,N):- Y=[Rel],input_value(Rel,N,Y,N,W).
input_value(Rel,M,Y,N,W):- reading_1(Value),chk_value(Value,Rel,N),
                write(' continue? (y/n) '),
                reading(A),!,str_test(A),
                input_tuple(Rel,M).

chk_value(Value,Rel,N):- length(Value,Ln),Ln == N,
                append([Rel],Value,Y),
                G =..Y,
                tell(prog_tem),assert(G),nl,write(G),put(46),
                told,system("cat prog_tem>>prog_out").

chk_value(Value,Rel,N):- nl,write(' **** Error ****'),nl,
                write(' **** Correct number of attributes should be'),
                write(' : '),write(N).

 /* SHOW ALL RELATIONS OF THE DATABASE ON THE SCREEN */
display_all_rel(Z):- e_r_set(E,R),append(E,R,X),display_relation(X,Z),nl.
display_relation([],W):- !,write('No relation exists').
display_relation(Y,W):- nl,write(' All of the relations in the database are:'),
              nl,V=[],disp_rel(Y,V,Z,W),tab(2).
disp_rel([],L,Z,W):- !,W=L.
disp_rel([A|B],L,Z,W):-tab(2),head(A,C),write(C),!,append(L,[C],Z),
               disp_rel(B,Z,Q,W).
/******************************** db5 ********************************/

query_help(X):- e_r_set(A,B),
        nl,write(' ***   ERM-BASED DATABASE QUERY SYSTEM   ***'),
        nl,write(' L : LIST ALL ATTRIBUTES'),
        nl,write(' F : FIND COMMAND'),
        nl,write(' P : PUT COMMAND'),
        nl,write(' R : RUN/EXECUTE THE QUERY'),
```

```
            nl,write(' B : BACK TO THE PREVIOUS STATUS'),
            nl,write(' E : EXIT THE SYSTEM'),
            nl,write(' ====>'),reading(Ans),
            head(Ans,A1),name(A1,A2),head(A2,A3),lower(A3,A4),
            q_select(A4,X).
 /* L */
q_select(108,A):- att_of_all_1(Atts),print_screen(Atts),
              query_help(A).

 /* R */
q_select(114,A):- nl,write('TO BE FINISHED LATER').

 /* B */
q_select(98,A):- base.

 /* E */
q_select(101,A):- halt.

 /* F */
q_select(102,A):- univ_query(Target,Restrict,R_Summary),
             query_parsing(Target,Restrict,L),
             parsing_flag(L,Target,Restrict,R_Summary).
parsing_flag(0,Target,Restr,R_Summary):-
        concate_phrase(Restr,T1,T2,Restr_phrase),
        par_convert(Restr_phrase,R_lst),
/*       sim_par([Restr_list],R_l),R_lst=R_l],*/
        att_vs_entity(Target,E_Target),
        target_reg(E_Target,R1,T_g,L),
        p_f_1(L,Target,R1,Restr,R_Summary,Restr_phrase,R_lst,E_Target).

parsing_flag(O,T,R,RS):- nl,write('Please do the query properly'),
                 !,query_help(Y).


p_f_1(0,Target,R1,Restr,R_Summary,Restr_phrase,Restr_list,E_Target):-
        att_vs_entity(R_Summary,Entity_Restr),
        delete_duplicate(Entity_Restr,E_Restr),
/*       print_test(Target), print_test([R1]),print_test(Restr),
        print_test(Restr_phrase),
        write('restrict list is'),write(Restr_list),*/
 /*      print_test(R_Summary),print_test(E_Target),
        print_test(E_Restr),    */
        q_run(Target,R1,Restr_list,E_Target,E_Restr,R_Summary),
        !,query_help(Y).
p_f_1(O,T,R1,X1,X2,X3,X4,S):- nl,write('Please do the query properly'),
                 !,query_help(Y).

 /* START THE FIND COMMAND */
univ_query(Target,Restrict,R_Summary):- write('Find  : '),!,
```

```
                    chk_target(Target,Restrict,R_Summary).

/* CHECKING ATTRIBUTE LIST OF INPUT FROM TARGET LIST */
chk_target(Target,Restrict,R_Summary):- att_of_all_1(L),!,take_query(Target),
                    not_sublist(Target,L,A),
                    act_target(A,Restrict,L,R_Summary).

/* THE INPUT OF THE RESTRICTION LIST */
act_target([],Restrict,L,R):- write('Where :'),tab(2),
                    take_query(Restrict),
                    att_summary(Restrict,R),
                    not_sublist(R,L,A),
                    chk_restrict(A).

/* GO BACK TO THE INPUT OF THE TARGET LIST */
act_target(A,Restrict,L):- write('the attributes is not correct'),
                  print_screen(A).

/* CHECKING WHETHER RESTRICT LIST IS PROPER */
chk_restrict([]).
chk_restrict(A):- nl,write(' **** The following input list is not'),
          write(' correct : '),nl,print_screen(A).
/* P */
q_select(112,A):- write('Put :'),
          write('here is put').


/* OTHERS */
q_select(_,A):- write('Select l/L, f/F, or p/P').
take_query(List):-reading(L),query_continue(L1),append(L,L1,List).
query_continue(A):- tab(9),reading_1(B),test_list(B,A).
test_list([],[]).
test_list(B,A):- query_continue(C),append(B,C,A).
/****************************** db6 ******************************/

/* LIST OF ALL NAMES OF ALL RELATIONS OF REPRESENTATIVE DATABASE */
all_relations(A):- all_entity(B),all_relationship(C),cat_list(B,C,A).

/* LIST OF ALL NAMES OF ALL ENTITY TYPES */
all_entity(A):- e_set(L),head_list(L,A).

/* LIST OF ALL NAMES OF ALL RELATIONSHIP TYPES */
all_relationship(A):- r_set(L),head_list(L,A).

/* COLLECT ALL HEAD OF SUBLIST OF A LIST INTO A LIST */
head_list([],[]).
head_list([H|T],S):-head(H,H1),head_list(T,S1),cat_list([H1],S1,S).

/* LIST OF ALL ATTRIBUTES OF ALL RELATIONS */
```

```
att_of_all(A):- att_of_entity(B),att_of_relationship(C),
          cat_list(B,C,A).

 /* LIST OF ALL ATTRIBUTES OF ALL RELATIONS WITHOUT
   INFORMATION OF "key" OR "no_key" */
att_of_all_1(A):- att_of_entity_1(B),att_of_relationship_1(C),
          cat_list(B,C,A).

 /* LIST OF ALL ATTRIBUTES OF ALL ENTITY TYPES  */
att_of_entity(A):- e_set(L),tail_list(L,A).

 /* LIST OF ALL ATTRIBUTES OF ALL ENTITY TYPE WITHOUT INFORMATION
   "key" OR "no-key" */
att_of_entity_1(A):- att_of_entity(B),head_list(B,C),abs_list_att(C,A).

 /* LIST OF ALL ATTRIBUTES OF ALL RELATIONSHIP TYPES WITHOUT
   INFORMATION OF "key" OR "no-key"  */
att_of_relationship_1(A):- att_of_relationship(B),head_list(B,C),
                abs_list_att(C,A).

 /* LIST OF ALL ATTRIBUTES OF ALL RELATIONSHIP TYPE */
att_of_relationship(A):- r_set(L),tail_list(L,A).

 /* COLLECT ALL TAIL OF SUBLIST OF A LIST INTO A LIST */
tail_list([],[]).
tail_list([H|T],S):- tail(H,H1),tail_list(T,S1),cat_list(H1,S1,S).

 /************************ db7 ********************************/

 /* MERGE TWO PATHS INTO A COMBINED PATH: e.g. [1,2,3,d,f,g,p,q,r]
   AND [1,2,3,v,p,q,r] WILL BE MERGED INTO [1,2,3,[[f,g],[v]],p,q,r] */
merge(A,B,C):- merge_head(A,B,H,T1,T2),merge_tail(T1,T2,H1,H2,T),
         rev(T,Ts),append(H,[[H1|[H2]]],S),append(S,Ts,C).

 /* MERGER THE SAME HEADS OF TWO LISTS */
merge_head([X|Y1],[X|Y2],[X|H0],P,Q):- merge_head(Y1,Y2,H0,P,Q).
merge_head(U,V,[],U,V).

 /* MERGE THE SAME TAILS OF TWO LISTS */
merge_tail(T1,T2,P,Q,T):- abstract_tail(S,T1,H1),
                abstract_tail(S,T2,H2),
                !,merge_tail(H1,H2,P,Q,TT),T=[S|TT].
merge_tail(X,Y,X,Y,[]).
 /************************** db8 ****************************/

paths(X,Y,L1):- setof(L,go(X,Y,L),L1).
go(Start,Dest,Route) :- go0(Start,Dest,[],R),rev(R,Route).

go0(X,X,T,[X|T]).
```

```
go0(Place,Y,T,R) :- legalnode(Place,T,Next),
            go0(Next,Y,[Place|T],R).

legalnode(X,Trail,Y) :- (a(X,Y) ; a(Y,X)),not(member(Y,Trail)).


a(X,Y) :- entity_relationship(X,Y,A,B,C).
/************************** db0 ********************************/


 /* COLLECT ATTRIBUTES OF ENTITY TYPE AND RELATIONSHIP TYPE */
e_r_set(L1,L2):-(e_set(L1);true),(r_set(L2),true),asserta(e_set(L1)),
          asserta(r_set(L2)).

 /* COLLECTION OF ATTRIBUTES AND ROLES INTO A SUBSET OF LIST OF
    ALL ENTITY TYPES */
e_set(L1):- setof([X|L],set_entity(X,L),L1).

 /* COLLECTION OF ATTRIBUTES AND ROLES INTO A SUBSET OF LIST OF ALL
    RELATIONSHIP TYPES */
r_set(L1):- setof([X|L],set_relationship(X,L),L1).

 /* THE RULE COLLECT ATTRIBUTES AND ROLES OF SAME ENTITY TYPE
    INTO A LIST */
set_entity(X,L):- setof([Y|[Z]],entity_type(X,Y,Z),L).

 /* THE RULE COLLECT ATTRIBUTES AND ROLES OF SAME RELATIONSHIP TYPE
    INTO A LIST */
set_relationship(X,L):- setof([Y|[Z]],relationship_type(X,Y,Z),L).

 /*ABSTRACT THE ATTRIBUTE LIST FROM A SPECIFIC RELATION NAME */
take_rel_name([],Rel,[]) :- !.
take_rel_name([A|B],Rel,Z):- head(A,X), X==Rel,!,tail(A,Z).
take_rel_name([A|B],Rel,Y):- !,take_rel_name(B,Rel,Y).

 /* 1.CHECKI WHETHER THE RELATION IS IN AN ENTITY LIST OR IN A
    RELATIONSHIP LIST. 2.RETURN THE ATTRIBUTE LIST OF THE
    RELATION */
check_rel(Relation,Attlist):- e_r_set(E,R),append(E,R,X),
                  take_rel_name(X,Relation,Attlist).

 /* INPUT DATA OF A RELATION CORRESPONDING TO THE ATTRIBUTE LIST */
display_tuple([]):- !.
display_tuple([A|B]):- head(A,X),write(X),tab(2),!,display_tuple(B).

 /* DISPLAY ALL OF THE ATTRIBUTES OF A RELATION */
display_rel(X,Attlist):- tab(3),check_rel(X,Attlist),display_tuple(Attlist).

 /*ABSTRACT NAMES OF ATTRIBUTES: e.g. abcd-c-8 => abcd */
```

```
abstract_name(X,X1):- name(X,Y),abs_name(Y,Y1),name(X1,Y1).
abs_name([A|B],[]):- A == 45.
abs_name([A|B],[A]):- B==[].
abs_name([A|B],[A|W]):- abs_name(B,W).

 /* ABSTRACT WHOLE NAMES OF ATTRIBUTE LIST */
abs_list_att([A|B],[A1]):- B == [],!,abstract_name(A,A1).
abs_list_att([A|B],[A1|W]):- abstract_name(A,A1),abs_list_att(B,W).



 /* AN ABSTRACTION LIST OF ATTRIBUTES FROM RESTRICTION LIST */
att_summary([],P):- P=[].
att_summary([A|B],[A|W]):- not(member_predicate(A)),!,
                  att_summary(B,W).
att_summary([A|B],P):- member_comp(A),B == [],!,tail(B,B1),att_summary(B1,P).
att_summary([A|B],P):- !,att_summary(B,P).

 /* INPUT A: AN ATTRIBUTE LIST; OUTPUT B: AN ENTITY TYPE LIST.
   (FIND THE ENTITY TYPE OF EACH ATTRIBUTE AND CONCATENATE THEM
   IN A LIST */
att_vs_entity([],[]).
att_vs_entity([A|T],[B|W]) :- not(member(A,T)),
                     (basic_entity(B,A,P); basic_rel(B,A,P1)),!,
                     att_vs_entity(T,W).
att_vs_entity([A|T],W):- att_vs_entity(T,W).

 /*DELETE REDUNDANT MEMBER OF A LIST. A: INPUT LIST; B: OUTPUT
  LIST WITHOUT DUPLICATE ELEMENT                    */
delete_duplicate([],[]).
delete_duplicate([A|T],[A|W]) :- not(member(A,T)),delete_duplicate(T,W).
delete_duplicate([A|T],W) :- delete_duplicate(T,W).
 /*****************************db_1*****************************/
 /* ADDS ATTRIBUTES TO THE SUITABLE RELATIONS */
all_att:-(add_attribute1(relationship_type(X,Y,key));true),
      (add_attribute2(entity_type(A,B,no_key));true),
      all_relation,rel_generate.

 /*ADDS THE ATTRIBUTES WHICH CONSTITUTE THE PRIMARY KEY TO THE
  RELATION OF n:m RELATIONSHIP (ADD KEYS TO THE RELATIONSHIP) */
add_attribute1(relationship_type(Y,V,key)):- entity_relationship(X,Y,cardn,U,V),
                        entity_relationship(Z,Y,cardn,T,W),
                        not(relationship_type(Y,V,key)),
                        assert(relationship_type(Y,V,key)),
                        fail.

 /*ADDS THE ATTRIBUTES WHICH CONSTITUTE THE PRIMARY KEY TO THE
  RELATION OF 1:m RELATIONSHIP (ADD KEYS TO THE RELATIONSHIP) */
add_attribute1(relationship_type(Y,V,key)):- entity_relationship(X,Y,card1,U,V),
```

```
                    entity_relationship(Z,Y,cardn,T,W),
                    not(relationship_type(Y,V,key)),
                    assert(relationship_type(Y,V,key)),
                    fail.

/*ADDS THE ATTRIBUTES OF 1:m RELATION  (ADD KEY OF ENTITY TO THE
  ENTITY OF ITS DEPENDENCY OR ITS EXIST DEPENDENCY */
/*add_attribute2(entity_type(X,W,no_key)):- entity_relationship(X,Y,card1,U,V),
                    entity_relationship(Z,Y,cardn,T,W),
                    X == Z,assert(entity_type(X,W,no_key)),
                    fail.*/

/* FIND ALL INSTANCES OF DATABASE  */
all_relation:- findall(X,(entity_type(X,Y,Z);relationship_type(X,Y,Z)),S),
        delete_duplicate(S,W),assertz(rel_set(W)).

/* GENERATE SET OF RELATION THAT HAS THREE ITEMA:RELATIONAL
   NAME, ATTRIBUTE SET OF THAT RELATION, KEY OF THAT RELATION */
rel_generate:- rel_set(X),create_rel(X).

create_rel([]).
create_rel([X|R]):- rel(X,Y,Z),!,take_att_name(Y,Y1),
            !,asserta(rel_info(X,Y1,Z)),!,create_rel(R).

/* THE RELATIONS(TABLES) OF THE DATABASE SYSTEM  */
rel(X,Y,Q):- entity_type(X,Q1,key),!,abstract_name(Q1,Q),!,
        check_rel(X,Y0),!,concate_att(Y0,Y).

rel(X,Y,Q):- relationship_type(X,Q1,key),!,abstract_name(Q1,Q),!,
        check_rel(X,Y0),concate_att(Y0,Y).

take_att_name([],[]).
take_att_name([A|B],[A1|B1]):- abstract_name(A,A1),take_att_name(B,B1).

concate_att([],[]).
concate_att([A|B],[A1|B1]):- head(A,A1),concate_att(B,B1).
/**************************db_2*******************************/
 /* ALL PREDICATE DEFINED IN THE SYSTEM */
predicate(X):- Y = [and,or,>,>=,<,<=,=,<>,not],name(C,[40]),
         name(D,[41]),append([C],[D],E),append(E,Y,X).

 /* CONJUNCTION LOGIC,NEGATION LOGIC, AND PARENTHESIS          */
symbol(X):- Y = [and,or,not],name(C,[40]),
       name(D,[41]),append([C],[D],E),append(E,Y,X).

/* CONJUNCTION LOGIC */
logic(X):- X = [and,or].

/* COMPARATIVE LOGIC */
```

```
comp_logic(X):- X = [>,>=,<,<=,=,<>].

 /* NEGATION */
negation(X):- X = [not].

 /*CHECKING WHETHER AN ELEMENT IS MEMBER OF "logic(defined above)"
   OR NOT */
member_logic(A):- logic(X),member(A,X).

 /*CHECKING IF AN ELEMENT IS MEMBER OF "comparative logic(defined above)" */
member_comp(A):- comp_logic(X),member(A,X).

 /* CHECKING MEMBER OF NEGAION */
member_negation(A):- negation(X),member(A,X).

 /* CHECKING MEMBER OF "symbol(defined above)"  */
member_symbol(A):- symbol(X),member(A,X).

 /* CHECKING MEMBER OF PREDICATE  */
member_predicate(A):- predicate(X),member(A,X).
/**************************db_3**********************************/

 /* THE FOLLOWING RULE CONCATENATE THE RELATIVE PHRASE OF
    RESTRICTION PART INTO S LIST */
concate_phrase([],[],[],[]).
concate_phrase([P|Q],J,W,[[P]|T]):- what_next([P|Q]),
                          (not member_comp(P),not member_negation(P))
                          ,concate_phrase(Q,J,W,T).
concate_phrase([P|Q],J,W,[Fi|T]) :- head(Q,Th),
                          member_comp(Th),
                          concate_restr(Q,J,W,S),
                          append([P],W,Fi),
                          concate_phrase(J,D,M,T).

concate_phrase(R,J,W,[W|T]) :- concate_restr(R,J,W,S),!,
                       concate_phrase(J,D,M,T).

 /* CONCATENATE THE REMAINING OF ">=,<>,<=,=,not" TO THE LIST */
concate_restr([A|B],G,[A|[W]],S) :- (member_comp(A); member_negation(A)),!,
                         head(B,W),tail(B,G).

 /*CHECKING NEXT ELEMENT IS NOT AN ELEMENT OF A LIST:
   (">=,<>,<=,=").   */
what_next([]).
what_next([A|B]) :- B == [].
what_next([A|B]) :- head(B,B1),!,(not member_comp(B1)).

 /* PARSE BOTH TARGET PART AND RESTRICT PART */
query_parsing(Target,R_List,L) :- !,parsing_target(Target),
```

```
                    parsing_restr(R_List,L).

/* PARSING OF THE TARGET PART */
parsing_target(Target).


/* PARSING OF THE RESTRICT PART: 1. CHECKING THE PARENTHESIS.  */
parsing_restr(R_List,L) :- chk_parenthesis(R_List,0,M,L).

/* 2. CHECKING THE NUMBER OF PARENTHESIS AND MAPPING THE
   PARENTHESIS */
chk_parenthesis([],0,M,0).
chk_parenthesis([],N,M,1):- nl,write('Parenthesis doesnot match').
chk_parenthesis(X,N,M,1) :- N < 0,nl,write('Parenthesis doesnot match').
chk_parenthesis([A|B],N,M,L) :- A == '(',M is N+1,!,
                    chk_parenthesis(B,M,O,L).
chk_parenthesis([A|B],N,M,L) :- A == ')',M is N-1,
                    !,chk_parenthesis(B,M,O,L).
chk_parenthesis([A|B],N,M,L) :- !,chk_parenthesis(B,N,M,L).


/* CONVERT LIST WHICH CONTAINS [(] AND [)] INTO A NESTED LIST:
   e.g. [a,[(],b,c,[)],d,e] ==> [a,[b,c],d,e]              */
par_convert(In,Out) :-  left_par(In,N),N> 0,par_convert_1(In,Out,N,M).
par_convert(In,In).
par_convert_1([],[],N,M).
par_convert_1([H|T],[H|Tout],N,M) :- H==['('],
                    par_convert_1(T,Tout,N,M).
par_convert_1([['('|T],[Y|Tout],N,M) :- p_x(T,X,T1,P,1),
                    par_convert(X,Y),
                    par_convert(T1,Tout).

/* CONVERT THE LIST BETWEEN FIRST [(] AND LAST [)] INTO '[' AND ']' */
p_x([[')'|T],[],T,N,1).
p_x([['('|T],[['('|T1],A,N,M):-
                    P is M+1,p_x(T,T1,A,N,P).
p_x([[')'|T],[[')'|T1],A,N,M):-
                    P is M-1,p_x(T,T1,A,N,P).
p_x([H|T],[H|T1],A,N,M)  :- p_x(T,T1,A,N,M).

left_par(In,Num) :- element_count(In,0,Num).

/* COUNT THE NUMBER OF AN ELEMENT IN A LIST */
element_count([],Num,Num).
element_count([I1|I2],N,P):- I1==['('],
                    element_count(I2,N,P).
element_count([I1|I2],N,P):- Num is N+1,element_count(I2,Num,P).
/* REMOVE THE REDUNDANT PARENTHESIS */
s_par([],[]).
```

```
s_par([A|B],[A|B1]):- atomic(A),s_par(B,B1).
s_par([A|B],[A1|B1]):- s_par_1(A,A1),s_par(B,B1).

 /* REMOVE REDUNDANT PARENTHESIS FOR EACH SUB-STATEMENT */
s_par_1([],[]).
s_par_1([A],P):- not(atomic(A)),length([A],1),s_par_1(A,P).
s_par_1([],[]).
s_par_1([H|T],[H|T1]):- atomic(H),s_par_1(T,T1).
s_par_1(A,B):- s_par(A,B).


/************************db_4*****************************/

 /* ANALYZING THE LOCAL REGION OF TARGET PART          */
target_reg(T_R,R,L,C1) :- rel_exist(T_R,L,R,C1,W).

 /* WHETHER SINGLE RELATIONSHIP EXIST IN THE LOCAL REGION */
rel_exist([],L,W,0,W).
rel_exist([H|T],L,H,C,W):-entity_relationship(A,H,B,F,D),member(H,L),
                s_nd(H,L1),!,rel_exist_1(T,L1,H,C).
rel_exist([H|T],L,R,0,W):-entity_relationship(H,A,B,K,D),
                s_nd(H,L),!,rel_exist(T,L,R,C,H).
rel_exist([H|T],L,R,C,W):-entity_relationship(H,A,B,K,D),
                s_nd(H,L1),intersection(L,L1,[R]),
                s_nd(R,L2), rel_exist_1(T,L2,R,C).
rel_exist([H|T],L,R,1,W):- nl,write('The attribute set of the target part is '),
                 write('not in the proper region').

rel_exist_1([],L,R,0).
rel_exist_1([H|T],L,R,C):-member(H,L),!,rel_exist_1(T,L,R,C).
rel_exist_1([H|T],L,R,1):- nl,write('The attribute set of the target part is '),
                 write('not in the proper region').

 /* SURROUNDING REGION OF AN ENTITY TYPE OR RELATIONSHIP  */
s_nd(A,[A|L]):- findall(X,entity_relationship(A,X,B,C,D),L),L==[].
s_nd(A,[A|L]):- findall(X,entity_relationship(X,A,B,C,D),L).
 /*****************************db_5*********************************/

 /* MANIPULATION OF ACCESS PATHS OF USER'S QUERY          */
 /* Paths     : A LIST OF LISTS FROM TARGET TO EACH DESTINATION */
 /* Ps        : A LIST OF ALL ACCESS PATHS                */
 /* Main      : MAIN ACCESS PATH.                         */
 /* M_set     : MAIN SET OF ACCESS PATH.                  */
 /* M_nodes   : ALL NODES OF MAIN SET OF ACCESS PATH(M_set)  */
 /* Inter_nodes: ALL NODES THAT ARE INVOLVED IN MORE THAN TWO  */
 /* ACCESS PATHS                                          */
access_paths(Start,Nodes,Paths,Ps,Main,M_set,M_nodes,Inter_nodes):-
          find_paths(Start,Nodes,Paths,Ps),find_main(Ps,Main,Start),
          main_set(Paths,Main,M_set),main_node(M_set,[],M_nodes),
```

```
        access_nodes(Ps,Inter_nodes).

/* FIND THE ACCESS PATHS OF USER'S QUERY */
find_paths(Start,[],[],[]).
find_paths(Start,[H|T],[P1|Pt],Ps):- paths(Start,H,P1),find_paths(Start,T,Pt,P),
                        append(P1,P,Ps).

/* OPTIMIZE THE ACCESS PATH          */
access_nodes([A|[]],[]).
access_nodes([A|B],Nodes):- inter_nd(A,B,Nh),access_nodes(B,Nd),
                    cat_list(Nh,Nd,Nodes).

/* NODES OF INTERSECTION OF ONE PATH WITH THE OTHERS */
inter_nd(A,[],[]).
inter_nd(A,[B|C],L):- intersection(A,B,L1),inter_nd(A,C,L2),
                cat_list(L1,L2,L).


/* FIND THE MAIN PATH: THE PATH WHICH HAS THE MAXIMUM NUMBER OF
   INTERSECTION WITH OTHER PATHS  */
find_main(Paths,Main,S):- length(Paths,1),head(Paths,Main).
find_main(Paths,Main,S):- head(Paths,H),inter_num(S,H,Paths,0,N,L1),
                main_path(H,Paths,Paths,N,Main,L1,L2).

/* FIND THE NUMBER OF INTERSECTION NODES OF ONE PATH WITH
   ALL OTHERS */
main_path(Main,[],Paths,N,Main,L1,L2).
main_path(M1,[M1|B],Paths,N1,Mg,L1,L2):-main_path(M1,B,Paths,N1,Mg,L1,L2).
main_path(M1,[M2|B],Paths,N1,Mg,L1,L2):-inter_num(S,M2,Paths,0,N2,L2),
                        select_main(N1,M1,N2,M2,Main,No,L1,L2,Ln),
                        main_path(Main,B,Paths,No,Mg,Ln,Lb).

/* SELECT THE MAIN BY COMPARING THE INTERSECTION NUMBER        */
select_main(N1,M1,N2,M2,M2,N2,L1,L2,L2):- N2>=N1,L2>L1.
select_main(N1,M1,N2,M2,M1,N1,L1,L2,L1).

/* FIND THE TOTAL NUMBER OF INTERSECTION NODES OF A PATH WITH
   ALL OTHER PATHS OF ACCESS PATHS                    */
inter_num(S,A,[],Num,Num,Lth):-length(A,Lth).
inter_num(S,A,[A|B],N1,Num,Lth):- inter_num(S,A,B,N1,Num,Lth).
inter_num(S,A,[B|C],N1,Num,Lth):- exclude(S,B,B1),intersection(A,B1,D),
                        length(D,L),
                        N is N1+L,inter_num(S,A,C,N,Num,Lth).

/* FIND THE MAIN SET OF ACCESS PATHS                  */
main_set([],Main,[]).
main_set([H|T],Main,H):- member(Main,H).
main_set([H|T],Main,M):- main_set(T,Main,M).
```

```
/* COLLECTING ALL NODES IN THE MAIN SET INTO A LIST          */
main_node([],L1,L1).
main_node([A|B],L1,L):- cat_list(L1,A,L2),main_node(B,L2,L).


/* THE ACCESS PATHS WHICH IS NOT IN THE MAIN ACCESS SET WILL BE
   SUMMARIED TO BRANCHES OF MAIN ACCESS SET                  */
abs_branch(M_st,M_nodes,[],[]).
abs_branch(M_st,M_nodes,[M_st|T],Bs):- abs_branch(M_st,M_nodes,T,Bs).
abs_branch(M_st,M_nodes,[H|T],[H1|T1]):- s_branch(M_nodes,H,H1),
                          abs_branch(M_st,M_nodes,T,T1).

/* PROCESS THE ACCESS SETS OTHER THAN MAIN ACCESS SET       */
/* 1.TAKE REVERSE LIST OF ANY ACCESS PATH.                  */
/* 2.CONCATENATE TO THE LIST UNTIL THE NODES IS IN THE M_set. */
/* 3.TAKE THE REVERSE LIST OF THE ABSTRACTED LIST.          */
s_branch(M_nodes,[],[]).
s_branch(M_nodes,[A|B],[A2|B1]):- rev(A,A0),br_summary(A0,A1,M_nodes),
                    rev(A1,A2),s_branch(M_nodes,B,B1).

/* EACH BRANCH START FROM THE LAST NODE WHICH INTERSECT WITH
   THE MAIN ACCESS SET                                      */
br_summary([H|T],[H],M_Nds):- member(H,M_Nds).
br_summary([H|T],[H|T1],M_Nds):- br_summary(T,T1,M_Nds).
/************************** db_6 ******************************/

q_mapping(In,Out):- q_run(In,Out).

q_run([],[]).
q_run(In,O):- restr_simple(In),exclude([],In,A),q_process(A,O).
q_run(In,O):- length(In,1),head(In,H),q_run_1(H,O).
q_run([H|T],[Q1|Q2]):- q_run_1(H,P1),q_run(T,P2),delete_duplicate(P1,Q1),
          delete_duplicate(P2,Qq),con_par_1(Qq,T,Q2).

con_par_1(Q,[],Q).
con_par_1(Q,[T|T1],R):- con_par(Q,T1,R).
con_par(Q,[],[Q]).
con_par(Q,T,Q).

q_run_1([],[]).
q_run_1(A,R):- simple_list(A),exclude([],A,A1),q_process_1(A1,R).
q_run_1(A,R):- restr_simple(A),exclude([],A,A1),q_process(A1,R).
q_run_1([H|T],[Q1|Q2]):- q_run_1(H,P1),q_run_1(T,P2), delete_duplicate(P1,Q1),
                delete_duplicate(P2,Q2),con_par_1(Qq,T,Q2).


/* CHECK WHETHER A LIST IS SIMPLE(THAT DOES NOT CONTAIN
   ANOTHER LIST */
simple_list([]).
```

```
simple_list([H|T]):- !,atomic(H),simple_list(T).
 /* CHECK WHETHER A LIST IS A SIMPLE TYPE (PREVIOUS RULE)    */
restr_simple([]).
restr_simple([A|B]):- simple_list(A),restr_simple(B).

 /* THE INPUT IS A LIST OF SIMPLE LIST: e.g. [[a,b,..],[c..d],[..],..] */
q_process([],[]).
q_process(W,L):- head(W,W1),head(W1,E),att_vs_entity([E],E1),
          head(E1,A),rel_res(A,B,W,L0),path_process(L0,A,L).

 /* THE INPUT IS A SIMPLE LIST: e.g. [a,b,.....],WHERE a,b,... ARE ATOMIC */
q_process_1([],[]).
q_process_1(W,L):- head(W,E),att_vs_entity([E],E1),head(E1,A),
           rel_res(A,B,[W],L0),path_process(L0,A,L).

q_execute([],[]).
q_ececute(A,R):- write(A).


 /* FIND THE KEY OF A INSTANCE              */
which_key(A,X):- relationship_type(A,X,key).
which_key(A,X):- entity_type(A,X,key).

 /* CONCATENATE ALL ATTRIBUTES OF A RELATION TO A LIST WITHOUT
   DDL PART */
rel_res_1(A,At,L):- check_rel(A,Att),abs_tuple(Att,At1),
             abs_list_att(At1,At),length(At,L).
abs_tuple([],[]).
abs_tuple([A|B],[X|Y]):- head(A,X),abs_tuple(B,Y).



 /* FROM THE RESTRICTION PART OF AN INSTANCE, FIND A KEY LIST
   OF THE INSTANCE*/
rel_res(A,B,With,Lst) :- rel_res_1(A,B,L),var_list(L,S),append([A],S,S1),
                which_key(A,K1),abstract_name(K1,K),
                search_tup(A,B,S1,With,Lst,K).

 /* SEARCHING FOR THE SUITABLE TUPLE AND DOING THE PROJECTION ON
   THE KEY */
search_tup(A,B,S1,W,L,K):- S2=..S1,findall(X,(with_restr(S2,X,W,A,B,K,O)),L).


 /*BEGIN THE EXECUTION OF RESTRICTION PART                  */
with_restr(S2,V,W,A,B,K,O):- call(S2),att_order(B,K,O,1),
                  arg(O,S2,V),map_with(W,S2,X,B).
map_with([],S2,Y,B).
map_with([H|T],S2,Val,B):- arg_n(H,Y,3,1),arg_n(H,Op,2,1),arg_n(H,Att,1,1),
                att_order(B,Att,Ord,1),arg(Ord,S2,Val),
```

```
                is_value(Val,Op,Y),nl,map_with(T,S2,X,B).

 /* FIND THE ORDER (POSITION) OF AN ATTRIBUTE IN A RELATION */
att_order([A|B],Att,P,P):- A==Att.
att_order([A|B],Att,Ord,P):- Q is P+1,att_order(B,Att,Ord,Q).


 /* FIND THE Nth ARGUMENT OF A LIST                    */
arg_n([A|B],A,P,P).
arg_n([A|B],Att,Ord,P):- Q is P+1,arg_n(B,Att,Ord,Q).

 /* CHECK WHETHER THE VALUE IS PROPER */
is_value(Value,'>=',Y):- Value >= Y.
is_value(Value,'<=',Y):- Value =< Y.
is_value(Value,'<>',Y):- Value == Y.
is_value(Value,'=',Y):-  Value == Y.
is_value(Value,'>',Y):- Value > Y.
is_value(Value,'<',Y):- Value < Y.

 /* CREATE A LIST OF VARIABLE WITH LENGTH N          */
var_list(0,[]).
var_list(L,[H|T]):- P is L -1,var_list(P,T).
rel_res_1(A,At,L):- check_rel(A,Att),abs_tuple(Att,At1),
                abs_list_att(At1,At).

 /* THE FUNCTION OF ER-SEMIJOIN IS PASSING A SET OF PARAMETER OF
    LOCAL REGION ON ERM AND GET A LIST OF PROJECTION OF A SET OF
    KEY:
    1: INPUT ENTITY TYPES AND RELATIONSHIP TYPE OF A LOCAL REGION.
    2: ENTER A SET OF KEY OF ENTITY TYPE AT ONE SIDE OF LOCAL REGION.
    3: PROJECT A SET OF KEY OF ENTITY TYPE AT ANOTHER SIDE OF LOCAL
       REGION.*/
er_semi_join(Ent1,Rel,Ent2,Att1,Att2):- entity_type(Ent1,K11,key),
                    entity_type(Ent2,K21,key), abstract_name(K11,K1),
                    abstract_name(K21,K2), rel_res_1(Rel,Att_set,L),
                    var_list(L,R2), append([Rel],R2,R1),
                    R=..R1, att_order(Att_set,K1,O1,1),
                    att_order(Att_set,K2,O2,1),
                    findall(Attri,in_key_set(R,O1,O2,Attri,Att1),Att2).

 /* PROJECTION ON RELATIONSHIP SET TO GET A SET OF KEY FROM
    ANOTHER SET OF KEY                              */
in_key_set(R,O1,O2,Attri,Att1):- call(R),arg(O1,R,Val),member(Val,Att1),
                        arg(O2,R,Attri).

 /************************** db_7 ******************************/

 /* FIND ACCESS PATHS OF USER'S QUERY, DOING THE ER-SEMIJOIN
    OPERATION UNTIL IT MEET THE TARGET REGION          */
```

```
path_process(L,P,F):- target_node(T),paths(T,P,D),p_process(D,L,F).

 /* CUT ACCESS PATHS INTO SEGMENT OF LOCAL REGION, AND OPERATE
   ER-SEMIJOIN ON THESE SEGMENTS OF LOCAL REGIONS          */
p_process([],L,[]).
p_process([[A|[]]],L,[A|[L]]).
p_process([D|Dt],L,F):- rev(D,D2),head(D2,H),
                er_process(D2,L,F1,H,K),delete_duplicate(F1,Q11),
                Q1=[K|[Q11]],p_process(Dt,L,F2),
                cat_list(Q1,Q2,O3),exclude([],O3,F).
er_process(D,L,L,Ek,Ek):- length(D,Leng),Leng < 3.
er_process(D,L,F,Es,Ek):- length(D,Leng),Leng>=3,[E1|T0]=D,[R|T1]=T0,[E2|_]=T1,
              er_semi_join(E1,R,E2,L,Att),er_process(T1,Att,F,E2,Ek).



 /* CHECK WHETHER THE TARGET PART OF USER'S QUERY IS IN A
   SINGLE INSTANCE OR NOT                  */
type_target([E],0):- length([E],1),asserta(target_node(E)).
type_target(E,1).

q_run(Target,R1,Restr_list,E_Target,E_Restr,R_Summary):-
   cat_list([R1],E_Target,E1),type_target(E1,Code),
   t_exe(Target,R1,Restr_list,E_Target,E_Restr,R_Summary,Code),
   retract(target_node(G)).


 /* PROJECTION OF TARGET REGION ON A SINGLE ENTITY TYPE */
/*t_exe(Target,R1,Restr_list,E_Target,E_Restr,R_Summary,0):-
   rm_logic(Restr_list,R_1),sim_par(R_1,Rest_1),
   q_mapping(Rest_1,Out),nl,
   write('**************************************************'),
   nl, write(Out).          */

 /* PROJECTION OF TARGET REGION ON THE RELATIONSHIP TYPE */
t_exe(Target,R1,R_list,E_Target,E_Restr,R_Summary,Q):-
   asserta(target_node(R1)),
   rm_logic(R_list,R_1),
   logic_operation(R_list,R_logic,C),
   sim_par(R_logic,Rg),
   q_mapping(R_1,O),sim_par(O,O1),q_merg(O1,Rg,Out),
   final_info(Target,R1,E_Target,Out,Result),
   write('_____'),nl,
   write(Result).

 /*REMOVE ALL THE REDUNDANT PARENTHESIS OF A LIST */
sim_par(I,O):- sim_p(I,O).

sim_p([A],B):- A=[D|E],!,sim_p(A,B).
```

```
sim_p(A,B):- !,s_parg(A,B).

s_parg([],[]).
s_parg(A,A):-simple_list(A).
s_parg([A|B],[A|B1]):- simple_list(A),s_parg(B,B1).
s_parg([A|B],[A|B1]):- atomic(A),s_parg(B,B1).
s_parg([A|B],[A1|B1]):- sim_p(A,A1),s_parg(B,B1).



/************************ db_8 ****************************/

/* DELETE LOGIC "AND"(WHICH IS THE DEFAULT), SET LOGIC "OR"
   IN THE RIGHT LEVEL OF PARENTHESIS                */
logic_operation([],[],0).
logic_operation(A,[or|B1],C):- place_logic(A,Lg,0),Lg==[or],!,level_ck(A,B1,C).
logic_operation(A,B1,C):- place_logic(A,[and],0),!, level_ck(A,B1,C).
logic_operation([A|B],B1,C):- place_logic([A|B],[or],0),!,level_ck(A,B1,C).


/* CHECK IS THERE ANOTHER LEVEL OF PARENTHESIS EXIST  */
level_ck([],[],0).
level_ck([[or]|B],B1,C):- level_ck(B,B1,C).
level_ck([[and]|B],B1,C):- level_ck(B,B1,C).
level_ck([A|B],[s|B1],C):- simple_list(A),level_ck(B,B1,C).
level_ck([A|B],[D|E],C):- logic_operation(A,D,C),level_ck(B,E,C).
level_ck([A|B],F,1).


/* AT EACH LEVEL, IF THERE ARE EAXCT ONE TYPE OF LOGIC
  (EITHER "AND" OR "OR"), THE SET THE LOGIC SIGNAL TO BE
  "0"(CORRECT), IF THERE ARE MORE THAN ONE TYPE OF LOGIC,
  THEN SET THE LOGIC SIGNAL TO BE "1"(INCORRECT,AMBIGUOUS) */
place_logic([],[and],0).
place_logic([],L,0).
place_logic([A|B],L,C):- A==[or],A==[and],place_logic(B,L,C).
place_logic([[or]|B],[or],C):- place_logic(B,[or],C).
place_logic([[and]|B],[and],C):- place_logic(B,[and],C).
place_logic(S,L,1).
q_merg(I,Ic,O):- target_node(R),q_m(R,I,Ic,O).

q_m(R,[],Ic,[]).
q_m(R,[A],Ic,[]):- atomic(A).
q_m(R,A,[or|Tc],O):- u_merg(R,A,Tc,O).
q_m(R,A,Tc,O):- int_merg(R,A,Tc,O).

u_merg(R,[],J,[]).
u_merg(R,[A],J,O):- simple_term(A),projection(R,A,O).
u_merg(R,[H|T],[s|Tc],O):- projection(R,H,O1),u_merg(R,T,Tc,O2),
```

```
                cat_list(O1,O2,O).

u_merg(R,[H|T],[Hc|Tc],O):- q_m(R,H,Hc,O1),u_merg(R,T,Tc,O2),
                cat_list(O1,O2,O).

int_merg(R,[],J,[]).
int_merg(R,A,J,O):- simple_term(A),projection(R,A,O).
int_merg(R,[H|T],[s|Tc],O):- projection(R,H,O1),int_merg(R,T,Tc,O2),
                do_inter(O1,O2,O).

int_merg(R,[H|T],[Hc|Tc],O):- q_m(R,H,Hc,O1),int_merg(R,T,Tc,O2),
                do_inter(O1,O2,O).
do_inter(O1,[],O1).
do_inter(O1,O2,O):- intersection(O1,O2,O).

 /* FOR A SIMPLE LIST, BEGINS THE STEP OF PROJECTION. IF IT'S
   NOT A SIMPLE LIST, CONTINUING THE RECURSIVE REACTION    */
q_decomp(T,Tc,O):- simple_list(T),target_node(R),projection(R,T,O).
q_decomp(T,Tc,O):- q_m(T,Tc,O).

 /* PROJECTION THE KEY SET OF RELATIONSHIP FROM ITS SURROUNDING
   ENTITY TYPES */
projection(R,[Lh|Lt],O):- entity_type(Lh,Lhk,key),abstract_name(Lhk,Key),
       rel_info(R,Att,K),length(Att,N),var_list(N,St),
       append([R],St,S1),Rs=..S1,att_order(Att,Key,Ord,1),
       head(Lt,Lg),
       findall(X,(call(Rs),Rs=..Xc,Xc=[A|X],arg(Ord,Rs,Gs),member(Gs,Lg)),O).


 /*REMOVE THE LOGIC OPERATOR FORM A LIST          */
rm_logic(In,Out):- place_logic(In,Log,C),!,rm_act(In,Log,Out).
rm_act([],P,[]).
rm_act([[or]|B],[or],B1):- !,rm_act(B,[or],B1).
rm_act([A|B],[or],[[A]|B1]):- simple_list(A),!,rm_act(B,[or],B1).
rm_act([A|B],[or],[[A1]|B1]):- !,rm_logic(A,A1),!,rm_act(B,[or],B1).
rm_act([[and]|B],[and],B1):- !,rm_act(B,[and],B1).
rm_act([A|B],[and],[A|B1]):- simple_list(A),!,rm_act(B,[and],B1).
rm_act([A|B],[and],[A1|B1]):- !,rm_logic(A,A1),!,rm_act(B,[and],B1).

simple_term([A|B]):- atomic(A),B=[B1],simple_list(B1).
 /****************************** db_9 ******************************/

/*PROJECTION THE VALUE OF TARGET PART IN THE LOCAL REGION        */
final_info(Target,R1,E_Target,Out,Result):- print_target_head(Target),nl,
     g_p(Target,E_Target,Tx,Ex,L),take_value(L,R1,Out,Result).

take_value(L,R,[],[]).
take_value(L,R,[H|T],[Res1|Res2]):- take_value_1(L,R,H,Res1),
                take_value(L,R,T,Res2).
```

```
take_value_1([],R,Ou,[]).
take_value_1([H|T],R,Ou,[O1|O2]):- [E|A]=H, rel_info(R,Att_r,Kr),
                        rel_info(E,Att_e,Ke),
                        att_order(Att_r,Ke,Ord,1),
                        arg_n(Ou,Val,Ord,1),
                        take_e_value(E,Att_e,Ke,Val,Va_z),
                        take_target_value(A,Att_e,Va_z,O1),
                        take_value_1(T,R,Ou,O2).

take_e_value(E,Att_e,Ke,Val,Va_z):- length(Att_e,N),var_list(N,List),
                      append([E],List,Lst),E_val=..Lst,
                      call(E_val),att_order(Att_e,Ke,Ok,1),
                      arg(Ok,E_val,Val),
                      E_val=..Pst,tail(Pst,Va_z).

/* TAKE THE VALUE OF ATTRIBUTE PUT INTO A LIST          */
take_target_value([],Att_e,Val_z,[]).
take_target_value([H|T],Att_e,Val,[V|V1]):- att_order(Att_e,H,Ord,1),
                              arg_n(Val,V,Ord,1),
                              take_target_value(T,Att_e,Val,V1).

 /* THE RULE GROUP ATTRIBUTES OF THE SAME RELATION INTO A LIST
   LEADED BY THE NAME OF THAT RELATION                */
g_p([],[],T,E,[]).
g_p(Ta,En,T,E,[[S|G]|G1]):- grouping(Ta,En,S,T,E,G),g_p(T,E,T1,E1,G1).

grouping([Th|Tt],[Eh|Et],Eh,T1,E1,[Th|G]):- grouping(Tt,Et,Eh,T1,E1,G).
grouping(T,E,S,T,E,[]).

/* PRINT THE HEAD ATTRIBUTE OF TARGET PART          */
print_target_head([]).
print_target_head([A|B]):- write(A),tab(2),print_target_head(B).
```

VITA

Hung-pin Chen was born in Yunlin, Taiwan, R.O.C., on August 8, 1953. He attended Chunan Elementary School and Chunan Junior High School in Chunan, Miaoli from September 1959 to July 1968. He graduated from Hsin-Chu High School in 1971. He received the B.S. degree from Tatung Institute of Technology in 1975 and M.S. degree from Cheng-Kung University in 1977. He served in the Combined Force Service of R.O.C from September 1977 to July 1979. From August 1979-1982, he taught in the Department of Chemical Engineering of Tatung Institute of Technology. In the spring of 1983, he enrolled as as graduate student of Lousiana State University. He is currently a candidate for the degree of Doctor of Philosophy with a major in Computer Science and Minor in Computer Graphics.

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Hun-ping Chen

Major Field: Computer Science

Title of Dissertation:

Query Processing on the Entity-Relationship Graph based Relational Database Systems

Approved:

Peter P. Chen
Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

1. Peter P. Chen
2. John [illegible]
3. [illegible]
4. S. [illegible]
5. Leslie Jones
6. Robert V. Perlis

Date of Examination:

Dec. 11, 1987