A FILE INTERFACE AND DATA DICTIONARY FOR THE ECRDBS HIGH LEVEL DATABASE SYSTEM

A Thesis

Presented to

the Faculty of the Department of Computer Science

College of Natural Sciences and Mathematics

University of Houston, University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Yu-Tung Wu

August, 1987

ACKNOWLEDGEMENT

I would like to thank my research advisor, Dr. Ramez A. Elmasri for his instructive tutelage and guidance. This thesis would not have been possible without his advice.

Thanks also go to the other members of my committee, Dr. S.H. Stephan Huang and Dr. Stanley N. Deming for their helpful comments and suggestions.

Also, I wish to thank other members of Dr. R. Elmasri's research group for their useful discussion. Special thanks are given to Miss Chien-Lee Yao for using her scanner program and Mr. Xingfang Lin for his help in preparing the initial draft of the University schema definition file.

Finally, I wish to thank my parents for their financial support and tremendous encouragement. Thanks also go to my wife and daughter for their patience and understanding.

A FILE INTERFACE AND DATA DICTIONARY FOR THE ECRDBS HIGH LEVEL DATABASE SYSTEM

An Abstract of

a Thesis

Presented to

the Faculty of the Department of Computer Science

University of Houston, University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Yu-Tung Wu

August, 1987

ABSTRACT

The Entity-Category-Relationship (ECR) data model captures important descriptive semantics of a database, such as generalization hierarchies and attribute inheritance. Since no complete ECR DBMS has been implemented before, an ECR DBMS, called ECRDBS, is implemented to demonstrate the usefulness of the above data modeling concepts. This thesis describes the design and implementation of two components of ECRDBS; the file interface and the data dictionary system.

In the first part of the thesis, the indexed file organization on the VAX/VMS system is used to implement a file interface which provides basic, record at a time, file commands for storage, retrieval and updating of information.

In the second part of the thesis, a data dictionary system based on the ECR model is implemented. It is responsible for parsing the data definition statements for a particular database and creation of data dictionary files. Function procedures are provided for the user and other components of the ECRDBS system to access the information stored in the data dictionary files.

TABLE OF CONTENTS

TABLE OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Database Systems and Their Uses

Database management is one of the most important functions provided by modern computer systems. In fact, very often, it is the principal justification for acquiring a computer. Database systems have been widely used in recent years and many organizations have become dependent on the continued and successful operation of database systems. A database system is basically a computer-based recordkeeping system [Date 81]. It allows an organization's data to be processed as an integrated whole and permits users to access data more naturally. Data integration enables more information to be produced from a given amount of data and reduces data duplication. Creation of program/data independence, maintaining data consistency and better data management are also achieved through data integration.

A database management system (DBMS) is the collection of software that allows users to use or modify the integrated stored data. A major role of the DBMS is to allow the user to deal with the data in abstract terms rather than as the computer stores the data.

1.2 Current Types of Database Systems

One important aspect of a DBMS is the data model. Data models are important database design tools and are also used to categorize DBMSs. Data models have two major components, the data definition language (DDL) and the data manipulation language (DML). DDL is a vocabulary for defining the structure of the database. DML is a vocabulary for defining the processing of the database.

The current commonly used data models in commercial DBMSs are the network model, the hierarchical model, and the relational model. These models have been used in the bulk of commercial database systems.

The network model (Codasyl DBTG model) [CODA 71, CODA 78] is typically a syntactic graphic data model. The basic methods of structuring data in this model are by records and sets. Different record types are declared with a specification of the data items in each record type and what kind of values are allowed for each item. The set specifies links between record occurrences in the database state. Specifically, a set type consists of an owner record type and a member record type. A set occurrence in the database state consists of one occurrence of a record of owner type and any number of occurrences of records of member type. Fig. 1 shows a simple DBTG schema, It specifies three record types and two set types, DEP_BEL and FAC_BEL. The BELONGS record type is a member of both set types which the DEPARTMENT record type is the owner of the DEP_BEL set type and the FACULTY record type is the owner of the FAC_BEL set type. BELONGS is called a linking record type and is used to represent the many to many logical relationship (binary relationship with common cardinality M : N) between two record types.

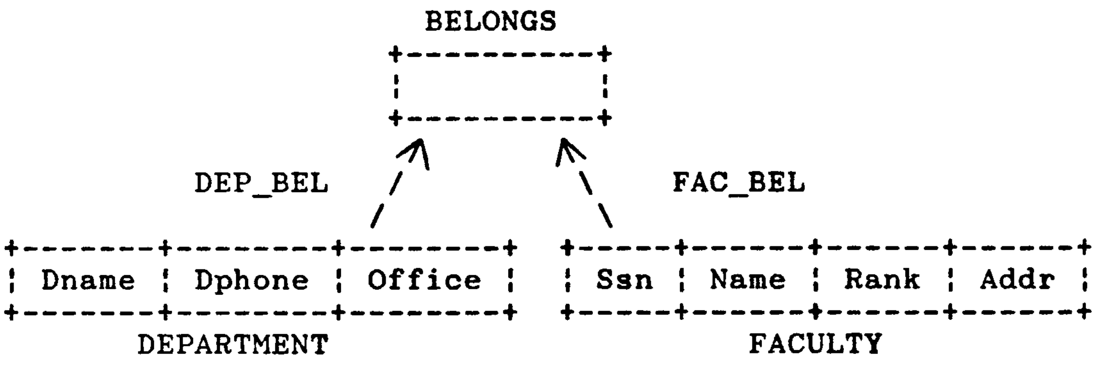


Figure 1 A DBTG schema

The basic operations used to change the database state are : store a record occurrence, delete a record occurrence, insert a record into a set occurrence, remove a record from a set occurrence and modify the values of items in a record.

Various integrity constraints can also be specified by declaring in the schema different membership classes and by specifying data items in record types as either being unique within a set occurrence or unique within all record occurrences of the record type. The membership class specifies whether or not a record occurrence of a given member record type is automatically inserted into a set occurrence of a given type and whether or not it is mandatory that once a member record occurrence is inserted into a set occurrence of a given type, the record must always be in a set occurrence of that type.

In addition to showing logical relationships between record occurrences, the links showing set membership in the network model are meant to represent access paths for the retrieval language to access the database. So DMLs of the network model are usually based on "one-record-at-a-time" access.

If links in the schema are restricted so that the database state is a set of trees of record occurrences, then this is said to be a hierarchical data model. In the hierarchical model, the record type at the top of the tree is called the root. The root, may have any number of dependents, each of these may have any number of lower-level dependents, and no dependent record occurrence can exist without its superior [Date 81]. The most widely used database system, IMS, is based on the hierarchical model. Both "one-record-at-a-time" access and higher-level languages exist for IMS.

The relational model [Codd 70] was proposed to eliminate the access dependencies present in the network and the hierarchical data models. In this data model, the database state consists of a set of named relations. Each relation is a subset of the Cartesian product space (D1 X D2 X .. X Dn) where the Di are sets of values called domains.

A relation is a set of tuples. The relations are displayed in tabular form where ordering of rows and columns are insignificant. A column (a field) is identified by the domain from which its values are drawn, and by its attribute name.

Fig. 2 shows relations corresponding to the same schema in Fig. 1. Relation BELONGS is used to represent the many to many logical relationship between relation DEPARTMENT and relation FACULTY.

DEPARTMENT

Dname	Dphone	Office
COSC	749-2761	PGH520
.	.	.
.	.	.

BELONGS

Dept_name	Fac_ssn
COSC	451769345
.	.
.	.

FACULTY

Ssn	Name	Rank	Addr
451769345	Mark William	Assoc. Prof.	11502 Leader
.	.	.	.
.	.	.	

Figure 2 Relations for a database schema

The retrieval language used with the relational data model is generally set oriented, such as Relational Calculus [Codd 71], Relational Algebra [Codd 72], and SEQUEL [Cham 76]. The update operations allowed in the relational models are the insertion and deletion of sets of tuples. These operations must be in accordance with the maintenance of the key of the relation. A key is just a column with values that each can uniquely identify a tuple in a relation.

Normal forms of relations are also introduced to prevent "update anomalies" [Codd 72a]. These anomalies correspond to database states which do not represent possible states of the abstract world being modelled. The definition of the normal forms depends on the analysis of functional dependencies. The concept of functional dependency can be easily explained, a column B is said to be functionally dependent on a column A of the same relation if and only if it is the case that for each distinct value of column A, there is necessarily exactly one distinct value of column B. Functional dependencies specify certain types of semantic features of the abstract world, but they are not sufficient to capture all of the semantic information necessary to maintain the database in a consistent state.

The relational model is much easier for the user to comprehend since it presents only a single type of structure, relation, while the network models presents several different types of structures, and the full specification of the network model is extremely complex [Date 74].

There are only two kinds of update operations, adding and deleting tuples, in the relational model, but the large number of constraints necessary to maintain the semantic integrity in a relational database could be very complex. On the other hand, in the network model, knowing what type of operation to use recognizes that certain types of predefined requirements must be satisfied by the update operation.

1.3 Semantic Data Models

The term semantic means "meaning". ANSI/SPARC [Klug 77] suggested a conceptual schema, existing between the internal schema (system or implementation's view) and the external schema (user's view), which provides semantic description of the database. So a complete data model should play two important roles, the semantic role and the representation role at the conceptual level of a DBMS architecture. That means that the data model should provide a vocabulary for expressing the meaning as well as the logical structure of the database data, and allow a straightforward translation from the conceptual schema into the physical data structures of the database [Elma 85].

However, most data models do not satisfy the two above-mentioned roles at the conceputal level with equal success. The network and the hierarchical models are oriented toward facilitating the translation from the conceptual schema into internal schema, and are also burdened with numerous construction rules and artificial constraints in the semantic role they play. In spite of theoretical foundations about functional dependencies and normalization, there is no implemented relational DBMS that uses this theory. So the weakness of the relational model is that users have to understand the implicit semantics of the relational schema in order to specify a query. That basically means the relational model fails to capture many important descriptive semantics of a database.

One of the most accepted conceptual data models has been the Entity-Relationship (ER) model [Chen 76, Chen 83]. The important contribution of the ER data model is the distinction between entities (objects) and relationships (connections among objects) among entities. The concepts of entities and relationships are closer to user's perceptions of the data than tabular forms in the relational model. Also the semantic properties such as common cardinalities (1:1, 1:N, M:N) for a relationship are clearly presented in ER model [Elma 85].

However, the ER model is not sufficient to represent some important data semantics. A lot of semantic data models have evolved [Elma 81, Elma 85] which explicitly represent some semantic concepts that are not directly represented in the ER model. These semantic concepts include, superclasses [Smit 77] (generalization categories, grouping of entities into superclasses), ISA-hierarchies [Mins 73] (subclass categories, grouping of entities into subclasses), roles [Bach 77] (grouping of entities according to the roles they play in relationships), and attribute inheritance which is related to both generalization and subclass categories. Hence an extended ER model, the Entity-Category-relationship model (ECR model) [Elma 80, Elma 81, Elma 83, Elma 84, Elma 85], was proposed to represent all the above semantic concepts. An additional construct, the concept of a category, was introduced to accomplish this. Also the attributes in the ECR model can be multi-valued, and more constraints are explicitly specified on attributes and relationships.

Apparently, the ECR model (an enhanced conceptual data model) satisfies the semantic role with great success. For the other role, the representation role, to be fulfilled, the ECR model must be translated into a representation data model such as relational model or network model.

The query language used with the ECR model is GORDAS which stands for Graph Oriented Data Selection Language [Elma 81, Elma 83, Elma 85]. GORDAS is a functional-type language [Back 78, Bune 79, Ship 81, Bune 82] and uses functional reference in specifying objects related to a given entity. The ECRDBS implemented at the University of Houston translates the GORDAS query into tree of relational algebra commands and represents the data internally as files similar to the relations used in the relational model. In this way, database applications are able to declare and reference data as viewed in the ECR model, which offers more descriptive semantics of a database.

1.4 Components of the ECRDBS

The ECRDBS software is divided into modules. Several modular subsystems are identified with well-specified interfaces among them. Basically there are six subsystems, the File system, the Data dictionary system, the Parser and translator system, the Optimization and retrieval system, the Transaction system and the Graphic query specification system.

The File system implements basic, record at a time, file commands for use by other parts of the system. The Data dictionary system is responsible for parsing the data definition and the file definition statements for a particular database schema. Thirty four function procedures are provided as the interface to the data dictionary system. The Parser and translator system parses a GORDAS query, and translates the query into an equivalent tree of relational algebra commands. The Optimization and retrieval system is used to optimize the relational algebra commands by performing transformation on the command tree and then executes the resulting relational algebra commands. The Transaction system is used to specify, compile and execute transactions on the database. The Graphical query specification system may be used to specify queries using the graphic interface instead of stating the queries directly in GORDAS. This thesis implements the File system and the Data dictionary system. Fig. 3 shows a block diagram of different components of the ECR DBMS [Elma 84].

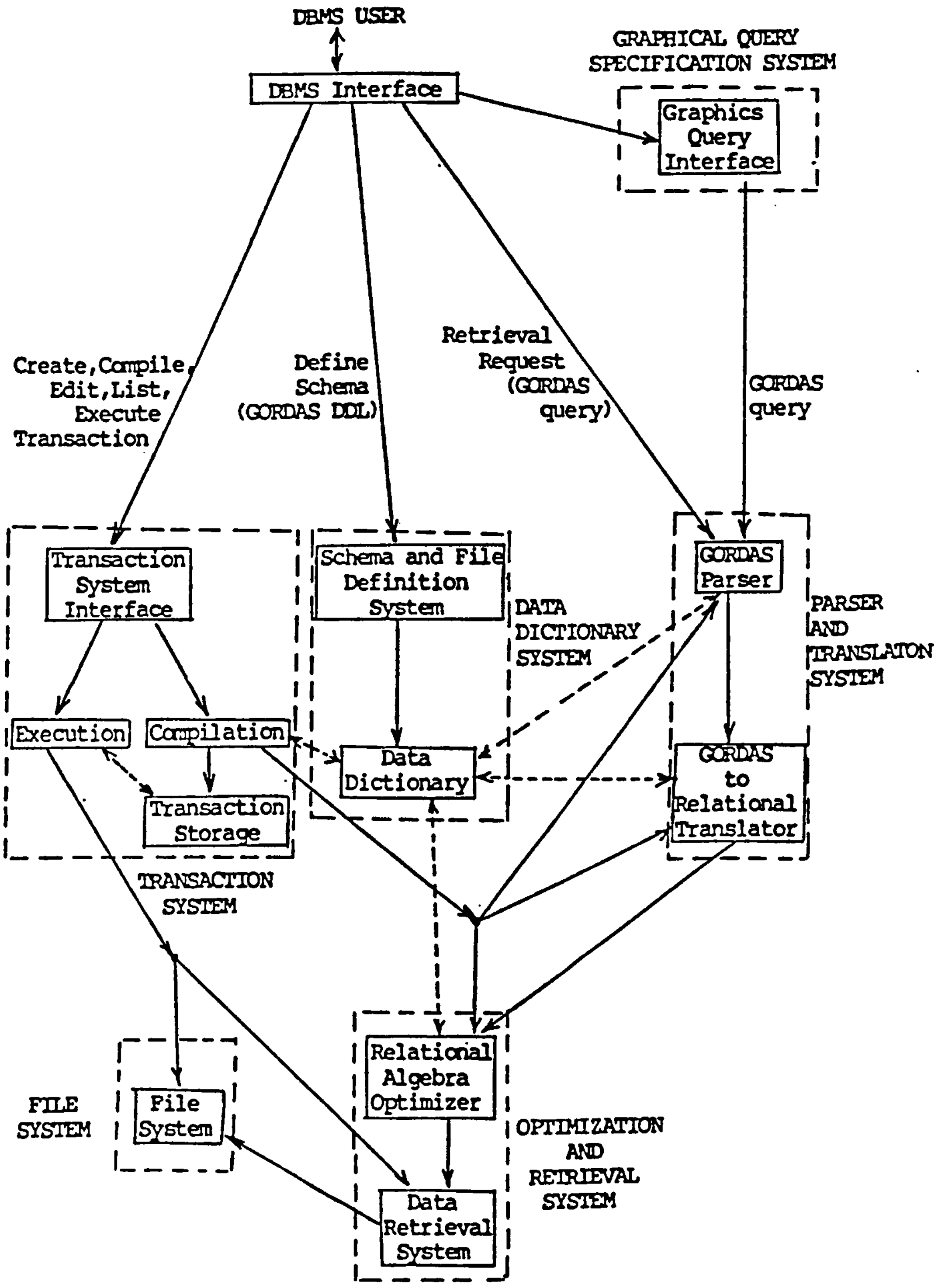

DBMS USER
DBMS Interface
GRAPHICAL QUERY SPECIFICATION SYSTEM
Graphics Query Interface
Create, Compile, Edit, List, Execute Transaction
Define Schema (GORDAS DDL)
Retrieval Request (GORDAS query)
GORDAS query
Transaction System Interface
Execution
Compilation
Transaction Storage
TRANSACTION SYSTEM
Schema and File Definition System
Data Dictionary
DATA DICTIONARY SYSTEM
GORDAS Parser
GORDAS to Relational Translator
PARSER AND TRANSLATON SYSTEM
FILE SYSTEM
File System
Relational Algebra Optimizer
Data Retrieval System
OPTIMIZATION AND RETRIEVAL SYSTEM

Figure 3 Components of the ECRDBS

CHAPTER 2

THE ENTITY-CATEGORY-RELATIONSHIP MODEL

2.1 Introduction to the ECR Model

The data model used in this thesis is the ECR model (an extended ER model) which stands for Entity-Category-Relationship model. The ECR model, like the ER model, views the world as consisting of entities and relationships among entities. Entities and relationships have attributes which provide information. Furthermore, in the ECR model, entities are not only grouped into entity types according to the similarity of basic attributes, but also are grouped into categories according to the roles which they may play in relationships. For example, categories FULLTIMEMPLOYEE (full-time employee), SCIENTIST, TECHNICIAN and CONSULTANT are defined on entity type EMPLOYEE. These are called "subclass categories". Entity types are disjoint; an entity belongs to one and only one entity type. Categories are not necessarily disjoint, because an entity may be a member of several categories. For example, an entity of type EMPLOYEE can also be a member of category FULLTIMEMPLOYEE and category SCIENTIST.

Category can also provide for grouping of dissimilar entities according to the similarity of their roles. For example, category OWNER (of a vehicle) is defined on both entity type PERSON and entity type CORPORATION. This is called 'generalization category'. Specific attributes can belong to categories if they exist. The set of attributes of a category is actually the union of the basic attributes of all entity types (some may even be categories) that the category is defined on and all attributes which have been specially defined for the category. This concept is called 'attribute inheritance'. So basically the ECR model has four components : entities, categories, attributes and relationships.

The ECR model extends the ER model in several important areas [Elma 81]. They are :

(1) The ECR model allows a direct representation for grouping of entities according to the roles the entities play in a relationship.

(2) A direct representation of subclasses (subclass categories) and superclasses (generalization categories) is possible in the ECR model.

(3) Multi-valued and composite attributes are directly represented, and are referenced just like single-valued attributes.

(4) A more complete and precise specification of the structural properties of relationships, and constraints on attributes is possible.

By incorporating all this rich semantic information, the ECR model can easily and naturally integrate user's views into a database schema, and perform crucial checks on semantic integrity constraints which are expected to hold on the data in the underlying database.

2.2 Entities, Categories, Attributes and Relationships

An entity is an object which exists in the real world and can be distinctly identified. An entity also has attributes which describe it. Entities which have similar basic attributes are classified into entity types. An entity type is similar to the entity set of the ER model [Chen 76]. Entities can also be classified into categories according to the roles they may play within relationships. The entities of a category C are specified as follows :

C = T1[S1] U T2[S2] U ... U Tn[Sn]

where each Ti is an entity type (or a category) called the defining entity type (or defining category). Each Si is a predicate called defining predicate. The predicate specifies the set of entities from Ti that belongs to C. The predicate is optional. If no predicate is given, then all entities in Ti belong to C [Elma 81].

A relationship is a connection between entities. An element of a relationship is usually called a relationship instance. A relationship R can be specified over n participants (entity types or categories) P1, ... , Pn, where n >= 2 and P1, ... , Pn are not necessarily distinct. Participants P1, ... , Pn are said to participate in the relationship R, and R is just a subset of the cartesian product P1 X ... X Pn.

Structural constraints on a relationship are used to specify the possible ways in which entities of a participant P can participate in a relationship R. The participation of an entity type (or a category) P in a relationship R can be specified by two numbers (i1, i2), where 0 <= i1 <= i2 and i2 > 0. The two numbers mean that each entity that is a member of P, must participate in at least i1, and in at most i2 relationship instances in R.

The numbers i1 and i2, also called cardinality constraints of a relationship, are a concise method of specifying structural constraints. A relationship R is total with respect to the participation of P if i1 >=1, and partial if i1 = 0. The participation is functional if i2 = 1, that means at most one appearance of each entity in

P will exist in R at a given moment. A relationship R is specific with respect to the participation of P, if the participation is total, and once an entity e from P is related to some relationship instance in R, that relationship instance cannot be removed unless the entity e itself is deleted [Elma 85].

In the GORDAS query language, a connection name (a participation name) is given for each participation in a relationship. The connection name is used as a means of functional reference to related entities. We will use the word class to stand for a set of entities (an entity type or a category) or a set of relationship instances (a relationship). When a class is used as a participant in a relationship, it means an entity type or a category only.

An attribute can be associated with either an entity type, a category or a relationship. Each attribute is defined on a value set, from which the values of the attribute are drawn. An attribute a, of an entity type T, or a category C, or a relationship R, defined on a value set V, is a function with domain T, C or R, and range POWER(V), where POWER(V) is the power set of V :

a : T -> POWER(V) or a : C -> POWER(V) or

a : R -> POWER(V)

The power set of a set is the collection of all possible subsets of that set, including the empty set. This definition of attributes allows both single-valued and multi-valued attributes to be modeled in a uniform manner.

If V is a set of single values, a is called a single attribute. If V is the cross product of several sets V1, ... , Vn (V = V1 X V2 X ... X Vn), then a is called a compound attribute. For example, if Fname (first name), Minit (middle initial) and Lname (last name) are single attributes, then Name (whole name) is a compound attribute over these three single attributes.

Constraints exist on attributes in order to represent real-word semantic constraints. The cardinality of an attribute is the number of values from value set V that can appear in a(x) for some element x (entity or relationship instance) in the domain D (entity type T, category C or relationship R) of a. The cardinality constraint for an attribute a is specified by two numbers (i1, i2), where 0 <= i1 <= i2 and i2 > 0, where i1 is the minimum number of values, and i2 is the maximum number of values from V that can appear in a(x) for some x in D. An attribute is single-valued if i2 = 1, and is multi-valued if i2 > 1. The attribute is total (no null values are allowed) if i1 >= 1, and is partial (null values allowed) if i1 = 0 [Elma 85].

An attribute is unique, if the values of that attribute can be used to uniquely identify entities of a given entity type. However, in the ECR model, an entity represents itself; it is not necessary to specify unique attributes for an entity.

2.3 ECR Diagram

The ECR diagram is an extension of the ER diagram [Chen 76]. Entity types, attributes and relationship are represented as in an ER diagram. Rectangular boxes represent entity types, diamond boxes represent relationships, ovals represent attributes. In an ECR diagram, hexagonal boxes are used to represent categories, and double ovals are used to represent multi-valued attributes. Usually ovals or double ovals are connected directly with rectangular boxes, diamond boxes or hexagonal boxes, but in the case of compound attribute, ovals or double ovals may also be connected with each other.

Partial participation is indicated by joining the participant (entity type or category) and the relationship by a single line. Total participation is indicated by a double line and specific participation is indicated by a triple line between the participant (entity type or category) and the relationship. Functional participation is represented by an arrow away from the participant (entity type or category) which has a structural constraint i2 equal to 1. The inclusion of an entity type (or a category) in a subclass category is indicated by drawing a line between the two and placing a set inclusion symbol on that line. For generalization category, the only difference is that there is a set union symbol (a letter U inside a circle) between the entity types (some may even be categories) that take part in the generalization category and the generalization category itself. This is because a generalization category is a subset of the union of several entity types or categories. At least two participation names (connection names) for each relationship are shown besides the lines indicating relationship participations. These names are used mainly for functional reference in GORDAS query.

Some conventions for ECR diagram are shown in Fig. 4. Fig. 5 demonstrates that common cardinalities (1:1, 1:N, M:N) for binary relationships can be easily deduced from the ECR diagram.

In Fig. 6, we show an ECR diagram for a Company database. EMPLOYEE is an entity type. SCIENTIST, TECHNICIAN, CONSULTANT, and FULLTIMEMPLOYEE are subclass categories defined on EMPLOYEE. Society (of SCIENTIST) and Location (of DIVISION) are two examples of multi-valued attribute. In Fig. 7, we show another ECR diagram for a University database. FACULTY and STUDENT are subclass categories defined on entity type PERSON, and GRAD_STUDENT is a subclass category further defined on category STUDENT. CURRENT_SECTION is also a subclass category which is defined on entity type SECTION. INSTR_RESEARCHER (stands for instructors and researchers) is a generalization category defined on both FACULTY and GRAD_STUDENT. Address, Name, Qtryear and Degrees are examples of compound attribute. Degrees is also a multi-valued attribute.

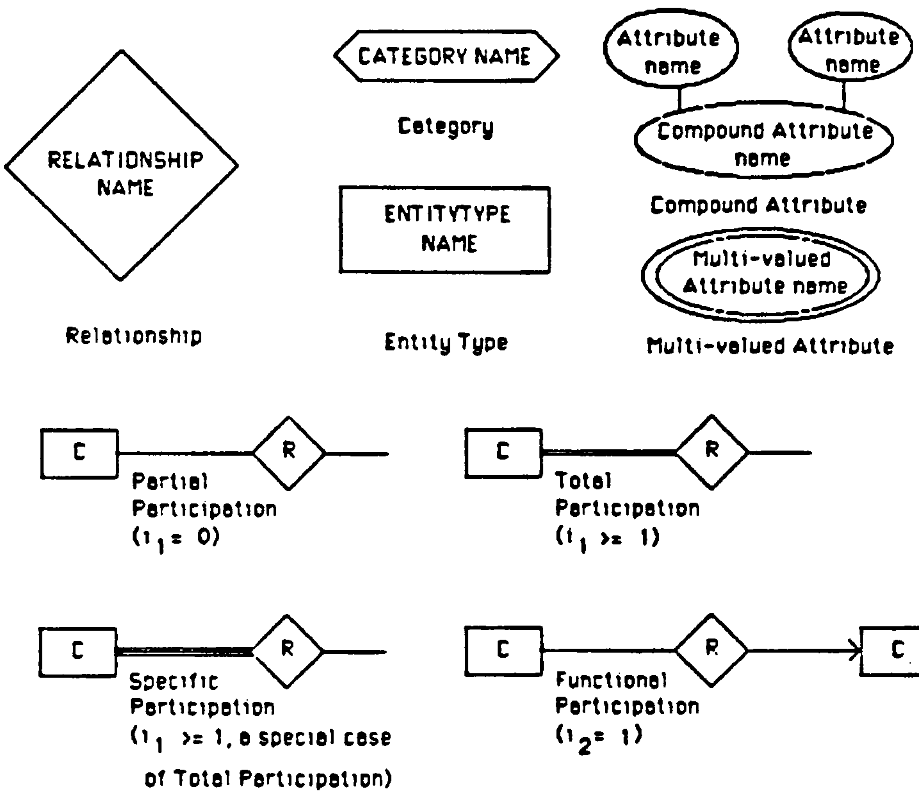

RELATIONSHIP NAME
Relationship
CATEGORY NAME
Category
ENTITYTYPE NAME
Entity Type
Attribute name
Attribute name
Compound Attribute name
Compound Attribute
Multi-valued Attribute name
Multi-valued Attribute
C
R
Partial Participation (i1 = 0)
C
R
Total Participation (i1 >= 1)
C
R
Specific Participation (i1 >= 1, a special case of Total Participation)
C
R
C
Functional Participation (i2 = 1)

Figure 4 Conventions for ECR diagram

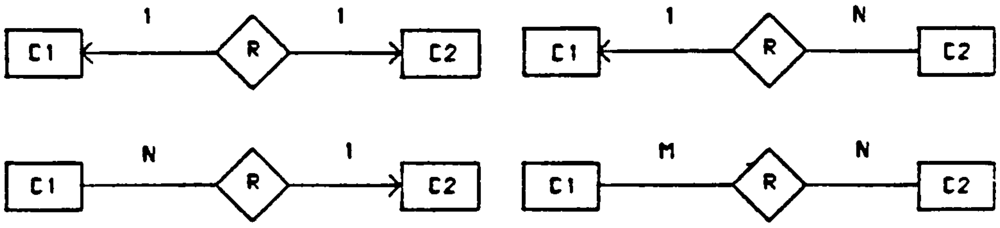


Figure 5 Common cardinalities (1:1, 1:N, M:N) for binary relationships

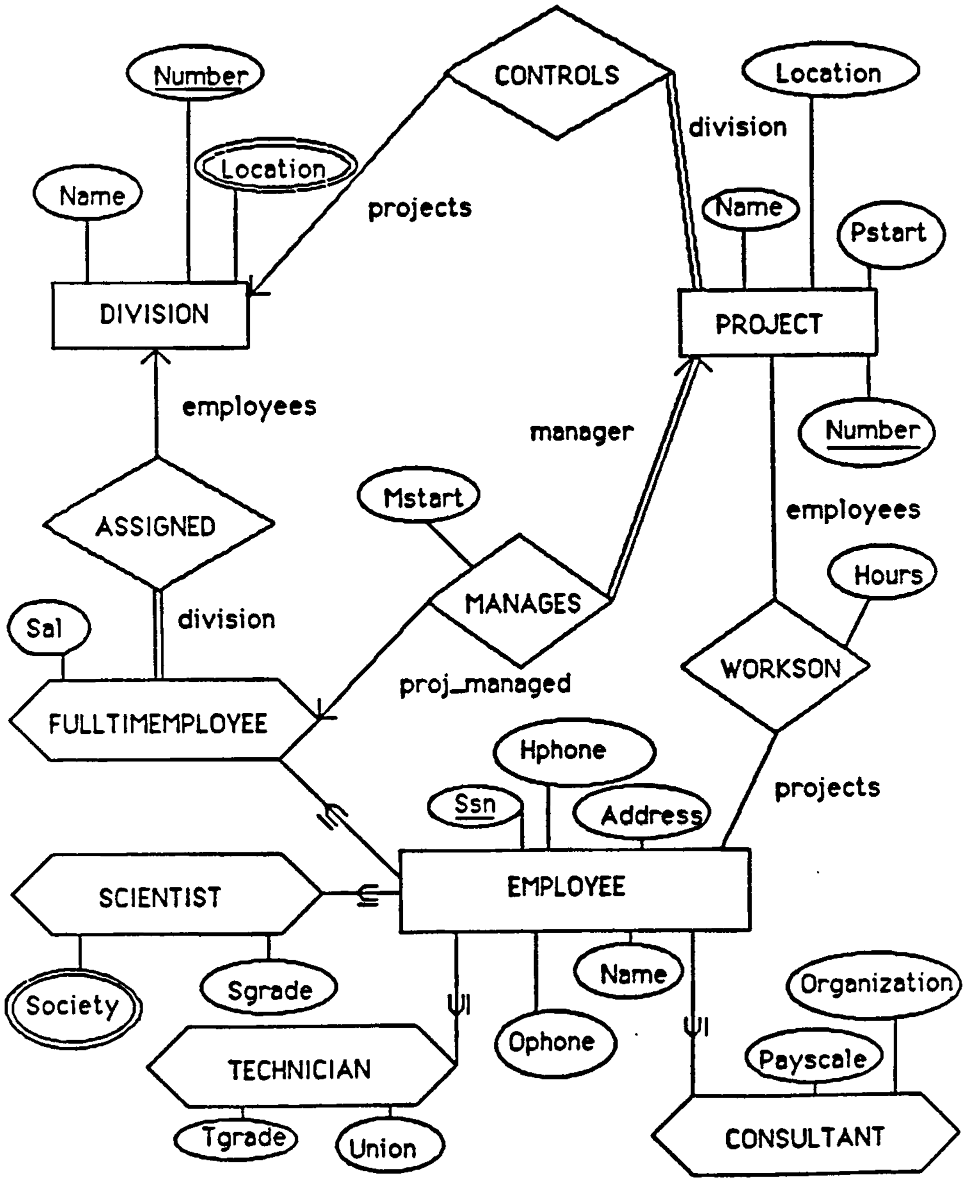


Figure 6 ECR diagram for a Company database

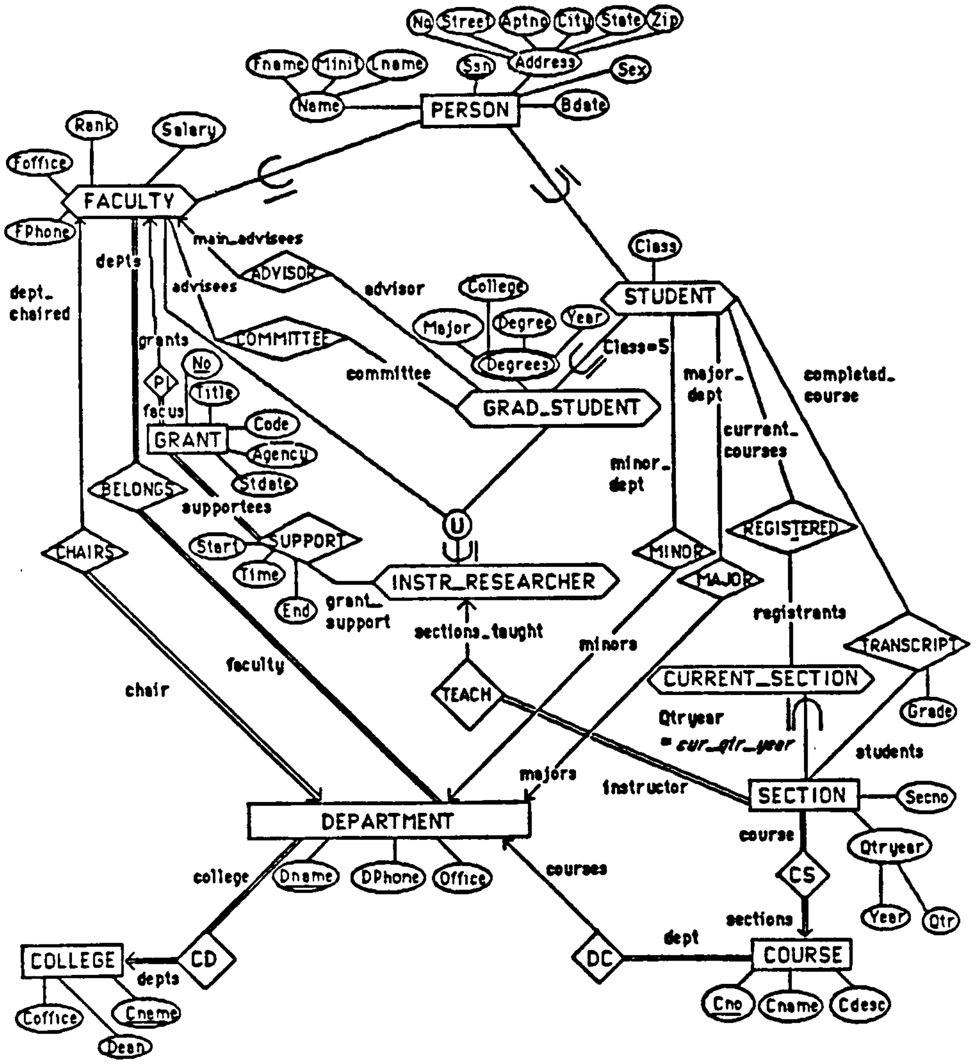


Figure 7 ECR diagram for an University database

CHAPTER 3

FILE INTERFACE AND FUNCTIONS

3.1 File System for a DBMS

The file system, a basic component of a database system, is used to manage all files in the database system. Each file consists of a number of records, and each record is composed of a number of related data fields. Some fields and some combinational fields can be keys used to uniquely identify records in a file. A file system is responsible for inserting, retrieving and updating records in files, and also supplies detailed information about fields, keys and records in files. Some file systems also provide security facilities against unauthorized access, modification or deletion of records, and offer information like the time of creation and latest modification of files.

A file system usually has a consistent physical organization. The selection of the physical organization is determined largely by the need for operational efficiency, fast response time and cost minimization. A file dictionary, a component of a file system, is also needed to store information such as file names and the format of the records comprising each file.

The basic features desired of a file system are fast access for retrieval, convenient updating, economy of storage, reliability, and maintenance of integrity of data stored in the file system.

3.2 Implementation of the File System

This thesis describes a FILE system for the ECRDBS. The File system for the ECRDBS provides basic, record at a time, file commands for storage, retrieval and updating of information stored in files. This file interface consists of fourteen procedures which offer both file level and record level operations. A File dictionary stored in the file 'FILES.DIC' contains information about fields and keys for all files of the ECRDBS.

The File system is implemented using the Indexed file organization and the Record Management Services (RMS) on the VAX/VMS system [Dec 83, Dec 84]. The File Description Language (FDL) is used to construct the description of an indexed file, and a process is spawned to create the file described by the FDL during OPEN_FILE operation (see Section 3.3). The information in the FDL file is also stored into the File dictionary.

3.2.1 File Dictionary

The File dictionary is used to keep the information describing files and their fields. All the information will be needed when retrieving, inserting or updating a record of the file. The File dictionary is stored in a file 'FILES.DIC'.

FILES.DIC

FIELD	TYPE	POSSIBLE VALUES	STARTPOS
FILETYPE	1 CHARACTER	'I', 'R', 'S'	1
SCHEMAFILENAME	35 CHARACTERS		2
FIELDNAME	16 CHARACTERS		37
COMBINATION	1 CHARACTER	'Y', 'N'	53
KEYNO	INTEGER	-1, 0, 1..254	54
STARTPOS	INTEGER	>= 1	58
LENGTH	INTEGER	>= 1	62
FIELDTYPE	1 CHARACTER	'I', 'R', 'S'	66
DUP	1 CHARACTER	'Y', 'N'	67
CHANGES	1 CHARACTER	'Y', 'N'	68
NULL_KEY	1 CHARACTER	'Y', 'N'	69

Figure 8 The structure of the file 'FILES.DIC'

Fig. 8 shows the structure of the File dictionary file 'FILES.DIC'. The SCHEMAFILENAME is just the subdirectory name concatenated with the file name. For example, [.COMPANY]EMPLOYEE is a SCHEMAFILENAME. COMPANY in this case is a subdirectory name, also a database schema name, because every ECR database schema is stored in a separate subdirectory. EMPLOYEE is the file name in this example. The actual physical file name in the VAX/VMS system is 'EMPLOYEE.DAT', because a file extension '.DAT' is added to the file name by the ECR File system. For FILETYPE (file type), 'I' stands for 'Indexed file', 'R' stands for 'Relative file', and 'S' stands for 'Sequential file'. Only Indexed file organization is implemented in this thesis. COMBINATION with 'Y' value, means the field is a combinational key field. If value of KEYNO (key number) is 0, then the field is a primary key. KEYNO with value -1, means the field is not a key. Any other value of KEYNO from 1 to 254 means the field is a secondary key. STARTPOS is the starting position of the field in a record. LENGTH is the number of bytes of the field in a record. For FIELDTYPE (field type), 'I' stands for integer, 'R' stands for real, and 'S' stands for character string. DUP, CHANGES and NULL_KEY are related to key field only. DUP with 'Y' value, means that duplicate key values are allowed for key field. CHANGES with 'Y' value, means that changes to key values are allowed. NULL_KEY with 'Y' value, means that null key values are allowed. The primary key (key 0) of the file 'FILES.DIC' is the combination key over fields SCHEMAFILENAME and FIELDNAME. A secondary key (key 1) is defined on field SCHEMAFILENAME. These two keys are used to facilitate retrieval of information from the File dictionary.

3.2.2 Data Structures Used in the File System

Before retrieving a record from an existing file, the correct position of the record in the file must be located. A condition tree, a binary tree, is used to pass the complex condition description to the COMPLEX_FIND operation (see Section 3.3), so that the record can be located.

Fig. 9 shows an example of the condition tree. In this example, Number, Salary, Bonus and Limit are all field names. The condition tree is evaluated to find the next record in the file which satisfies the complex condition. If the user just wants to get records sequentially according to the order of the primary key, a nil TREE pointer can be used.

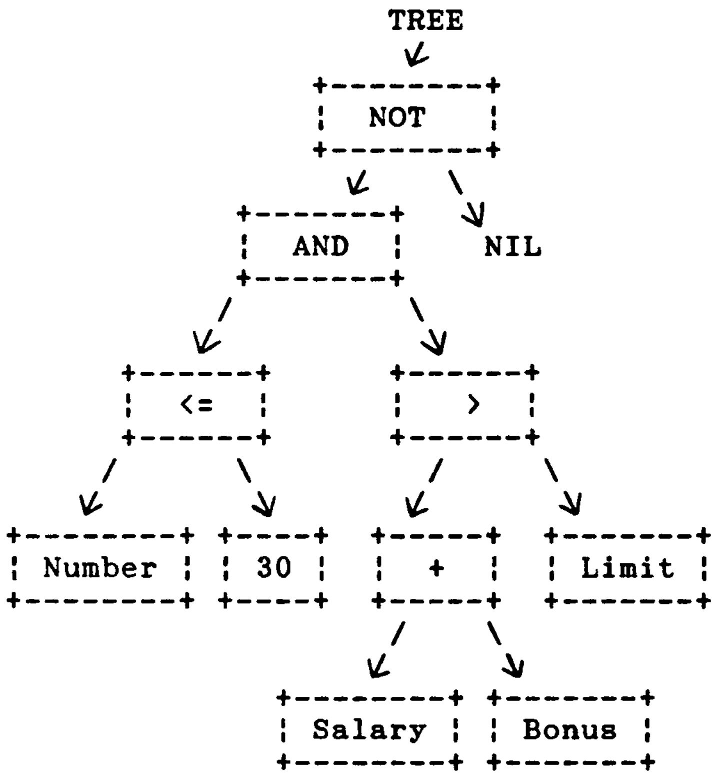


Figure 9 An example of condition tree

The data structure for each tree node is listed in the following.

```
VSTR512 = VARYING [512] OF CHAR;
STRINGTYPE = PACKED ARRAY [1..16] OF CHAR;
LEAF_TYPE = (AL, SL, IL, RL);
TREE_TYPE = ^TREE_NODE;
TREE_NODE = RECORD
              LEFT, RIGHT : TREE_PTR;
              OPERATOR : INTEGER;
              CASE LEAF : LEAF_TYPE OF
                AL : (ATTR_NAME : STRINGTYPE;
                      ATTR_TYPE : LEAF_TYPE);
                SL : (STR_CONST : VSTR512);
                IL : (INT_CONST : INTEGER);
                RL : (REAL_CONST : REAL);
             END;
```

The OPERATOR of a TREE_NODE contains an integer value which represents a logical, an arithmetic or a comparison operator. The possible values of OPERATOR are listed in the following.

1 (NOT), 2 (AND), 3 (OR), 4 (=), 5 (<>), 6 (>), 7 (>=), 8 (<), 9 (<=), 10 (+, Real or Mixed), 11 (-, Real or Mixed), 12 (*, Real or Mixed), 13 (/, Real or Mixed), 14 (+, Integer), 15 (-, Integer), 16 (*, Integer), 17 (DIV, integer), 18 (REM, Integer), 19 (MOD, Integer)

Mixed means real number mixed with integer number. The result of calculation of this kind of mixed operands is always real. If the value of OPERATOR equals to -1, then this tree node is a leaf node. A leaf node contains only field name or constant value instead of the logical, arithmetic or comparison operator. For every leaf node, leaf type can be AL, SL, IL or RL. AL stands for attribute (i.e. field), SL stands for character string constant, IL stands for integer constant, RL stands for real constant. If leaf type is AL, then ATTR_NAME (field name) must be specified, else a constant must be provided.

The only limitation to the condition tree is that there can only be one descendent tree node on the right subtree descending from the tree node with OPERATOR value between 4 and 9.

In order to maintain all opened physical data files, a linked list of logical file units is built for each ECRDBS session. Each logical file unit is composed of the following components.

(1) SCHEMAFILENAME (subdirectory name + file name)

(2) FILENO (file number, any integer number)

A physical file can be opened more than once with different file numbers and all copies of the file can stay open at the same time.

(3) STRFILE (a pointer to the actual physical file)

(4) SIMP_FIND (a boolean variable)

SIMP_FIND with a true value is used to indicate that a SIMPLE_FIND operation (see Section 3.3) has been done and is still valid, so that the SIMP_FINDNEXT operation (see Section 3.3) can then be executed on this file.

(5) KEYED (keyed access)

IF SIMP_FIND is true, then KEYED with a true value is used to indicate that the field used in the SIMPLE_FIND operation is a key and keyed access on indexed file should be done when executing a SIMP_FINDNEXT operation.

(6) FIELDNAME (field name)

If SIMP_FIND is true, then FIELDNAME stores the name of the field used in the SIMPLE_FIND operation.

(7) STARTPOS (starting position, positive integer)

If SIMPLE_FIND is true, then STARTPOS stores the starting position of the field used in SIMPLE_FIND operation.

(8) LEN (field length, positive integer)

If SIMP_FIND is true, then LEN stores the number of bytes of the field used in SIMPLE_FIND operation.

(9) COMP_OP (comparison operator : EQL, GEQ, GTR, NEQ, LTH, LEQ)

If SIMP_FIND is true, then COMP_OP stores the comparison operator used in SIMPLE_FIND operation.

(10) CONST_V (constant value)

If SIMPLE_FIND is true, the CONST_V stores the constant value used in the SIMPLE_FIND operation. A constant value is actually a variant record of type CONST_VALUE.

```
CONST_VALUE = RECORD
                   CASE TAG : TAG_TYPE OF
                     0 : (INUM : INTEGER);
                     1 : (CNUM : VSTR512);
                     2 : (RNUM : REAL);
                     3 : (INUM1, INUM2 : INTEGER);
                     4 : (TWINCNUM : STR8);
              END;

STR8 = PACKED ARRAY [1..8] of CHAR:
```

TAG with value 3 and 4 are used for combinational integer key values.

(11) COMP_FIND (a boolean variable)

COMP_FIND with a true value is used to indicate that a COMPLEX_FIND operation (see Section 3.3) has been done and is still valid. So that the COMP_FINDNEXT operation (see Section 3.3) can be then executed on this file.

(12) TREE (a condition tree pointer)

If COMP_FIND is true, then TREE is a pointer pointing to a condition tree used in the COMPLEX_FIND operation. This condition tree can then be used in COMP_FINDNEXT operation.

(13) RE_SET (reset, a boolean variable)

RE_SET with a true value is used to indicate that a RESET_FILE operation (see Section 3.3) has been done on this file, and the previous SIMPLE_FIND or COMPLEX_FIND operation is now invalid. No SIMP_FINDNEXT or COMP_FINDNEXT operation can be executed at this point.

(14) UPDATE_BUF (update buffer, of TYPE VSTR512)

The UPDATE_BUF is used to store the content of the record which has just been modified by MODIFY_RECORD operation (see Section 3.3). This buffer is used for comparison purpose, so that all records in a file satisfying a specific condition can be modified using MODIFY_RECORD operation, and no record will be modified more than once with the same condition.

In order to save space in a file and make the implementation of the File system using PASCAL language possible, all data are stored in the file as if they were character strings. Two kinds of variant records are used to code integers or reals as character strings before they are written to a file. This data is actually in integer or real (binary form). These two kinds of variant records are listed in the following. By setting TAG to 1 in VAR_RECORD, any integer or real number can be stored as four bytes into a file. The TWIN_RECORD is used for the same purpose but is for combination integer key field only.

```
VAR_RECORD = RECORD
               CASE TAG : TAG_TYPE OF
                 0 : (INUM : INTEGER);
                 1 : (CNUM : STR4);
                 2 : (RNUM : REAL);
             END;

TWIN_INTRECORD = RECORD
                   CASE TAG : TAG_TYPE OF
                     0 : (INUM1, INUM2 : INTEGER);
                     1 : (TWINCNUM : STR8);
                 END;

STR4 = PACKED ARRAY [1..4] OF CHAR;
```

3.3 File System Procedures and Interface

The ECRDBS File system provides an interface which consists of five file commands and nine record commands. The file commands are CREATE_FILE, DELETE_FILE, OPEN_FILE, CLOSE_FILE AND RESET_FILE. The record commands are INSERT_RECORD, SIMPLE_FIND, SIMP_FINDNEXT, DELETE_RECORD, READ_RECORD, MODIFY_RECORD, FINDSEQ_NEXT, COMPLEX_FIND AND COMP_FINDNEXT. The READ_RECORD, DELETE_RECORD and MODIFY_RECORD commands can only be executed on the current record of a file. The current record must first be located by one of the following commands : SIMPLE_FIND, SIMP_FINDNEXT, COMPLEX_FIND, COMP_FINDNEXT, RESET_FILE, FINDSEQ_NEXT. These fourteen operations of the file interface are listed in the following.

(1) CREATE_FILE

Purpose : Create an indexed file according to the file specification given.

Input : FILETYPE (filetype 'I', 'R' or 'S')

FILENAME (file name, of type PACKED ARRAY [1..16] OF CHAR)

FIELDS (a pointer pointing to a linked list of records of type FIELDDEF)

```
PTR_FIELDDEF = ^FIELDDEF;
```

```
FIELDDEF = RECORD
             FIELDNAME : STRINGTYPE;
             COMBINATION : CHAR;
             KEYNO : INTEGER;
             STARTPOS : INTEGER;
             LENGTH : INTEGER;
             FIELDTYPE : CHAR;
             DUP : CHAR;
             CHANGES : CHAR;
             NULL_KEY : CHAR;
             NEXT : PTR_FIELDDEF;
           END;
```

Record of type FIELDDEF (field definition) contains the same information as described in the File dictionary.

Output : None

Major actions taken :

(a) Make sure all field definitions are correct.

(b) Write field definitions for the given file into the File dictionary 'FILES.DIC'.

(c) Create FDL file according to field definitions.

(d) Spawn a process to create an indexed file according to the FDL file specification.

(2) DELETE_FILE

Purpose : Delete an indexed file.

Input : FILENAME (file name)

Output : None

Major actions taken :

(a) Make sure all copies of the file (with different file numbers, of course) are closed

(b) Remove all records in File dictionary 'FILES.DIC' that are related to the given file.

(c) Spawn a process to delete the indexed file.

(3) OPEN_FILE

Purpose : Open an indexed file in the File system.

Input : FILENAME (file name)

FILENO (file number)

Output : None

Major actions taken :

(a) Make sure that the File dictionary already contains information about the given file.

(b) Set up a logical file unit, and put it into the linked list of logical file units.

(c) Open the indexed file in the VAX/VMS system. (specifying that the file is for sharing.)

(4) CLOSE_FILE

Purpose : Close an indexed file in the File system.

Input : FILENAME (file name)

FILENO (File number)

Output : None

Major actions taken :

(a) Make sure that a logical file unit already existing matches the file name and file number given.

(b) Close the indexed file in the VAX/VMS system.

(c) Delete the logical file unit from the linked list of logical file units.

(5) RESET_FILE

Purpose : Reset an indexed file for sequential access according to the order of the given key number.

Input : FILENAME (filename)
FILENO (file number)
KEYNO (key number)

Output : FOUND (A true value indicates that the first record has been located.)

Major actions taken :

(a) Make sure that the File dictionary contains information about the given file.

(b) Reset the file according to the order of the given key number.

(c) Set FOUND to true, if the file is not empty.

(6) INSERT_RECORD

Purpose : Insert a record into the given indexed file.

Input : FILENAME (file name)
FILENO (file number)
FIELDLIST (a pointer to a linked list of records of

```
              type FIELD_UNIT)

        PTR_FIELDUNIT = ^FIELD_UNIT;
        FIELD_UNIT = RECORD
                       FIELDNAME : STRINGTYPE;
                       NEXT : PTR_FIELDUNIT;
                       CASE TAG : TAG_TYPE OF
                         0 : (INUM : INTEGER);
                         1 : (CNUM : VSTR512);
                         2 : (RNUM : REAL);
                         3 : (INUM1, INUM2 : INTEGER);
                         4 : (TWINCNUM : STR8);
                      END;
```

Each record of type FIELDUNIT contains a field name and a field value.

Output : INSERT_STATUS (an integer)

(A value of 0 indicates that the INSERT_RECORD operation is successful. A value of 5 indicates that the duplication key value error occurs. A value of -1 indicates other error occurs.)

Major actions taken :

(a) Make sure that the file has already been opened by the File system.

(b) Get field definitions from the File dictionary.

(c) Put field values from FIELD_UNIT records into a buffer. For surrogate key field (see Section 4.2.5), an unique integer number is inserted automatically.

(d) Insert that buffer (as a record) into the file.

(e) Return INSERT_STATUS to indicate whether the

INSERT_RECORD operation is successful or not.

(7) SIMPLE_FIND

Purpose : Use the field name, comparison operator and constant value given to locate a record in the file satisfying this simple condition.

Input : FILENAME (file name)

FILENO (file number)

FIELDNAME (field name)

COMP_OP (comparison operator : EQL, GEQ, GTR, NEQ, LTH, LEQ)

CONST_V (constant value, see Section 3.2.2)

Output : FOUND (A true value indicates that a record has been located.)

Major actions taken :

(a) Make sure that the file has already been opened by the File system.

(b) Get the field definition information for the given field from the File dictionary.

(c) Find the first record that satisfies this simple condition.

(d) Record the information into the corresponding logical file unit, so that SIMP_FINDNEXT operation can use this information.

(e) Set FOUND to true if a record has been located.

(8) SIMP_FINDNEXT

Purpose : Locate the next record satisfying the same condition as specified in the previous SIMPLE_FIND operation.

Input : FILENAME (file name)
FILENO (file number)

Output : FOUND (A true value indicates that a record has been located.)

Major actions taken :

(a) Make sure that the file has already been opened by the File system.

(b) Get information about previous SIMPLE_FIND operation from the corresponding logical file unit, then locate the next record satisfying the same condition.

(c) Set FOUND to true if a record has been located.

(9) DELETE_RECORD

Purpose : Delete the current record from the given file.

Input : FILENAME (file name)
FILENO (file number)

Output : None

Major actions taken :

(a) Make sure that the file has already been opened by the File system, and that a current record has been specified by one of the FIND operations or RESET_FILE.

(b) Delete the current record from the given file.

(10) READ_RECORD

Purpose : Read the current record from the given file.

Input : FILENAME (file name)

FILENO (file number)

Output : FIELDLIST (a linked list of records of type FIELD_UNIT)

Each record of type FIELD_UNIT will contain field name and field value fetched from the current record.

Major actions taken :

(a) Make sure that the file has already been opened by the File system, and that a current record has been specified by one of the FIND operations or RESET_FILE.

(b) Get field definitions for the given file from the File dictionary.

(c) Get field values from the current record of the given file and store them together with field name into records of type FIELD_UNIT.

(d) Return the pointer FIELDLIST which points to a linked list of records of type FIELD_UNIT. This contains the field values of the record.

(11) MODIFY_RECORD

Purpose : Modify the current record of the given file.

Input : FILENAME (file name)

FILENO (file number)

FIELDLIST (a linked list of records of type FIELD_UNIT)

Each record of type FIELD_UNIT contains field name and field value used to modify the current record.

Output : STOP (A true value indicates that no more records in the file are to be modified.)

Major actions taken :

(a) Make sure that the file has already been opened by the File system.

(b) If the current record has not yet been modified by the same FIELDLIST, then modify the current record, else modify the next record (if any) satisfying the same condition as that of the current record.

(c) Set STOP to true, if no more records in the given file satisfy the same condition.

(12) FINDSEQ_NEXT

Purpose : Locate the next record according to the order specified by the previous RESET_FILE operation.

Input : FILENAME (file name)

FILENO (file number)

Output : FOUND (A true value indicates that a record has been located for sequential access.)

Major actions taken :

(a) Make sure that the file has already been opened by the File system and also been reset by a previous RESET_FILE operation.

(b) Locate the next record according to the order specified in the previous RESET_FILE operation.

(c) Set FOUND to true if a record has been located.

(13) COMPLEX_FIND

Purpose : Use a condition tree to locate a record in the file satisfying the complex condition described by the condition tree.

Input : FILENAME (file name)
FILENO (file number)
TREE (a pointer to a condition tree)

Output : FOUND (A true value indicates that a record has been located by the COMPLEX_FIND operation.)

Major actions taken :

(a) Make sure that the file has already been opened by the File system.

(b) Use a simple condition (a field name, a comparison operator and a constant value) existing in a subtree of the condition tree to find a record satisfying the simple condition chosen. Preference is given to a key, and the primary key is the best choice. Also, preference is given to a comparison operator among EQL, GEQ and GTR. If no record satisfies the simple condition chosen, then set FOUND to false and return to the calling program.

(c) Evaluate the condition tree. If the condition tree is evaluated to be true for the record located in (b), then set FOUND to true and record information about COMPLEX_FIND operation into the corresponding logical file unit so that COMP_FINDNEXT operation can use them, else go to (b).

(14) COMP_FINDNEXT

Purpose : Locate the next record satisfying the same complex condition as specified in the condition tree of the previous COMPLEX_FIND operation.

Input : FILENAME (file name)
FILENO (file number)

Output : FOUND (A true value indicates that a record has been located.)

Major actions taken :

(a) Make sure that the file has already been opened by the File system.

(b) Get information about last COMPLEX_FIND operation from the corresponding logical file unit, and then locate the next record satisfying the same complex condition.

(c) Set FOUND to true if a record has been located.

3.4 Limitations of the File System

Since the File system is built on top of the indexed file organization of the VAX/VMS system, there are some limitations inherited directly from the VAX/VMS Record Management Services. Other limitations are arbitrarily set to optimize the performance of the File system. All limitations are listed in the following.

(1) The maximum number of fields in a record is 50.

(2) The maximum number of bytes a field can have is 512.

(3) The maximum number of characters in a file name is 16. (The File system will automatically append a file extension .DAT to the file name.)

(4) The maximum number of characters in a field name is 16.

(5) The maximum number of characters in a schemafilename (subdirectory name + file name) is 35.

(6) The key number can be -1 (non-key) and from 0 to 254.

(7) The maximum number of files that can be simultaneously opened is 74.

(8) Key of type String cannot exceed 255 characters.

(9) Primary key (key 0) cannot have 'Y' for CHANGES or 'Y' for NULL_KEY in the file definition. However, primary

key can have 'Y' for DUPLICATES.

(10) Combination key can only be set on several contiguous string fields or two contiguous integer fields.

(11) The length of noncombinational integer or real fields is always 4.

(12) Noncombinational key can only be set on character string or integer field.

CHAPTER 4

SCHEMA DEFINITION FOR THE ECRDBS

4.1 Role of the Data Dictionary System

A data dictionary is a principal component of a database system. It handles and controls the data information. It enables management to enforce data definition standards; it supplies information about the creation, usage and relationships of data; it eliminates the data redundancy and data inconsistency; it aids the security of sensitive data definitions against unauthorized use [Chan 83].

A data dictionary provides assistance to the database administrator in cataloging and maintaining the database design, so that the data dictionary becomes one of the most important tools in database administration. An integrated data dictionary is also used by the database system software whenever the software needs information on the structure of the data. So basically it is a tool to help the database administrator, system analyst, application programmer and user to plan, control and use the data stored in the database, and hence is an essential part of a database system.

This thesis describes a data dictionary system based on the Entity-Category-Relationship model. The Data dictionary system for the ECR model is responsible for parsing the data definition and the file definition statements for a particular database schema. These statements are analyzed and checked for correctness and consistency. The information in these statements is then stored in data dictionary files for use when queries and transactions are specified and executed.

The data definition statements specify an ECR schema, the file definition statements specify the data file descriptions, as well as their correspondence to the ECR schema. Mapping ECR requests to the data files is accomplished during translation.

Because of the complexity of the ECR model (relative to the relational model), thirteen data dictionary files have been created to store and cross-reference the information. Some of the data in the Data dictionary is stored redundantly in order to facilitate retrieval of the same information based on different selection criteria. However, the redundant storage is consistent since it is always controlled by the Data dictionary system when the schema is defined.

Retrieval of information from the Data dictionary is via function procedures with appropriate parameters. Both DBMS system routines and DBMS users (using a user friendly interface) can retrieve descriptions from the Data dictionary [Elma 84].

4.2 Data Definition Language for the ECRDBS

The statement for defining the name of an ECR schema is the SCHEMA statement. To specify the constructs of an ECR schema, we have four types of DEFINE statements : DEFINE VALUESET, DEFINE ENTITYTYPE, DEFINE CATEGORY, and DEFINE RELATIONSHIP. In addition, a DEFINE FILE statement is used to specify the file structures corresponding to the ECR schema. The BNF symbols used in data definition language are :

```
[x] : x occurs 0 or 1 times.
{x} : x occurs 0 or more times.
x ¦ y : x or y.
(x¦y¦...¦z) : x or y or ... or z, used to group
                 alternatives.
'x' : x is a literal metasymbol.
```

The BNF for SCHEMA statement and DEFINE statement follows :

```
<schema def> ::= SCHEMA <schema name> <define statement>
                  {\<define statement>} END
<define statement> ::= DEFINE (<value set>¦<entity type>¦
                        <category>¦<relationship>¦<file>)
<schema name> ::= <string of characters>
<string of characters> ::= <letter or digit>
                            {<letter or digit>}
<letter or digit> ::= <letter>¦<digit>
<digit> ::= 0¦1¦2¦3¦4¦5¦6¦7¦8¦9
<letter> ::= A¦B¦C¦D¦E¦F¦G¦H¦I¦J¦K¦L¦M¦N¦O¦P¦Q¦R¦S¦T¦U¦V¦W¦X
             ¦Y¦Z¦a¦b¦c¦d¦e¦f¦g¦h¦i¦j¦k¦l¦m¦n¦o¦p¦q¦r¦s¦t¦u¦
             v¦w¦x¦y¦z¦_
```

4.2.1 Value Set Definition

A value set is defined by its name and description. The value set names must be unique because they are used to refer to the value set when an attribute is defined. The value set descriptions can be one of the following :

(1) A set of explicit values. This specification is used for a small value set with values of irregular structure.

(2) A standard data type. It can be integer, real, or string. The standard type can also be restricted to a

subrange.

(3) By reference to an attribute of some given class, such as entity type, category or relationship. The value set is defined to be the set of values that currently exists in the database for the given attribute of the given class.

BNF for value set definition :

```
<value set> ::= VALUESET <value set name> AS
                <value set description>
<value set description> ::= <explicit value set>¦
                            <standard value set>¦
                            <reference value set>
<explicit value set> ::= '{'<constant>{,<constant>}'}'
<standard value set> ::= STRING <integer>¦INTEGER [[RANGE]
                         <integer>[:<integer>]]¦REAL
                         [[INTERVAL]<real>[:<real>]]
<reference value set> ::= <attribute name> OF <class name>:
                          <low level domain>
<low level domain> ::= <standard value set>
<value set name> ::= <string of characters>
<constant> ::= <string of characters>
<attribute name> ::= <string of characters>
<class name> ::= <string of characters>
```

4.2.2 Entity type Definition

An entity type is defined by its name and attribute list. The entity type name must be unique, since it is used by the user to specify the entity type when formulating a query or transaction. Each attribute in the attribute list must have a unique name and is associated with a value set.

The cardinality of an attribute is specified by the optional integers following MIN and MAX. The default values are MIN 1 and MAX 1. The optional key word UNIQUE specifies a unique attribute, and may be used to define both single and compound attributes. MIN cardinality must be greater than or equal to 0. If MIN cardinality is equal to 0, then the attribute is partial (null values are allowed), else the attribute is total (no null values are allowed). MAX cardinality must be greater than or equal to 1. If MAX cardinality is equal to 1, the attribute is single-valued, else the attribute is multi-valued.

BNF for entity type definition :

```
<entity type> ::= ENTITYTYPE <entity type name> ATTRIBUTES
                  <attribute list>
<attribute list> ::= <constrained attribute>
                     {,<constrained attribute>}
```

```
<constrained attribute> ::= <attribute name> VALUESET
                            <value set name>[UNIQUE] [MIN
                            <integer>] [MAX <integer>]
<entity type name> ::= <string of characters>
<attribute name> ::= <string of characters>
```

4.2.3 Category Definition

A category is defined by its name, the entity type specifications and an optional attribute list. Each entity type specification is composed of entity type specs, and optional predicates (logical expression). Each entity type spec can be an entity type or a category. A predicate selects the subset of a entity type spec which specifies the members of the category. The entities in a category is defined to be the union of those subsets of entity type specs. If the optional predicate is not given, any member from the entity type spec can be a member of the category. Specific attributes for the category (if any) are defined by an attribute list.

BNF for category definition :

```
<category> ::= CATEGORY <category name> FROM
               <entity type specifications> [[SPECIFIC]
               ATTRIBUTES <attribute list>]
```

```
<entity type specifications> ::= <entity type spec>
                                 [:<predicate>]
                                 {,entity type spec>
                                 [:<predicate>]}
<entity type spec> ::= (<entity type name>¦<category name>)
<predicate> ::= <attribute name> = <constant>
<constant> ::= <string of characters>
<category name> ::= <string of characters>
<attribute name> := <string of characters>
```

4.2.4 Relationship Definition

A relationship is specified by its name, participation list and optional attribute list. The relationship name must be unique from all other relationship names, entity type names and category names in the schema.

In a relationship, there must exist at least two participations. Each participation of a category or entity type in the relationship is specified by the category name or entity type name, a pair of participation names and the structural constraints on the participation (the integers after MIN and MAX).

MIN cardinality constraint must be greater than or equal to 0. If MIN cardinality constraint is equal to 0, then the participation is partial, else the participation is total. MAX cardinality constraint must be greater than or equal to 1. If MAX cardinality constraint is equal to 1, then the participation is functional. The default values for structural constraints are MIN 0, MAX maxint (the largest integer value in PASCAL). If MIN > 0, we can also specify the DLT option. In this case if some relationship tuple specified for deletion causes some instance e in P (participant of the relationship) to be related to less than MIN cardinality tuples, then both the relationship tuple and the related instance e from P are deleted. On the other hand, if DLT is not defined, then a transaction will be rejected if it contains delete-relationship operation which violates the MIN structural constraint.

In addition to the structural constraints of a relationship, another constraint called basic relationship constraint must always hold. The basic relationship constraint specifies that each relationship tuple must relate to instances that currently exist in the entity types or categories that participate in the relationship. This constraint is implicit in the ECR model definition for relationships. In order to maintain the basic relationship const~~ant~~raint, if PD (Prohibit Deletion) is defined, then a transaction will be rejected if it contains an operation which deletes a participant instance that participates in some relationship tuple. If PD is not defined, then both the participant instance and the relationship tuples in which it participates are removed [Elma 80].

A relationship R is SPECIFIC with respect to the participation of P, if the participation is total and once an instance e from P is related to some relationship instance in R, that relationship instance can not be deleted unless e itself is deleted. If SPECIFIC is defined for a participant P, then PD (Prohibit Deletion) must not be defined on the basic relationship constraint.

BNF for relationship definition :

```
<relationship> ::= RELATIONSHIP <relationship name> FROM
                   <participation list> [ATTRIBUTES
                   <attribute list>]
<participation list> ::= <participation>, <participation>
                         {,<participation>}
<participation> ::= (<category name>¦<entity type name>)
                    [SPECIFIC] [PD]'('<participation name1>,
                    <participation name2>')'[MIN <integer>
                    [DLT]] [MAX <integer>]
<relationship name> ::= <string of characters>
<participation name1> ::= <string of characters>
```

<participation name2> ::= <string of characters>

4.2.5 Data File Definition

A data file is defined by its name, file type, corresponding class name or attribute name, and field list. The file type can be INDEX, REL (relative) or SEQ (sequential). Only INDEX file organization are implemented in this thesis. The corresponding class name can be either entity type name, category name or relationship name. The corresponding attribute must be the name of a multi-valued attribute name.

Every field is defined by field name, starting location, fieldtype, length (field length), keyno (key number), combination (combinational key), dup (duplicate key value), changes (change of key values) and null_key (null key value). Fieldtype can be I (integer), R (real) or S (character string). Keyno can be -1 (non-key), 0 (primary key) or any number between 1 and 254 (secondary key). The value for combination, dup, changes and null_key are either 'Y' or 'N'.

The CORRESPONDS clause is used to relate the field to an attribute of a class (entity type, category or relationship). Since subclass categories have no corresponding files, all corresponding fields of subclass categories are put into the file corresponding to the entity type (or category) they are defined on. So a field (defined by a SUBCATEGORY clause) is needed for each subclass category, to indicate which record in the file belong to the subcategory. On the other hand, Generalization categories have their own corresponding files and the GENCATEGORY clause is used to define the primary key field of that file. The unique value of the primary key is generated by the File system automatically (as a surrogate key). An example of data file definition for a file related to a generalization category is shown in Fig. 10. In this example, the DEFINING clause is used to define join fields between the file corresponding to the generalization category and the file corresponding to the class (an entity type or a category) that takes part in the generalization category.

For entity types that do not have unique keys (weak entity types), the SURROGATE clause is used to define a unique primary key field (surrogate key) for that entity type. The value of the surrogate key is automatically generated by the file system. The REFERS clause is used to define join fields for two files. Since the corresponding field of a multi-valued attribute are stored in a different

```
DEFINE X_FILE TYPE INDEX CORRESPONDS X_CATEGORY FIELDS
 X_key 1  I 4  0  N N N N GENCATEGORY,
 X_a   5  S 9  1  N N N N DEFINING A_CLASS A_FILE A_field,
 X_b   14 S 8  2  N N N N DEFINING B_CLASS B_FILE B_field,
 X_c   22 S 7  3  N N N N DEFINING C_CLASS C_FILE C_field,
 X_d   29 S 4  -1 N Y Y Y CORRESPONDS Xd_attr OF X_CATEGORY\
```

```
A_attr (unique)      B_attr (unique)      C_attr (unique)
A_field (key 0)      B_field (key 0)      C_field (key 0)
+-------------+      +-------------+      +-------------+
¦ A_CLASS     ¦      ¦  B_CLASS    ¦      ¦  C_CLASS    ¦
¦ (A_FILE)    ¦      ¦  (B_FILE)   ¦      ¦  (C_FILE)   ¦
+-------------+      +-------------+      +-------------+
       \                    ¦                    /
        \                   ¦                   /
         \                  ¦                  /
       +-------------------------------------------+
       ¦  X_CATEGORY (a generalization category)   ¦
       ¦  (X_FILE)                                 ¦
       +-------------------------------------------+
                     X_key (key 0)
                     X_a   (key 1)
                     X_b   (key 2)
                     X_c   (key 3)
                     X_d   (non-key) -- Xd_attr
```

Figure 10 An example of data file definition
for generalization category

file (multi-valued file), the short REFERS clause without FOR key word is used to define a join field (a key field in the multi-valued file) corresponding to the entity type, category or relationship which the multi-valued attribute belongs to. The long REFERS clause with FOR key word is used to define join fields between two files for which a relationship exists between two classes.

By using the appropriate relational join specifications and surrogate keys, all relevant information concerning relationships, categories and multi-valued attributes can be stored into tabular forms used in relational model [Elma 85].

BNF for data file definition :

```
<file> ::= FILE <filename> TYPE <file type> CORRESPONDS
           (<class name>¦<attribute name>)
           FIELDS
           <field name> <starting location> <fieldtype>
           <length> <keyno> <combination> <dup> <changes>
           <null_key> [(CORRESPONDS <attribute name> OF
           <class name> ¦ REFERS <field name2> OF
           <file name2> [FOR <relationship name>
           <class name1> <class name2> <participation name1>
           <participation name2>] ¦ SUBCATEGORY
           <category name> ¦ GENCATEGORY ¦ DEFINING
           <class name> <file name> <field name>
           ¦ SURROGATE)]
           {,<field name> <starting location> <fieldtype>
           <length> <keyno> <combination> <dup> <changes>
           <null_key> [(CORRESPONDS <attribute name> OF
           <class name> ¦ REFERS <field name2> OF
```

```
            <file name2> [FOR <relationship name>
            <class name1> <class name2> <participation name1>
            <participation name2>] ¦ SUBCATEGORY
            <category name> ¦ GENCATEGORY ¦ DEFINING
            <class name> <file name> <field name>
            ¦ SURROGATE)]}
<filename> ::= <string of characters>
<file type> ::= INDEX ¦ REL ¦ SEQ
<field name> ::= <string of characters>
<starting location> ::= <integer>
<fieldtype> ::= I ¦ R ¦ S
<length> ::= <integer>
<keyno> := <integer>
<attribute name> ::= <string of characters>
<class name> ::= <string of characters>
<combination> ::= Y ¦ N
<dup> ::= Y ¦ N
<changes> ::= Y ¦ N
<null_key> ::= Y ¦ N
```

4.2.6 Naming restrictions in an ECR schema

In order to avoid ambiguity in the GORDAS query statements, some naming restrictions are needed in an ECR schema and are listed in the following.

(1) All value set names must be unique.

(2) All entity type, category, and relationship names must be unique.

(3) All data file names must be unique.

(4) All attribute names of a particular class (entity type, category or relationship) must be unique.

(5) All field names of a particular file must be unique.

(6) All participation names (connection names) directly related to a particular entity type or a category must be unique.

(7) Entity type names, category names, relationship names and file names are all in upper case.

(8) Participation names (connection names) are all in lower case.

(9) The first character of attribute names and field names are in upper case, the rest of the characters are in lower cases.

(10) The names of the attributes of a binary relationship (relationship in which only two classes participate) must be different from the names of the attributes of the two classes (entity types or categories) that participate in the relationship.

4.3 Example DDLs for Two Databases

The data definition files for the two ECR schemas Company and University (ECR diagrams in Fig. 6 and Fig. 7) are listed in the following to illustrate the definitions of value set, entity type, category, relationship and data file.

4.3.1 Company Database

```
SCHEMA COMPANY


DEFINE VALUESET SOCSECNUM AS INTEGER 0:999999999\
DEFINE VALUESET SALARIES AS REAL 0.0:999999.99\
DEFINE VALUESET LOCATIONCODES AS {MN01,OH20,AZ05,CA06,CA08,
                                  MN09,TX01,TX05,TX25}\
DEFINE VALUESET STATES AS {MINNESOTA,OHIO,ARIZONA,
                           CALIFORNIA,TEXAS}\
DEFINE VALUESET PEOPLENAMES AS STRING 15\
DEFINE VALUESET PHONENUM AS STRING 12\
DEFINE VALUESET PROJECTNAMES AS NAME OF PROJECT: STRING 15\
DEFINE VALUESET PROJECTNUMS AS NUMBER OF PROJECT:INTEGER\
DEFINE VALUESET DIVISIONNAMES AS NAME OF DIVISION:STRING 15\
DEFINE VALUESET STARTDATES AS STRING 8\
DEFINE VALUESET WEEKLYHOURS AS REAL 0:40\
DEFINE VALUESET SCIENTISTGRADES AS {S1,S2,S3,S4,S5,E,F}\
DEFINE VALUESET TECHNICIANGRADES AS {T1,T2,T3,T4,T5,T6,T7,
                                     T8,T9}\
DEFINE VALUESET DIVISIONNUMS AS INTEGER\
DEFINE VALUESET ADDRESSES AS STRING 30\
DEFINE VALUESET TRADEUNIONS AS {UAW,TEAMSTERS}\
DEFINE VALUESET ORGANIZATIONS AS STRING 15\
DEFINE VALUESET TECHSOCIETIES AS {ACM,IEEE,ASME,AMS}\


DEFINE ENTITYTYPE DIVISION ATTRIBUTES
  Number           VALUESET DIVISIONNUMS UNIQUE,
  Name             VALUESET DIVISIONNAMES UNIQUE,
  Location         VALUESET LOCATIONCODES MIN 1 MAX 20\

DEFINE ENTITYTYPE EMPLOYEE ATTRIBUTES
  Ssn              VALUESET SOCSECNUM UNIQUE,
  Name             VALUESET PEOPLENAMES,
  Hphone          VALUESET PHONENUM MIN 0 MAX 1,
  Ophone          VALUESET PHONENUM MIN 0 MAX 1,
  Address         VALUESET ADDRESSES MIN 1 MAX 1\

DEFINE ENTITYTYPE PROJECT ATTRIBUTES
  Number          VALUESET PROJECTNUMS UNIQUE,
  Name            VALUESET PROJECTNAMES UNIQUE,
  Location        VALUESET LOCATIONCODES,
  Pstart          VALUESET STARTDATES\


DEFINE CATEGORY FULLTIMEMPLOYEE FROM EMPLOYEE ATTRIBUTES
  Sal             VALUESET SALARIES\

DEFINE CATEGORY CONSULTANT FROM EMPLOYEE ATTRIBUTES
  Organization    VALUESET ORGANIZATIONS,
  Payscale        VALUESET SALARIES\

DEFINE CATEGORY SCIENTIST FROM EMPLOYEE ATTRIBUTES
  Sgrade          VALUESET SCIENTISTGRADES,
  Society         VALUESET TECHSOCIETIES MAX 2\

DEFINE CATEGORY TECHNICIAN FROM EMPLOYEE ATTRIBUTES
  Tgrade          VALUESET TECHNICIANGRADES,
  Union           VALUESET TRADEUNIONS\


DEFINE RELATIONSHIP ASSIGNED FROM
  FULLTIMEMPLOYEE (division, employees) MIN 1 MAX 1,
  DIVISION (employees, division)\

DEFINE RELATIONSHIP CONTROLS FROM
  DIVISION (projects,division),
  PROJECT (division,projects) MIN 1 MAX 1\

DEFINE RELATIONSHIP MANAGES FROM
  FULLTIMEMPLOYEE (proj_managed, manager) MIN 0 MAX 1,
  PROJECT (manager, proj_managed) MIN 1 MAX 1
 ATTRIBUTES
  Mstart          VALUESET STARTDATES\

DEFINE RELATIONSHIP WORKSON FROM
  EMPLOYEE (projects, employees),
  PROJECT (employees, projects)
 ATTRIBUTES
  Hours           VALUESET WEEKLYHOURS\
DEFINE FILE EMPLOYEE TYPE INDEX CORRESPONDS EMPLOYEE
  FIELDS
    Ssn          1   I 4  0  N N N N
                             CORRESPONDS Ssn OF EMPLOYEE,
    Name         5   S 15 1  N Y N N
                             CORRESPONDS Name OF EMPLOYEE,
    Hphone       20  S 12 -1 N Y Y Y
                             CORRESPONDS Hphone OF EMPLOYEE,
    Ophone       32  S 12 -1 N Y Y Y
                             CORRESPONDS Ophone OF EMPLOYEE,
    Address      44  S 30 -1 N Y Y Y
                             CORRESPONDS Address OF
                             EMPLOYEE,
    Fulltime     74  S 1  3  N Y Y Y
                             SUBCATEGORY FULLTIMEMPLOYEE,
    Sal          75  R 4  -1 N Y Y Y
                             CORRESPONDS Sal OF
                             FULLTIMEMPLOYEE,
    Divnum       79  I 4  2  N Y Y Y
                             REFERS Number OF DIVISION FOR
                             ASSIGNED FULLTIMEMPLOYEE
                             DIVISION division employees,
    Consultant   83  S 1  4  N Y Y Y
                             SUBCATEGORY CONSULTANT,
    Organization 84  S 15 -1 N Y Y Y
                             CORRESPONDS Organization OF
                             CONSULTANT,
    Payscale     99  R 4  -1 N Y Y Y
                             CORRESPONDS Payscale OF
                             CONSULTANT,
    Scientist    103 S 1  5  N Y Y Y
                             SUBCATEGORY SCIENTIST,
    Sgrade       104 S 32 -1 N Y Y Y
                             CORRESPONDS Sgrade OF
                             SCIENTIST,
    Technician   136 S 1  6  N Y Y Y
                             SUBCATEGORY TECHNICIAN,
    Tgrade       137 S 32 -1 N Y Y Y
                             CORRESPONDS Tgrade OF
                             TECHNICIAN,
    Union        169 S 32 -1 N Y Y Y
                             CORRESPONDS Union OF
                             TECHNICIAN\

DEFINE FILE SOCIETY TYPE INDEX CORRESPONDS Society
  FIELDS
    Ssn          1   I 4  0   N Y N N
                              REFERS Ssn OF EMPLOYEE,
    Society       5    S 32 1    N Y Y Y
                                 CORRESPONDS Society OF
                                 SCIENTIST\

DEFINE FILE DIVISION TYPE INDEX CORRESPONDS DIVISION
  FIELDS
    Number        1    I 4  0    N N N N
                                 CORRESPONDS Number OF
                                 DIVISION,
    Name          5    S 15 1    N N N N
                                 CORRESPONDS Name OF DIVISION\

DEFINE FILE LOCATION TYPE INDEX CORRESPONDS Location
  FIELDS
    Number        1    I 4  0    N Y N N
                                 REFERS Number OF DIVISION,
    Location      5    S 32 1    N Y Y Y
                                 CORRESPONDS Location OF
                                 DIVISION\

DEFINE FILE PROJECT TYPE INDEX CORRESPONDS PROJECT
  FIELDS
    Number        1    I 4  0    N N N N
                                 CORRESPONDS Number OF PROJECT,
    Name          5    S 15 1    N N N N
                                 CORRESPONDS Name OF PROJECT,
    Location      20   S 32 -1   N Y Y Y
                                 CORRESPONDS Location OF
                                 PROJECT,
    Pstart        52   S 8  -1   N Y Y Y
                                 CORRESPONDS Pstart OF PROJECT,
    Divnum        60   I 4  3    N Y Y Y
                                 REFERS Number OF DIVISION FOR
                                 CONTROLS PROJECT DIVISION
                                 division projects,
    Mgrssn        64   I 4  2    N Y Y Y
                                 REFERS Ssn OF EMPLOYEE FOR
                                 MANAGES PROJECT
                                 FULLTIMEMPLOYEE
                                 manager proj_managed,
    Mstart        68   S 8  -1   N Y Y Y
                                 CORRESPONDS Mstart OF MANAGES\

DEFINE FILE WORKSON TYPE INDEX CORRESPONDS WORKSON
  FIELDS
    Hours         1    R 4  -1   N Y Y Y
                                 CORRESPONDS Hours OF WORKSON,
    Empssn        5    I 4  1    N Y N N
```

```
                                        REFERS Ssn OF EMPLOYEE FOR
                                        WORKSON WORKSON EMPLOYEE
                                        employees projects,
    Pnum          9    I 4   2          N Y N N
                                        REFERS Number OF PROJECT FOR
                                        WORKSON WORKSON PROJECT
                                        projects employees,
    Essnpnum      5    I 8   0          Y N N N

END
```

4.3.2 University Database

```
SCHEMA UNIVERSITY


DEFINE VALUESET SOCSECNUM AS STRING 9\
DEFINE VALUESET PERSONLNAMES AS STRING 15\
DEFINE VALUESET PERSONFNAMES AS STRING 15\
DEFINE VALUESET PERSONMNAMES AS STRING 1\
DEFINE VALUESET PERSONNAMES AS STRING 31\
DEFINE VALUESET STREETNUMS AS STRING 6\
DEFINE VALUESET STREETNAMES AS STRING 15\
DEFINE VALUESET APTNUMS AS STRING 4\
DEFINE VALUESET CITYNAMES AS STRING 15\
DEFINE VALUESET STATENAMES AS STRING 15\
DEFINE VALUESET ZIP AS STRING 5\
DEFINE VALUESET ADDRESSES AS STRING 60\
DEFINE VALUESET DAYS AS STRING 10\
DEFINE VALUESET SEXES AS {M,F}\
DEFINE VALUESET SALARIES AS REAL INTERVAL 0.0:999999.99\
DEFINE VALUESET RANKS AS STRING 10\
DEFINE VALUESET OFFICES AS STRING 10\
DEFINE VALUESET PHONES AS STRING 12\
DEFINE VALUESET GRANTITLES AS Title OF GRANT:STRING 15\
DEFINE VALUESET GRANTNUMS AS INTEGER\
DEFINE VALUESET GRANCODES AS STRING 10\
DEFINE VALUESET GRANTAGENCIES AS Agency OF GRANT:STRING 15\
DEFINE VALUESET SUPPORTDURATIONS AS Time OF SUPPORT:INTEGER
                RANGE 0:10\
DEFINE VALUESET COLLEGENAMES AS STRING 15\
DEFINE VALUESET DEPTNAMES AS STRING 15\
DEFINE VALUESET COURSENAMES AS STRING 15\
DEFINE VALUESET COURSENUMS AS STRING 10\
DEFINE VALUESET COURSEDESC AS STRING 20\
DEFINE VALUESET SECTIONNUMS AS STRING 5\
DEFINE VALUESET SECTIONQTRS AS {SPRING,SUMMER1,SUMMER2,
                                FALL,WINTER}\
DEFINE VALUESET YEARS AS STRING 4\
DEFINE VALUESET QTRYEARS AS STRING 11\
DEFINE VALUESET CLASSES AS INTEGER RANGE 1:20\
DEFINE VALUESET GRADES AS {A,B,C,D,F,W,I,P,S,U}\
DEFINE VALUESET DEGREE AS {AA,BA,BS,MA,MS,MBA,PHD,JD,MD}\
DEFINE VALUESET DEGREES AS STRING 38\


DEFINE ENTITYTYPE PERSON ATTRIBUTES
  Ssn               VALUESET SOCSECNUM UNIQUE,
  Lname             VALUESET PERSONLNAMES,
  Fname             VALUESET PERSONFNAMES,
  Minit             VALUESET PERSONMNAMES,
  Name              VALUESET PERSONNAMES,
  No                VALUESET STREETNUMS,
  Street            VALUESET STREETNAMES,
  Aptno             VALUESET APTNUMS,
  City              VALUESET CITYNAMES,
  State             VALUESET STATENAMES,
  Zip               VALUESET ZIP,
  Address           VALUESET ADDRESSES,
  Sex               VALUESET SEXES,
  Bdate             VALUESET DAYS\

DEFINE ENTITYTYPE GRANT ATTRIBUTES
  No                VALUESET GRANTNUMS UNIQUE,
  Code              VALUESET GRANCODES UNIQUE,
  Title             VALUESET GRANTITLES,
  Agency            VALUESET GRANTAGENCIES,
  Stdate            VALUESET DAYS\

DEFINE ENTITYTYPE DEPARTMENT ATTRIBUTES
  Dname             VALUESET DEPTNAMES UNIQUE,
  Office            VALUESET OFFICES,
  Dphone            VALUESET PHONES\

DEFINE ENTITYTYPE COLLEGE ATTRIBUTES
  Cname             VALUESET COLLEGENAMES UNIQUE,
  Coffice           VALUESET OFFICES,
  Dean              VALUESET PERSONNAMES\

DEFINE ENTITYTYPE SECTION ATTRIBUTES
  Secno             VALUESET SECTIONNUMS,
  Qtr               VALUESET SECTIONQTRS,
  Year              VALUESET YEARS,
  Qtryear           VALUESET QTRYEARS\
DEFINE ENTITYTYPE COURSE ATTRIBUTES
  Cno               VALUESET COURSENUMS UNIQUE,
  Cname             VALUESET COURSENAMES,
  Cdesc             VALUESET COURSEDESC\


DEFINE CATEGORY FACULTY FROM PERSON ATTRIBUTES
  Salary            VALUESET SALARIES,
  Rank              VALUESET RANKS,
  Foffice           VALUESET OFFICES,
  Fphone            VALUESET PHONES\

DEFINE CATEGORY STUDENT FROM PERSON ATTRIBUTES
  Class             VALUESET CLASSES\

DEFINE CATEGORY GRAD_STUDENT FROM STUDENT:Class=5 ATTRIBUTES
  College           VALUESET COLLEGENAMES MIN 0 MAX 10,
  Major             VALUESET DEPTNAMES MIN 0 MAX 10,
  Year              VALUESET YEARS MIN 0 MAX 10,
  Degree            VALUESET DEGREE MIN 0 MAX 10,
  Degrees           VALUESET DEGREES MIN 0 MAX 10\

DEFINE CATEGORY INSTR_RESEARCHER FROM FACULTY,GRAD_STUDENT\

DEFINE CATEGORY CURRENT_SECTION FROM SECTION
                                      :Qtryear=cur_qtr_year\


DEFINE RELATIONSHIP ADVISOR FROM                 15
  FACULTY (main_advisees, advisor) MIN 0 MAX 10,
  GRAD_STUDENT (advisor, main_advisees) MIN 0 MAX 1\

DEFINE RELATIONSHIP COMMITTEE FROM            20
  FACULTY (advisees, committee) MIN 0 MAX 10,
  GRAD_STUDENT (committee, advisees) MIN 0 MAX 10\
                                                20
DEFINE RELATIONSHIP BELONGS FROM
  FACULTY (depts, faculty) MIN 1 MAX 5,
  DEPARTMENT (faculty, depts) MIN 1 MAX 100\

DEFINE RELATIONSHIP CHAIRS FROM
  FACULTY (dept_chaired, chair) MIN 0 MAX 1,
  DEPARTMENT (chair, dept_chaired) MIN 1 MAX 1\

DEFINE RELATIONSHIP PI FROM
  FACULTY (grants, facus) MIN 0 MAX 20,
  GRANT (facus, grants) MIN 1 MAX 1\
DEFINE RELATIONSHIP SUPPORT FROM
  GRANT (supportees, grant_support) MIN 1 MAX 10,
  INSTR_RESEARCHER (grant_support, supportees) MIN 0 MAX 10
 ATTRIBUTES
  Start             VALUESET DAYS,
  Time              VALUESET SUPPORTDURATIONS,
  End               VALUESET DAYS\

DEFINE RELATIONSHIP MINOR FROM
  STUDENT (minor_dept, minors) MIN 0 MAX 1,
  DEPARTMENT (minors, minor_dept) MIN 0\

DEFINE RELATIONSHIP MAJOR FROM
  STUDENT (major_dept, majors) MIN 0 MAX 1,
  DEPARTMENT (majors, major_dept) MIN 0\
                                      1 ✓
DEFINE RELATIONSHIP REGISTERED FROM
  STUDENT (current_courses, registrants) MIN 0 MAX 10,
  CURRENT_SECTION (registrants, current_courses) MIN 5
                                                 MAX 100\

DEFINE RELATIONSHIP TRANSCRIPT FROM
  STUDENT (completed_course, students),
  SECTION (students, completed_course) MIN 5
 ATTRIBUTES
  grade             VALUESET GRADES\

DEFINE RELATIONSHIP CS FROM
  SECTION (course, sections) MIN 1 MAX 1,
  COURSE (sections, course) MIN 1\
                                Ø ←
DEFINE RELATIONSHIP DC FROM
  DEPARTMENT (courses, dept) MIN 0 MAX 100,
  COURSE (dept, courses) MIN 1 MAX 1\

DEFINE RELATIONSHIP CD FROM
  DEPARTMENT (college, depts) MIN 1 MAX 1,
  COLLEGE (depts, college) MIN 1\

DEFINE RELATIONSHIP TEACH FROM
  INSTR_RESEARCHER (sections_taught, instructor) MIN 0
                                                 MAX 5,
  SECTION (instructor, sections_taught) MIN 1 MAX 1\

DEFINE FILE PERSON TYPE INDEX CORRESPONDS PERSON
  FIELDS
    Ssn          1   S 9   0  N N N N
                              CORRESPONDS Ssn OF PERSON,
Lname       10  S 15 -1 N Y Y N
                        CORRESPONDS Lname OF PERSON,
Fname       25  S 15 -1 N Y Y N
                        CORRESPONDS Fname OF PERSON,
Minit       40  S 1  -1 N Y Y N
                        CORRESPONDS Minit OF PERSON,
Name        10  S 31 1  Y Y Y N
                        CORRESPONDS Name OF PERSON,
Streetnum   41  S 6  -1 N Y Y Y
                        CORRESPONDS No OF PERSON,
Street      47  S 15 -1 N Y Y Y
                        CORRESPONDS Street OF PERSON,
Aptno       62  S 4  -1 N Y Y Y
                        CORRESPONDS Aptno OF PERSON,
Cityname    66  S 15 -1 N Y Y Y
                        CORRESPONDS City OF PERSON,
State       81  S 15 -1 N Y Y Y
                        CORRESPONDS State OF PERSON,
Zipcode     96  S 5  -1 N Y Y Y
                        CORRESPONDS Zip OF PERSON,
Address     41  S 60 2  Y Y Y Y
                        CORRESPONDS Address OF PERSON,
Sex         101 S 1  -1 N Y N Y
                        CORRESPONDS Sex OF PERSON,
Bdate       102 S 10 -1 N Y Y Y
                        CORRESPONDS Bdate OF PERSON,
Faculty     112 S 1  3  N Y Y Y
                        SUBCATEGORY FACULTY,
Rank        113 S 10 -1 N Y Y Y
                        CORRESPONDS Rank OF FACULTY,
Salary      123 R 4  -1 N Y Y Y
                        CORRESPONDS Salary OF FACULTY,
Foffice     127 S 10 -1 N Y Y Y
                        CORRESPONDS Foffice OF FACULTY,
Fphone      137 S 12 -1 N Y Y Y
                        CORRESPONDS Fphone OF FACULTY,
Student     149 S 1  4  N Y Y Y
                        SUBCATEGORY STUDENT,
Class       150 I 4  -1 N Y Y Y
                        CORRESPONDS Class OF STUDENT,
Major       154 S 15 5  N Y Y Y
                        REFERS Dname OF DEPARTMENT FOR
                        MAJOR STUDENT DEPARTMENT
                        major_dept majors,
Minor       169 S 15 6  N Y Y Y
                        REFERS Dname OF DEPARTMENT FOR
                        MINOR STUDENT DEPARTMENT
                        minor_dept minors,
    Grad_stud    184 S 1  7  N Y Y Y
                             SUBCATEGORY GRAD_STUDENT,
    Fac_ssn      185 S 9  -1 N Y Y Y
                             REFERS Ssn OF PERSON FOR
                             ADVISOR GRAD_STUDENT FACULTY
                             advisor main_advisees\

DEFINE FILE INSTR_RES TYPE INDEX CORRESPONDS
                                 INSTR_RESEARCHER
  FIELDS
    Instr_res_id  1   I 4  0  N N N N
                              GENCATEGORY,
    Faculty_ssn   5   S 9  1  N N N Y
                              DEFINING FACULTY PERSON Ssn,
    Res_ssn       14  S 9  2  N N N Y
                              DEFINING GRAD_STUDENT PERSON
                              Ssn\

DEFINE FILE COMMITTEE TYPE INDEX CORRESPONDS COMMITTEE
  FIELDS
    Faculty_grad  1   S 18 0  Y N N N,
    Facultyssn    1   S 9  1  N N N N
                              REFERS Ssn OF PERSON FOR
                              COMMITTEE COMMITTEE FACULTY
                              committee advisees,
    Gradstudssn   10  S 9  2  N N N N
                              REFERS Ssn OF PERSON FOR
                              COMMITTEE COMMITTEE
                              GRAD_STUDENT
                              advisees committee\

DEFINE FILE GRANT TYPE INDEX CORRESPONDS GRANT
  FIELDS
    Number   1   I 4  0  N N N N
                         CORRESPONDS No OF GRANT,
    Code     5   S 10 1  N N N N
                         CORRESPONDS Code OF GRANT,
    Title    15  S 15 2  N Y Y N
                         CORRESPONDS Title OF GRANT,
    Agency   30  S 15 3  N Y Y Y
                         CORRESPONDS Agency OF GRANT,
    Fssn     45  S 9  4  N Y Y Y
                         REFERS Ssn OF PERSON FOR PI GRANT
                         FACULTY facus grants,
    Stdate   54  S 10 -1 N Y Y Y
                         CORRESPONDS Stdate OF GRANT\

DEFINE FILE SUPPORT TYPE INDEX CORRESPONDS SUPPORT
```

```
  FIELDS
    Support_id   1    I 8  0  Y N N N,
    Instr_res_id 1    I 4  1  N Y N N
                              REFERS Instr_res_id OF
                              INSTR_RESEARCHER FOR SUPPORT
                              SUPPORT INSTR_RESEARCHER
                              supportees grant_support,
    Grantnum     5    I 4  2  N Y N N
                              REFERS No OF GRANT FOR SUPPORT
                              SUPPORT GRANT grant_support
                              supportees,
    Start        9    S 10 -1 N Y Y Y
                              CORRESPONDS Start OF SUPPORT,
    End          19   S 10 -1 N Y Y Y
                              CORRESPONDS End OF SUPPORT,
    Time         29   I 4  -1 N Y Y Y
                              CORRESPONDS Time OF SUPPORT\

DEFINE FILE BELONGS TYPE INDEX CORRESPONDS BELONGS
  FIELDS
    Fssn_dname   1    S 24 0  Y N N N,

    Fssn         1    S 9  1  N N N N
                              REFERS Ssn OF PERSON FOR
                              BELONGS BELONGS FACULTY
                              faculty depts,
    Dname        10   S 15 2  N N N N
                              REFERS Dname OF DEPARTMENT FOR
                              BELONGS BELONGS DEPARTMENT
                              depts faculty\

DEFINE FILE DEPARTMENT TYPE INDEX CORRESPONDS DEPARTMENT
  FIELDS
    Dname        1    S 15 0  N N N N
                              CORRESPONDS Dname OF
                              DEPARTMENT,
    Dphone       16   S 12 -1 N Y Y Y
                              CORRESPONDS Dphone OF
                              DEPARTMENT,
    Office       28   S 10 -1 N Y Y Y
                              CORRESPONDS Office OF
                              DEPARTMENTS,
    Collegename  38   S 15 1  N Y Y Y
                              REFERS Cname OF COLLEGE FOR CD
                              DEPARTMENT COLLEGE college
                              depts,
    Chairssn     53   S 9  2  N N Y N
                              REFERS Ssn OF PERSON FOR
```

```
                                   CHAIRS DEPARTMENT FACULTY
                                   chair dept_chaired\

DEFINE FILE COLLEGE TYPE INDEX CORRESPONDS COLLEGE
  FIELDS
    Cname      1    S 15 0  N N N N
                            CORRESPONDS Cname OF COLLEGE,
    Coffice    16   S 10 -1 N Y Y Y
                            CORRESPONDS Coffice OF COLLEGE,
    Dean       26   S 31 1  N N Y Y
                            CORRESPONDS Dean OF COLLEGE\

DEFINE FILE COURSE TYPE INDEX CORRESPONDS COURSE
  FIELDS
    Cnumber     1    S 10 0  N N N N
                             CORRESPONDS Cno OF COURSE,
    Cname       11   S 15 1  N Y Y N
                             CORRESPONDS Cname OF COURSE,
    Cdesc       26   S 20 -1 N Y Y Y
                             CORRESPONDS Cdesc OF COURSE,
    Offer_dept 46   S 15 2  N Y Y N
                             REFERS Dname OF DEPARTMENT FOR
                             DC COURSE DEPARTMENT
                             dept courses\

DEFINE FILE DEGREES TYPE INDEX CORRESPONDS Degrees
  FIELDS
    Gradssn_major  1    S 43 0  Y N N N,

    Gradssn        1    S 9  1  N Y N N
                                REFERS Ssn OF PERSON,

    Major          10   S 15 2  N Y N N
                                CORRESPONDS Major OF
                                GRAD_STUDENT,

    Degree         25   S 4  3  N Y Y Y
                                CORRESPONDS Degree OF
                                GRAD_STUDENT,
    College        29   S 15 -1 N Y Y Y
                                CORRESPONDS College OF
                                GRAD_STUDENT,
    Year           44   S 4  -1 N Y Y Y
                                CORRESPONDS Year OF
                                GRAD_STUDENT,
    Degrees        10   S 38 4  Y Y Y N
                                CORRESPONDS Degrees OF
                                GRAD_STUDENT\
DEFINE FILE SECTION TYPE INDEX CORRESPONDS SECTION
  FIELDS
    Section_id      1   I 4  0  N N N N
                                SURROGATE,
    Cnumber         5   S 10 1  N Y Y N
                                REFERS Cnumber OF COURSE FOR
                                CS SECTION COURSE course
                                sections,
    Secnum          15  S 5  2  N Y Y N
                                CORRESPONDS Secno OF SECTION,

    Qtr             20  S 7  -1 N Y Y N
                                CORRESPONDS Qtr OF SECTION,
    Year            27  S 4  -1 N Y Y N
                                CORRESPONDS Year OF SECTION,
    Qtryear         20  S 11 3  Y Y Y Y
                                CORRESPONDS Qtryear OF
                                SECTION,
    Insres_id       31  I 4  4  N Y Y Y
                                REFERS Instr_res_id OF
                                INSTR_RESEARCHER FOR TEACH
                                SECTION INSTR_RESEARCHER
                                instructor sections_taught,
    Current_sec     35  S 1  -1 N Y Y Y
                                SUBCATEGORY CURRENT_SECTION\

DEFINE FILE TRANSCRIPT TYPE INDEX CORRESPONDS TRANSCRIPT
  FIELDS
    Trans_id        1   I 4  0  N N N N
                                SURROGATE,
    Stud_ssn        5   S 9  1  N Y N N
                                REFERS Ssn OF PERSON FOR
                                TRANSCRIPT TRANSCRIPT STUDENT
                                students completed_course,
    Section_id      14  I 4  2  N Y N N
                                REFERS Section_id OF SECTION
                                FOR TRANSCRIPT TRANSCRIPT
                                SECTION completed_course
                                students,
    Grade           18  S 1  -1 N Y Y Y
                                CORRESPONDS Grade OF
                                TRANSCRIPT\

DEFINE FILE REGISTERED TYPE INDEX CORRESPONDS REGISTERED
  FIELDS
    Reg_id          1   I 4  0  N N N N
                                SURROGATE,
    Stud_ssn        5   S 9  1  N Y N N
```

```
                                       REFERS Ssn OF PERSON FOR
                                       REGISTERED REGISTERED STUDENT
                                       registrants current_courses,
    Section_id     14  I 4  2  N Y N N
                                       REFERS Section_id OF SECTION
                                       FOR REGISTERED REGISTERED
                                       CURRENT_SECTION
                                       current_courses registrants
END
```

CHAPTER 5

DATA DICTIONARY IMPLEMENTATION

5.1 The Need for Data Dictionary Functions

Because the ECR model is a data model with rich semantics, the Data dictionary of the ECRDBS is more complicated than other data dictionaries based on other data models. There is a total of thirteen data dictionary files for each database schema. Each schema is stored in a separate subdirectory. Also, the main ECRDBS directory has two files 'FILES.DIC' (the File dictionary) and 'SCHEMA.DIC' which contain useful information for the Data dictionary system. The 'SCHEMA.DIC' file contains all database schema names existing in the whole ECRDBS.

In order to save the trouble of knowing the detailed structure of each data dictionary file, and facilitate retrieval of some information stored in several data dictionary files, thirty four functional procedures are provided for user or other components of the ECRDBS to retrieve information stored in the Data dictionary system.

By using the Data dictionary functions, users can get all information stored in the Data dictionary and the File dictionary without worrying about any changes that could be made to both the Data dictionary and the File dictionary, and also avoid coding complicated functional procedures themselves.

5.2 Data Dictionary Files for the ECRDBS

Each ECR database schema has its own Data dictionary which is composed of thirteen dictionary files. These thirteen dictionary files provide information about attributes, value sets, entity types, categories, relationships and data files in an ECR database schema.

Each data dictionary file is described in the following with a simple example.

(1) VALUESETDBD

Description : An indexed file composed of records of type VALUESETDBDRECORD.

```
VALUESETDBDRECORD = RECORD
                      VALUESETNAME : [KEY(0)]STRINGTYPE;
                      REF : BOOLEAN;
                      ATTRIBUTENAME, CLASSNAME : STRINGTYPE;
                      CASE VALUESETTYPE : CHAR OF
                        'S' : (LEN : INTEGER);
                        'E' : (COUNT : INTEGER);
                        'I' : (MINI, MAXI : INTEGER);
                        'R' : (MINR, MAXR : REAL)
                    END;
```

This file stores information about value sets. Field VALUESETNAME (value set name) is the primary key of this file. If REF is true, then this value set is a reference set, and ATTRIBUTENAME and CLASSNAME store the information about the reference value set; else this value set can be an explicit value set (with VALUESETTYPE value 'E') or a standard value set (with VALUESETTYPE value 'S', 'I' or 'R'). The field COUNT stores the number of explicit values in an explicit value set. The field LEN stores the length of a string type standard value set. The fields MINI, MAXI store the subrange of the integer type standard value set. The fields MINR, MAXR store the subrange of a real type standard value set. VALUESETDBD is the only dictionary file that is not managed by the ECR File system.

Example : Part of VALUESETDBD for the Company database.

VALUESETNAME	REF	ATTRIBUTENAME	CLASSNAME	VALUESETTYPE	LEN
PROJECTNAMES	TRUE	Name	PROJECT	S	15

VALUESETNAME	REF	VALUESETTYPE	COUNT
TECHSOCIETIES	FALSE	E	4

VALUESETNAME	REF	VALUESETTYPE	LEN
PEOPLENAMES	FALSE	S	15

VALUESETNAME	REE	VALUESETTYPE	MINI	MAXI
SOCSECNUM	FALSE	I	0	999999999

VALUSETNAME	REF	VALUESETTYPE	MINR	MAXR
WEEKLYHOURS	FALSE	R	0	40

(2) EXPLICITDBD

Description : An indexed file with two fields.

```
[1] VALUESETNAME : STRINGTYPE;
[2] DATA_VALUE : DATA_VALUETYPE;
```

(DATA_VALUETYPE is PACKED ARRAY [1..32] OF CHAR)

This file stores data values of all explicit value sets. The primary key (key 0) is the combination of field VALUESETNAME (value set name) and field DATA_VALUE (data value). Key 1 is on field VALUESETNAME.

Example : Part of EXPLICITDBD for the Company database.

VALUESETNAME	DATA_VALUE
TECHSOCIETIES	ACM
TECHSOCIETIES	IEEE
TECHSOCIETIES	ASME
TECHSOCIETIES	AMS
TRADEUNIONS	UAW
TRADEUNIONS	TEAMSTERS

(3) ATTRIBUTEDBD

Description : An indexed file with nine fields.

```
[1] ATTRIBUTENAME : STRINGTYPE;
[2] BELONGTO : STRINGTYPE;
[3] OBJECT_TYPE : INTEGER;
[4] VALUESETNAME : STRINGTYPE;
[5] UNIQUE : CHAR;
[6] MIN : INTEGER;
[7] MAX : INTEGER;
[8] CORRES_FILE : STRINGTYPE;
[9] CORRES_FIELD : STRINGTYPE;
```

This file stores information about attributes. The primary key is the combination of fields ATTRIBUTENAME (attribute name) and BELONGTO (the class name which the attribute belongs to). Key 1 is on field ATTRIBUTENAME. Key 2 is on field BELONGTO. Key 3 is the combination of fields CORRES_FILE (corresponding file) and CORRES_FIELD (corresponding field). The field OBJECT_TYPE contains an integer indicating the object type of the class name stored in field BELONGTO. The possible object types of a database schema are listed in the following.

1 : Attributes   2 : Entity types

3 : Categories (Subclass categories)

4 : Relationships

5 : Participation names (Connection names)

6 : Value sets

7 : Gcategories (Generalization categories)

The OBJECT_TYPE field in ATTRIBUTEDBD can only have value 2, 3, 4 or 7. The field UNIQUE with a 'T' value indicates that this attribute is unique. The field MIN and field MAX contain the cardinality constraints of the attribute.

Example : Part of ATTRIBUTEDBD for the Company database.

ATTRIBUTENAME	BELONGTO	OBJECT_TYPE	VALUESETNAME	UNIQUE
Number	DIVISION	2	DIVISIONNUMS	T
Union	TECHNICIAN	3	TRADEUNIONS	F
Hours	WORKSON	4	WEEKLYHOURS	F

MIN	MAX	CORRES_FILE	CORRES_FIELD
1	1	DIVISION	Number
1	1	EMPLOYEE	Union
1	1	WORKSON	Hours

(4) TOTALDBD

Description : An indexed file with two fields.

```
[1] OBJECT_NAME : STRINGTYPE;
[2] OBJECT_TYPE : INTEGER;
```

This file stores all object names and object types of a database schema. The primary key is on field OBJECT_NAME (object name).

Example : Part of TOTALDBD for the Company database.

OBJECT_NAME	OBJECT_TYPE
Ssn	1
EMPLOYEE	2
TECHNICIAN	3
WORKSON	4
employees	5
SOCSECNUM	6

(5) ENTITYDBD

Description : An indexed file with five fields.

```
[1] ENTITYTYPENAME : STRINGTYPE;
[2] PARTICIP_NAME1 : STRINGTYPE;
[3] PARTICIP_NAME2 : STRINGTYPE;
[4] RELATION_NAME : STRINGTYPE;
[5] RELATED_TO : STRINGTYPE;
```

Each record of this file contains an entity type name (field ENTITYTYPENAME), the class name (field RELATED_TO) it is in relationship with, the relationship name (field RELATIONSHIP) and the participation names (fields PARTICIP_NAME1, PARTICIP_NAME2) for both participants of the relationship. The primary key is the combination of field ENTITYTYPE and field PARTICIP_NAME1. Key 1 is on field ENTITYTYPENAME. Key 2 is on field PARTICIP_NAME1. If the relationship is not binary (i.e. more than two participants) then the RELATED_TO field will contain blanks.

Example : Part of ENTITYDBD for the Company database.

ENTITYTYPENAME	PARTICIP_NAME1	PARTICIP_NAME2
DIVISION	projects	division
PROJECT	division	projects
PROJECT	manager	proj_managed

RELATION_NAME	RELATED_TO
CONTROLS	PROJECT
CONTROLS	DIVISION
MANAGES	FULLTIMEMPLOYEE

(6) CATEGORYRELDBD

Description : An indexed file with five fields.

```
[1] CATEGORYNAME : STRINGTYPE;
[2] PARTICIP_NAME1 : STRINGTYPE;
[3] PARTICIP_NAME2 : STRINGTYPE;
[4] RELATION_NAME : STRINGTYPE;
[5] RELATED_TO : STRINGTYPE;
```

Each record of this file contains a category name (field CATEGORYNAME), the class name (field RELATED_TO) it is in relationship with, the relationship name (field

RELATION_NAME) and the participation names (fields PARTICIP_NAME1, PARTICIP_NAME2) for both participants of the relationship. The primary key of this file is the combination of field CATEGORYNAME and field PARTICIP_NAME1. Key 1 is on field CATEGORYNAME. Key 2 is on field PARTICIP_NAME1. If the relationship is not binary, then the RELATED_TO field will contain blanks.

Example : CATEGORYRELDBD for the Company database.

```
CATEGORYNAME     |PARTICIP_NAME1  |PARTICIP_NAME2
-----------------+----------------+---------------
FULLTIMEMPLOYEE  |proj_managed    |manager
FULLTIMEMPLOYEE  |division        |employees

RELATION_NAME  |RELATED_TO
---------------+---------------
MANAGES        |PROJECT
ASSIGNED       |DIVISION
```

(7) CATEGORYDBD

Description : An indexed file with eight fields.

```
[1] CATEGORYNAME : STRINGTYPE;
[2] DEFINED_ON_CLASS : STRINGTYPE;
[3] CLASS_TYPE: CHAR;
[4] ATTRIBUTE : STRINGTYPE;
[5] CONST_VAL : STRINGTYPE:
[6] CORRES_FILE : STRINGTYPE;
[7] CORRES_FIELD : STRINGTYPE;
[8] CTYPE : CHAR;
```

This file stores all information about categories. The primary key is the combination of field CATEGORYNAME (category name) and field DEFINED_ON_CLASS (the class name that the category is defined on). Key 1 is on field CATEGORYNAME. Key 2 is on field DEFINED_ON_CLASS. Key 3 is on field CLASS_TYPE (the class type of the DEFINED_ON_CLASS). CLASS_TYPE can have value 'E' (for entity type) or 'C' (for category). The field ATTRIBUTE (attribute name) and the field CONST_VAL (constant value) specify the defining predicate used to select entities from DEFINED_ON_CLASS that belongs to the category. If no defining predicate is present, then both ATTRIBUTE and CONST_VAL contain blanks. Field CTYPE (category type) can have value 'G' (for generalization category) or 'S' (for subclass category). The field CORRES_FILE contains the file name corresponding to the category. The field CORRES_FIELD (corresponding field) contains a field name (in the corresponding file) which is used to specify the category. This corresponding field name is fetched from the field name defined by SUBCATEGORY or GENCATEGORY clause in the data file definition statement.

Example : Part of CATEGORYDBD for the University database.

CATEGORYNAME	DEFINED_ON_CLASS	CLASS_TYPE	ATTRIBUTE
STUDENT	PERSON	E	
GRAD_STUDENT	STUDENT	C	Class
INSTR_RESEARCHER	FACULTY	C	
INSTR_RESEARCHER	GRAD_STUDENT	C	

CONST_VAL	CORRES_FILE	CORRES_FIELD	CTYPE
	PERSON	Student	S
5	PERSON	Grad_student	S
	INSTR_RES	Instr_res_id	G
	INSTR_RES	Instr_res_id	G

(8) RELATIONSHIPDBD

Description : An indexed file with ten fields.

```
[1] RELATION_NAME : STRINGTYPE;
[2] RELATION_FROM : STRINGTYPE;
[3] PARTICIP_NAME1 : STRINGTYPE;
[4] PARTICIP_NAME2 : STRINGTYPE;
[5] FROM_TYPE : CHAR;
[6] SPECIFIC : CHAR;
[7] PD : CHAR;
[8] DLT : CHAR;
[9] MIN : INTEGER;
[10] MAX : INTEGER;
```

This file stores all information about relationships. Field RELATION_NAME contains a relationship name. Field RELATION_FROM contains the name of a participant of this relationship. Field PARTICIP_NAME1 is the participation name for the participant specified and field PARTICIP_NAME2 is the participation name for the unspecified participant in this relationship. Field FROM_TYPE specifies the type of RELATION_FROM. FROM_TYPE can have value 'E' (for entity type) or value 'C' (for category). Field SPECIFIC with value 'T' means that the relationship is specific. Field PD and field DLT are explained in more detail in Relationship Definition (section 4.2.4). Fields MIN and MAX specify the structural constraints of the relationship. The primary key is the combination of field RELATION_NAME, field RELATION_FROM and field PARTICIP_NAME1. Key 1 is on field RELATION_NAME. Key 2 is on field RELATION_FROM. Key 3 is the combination of field RELATION_FROM and field PARTICIP_NAME1.

Examples : Part of RELATIONSHIPDBD for the Company database.

RELATION_NAME	RELATION_FROM	PARTICIP_NAME1
ASSIGN	DIVISION	employees
ASSIGN	FULLTIMEMPLOYEE	division
WORKSON	EMPLOYEE	projects

PARTICIP_NAME2	FROM_TYPE	SPECIFIC	PD	DLT	MIN	MAX
division	E	F	F	F	0	(MAXINT)
employees	C	F	F	F	1	1
employees	E	F	F	F	0	(MAXINT)

(MAXINT is the largest integer value 2,147,483,647 on the VAX/VMS system)

(9) JOINDBD

Description : An indexed file with nine fields.

```
          [1] CLASS_NAME1 : STRINGTYPE;
          [2] PARTICIP_NAME1 : STRINGTYPE;
          [3] CLASS_NAME2 : STRINGTYPE;
          [4] PARTICIP_NAME2 : STRINGTYPE;
          [5] RELATION_NAME : STRINGTYPE;
          [6] FILE1 : STRINGTYPE;
          [7] JOINFIELD1 : STRINGTYPE;
          [8] FILE2 : STRINGTYPE;
          [9] JOINFIELD2 : STRINGTYPE;
```

This file stores join information for relationships. Each record of this file contains a relationship name (field RELATION_NAME), names of participants (fields CLASS_NAME1, CLASS_NAME2) in this relationship, participation names (fields PARTICIP_NAME1, PARTICIP_NAME2) for this relationship, join files (fields FILE1, FILE2) and join fields (field JOINFIELD1, JOINFIELD2) for these two participants. The primary key is the combination of field CLASS_NAME1 and field PARTICIP_NAME1. Key 1 is the combination of field CLASS_NAME2 and field PARTICIP_NAME2.

Example : Part of JOINDBD for the Company database.

CLASS_NAME1	PARTICIP_NAME1	CLASS_NAME2	PARTICIP_NAME2
FULLTIMEMPLOYEE	division	DIVISION	employees
DIVISION	projects	PROJECT	division

RELATION_NAME	FILE1	JOINFIELD1	FILE2	JOINFIELD2
ASSIGNED	EMPLOYEE	Divnum	DIVISION	Number
CONTROLS	DIVISION	Number	PROJECT	Divnum

(10) MVATTRDBD

Description : An indexed file with seven fields.

```
[1] ATTRIBUTENAME : STRINGTYPE;
[2] CLASSNAME : STRINGTYPE;
[3] MV_FILE : STRINGTYPE;
[4] MV_JOINFIELD : STRINGTYPE;
[5] FILE2 : STRINGTYPE;
[6] JOINFIELD2 : STRINGTYPE;
[7] MV_FIELD : STRINGTYPE;
```

Each record of this file contains a multi-valued attribute (field ATTRIBUTENAME), a class name (field CLASSNAME) that the multi-valued attribute belongs to, a multi-valued file (field MV_FILE), a join field (field MV_JOINFIELD) in the multi-valued file, the corresponding file (field FILE2) of CLASSNAME, the join field (field JOINFIELD2) in FILE2, and a field (field MV_FIELD) corresponding to the multi-valued attribute. The primary key of this file is the combination of field ATTRIBUTENAME and field CLASSNAME. Key 1 is on field ATTRIBUTENAME. Key 2 is on field CLASSNAME.

Examples : MVATTRDBD for the Company database.

ATTRIBUTENAME	CLASSNAME	MV-FILE	MV_JOINFIELD
Society	SCIENTIST	SOCIETY	Ssn
Location	DIVISION	LOCATION	Number

FILE2	JOINFIELD2	MV_FIELD
EMPLOYEE	Ssn	Society
DIVISION	Number	Location

(11) GENCATJOINDBD

Description : An indexed file with six fields.

```
[1] GEN_CATEGORYNAME : STRINGTYPE;
[2] CLASSNAME2 : STRINGTYPE;
[3] GEN_FILE : STRINGTYPE;
[4] GEN_JOINFIELD : STRINGTYPE;
[5] FILE2 : STRINGTYPE;
[6] JOINFIELD2 : STRINGTYPE;
```

This file stores join information about generalization categories. Each record of this file contains the name of a generalization category (field GEN_CATEGOTYNAME), the name of a class (field CLASSNAME2) which takes part in the generalization category, the corresponding file (field GEN_FILE) of the generalization category, the corresponding file (field FILE2) of CLASSNAME2, and join fields (fields GEN_JOINFIELD, JOINFIELD2) of these two files. The primary key is the combination of field GEN_CATEGORYNAME and field CLASSNAME2. Key 1 is on field GEN_CATEGORYNAME. Key 2 is on field CLASSNAME2.

Example : GENCATJOINDBD for the University database.

GEN_CATEGORYNAME	CLASSNAME2	GEN_FILE	GEN_JOINFIELD
INSTR_RESEARCHER	FACULTY	INSTR_RES	Faculty_ssn
INSTR_RESEARCHER	GRAD_STUDENT	INSTR_RES	Res_ssn

FILE2	JOINFIELD2
PERSON	Ssn
PERSON	Ssn

(12) KEYVALUESDBD

Description : An indexed file with three fields.

```
[1] FILENAME : STRINGTYPE;
[2] KEY_FIELD : STRINGTYPE;
[3] HIGHEST_KEY : INTEGER;
```

Each record of this file stores the name of a surrogate key (field KEY_FIELD), the name of the file (field FILENAME) which the surrogate key field belongs to, and the current value (field HIGHEST_KEY) of the surrogate key. The initial value of the field HIGHEST_KEY is zero. The value of HIGHEST_KEY is increased by one during each INSERT_RECORD operation operated on the file specified in the FILENAME. The primary key of this file is the combination of field FILENAME and field KEY_FIELD. Key 1 is on field FILENAME.

Example : Part of KEYVALUESDBD for the University database.

FILENAME	KEY_FIELD	HIGHEST_KEY
INSTR_RES	Instr_res_id	0
SECTION	Section_id	0
REGISTERED	Reg_id	0

(13) CLASSFILEDBD

Description : An indexed file with four fields.

```
[1] CLASSNAME : STRINGTYPE;
[2] CORRES_FILE : STRINGTYPE;
[3] SURROGATE_KEY : STRINGTYPE;
[4] MAJOR : CHAR;
```

Each record of this file contains the name of a class (field CLASSNAME), the name of the corresponding file (field CORRES_FILE) of this class, the name of a surrogate key (field SURROGATE_KEY) for this file, and a major indicator (field MAJOR). If no surrogate key is defined for this file, then the field SURROGATE_KEY contains blanks. Since each class can have more than one corresponding file, the field MAJOR with value 'Y' is used to indicate that this file is the major corresponding file of the class. The primary key of this file is on field CLASSNAME. Key 1 is on field CORRES_FILE. Key 2 is on field SURROGATE_KEY.

Example : Part of CLASSFILEDBD for the University database.

CLASSNAME	CORRES_FILE	SURROGATE_KEY	MAJOR
INSTR_RESEARCHER	INSTR_RES	Instr_res_id	Y
SECTION	SECTION	Section_id	Y
CS	SECTION		N
TEACH	SECTION		N
CURRENT_SECTION	SECTION		N
REGISTERED	REGISTERED	Reg_id	Y

5.3 Data Dictionary Procedures and Functions

There are thirty four functional procedures provided for the user or other component of the ECR DBMS to retrieve information stored in the Data dictionary, File dictionary, and 'SCHEMA.DIC' file. These thirty four functional procedures are listed in the following. Appropriate parameters for each functional procedure are explained in detail.

(1) SET_SCHEMA

Purpose : Set the current schema to the given schema name.

Files accessed : None.

Input : SCHEMA (a schema name).

Output : None.

(2) DELETE_DATABASE

Purpose : Delete a database (an ECR schema).

Files accessed : SCHEMA.DIC, FILES.DIC.

Input : SCHEMANAME (a schema name).

Output : None.

(3) GET_SCHEMA

Purpose : Get all schema names in the ECR DBMS.

Files accessed : SCHEMA.DIC.

Input : None.

Output : All schema names are stored in a linked list, and the pointer pointing to this linked list is returned.

(4) FIELD_INFO

Purpose : Get field definition (as described in the File dictionary) for a field, given a file name and a field name.

Files accessed : FILES.DIC

Input : FILENAME (a file name).
FIELDNAME (a field name).

Output : Field definition is stored in a record and the pointer pointing to this record is returned. If no such field is found, then a NIL pointer is returned.

(5) FILE_FIELD_INFOS

Purpose : Get all field definitions for fields in a file, given a filename.

Files accessed : FILES.DIC.

Input : FILENAME (a file name).

Output : All field definitions are stored in a linked list, and the pointer pointing to this linked list is returned. If no such file is found, then a NIL pointer is returned.

(6) FILE_FIELD_NAMES

Purpose : Get all field names and indicate all combination key fields, given a file name.

Files accessed : FILES.DIC.

Input : DATAFILENAME (a data file name).

Output : All field names and combination key indicators are stored in a linked list and pointed at by FIELDNAMERECPTR.

Remark :

(a) FIELDNAMEREC is of type PTR_FIELDNAMEREC.

```
PTR _FIELDNAMEREC = ^FIELDNAMEREC;
FILENAMEREC = RECORD
                FIELDNAME : STRINGTYPE;
                FTYPE : CHAR;
                NEXT : PTR_FIELDNAMEREC;
               END;
```

(b) Combination key indicator (FTYPE) can have value 'C' (for combination key field) or value 'F' (for noncombinational field).

(7) UNIQUE_FIELDS

Purpose : Get all unique field names, given a file name.

Files accessed : FILES.DIC.

Input : FILENAME (a file name).

Output : All unique key field names are stored in a linked list, and the pointer pointing to this linked list is returned. If no unique fields are found, then return a NIL pointer.

(8) OBJECT_TYPE

Purpose : Get the object type, given an object name of the current schema.

Files accessed : TOTALDBD.

Input : ONAME (an object name).

Output : a schema object type

0 -- object names not found	1 -- attributes
2 -- entity types	3 -- subclass categories
4 -- relationship	6 -- value sets
5 -- participation names (connection names)	
7 -- generalization categories	

(9) GET_OBJECT_INFO

Purpose : Get all object names and object types of the current schema (not including value sets).

Files accessed : TOTALDBD.

Input : None.

Output : All object names and object types are stored in a linked list, and the pointer pointing to this linked list is returned.

(10) ALL_CLASSNAMES

Purpose : Get all class names (entity type names, category names and relationship names) of the current schema.

Files accessed : TOTALDBD.

Input : None.

Output : All class names are stored in a linked list, and the pointer pointing to this linked list is returned.

(11) VALUESET_INFO

Purpose : Get all information about a value set, given an attribute name and a class name.

Files accessed : ATTRIBUTEDBD, VALUESETDBD.

Input : ATTR_NAME (an attribute name).
CLASSNAME (a class name).

Output : All information about the value set of an attribute is stored in a record and pointed at by VALPTR. If no such attribute name and class name are found, then VALPTR is set to NIL.

Remark :

(a) VALPTR is of type PTR_VALUESETRECORD.

```
PTR_VALUESETRECORD = ^VALUESETDBDRECORD;
VALUESETDBDRECORD = RECORD
                      VALUESETNAME : STRINGTYPE;
                      REF : BOOLEAN;
                      ATTRIBUTENAME : STRINGTYPE;
                      CLASSNAME : STRINGTYPE;
                      CASE VALUESETTYPE : CHAR OF
                        'S' : (LEN : INTEGER);
                        'E' : (COUNT : INTEGER);
                        'I' : (MINI, MAXI : INTEGER);
                        'R' : (MINR, MAXR : REAL)
                    END;
```

(12) EXPLICIT_VALUE

Purpose : Get all explicit data values for an explicit value set, given a value set name.

Files accessed : EXPLICITDBD.

Input : VNAME (a valueset name).

Output : All explicit data values are stored in a linked list pointed at by EXPTR. If no such explicit value set is found, then EXPTR is set to NIL.

Remark :

(a) EXPTR is of type PTR_EXPLICIT.

```
VSTR32 = VARYING [32] OF CHAR;
PTR_EXPLICIT = ^EXPLICIT_UNIT;
EXPLICIT_UNIT = RECORD
                  DATA_VALUE : VSTR32;
                  NEXT : PTR_EXPLICIT;
                END;
```

(13) GET_CLASS_UNIQKEYS

Purpose : Get attribute names, corresponding file names, corresponding field names for all unique attributes in a given class.

Files accessed : ATTRIBUTEDBD, FILES.DIC.

Input : CLASSNAME (a class name).

Output : All attribute names, file names, field names are stored in a linked list, and the pointer pointing to this linked list is returned. The node containing the primary key of the corresponding file will be put at the front of the list.

(14) BELONG_TO

Purpose : Check whether an attribute belongs to a class.

Files accessed : ATTRIBUTEDBD.

Input : ATTR_NAME (an attribute name).
CLASSNAME (a class name).

Output : A boolean value TRUE or FALSE.

(15) ATTRIBUTE_NAME

Purpose : Get all attribute names which belong to a class.

Files accessed : ATTRIBUTEDBD.

Input : CLASSNAME (a class name).

Output : All attribute names are stored in a linked list pointed by ATTRPTR. If no attributes belong to the given class, then ATTRPTR is set to NIL.

Remark :

(a) ATTRPTR is of type PTR_ATTRIBUTEREC.

```
PTR_ATTRIBUTEREC = ^ATTRIBUTEREC;
ATTRIBUTEREC = RECORD
                 ATTRIBUTENAME : STRINGTYPE;
                 NEXT : PTR_ATTRIBUTEREC;
               END;
```

(16) ATTRIBUTE_UNIQUE

Purpose : Check whether an attribute of a class is unique.

Files accessed : ATTRIBUTEDBD.

Input : ATTR_NAME (an attribute name).

CLASSNAME (a class name).

Output : A boolean value TRUE or FALSE.

(17) CHECK_MV_ATTR

Purpose : Check whether an attribute of a class is a multi-valued attribute.

Files accessed : ATTRIBUTEDBD.

Input : ATTR_NAME (an attribute name).

CLASSNAME (a class name).

Output : A boolean value TRUE or FALSE.

(18) GET_FIELD_ATTR_CLASS

Purpose : Get all corresponding attribute names, class names, field names, field type, field length, and combination field indicator, given a file name.

Files accessed : FILES.DIC, ATTRIBUTEDBD, CLASSFILEDBD.

Input : FILENAME (a file name).

Output : All attribute names, class names, field names, field type, field length, combination field indicators are stored in a linked list, and the pointer pointing to this linked list is returned.

Remark : Combination field indicator can have value 'C' (for combination key field) or value 'F' (for noncombinational field).

(19) ENTITY_RELATE_NAME

Purpose : Get all class names (entity type names or category names) and participation names related to a given entity type.

Files accessed : ENTITYDBD.

Input : ENTITYTYPENAME (an entity type name).

Output : All class names, participation names are stored in a linked list pointed by RELPTR. If no related class names are found, then RELPTR is set to NIL.

Remark : RELPTR is of type PTR_RELATED.

```
PTR_RELATED = ^RELATEDREC;
RELATEDREC = RECORD
               RELATED_TO : STRINGTYPE;
               PARTICIP_NAME1 : STRINGTYPE;
               PARTICIP_NAME2 : STRINGTYPE;
               NEXT : PTR_RELATED;
             END;
```

(20) CHECK_CONNECTION

Purpose : Get relationship name, related class name (category name or entity type name), given a class name (entity type name or category name) and a connection name.

Files accessed : TOTALDBD, ENTITYDBD, CATEGORYRELDBD.

Input : CLASSNAME (a class name).
CNAME (a connection name).

Output : RNAME (a relationship name).
CLASSNAME2 (related class name).
YESNO (a boolean value TRUE or FALSE indicating whether this relationship existed or not).

(21) CREATE_CTREE

Purpose : Get a C tree (category tree) which contains information about all defining categories and defined on classes (entity types or categories) related to the given class (as root of the C tree).

Files accessed : CATEGORYDBD.

Input : CLASSNAME (a class name).

Output : A pointer pointing to the C tree (with the given class as the root) is returned.

Remark :

(a) A pointer pointing to a C tree (category tree) is of type CTREE_PTR.

```
CTREE_PTR = ^CNODETYPE;
CNODETYPE = RECORD
              PARENT_PTR : CTREE_PTR;
              DEFINING_TREE : CTREE_PTR;
              DEFINEDON_TREE : CTREE_PTR;
              CLASSNAME : STRINGTYPE;
              CTYPE : CHAR;
              LEVEL : INTEGER;
              NEXT_PTR : CTREE_PTR;
              NEXT_QPTR : CTREE_PTR;
            END;
```

CTYPE (category type) can have value 'S' (for subclass category), value 'G' (for generalization category) or value 'R' (for root of a C tree). The root of a C tree has NIL value for its PARENT_PTR. LEVEL contains an integer value indicating level of a node from the root of a C tree. The root of a C tree has a LEVEL value 0. Other nodes have a LEVEL value greater than 0 and a nonnil PARENT_PTR. DEFINING_TREE is a pointer pointing to a subtree with root of the subtree containing the defining category of the specified node. DEFINEDON_TREE is a pointer pointing to a subtree with the root of the subtree containing the defined on class (entity type or category) of the specified node. NEXT_PTR is a pointer pointing to the next brother node (with the same LEVEL value) of the specified node. NEXT_QPTR (next queue pointer) is just a pointer used in programming nonrecursive functions for constructing a C tree and retrieving information from a C tree.

(b) A C tree for an ECR database schema example.

Fig. 11 shows the ECR diagram for this database schema example (showing only entity types and categories). CATEGORYDBD for this ECR database schema example is shown below.

CATEGORYNAME	DEFINED_ON_CLASS	CLASS_TYPE	...	CTYPE
E	D	E	...	S
H	D	E	...	S
M	H	C	...	S
L	H	C	...	S
R	M	C	...	S
K	I	E	...	G
K	J	C	...	G
J	G	E	...	G
J	H	C	...	G
H	A	E	...	G
H	B	E	...	G

Fig. 12 shows a C tree (with root class H) for this ECR database schema example.

(22) GET_DEFINE_LIST

Purpose : Get all class names (entity type names or category names), category types and category levels related to the given class (entity type or category) in the order of ascending level value.

Files accessed : CATEGORYDBD.

Input : CLASSNAME (a class name).

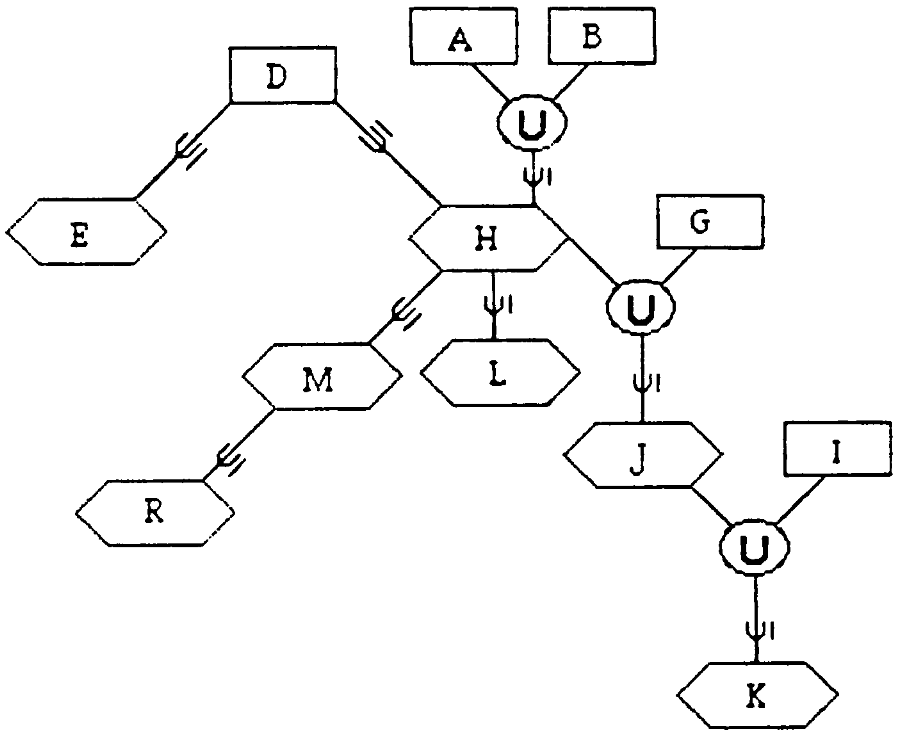
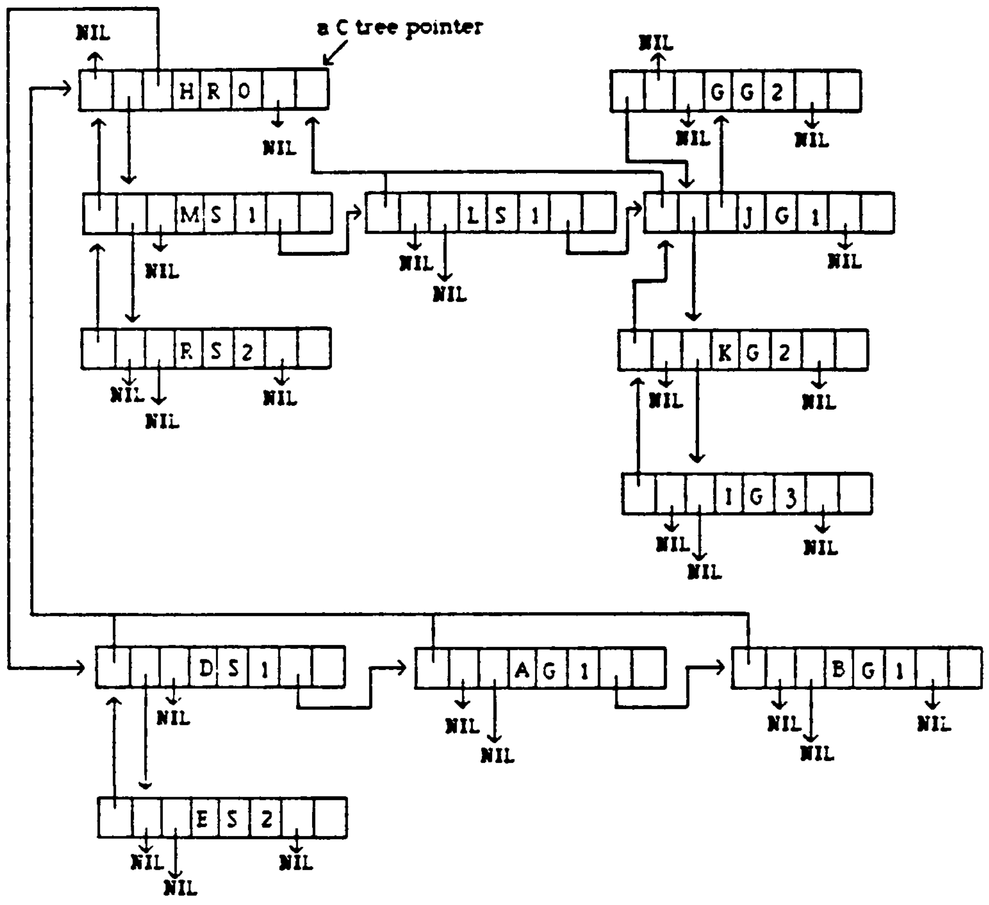


Figure 11 ECR diagram for a database schema example
(showing only entity types and categories)



PARENT_PTR	DEFINING_TREE	DEFINEDON_TREE	CLASSNAME	CTYPE	LEVEL	NEXT_PTR	NEXT_QPTR

Figure 12 A C tree (with root class H) for a database schema example

Output : All class names, category types and levels are stored in a linked list in the order of ascending level value, and the pointer pointing to this linked list is returned.

Remark :

(a) The output pointer is of type CLISTPTR.

```
CLISTPTR = ^NAME_NODE;
NAME_NODE = RECORD
              CTYPE : CHAR;
              CLASSNAME : STRINGTYPE;
              LEVEL : INTEGER;
              NEXT : CLISTPTR;
            END;
```

(b) The output of this functional procedure for the C tree example given in Fig. 12 is shown below.

```
           +-+-+-+-+   +-+-+-+-+   +-+-+-+-+    +-+-+-+-+
pointer -> ¦S¦M¦1¦-¦-> ¦S¦L¦1¦-¦-> ¦G¦J¦1¦-¦->  ¦S¦D¦1¦-¦-+
           +-+-+-+-+   +-+-+-+-+   +-+-+-+-+    +-+-+-+-+ ¦
                                                          ¦
 +--------------------------------------------------------+
 ¦    +-+-+-+-+   +-+-+-+-+   +-+-+-+-+   +-+-+-+-+
 +--> ¦G¦A¦1¦-¦-> ¦G¦B¦1¦-¦-> ¦S¦R¦2¦-¦-> ¦G¦K¦2¦-¦-------+
      +-+-+-+-+   +-+-+-+-+   +-+-+-+-+   +-+-+-+-+       ¦
                                                          ¦
 +--------------------------------------------------------+
 ¦    +-+-+-+-+   +-+-+-+-+   +-+-+-+-+
 +--> ¦G¦G¦2¦-¦-> ¦S¦E¦2¦-¦-> ¦G¦I¦3¦-¦-> NIL
      +-+-+-+-+   +-+-+-+-+   +-+-+-+-+
```

(23) DEFINED_ON :

Purpose : Check whether a class name is defined on the given defined name.

Files accessed : CATEGORYDBD.

Input : CLASSNAME (a class name).

DEFINED_NAME (another class name).

Output : A boolean value TRUE or FALSE.

Remark : This functional procedure works for both subclass and generalization categories.

(24) REPEAT_DEFINEDON_SUBCLASS

Purpose : Get all level superclass class names of a given subclass category.

Files accessed : CATEGORYDBD.

Input : CATEGORYNAME (a subclass category name).

Output : All superclass class names are stored in a linked list pointed by DEFINEDPTR. If no defined on superclasses are found, then set DEFINEDPTR to NIL.

Remark :

(a) DEFINEDPTR is of type PTR_DEFINEDON.

```
PTR_DEFINEDON = ^DEFINEDREC;

DEFINEDREC = RECORD
               DEFINE : STRINGTYPE;
               NEXT : PTR_DEFINEDON;
             END;
```

(b) This procedure works for subclass category only.

(25) DEFINING_SUBCLASS

Purpose : Get all first level subclass defining class names of a given class.

Files accessed : CATEGORYDBD.

Input : CLASSNAME (a class name).

Output : All defining subclass class names are stored in a linked list pointed by DEFININGPTR. If no defining subclasses are found, then set DEFININGPTR to NIL.

Remark :

(a) DEFININGPTR is of type PTR_DEFINING.

PTR_DEFINING = PTR_DEFINEDON;

(b) This procedure works for subclass category only.

(26) REPEAT_DEFINING_SUBCLASS

Purpose : Get all level defining subclass class names of a given class.

Files accessed : CATEGORYDBD.

Input : CLASSNAME (a class name).

Output : All level defining class names are stored in a linked list pointed by DEFININGPTR. If no defining subclasses are found, then set DEFININGPTR to NIL.

Remark :

(a) DEFININGPTR is of type PTR_DEFINING.

(b) This procedure works for subclass category only.

(27) CATEGORY_RELATE_NAME

Purpose : Get all class names (entity type names or category names) and participation names related to a category.

Files accessed : CATEGORYRELDBD.

Input : CATEGORYNAME (a category name).

Output : All class names and participation names are stored in a linked list pointed at by RELPTR. If no related class names are found, then set RELPTR to NIL.

Remark

(a) RELPTR is of type PTR_RELATED.

(28) GET_CONN_MIN_MAX

Purpose : Get MIN and MAX structural constraints for one end of a participation in a relationship, given a class name (entity type name or category name) and a connection name.

Files accessed : RELATIONSHIPDBD.

Input : CLASSNAME (a class name).
CNAME (a connection name).

Output : MIN, MAX structural constraints.

(29) CHECK_RELATIONSHIP

Purpose : Get relationship name, related class name (entity type name or category name), connection name for the related class, and MIN, MAX structural constraints for both ends of participations in the relationship, given a class name (entity type name or category name) and a connection name.

Files accessed : TOTALDBD, CATEGORYRELDBD, ENTITYDBD, RELATIONSHIPDBD.

Input : CLASSNAME (a class name).
CNAME (a connection name).

Output : RNAME (a relationship name).
CNAME2 (connection name for the other end of participation in a relationship).
CLASSNAME2 (Class name for the other end of participation in a relationship).
MIN2, MAX2 (MIN, MAX structural constraints for related participation).
MIN, MAX (MIN, MAX structural constraints for the given CLASSNAME and CNAME).
YESNO (a boolean value TRUE or FALSE, indicating whether this relationship existed or not).

(30) GET_JOIN_ATTRIBUTE

Purpose : Get join information of a relationship, given a class name (entity type name or category name) and a connection name.

Files accessed : JOINDBD, RELATIONSHIPDBD.

Input : CLASSNAME (a class name).
CNAME (a connection name).

Output : MTON (a boolean value TRUE or FALSE, indicating whether the relationship is M to N or not).

FILE1, FILE2 (two join files, if relationship is not M to N).

FILE3, FILE4 (more join files needed, if relationship is M to N).

JOINFIELD1, JOINFIELD2, JOINFIELD3, JOINFIELD4 (join fields for each join file).

If MTON is TRUE, then FILE2 and FILE4 are the same file (the corresponding file of this relationship).

(31) CORRES_FILES

Purpose : Get a corresponding data file name and corresponding multi-valued file names and multi-valued field names, given a class name.

Files accessed : MVATTRDBD, CLASSFILEDBD.

Input : CLASSNAME (a class name).

Output : PRIM_FILENAME (a corresponding data file name).

All corresponding multi-valued file names and multi-valued field names are stored in a linked list pointed by MV_RECPTR. If no such class name is found, then PRIM_FILENAME is set to blanks. If no multi-valued fields are found, then MV_RECPTR is set to NIL.

Remark :

(a) MV_RECPTR is of type PTR_MVREC.

```
      PTR_MVREC = ^MV_REC;
      MV_REC = RECORD
                 MV_FILENAME, MV_FIELDNAME : STRINGTYPE;
                 NEXT : PTR_MVREC;
               END;
```

(32) GET_MV_JOININFO

Purpose : Get multi-valued join information, given a multi-valued attribute name and a class name.

Files accessed : MVATTRDBD.

Input : CLASSNAME (a class name).

ATTRIBUTENAME (a multi-valued attribute name).

Output : MV_FILE (multi-valued file name).

MV_FIELD (multi-valued field name).

MV_JOINFIELD (the join field of the multi-valued file).

FILE2 (the main data file to join with).

JOINFIELD2 (the join field of the main data file).

If no such multi-valued attribute is found, then all output are set to blanks.

(33) GET_GENCAT_INFO

Purpose : Get join information of a generalization category, given a generalization category name and name of a class which takes part in the generalization.

Files accessed : GENCATJOINDBD.

Input : GEN_CATEGORY (a generalization category name).

CLASSNAME2 (name of a class taken part in the generalization category).

Output : GEN_FILE (main generalization category data file).

GEN_JOINFIELD (join field of the generalization data file).

FILE2 (data file of the class taken part in the generalization).

JOINFIELD2 (join field of FILE2).

(34) GET_SURROGATE_KEY

Purpose : Get surrogate key (if any) and corresponding file name, for a given class.

Files accessed : CLASSFILEDBD.

Input : CLASSNAME (a class name).

Output : Corresponding file name and surrogate key name are stored in a record, and this record is returned. If no surrogate key exists for the given class, then surrogate key name is set to blanks.

CHAPTER 6

CONCLUSION

The purpose of this thesis is to describe two components of the ECR database management system, the File system and the Data dictionary system.

The File system is built on the Indexed file organization of the VAX/VMS system, and offers a file interface for the other components of the ECR DBMS to access information stored in the data files. The Data dictionary system provides the schema definition parsing facilities and functional procedures for accessing information in the data dictionary files.

More changes can be made to the File system and the Data dictionary system to improve the efficiency of the whole ECRDBS. Offering concurrency control in the ECR File system and loading the ECR Data dictionary into the main memory during run time of the ECR DBMS are being proposed [Lin 87]. As main memory becomes less expensive and more high speed processors are available, the DBMS based on the ECR model will be more efficient.

The ECR model provides a semantically rich means of modeling user's views into a conceptual database schema, and information concerning semantic integrity constraints are stored in the Data dictionary for the ECR DBMS to maintain the semantic integrity of the database. Consequently, database applications can easily and naturally declare, reference and update data as viewed in the ECR model. Since the ECR model offers more descriptive semantics of a database and acceptable performance of the ECR DBMS can be achieved in the near future, therefore the ECR model will be no doubt the data model of the future.

APPENDIX

PROGRAM DESCRIPTION

The programs in this thesis are divided into four parts. They are :

I. Program for installing the ECR database system.

II. Program for creating the file interface.

III. Program for parsing the ECR schema definition.

IV. Program for creating data dictionary procedures and functions.

I. Program for installing the ECR database system.

(A) Description of the program 'INSTALLSYS'.

Program 'INSTALLSYS' is used to install the whole ECR database system by creating two empty files 'FILES.DIC' and 'SCHEMA.DIC'.

FILES.DIC -- to store the file dictionary

SCHEMA.DIC -- to store the schema names

(B) How to run the 'INSTALLSYS' program.

```
$RUN INSTALLSYS
```

II. Program for creating the file interface.

(A) Description of the program 'FILESYS'.

There are two levels of basic operations in the file interface. They are file level and record level operations.

There are a total of fourteen procedures one can use to perform these operations on files or records. The detailed procedures are listed in the following.

(1) File level operations

Data structure :

```
STR16 = PACKED ARRAY [1..16] OF CHAR;
STRINGTYPE = STR16;
FILENAMETYPE = STR16;

PTR_FIELDDEF = ^FIELDDEF;
FIELDDEF = RECORD
             FIELDNAME : STRINGTYPE;
             COMBINATION : CHAR;
             KEYNO : INTEGER;
             STARTPOS : INTEGER;
             LENGTH : INTEGER;
             FIELDTYPE : CHAR;
             DUP : CHAR;
             CHANGES : CHAR;
             NULL_KEY : CHAR;
             NEXT : PTR_FIELDDEF;
           END;
```

COMBINATION : ('Y' or 'N'), 'Y' for combination key

KEYNO : (-1..254), -1 for nonkey, 0 for primary key,
        1..254 for secondary keys

STARTPOS : starting position of a field in a record

LENGTH : length of a field in a record

FIELDTYPE : ('I, 'R', or 'S'), 'I' for integer,
            'R' for real, 'S' for character string

DUP : ('Y' or 'N'), 'Y' means duplicate key value allowed

CHANGES : ('Y' or 'N'), 'Y' means changes to key values allowed

NULL_KEY : ('Y' or 'N'), 'Y' means null key values allowed

(a) PROCEDURE CREATE_FILE (FILETYPE : CHAR; FILENAME : FILENAMETYPE; VAR FIELDS : PTR_FIELDDEF);

FILETYPE : ('I', 'R' OR 'S')

'I' for indexed file

'R' for relative file

'S' for sequential file

only indexed file organization is implemented in this thesis.

FILENAME should be 16 characters long without file extension. A '.DAT' extension will be added to the filename by the file system.

A linked list of FIELDDEF records containing field definitions should be provided and pointed at by FIELDS.

(b) PROCEDURE DELETE_FILE (FILENAME : FILENAMETYPE);

(c) PROCEDURE OPEN_FILE (FILENAME : FILENAMETYPE; FILENO : INTEGER):

FILENO can be any integer number. A file can be opened with different file numbers for concurrent read access.

(d) PROCEDURE CLOSE_FILE (FILENAME : FILENAMETYPE; FILENO :

```
                              INTEGER);
```

(e) PROCEDURE RESET_FILE (FILENAME : FILENAMETYPE; FILENO,

```
                           KEYNO : INTEGER; VAR FOUND : BOOLEAN);
```

Procedure RESET_FILE is used for sequential read access of a file in the order of the given key. If the file is not empty, then FOUND will be true.

(2) Record level operations

Data structure :

```
   STR8 = PACKED ARRAY [1..8] OF CHAR;
   VSTR512 = VARYING [512] OF CHAR;

   PTR_FIELDUNIT = ^FIELD_UNIT;
   FIELD_UNIT = RECORD
                  FIELDNAME : STRINGTYPE;
                  NEXT : PTR_FIELDUNIT;
                  CASE TAG : TAG_TYPE;
                    0 : (INUM : INTEGER);
                    1 : (CNUM : VSTR512);
                    2 : (RNUM : REAL);
                    3 : (INUM1, INUM2 : INTEGER);
                    4 : (TWINCNUM : STR8);
                END;

   COMPARISON_OP = (EQL, GEQ, GTR, NEQ, LTH, LEQ);
   CONST_VALUE = RECORD
                   CASE TAG : TAG_TYPE OF
                     0 : (INUM : INTEGER);
                     1 : (CNUM : VSTR512);
                     2 : (RNUM : REAL);
                     3 : (INUM1, INUM2 : INTEGER);
                     4 : (TWINCNUM : STR8);
                 END;

   LEAF_TYPE = (AL, SL, IL, RL);
   TREE_PTR = ^TREE_NODE;
   TREE_NODE = RECORD
                 LEFT, RIGHT : TREE_PTR;
                 OPERATOR : INTEGER;
                 TREE_TYPE : LEAF_TYPE;
                 CASE LEAF : LEAF_TYPE OF
```

```
                    AL : (ATTR_NAME : STRINGTYPE;
                          ATTR_TYPE : LEAF_TYPE);
                    SL : (STR_CONST : VSTR512);
                    IL : (INT_CONST : INTEGER);
                    RL : (REAL_CONST : REAL);
                END;
```

INUM : integer number

CNUM : character string

RNUM : real number

INUM1, INUM2 : integer combination key values

(a) PROCEDURE INSERT_RECORD (FILENAME : FILENAMETYPE; FILENO
: INTEGER; FIELDLIST : PTR_FIELDUNIT;
VAR INSERT_STATUS : INTEGER);

A linked list of FIELD_UNIT records containing field names and field values should be provided and pointed at by FIELDLIST. If the record has been inserted successfully, then INSERT_STATUS will be set to 0, else set to -1. The only exception is that when duplication key value error occurs, the INSERT_STATUS is set to 5.

(b) PROCEDURE SIMPLE_FIND (FILENAME : FILENAMETYPE; FILENO :
INTEGER ; FIELDNAME : STRINGTYPE; COMP_OP :
COMPARISON_OP; CONST_V : CONST_VALUE; VAR FOUND
: BOOLEAN);

Field value is supplied in CONST_V.

(c) PROCEDURE SIMP_FINDNEXT (FILENAME : FILENAMETYPE; FILENO
: INTEGER; VAR FOUND : BOOLEAN);

This procedure uses the same condition provided in the previous SIMPLE_FIND to find the next record satisfying the condition.

(d) PROCEDURE DELETE_RECORD (FILENAME : FILENAMETYPE; FILENO : INTEGER);

This procedure deletes the record which has been located by a previous SIMPLE_FIND or COMPLEX_FIND.

(e) PROCEDURE READ_RECORD (FILENAME : FILENAMETYPE; FILENO : INTEGER; VAR FIELDLIST : PTR_FIELDUNIT);

A linked list of FIELD_UNIT records containing field names and field values will be returned and pointed at by FIELDLIST.

(f) PROCEDURE MODIFY_RECORD (FILENAME : FILENAMETYPE; FILENO : INTEGER; FIELDLIST : PTR_FIELDUNIT; VAR STOP : BOOLEAN);

A linked list of FIELD_UNIT records containing field names and field values should be provided and pointed at by FIELDLIST.

(g) PROCEDURE FINDSEQ_NEXT (FILENAME : FILENAMETYPE; FILENO: INTEGER; VAR FOUND : BOOLEAN);

This procedure gets the next record according to the order specified by the previous RESET_FILE.

(h) PROCEDURE COMPLEX_FIND (FILENAME : FILENAMETYPE; FILENO : INTEGER; TREE : 'TREE_PTR; VAR FOUND : BOOLEAN);

TREE is a pointer to a condition tree and should be supplied to this procedure.

OPERATOR of the TREE_NODE :

1 (NOT)   2 (AND)   3 (OR)   4 (=)   5 (<>)

6 (>)   7 (>=)   8 (<)   9 (<=)

10 (+ real or mixed)   11 (- real or mixed)

12 (* real or mixed)   13 (/ real or mixed)

14 (+ integer)   15 (- integer)   16 (* integer)

17 (DIV integer)   18 (REM integer)   19 (MOD integer)

Mixed means real number mixed with integer number.

If OPERATOR equals to -1, then this TREE_NODE is a LEAF_NODE. For LEAF_NODE, LEAF_TYPE can be AL, SL, IL or RL.

LEAF_TYPE :	AL (attribute)
	SL (character string constant)
	IL (integer constant)
	RL (real constant)

(i) PROCEDURE COMP_FINDNEXT (FILENAME : FILENAMETYPE; FILENO : INTEGER; VAR FOUND : BOOLEAN);

This procedure uses the same condition provided in the previous COMPLEX_FIND to find the next record satisfying the condition.

(B) Examples of calling procedures in program 'FILESYS'.

(1) Sequential read according to a given key order

```
OPEN_FILE(FILENAME, FILENO);
RESET_FILE(FILENAME, FILENO, KEYNO, FOUND);
WHILE FOUND DO
  BEGIN
    READ_RECORD(FILENAME, FILENO, FIELDLIST);
    FINDSEQ_NEXT(FILENAME, FILENO, FOUND);
  END;
CLOSE_FILE(FILENAME, FILENO);
```

(2) Nonsequential read (indexed access)

(a) using SIMPLE_FIND and SIMP_FINDNEXT

```
OPEN_FILE(FILENAME, FILENO);
SIMPLE_FIND(FILENAME, FILENO, FIELDNAME, COMP_OP,
            CONST_V, FOUND);
WHILE FOUND DO
  BEGIN
    READ_RECORD(FILENAME, FILENO, FIELDLIST);
    SIMP_FINDNEXT(FILENAME, FILENO, FOUND);
  END;
CLOSE_FILE(FILENAME, FILENO);
```

(b) using COMPLEX_FIND and COMP_FINDNEXT

```
OPEN_FILE(FILENAME, FILENO);
COMPLEX_FIND(FILENAME, FILENO, TREE, FOUND);
WHILE FOUND DO
  BEGIN
    READ_RECORD(FILENAME, FILENO, FIELDLIST);
    COMP_FINDNEXT(FILENAME, FILENO, FOUND);
  END;
CLOSE_FILE(FILENAME, FILENO);
```

(3) Modify records in a file

(a) using SIMPLE_FIND

```
OPEN_FILE(FILENAME, FILENO);
SIMPLE_FIND(FILENAME, FILENO, FIELDNAME, COMP_OP,
            CONST_V, FOUND);
IF FOUND THEN
  STOP := FALSE
ELSE
  STOP := TRUE;
```

```
    WHILE NOT STOP DO
      MODIFY_RECORD(FILENAME, FILENO, FIELDLIST, STOP);
    CLOSE_FILE(FILENAME, FILENO);
```

(b) using COMPLEX_FIND

```
    OPEN_FILE(FILENAME, FILENO);
    COMPLEX_FIND(FILENAME, FILENO, TREE, FOUND);
    IF FOUND THEN
      STOP := FALSE
    ELSE
      STOP := TRUE;
    WHILE NOT STOP DO
      MODIFY_RECORD(FILENAME, FILENO, FIELDLIST, STOP);
    CLOSE_FILE(FILENAME, FILENO);
```

(4) Delete records in a file

(a) using SIMPLE_FIND

```
   OPEN_FILE(FILENAME, FILENO);
   SIMPLE_FIND(FILENAME, FILENO, FIELDNAME, COMP_OP,
               CONST_V, FOUND);
   WHILE FOUND DO
     BEGIN
       DELETE_RECORD(FILENAME, FILENO);
       SIMPLE_FIND(FILENAME, FILENO, FIELDNAME, COMP_OP,
                   CONST_V, FOUND):
     END;
   CLOSE_FILE(FILENAME, FILENO);
```

(b) using COMPLEX_FIND

```
    OPEN_FILE(FILENAME, FILENO);
    COMPLEX_FIND(FILENAME, FILENO, TREE, FOUND);
    WHILE FOUND DO
      BEGIN
        DELETE_RECORD(FILENAME, FILENO);
        COMPLEX_FIND(FILENAME, FILENO, TREE, FOUND);
      END;
    CLOSE_FILE(FILENAME, FILENO);
```

(5) Insert a record into a file

```
 OPEN_FILE(FILENAME, FILENO);
 INSERT_RECORD(FILENAME, FILENO, FIELDLIST,
               INSERT_STATUS);
 CLOSE_FILE(FILENAME, FILENO);
```

(6) Create a file, Delete a file

```
CREATE_FILE(FILETYPE, FILENAME, FIELDS);

DELETE_FILE(FILENAME);
```

(C) How to use the 'FILESYS' program.

Any program calling procedures in the program 'FILESYS' should do the following steps first.

(1) Put [INHERIT ('FILESYS')] in the first line of the program.

(2) Call the following two procedures in the beginning of the main program block.

```
INIT_FILESYS;
SET_SCHEMA(SCHEMA_NAME);
```

INIT_FILESYS initializes the file interface, and SET_SCHEMA restricts all file and record operations to a subdirectory(an ECR schema) under the current directory.

(3) Link program with 'FILESYS'.

(D) Error messages

```
                   Error messages
---------------------------------------------------------

CREATE_FILE ERROR -- FIELD DEFINITION ERROR
CREATE_FILE ERROR -- INDEX TYPE DEFINITION ERROR
CREATE_FILE ERROR -- COMBINATION INDEX DEFINITION ERROR
CREATE_FILE ERROR -- FILE ALREADY IN DICTIONARY
DELETE_FILE ERROR -- FILE NOT IN DICTIONARY
DELETE_FILE ERROR -- FILE NOT YET CLOSED
OPEN_FILE ERROR -- FILE NOT IN DICTIONARY
OPEN_FILE ERROR -- FILE WITH SAME FILENAME AND SAME FILE
                   NUMBER ALREADY OPENED
RESET_FILE ERROR -- NO SUCH KEY NUMBER
RESET_FILE ERROR -- FILE NOT YET OPENED
CLOSE_FILE ERROR -- FILE NOT YET OPENED
```

```
INSERT_RECORD ERROR -- FILE NOT YET OPENED
INSERT_RECORD ERROR -- FIELD NOT FOUND
INSERT_RECORD ERROR -- INSERTED FIELD STRING TOO LONG
INSERT_RECORD ERROR -- NULL VALUE NOT ALLOWED
INSERT_RECORD ERROR -- DUPLICATE KEY VALUES NOT ALLOWED
                       FOR KEY DUP = N
SIMPLE_FIND ERROR -- FILE NOT YET OPENED
SIMPLE_FIND ERROR -- FIELD NOT FOUND
SIMPLE_FIND ERROR -- CONSTANT STRING VALUE TOO LONG
SIMP_FINDNEXT ERROR -- FILE NOT YET OPENED
SIMP_FINDNEXT ERROR -- SIMPLE_FIND NOT ALREADY DONE OR
                       HAS BEING CLEARED
DELETE_RECORD ERROR -- FILE NOT YET OPENED
DELETE_RECORD ERROR -- RECORD UNDEFINED
READ_RECORD ERROR -- FILE NOT YET OPENED
READ_RECORD ERROR -- RECORD UNDEFINED
MODIFY_RECORD ERROR -- FILE NOT YET OPENED
MODIFY_RECORD ERROR -- RECORD UNDEFINED
MODIFY_RECORD ERROR -- FIELD NOT FOUND
MODIFY_RECORD ERROR -- UPDATE STRING VALUE TOO LONG
FINDSEQ_NEXT ERROR -- FILE NOT YET OPENED
FINDSEQ_NEXT ERROR -- RESET_FILE NOT ALREADY DONE, OR
                      HAS BEEN CLEARED
COMPLEX_FIND ERROR -- FILE NOT YET OPENED
COMPLEX_FIND ERROR -- ATTRIBUTE NOT FOUND
COMPLEX_FIND ERROR -- INVALID OPERATOR
COMPLEX_FIND ERROR -- TREE LEAF TYPE MISMATCH
COMP_FINDNEXT ERROR -- FILE NOT YET OPENED
COMP_FINDNEXT ERROR -- COMPLEX_FIND NOT ALREADY DONE, OR
                       HAS BEEN CLEARED
```

Explanation :

Possible causes for FIELD DEFINITION ERROR could be the following.

(1) fields not contiguous

(2) first field not starting from position 1

(3) integer field (not integer combination index) and real field, field length not equal to 4

Possible causes for COMBINATION INDEX DEFINITION ERROR could be the following.

(1) combination index not corresponding to contiguous string fields or two contiguous integer fields

(2) combination index not on fields of type string or type integer

III Program for parsing the ECR schema definition.

(A) Description of the program 'SCHEMA'.

Program 'SCHEMA' is used to parse the ECR definition language in an ECR schema definition file. Thirteen data dictionary files and data files for the database based on the schema definition are created. Schema name is stored into 'SCHEMA.DIC' file and information about each file, record, and field are stored into 'FILES.DIC' file.

If the schema name given in the definition file is not in 'SCHEMA.DIC' file and there is no errors in parsing the definition file, then a subdirectory is created and all data dictionary files and data files are put into the subdirectory created. The name of the subdirectory is the same as the schema name. If the schema name given already existed in 'SCHEMA.DIC', then an error message is displayed and no action will be taken. If there is an error during parsing the definition file, then all files related to the schema are deleted, the subdirectory is deleted, schema name is deleted from the 'SCHEMA.DIC' file and all records in 'FILES.DIC' related to the schema are deleted.

The procedure TRANSLATE_SCHEMA in program 'SCHEMA' calls other procedures to do the parsing of the schema definition language.

PROCEDURE TRANSLATE_SCHEMA (DEF_SCHEMAFILE : VSTR39);

The name of the schema definition file is provided by DEF_SCHEMAFILE. VSTR39 is a type of VARYING [39] OF CHAR.

(B) How to use the 'SCHEMA' program.

Any program calling procedure TRANSLATE_SCHEMA in the program 'SCHEMA' should do the following steps first.

(1) Put [INHERIT ('SCHEMA_FILEDEF', 'SCANNER', 'FILESYS')] in the first line of the program.

(2) Call the following two procedures in the beginning of the main program block.

```
    INIT_FILESYS;
    SET_SCHEMA(SCHEMA_NAME);
```

(3) Call procedure TRANSLATE_SCHEMA inside the program.

```
    TRANSLATE_SCHEMA(DEF_SCHEMAFILE);
```

(4) Link program with 'SCHEMA', 'MOD1' and 'FILESYS'. Program 'MOD1' is a scanner provided by Miss Yao to fetch tokens from the schema definition file.

(C) Data dictionary files.

Thirteen data dictionary files are generated during parsing an schema definition file. The detailed data structure of each data dictionary files are listed in the following.

(1) File 'VALUESETDBD.DAT'.

```
VALUESETDBDRECORD = RECORD
                       VALUESETNAME : [KEY(0)]STRINGTYPE;
                       REF : BOOLEAN;
                       ATTRIBUTENAME, CLASSNAME : STRINGTYPE;
                       CASE VALUESETTYPE : CHAR OF
                         'S' : (LEN : INTEGER);
                         'E' : (COUNT : INTEGER);
                         'I' : (MINI, MAXI : INTEGER);
                         'R' : (MINR, MAXR : REAL)
                    END;
```

```
VALUESETDBD : FILE OF VALUESETDBDRECORD;
```

Default :

```
   MINI : -MAXINT (-2147483647),  MAXI : MAXINT (2147483647)

   MINR : -1.70E38,  MAXR : 1.70E38
```

ATTRIBUTENAME and CLASSNAME are meaningful, only when REF equals to TRUE.

Access method :

```
   OPEN(VALUESETDBD, '[.'+SCHEMANAME+']VALUESETDBD.DAT', OLD
       , ORGANIZATION := KEYED, ACCESS_METHOD := KEYED);
```

(2) File 'EXPLICITDBD.DAT'.

File 'EXPLICITDBD.DAT' is an indexed file with two fields.

```
        [1] VALUESETNAME : STRINGTYPE;
        [2] DATA_VALUE : DATA_VALUETYPE;
```

Key 0 (EXPLICIT_PRIM) is the combination of field [1] and field [2]. Key 1 is field [1].

DATA_VALUETYPE is PACKED ARRAY [1..32] OF CHAR;

Access method :

```
   OPEN_FILE('EXPLICITDBD     ', FILENO);
```

(3) File 'ATTRIBUTEDBD.DAT'.

File 'ATTRIBUTEDBD.DAT' is an indexed file with nine fields.

```
[1] ATTRIBUTENAME : STRINGTYPE;
[2] BELONGTO : STRINGTYPE;
[3] OBJECT_TYPE : INTEGER;
[4] VALUESETNAME : STRINGTYPE;
[5] UNIQUE : CHAR;
[6] MIN : INTEGER;
[7] MAX : INTEGER;
[8] CORRES_FILE : STRINGTYPE;
[9] CORRES_FIELD : STRINGTYPE;
```

Key 0 (ATTRIBUTE_PRIM) is the combination of field [1] and field [2]. Key 1 is field [1]. Key 2 is field [2]. Key 3 (FILE_FIELD) is the combination of field [8] and field [9] .

Default :

```
UNIQUE : 'F',     MIN : 1,     MAX : 1
```

Access method :

```
OPEN_FILE('ATTRIBUTEDBD    ', FILENO);
```

(4) File 'TOTALDBDD.DAT'.

File 'TOTALDBD.DAT' is an indexed file with two fields.

```
[1] OBJECT_NAME : STRINGTYPE;
[2] OBJECT_TYPE : INTEGER;
```

OBJECT_TYPE :

```
1 : ATTRIBUTES,   2 : ENTITYTYPES
3 : CATEGORIES    4 : RELATIONSHIPS
5 : PARTICIPATION NAMES (CONNECTION NAMES)
6 : VALUESETS     7 : GCATEGORIES
```

Key 0 is field [1].

Access method :

```
   OPEN_FILE('TOTALDBD        ', FILENO);
```

(5) File 'ENTITYDBD.DAT'.

File 'ENTITYDBD.DAT' is an indexed file with five fields.

```
        [1] ENTITYTYPENAME : STRINGTYPE;
        [2] PARTICIP_NAME1 : STRINGTYPE;
        [3] PARTICIP_NAME2 : STRINGTYPE;
        [4] RELATION_NAME : STRINGTYPE;
        [5] RELATED_TO : STRINGTYPE;
```

Key 0 (ENTITY_PRIM) is the combination of field [1] and field [2]. Key 1 is field [1]. Key 2 is field [2].

If the relationship is not binary (more than two participants), then RELATED_TO will contain blanks.

Access method :

```
   OPEN_FILE('ENTITYDBD       ', FILENO);
```

(6) File 'CATEGORYRELDBD.DAT'.

File 'CATEGORYRELDBD.DAT' is an indexed file with five fields.

```
        [1] CATEGORYNAME : STRINGTYPE;
        [2] PARTICIP_NAME1 : STRINGTYPE;
        [3] PARTICIP_NAME2 : STRINGTYPE;
        [4] RELATION_NAME : STRINGTYPE;
        [5] RELATED_TO : STRINGTYPE;
```

Key 0 (CATEGORYREL_PRIM) is the combination of field [1] and field [2]. Key 1 is field [1]. Key 2 is field [2].

Access method :

```
   OPEN_FILE('CATEGORYRELDBD  ', FILENO);
```

(7) File 'CATEGORYDBD.DAT'.

File 'CATEGORYDBD.DAT' is an indexed file with eight fields.

```
[1] CATEGORYNAME : STRINGTYPE;
[2] DEFINED_ON_CLASS : STRINGTYPE;
[3] CLASS_TYPE: CHAR;
[4] ATTRIBUTE : STRINGTYPE;
[5] CONST_VAL : STRINGTYPE:
[6] CORRES_FILE : STRINGTYPE;
[7] CORRES_FIELD : STRINGTYPE;
[8] CTYPE : CHAR;
```

Key 0 (CATEGORY_PRIM) is the combination of field [1] and field [2]. Key 1 is field [1]. Key 2 is field [2].

Field [3] CLASS_TYPE can be 'E' or 'C' (for entitytype or category). Field [8] CTYPE can be 'G' or 'S' (for generalization or subclass category).

If no <predicate> on DEFINED_ON_CLASS, then field [4] ATTRIBUTE and field [5] CONST_VAL will contain blanks.

Access method

```
OPEN_FILE('CATEGORYDBD     ', FILENO);
```

(8) File 'RELATIONSHIPDBD.DAT'.

File 'RELATIONSHIPDBD.DAT' is an indexed file with ten fields.

```
[1] RELATION_NAME : STRINGTYPE;
[2] RELATION_FROM : STRINGTYPE;
[3] PARTICIP_NAME1 : STRINGTYPE;
[4] PARTICIP_NAME2 : STRINGTYPE;
[5] FROM_TYPE : CHAR;
[6] SPECIFIC : CHAR;
[7] PD : CHAR;
[8] DLT : CHAR;
[9] MIN : INTEGER;
```

```
      [10] MAX : INTEGER;
```

Key 0 (RELATION_PRIM) is the combination of field [1], field [2] and field [3]. Key 1 is field [1]. Key 2 is field [2]. Key 3 (RELATION_TRD) is the combination of field [2] and field [3].

Default :

```
   SPECIFIC : 'F',     PD : 'F',     DLT : 'F'
   MIN : 0,     MAX : MAXINT
```

Field [5] FROM_TYPE can be 'E' or 'C'. Field [6] SPECIFIC can be 'T' or 'F'. Field [7] PD can be 'T' or 'F'. Field [8] DLT can be 'T' or 'F'.

Access method :

```
   OPEN_FILE('RELATIONSHIPDBD ', FILENO);
```

(9) File 'JOINDBD.DAT'.

File 'JOINDBD.DAT' is an indexed file with nine fields.

```
      [1] CLASS_NAME1 : STRINGTYPE;
      [2] PARTICIP_NAME1 : STRINGTYPE;
      [3] CLASS_NAME2 : STRINGTYPE;
      [4] PARTICIP_NAME2 : STRINGTYPE;
      [5] RELATION_NAME : STRINGTYPE;
      [6] FILE1 : STRINGTYPE;
      [7] JOINFIELD1 : STRINGTYPE;
      [8] FILE2 : STRINGTYPE;
      [9] JOINFIELD2 : STRINGTYPE;
```

Key 0 (JOIN_PRIM) is the combination of field [1] and field [2]. Key 1 (JOIN_SCND) is the combination of field [3] and field [4].

Access method :

```
   OPEN_FILE('JOINDBD          ', FILENO);
```

(10) File 'MVATTRDBD.DAT'.

File 'MVATTRDBD.DAT' is an indexed file with seven fields.

```
       [1] ATTRIBUTENAME : STRINGTYPE;
       [2] CLASSNAME : STRINGTYPE;
       [3] MV_FILE : STRINGTYPE;
       [4] MV_JOINFIELD : STRINGTYPE;
       [5] FILE2 : STRINGTYPE;
       [6] JOINFIELD2 : STRINGTYPE;
       [7] MV_FIELD : STRINGTYPE;
```

Key 0 (MVATTR_PRIM) is the combination of field [1] and field [2]. Key 1 is field [1]. Key 2 is field [2].

Access method :

```
   OPEN_FILE('MVATTRDBD        ', FILENO);
```

(11) File 'GENCATJOINDBD.DAT'.

File 'GENCATJOINDBD.DAT' is an indexed file with six fields.

```
       [1] GEN_CATEGORYNAME : STRINGTYPE;
       [2] CLASSNAME2 : STRINGTYPE;
       [3] GEN_FILE : STRINGTYPE;
       [4] GEN_JOINFIELD : STRINGTYPE;
       [5] FILE2 : STRINGTYPE;
       [6] JOINFIELD2 : STRINGTYPE;
```

Key 0 (GEN_PRIM) is the combination of field [1] and field [2]. Key 1 is field [1]. Key 2 is field [2].

Access method :

```
   OPEN_FILE ('GENCATJOINDBD    ', FILENO);
```

(12) File 'KEYVALUESDBD.DAT'.

File 'KEYVALUESDBD.DAT' is an indexed file with three fields.

```
[1] FILENAME : STRINGTYPE;
[2] KEY_FIELD : STRINGTYPE;
[3] HIGHEST_KEY : INTEGER;
```

Key 0 (KEYV_PRIM) is the combination of field [1] and field [2]. Key 1 is field [1].

Access method :

```
OPEN_FILE('KEYVALUESDBD    ', FILENO);
```

(13) File 'CLASSFILEDBD.DAT'.

File 'CLASSFILEDBD.DAT' is an indexed file with four fields.

```
[1] CLASSNAME : STRINGTYPE;
[2] CORRES_FILE : STRINGTYPE;
[3] SURROGATE_KEY : STRINGTYPE;
[4] MAJOR : CHAR;
```

Key 0 is field [1]. Key 1 is field [2].

Key 2 is field [3].

Field [4] MAJOR can be 'Y' or 'N'. This field indicates which classname is the major corresponding classname of a given file.

Access method :

```
OPEN_FILE('CLASSFILEDBD    ', FILENO);
```

(D) Error messages.

Two kinds of error messages are provided by program 'Schema'.

(1) Syntax error messages

(2) Database definition error (DBD ERROR) messages

Unexpected token or illegal token will produce an syntax error. Inconsistency in the schema definition will produce an database definition error.

Error messages for SYNTAX ERROR

```
------------------------------------------------------------

SYNTAX ERROR -- INVALID <CONSTANT>
SYNTAX ERROR -- "," OR "}" EXPECTED
SYNTAX ERROR -- INVALID <STANDARD VALUE SET>
SYNTAX ERROR -- INVALID <REFERENCE VALUE SET>
SYNTAX ERROR -- ":" EXPECTED
SYNTAX ERROR -- "DEFINE" EXPECTED
SYNTAX ERROR -- "\" OR "END" EXPECTED
SYNTAX ERROR -- INVALID <VALUE SET NAME>
SYNTAX ERROR -- "AS" EXPECTED
SYNTAX ERROR -- INVALID <INTEGER>
SYNTAX ERROR -- "\", "END", OR <CONSTRAINED ATTRIBUTE>
                EXPECTED
SYNTAX ERROR -- INVALID <CONSTRAINED ATTRIBUTE>
SYNTAX ERROR -- INVALID <ATTRIBUTE NAME>
SYNTAX ERROR -- "VALUESET" EXPECTED
SYNTAX ERROR -- INVALID <VALUE SET NAME>
SYNTAX ERROR -- INVALID <ENTITY TYPE NAME>
SYNTAX ERROR -- "ATTRIBUTES" EXPECTED
SYNTAX ERROR -- "VALUESET", "CATEGORY","FILE","RELATIONSHIP"
                OR "ENTITYTYPE" EXPECTED
SYNTAX ERROR -- "SCHEMA" EXPECTED
SYNTAX ERROR -- INVALID <SCHEMA NAME>
SYNTAX ERROR -- INVALID <CATEGORY NAME>
SYNTAX ERROR -- "FROM" EXPECTED
SYNTAX ERROR -- INVALID <ENTITY TYPE NAME> OR
                <CATEGORY NAME>
SYNTAX ERROR -- ",", ":", "END", "\", "SPECIFIC", OR
                "ATTRIBUTES" EXPECTED
SYNTAX ERROR -- INVALID <RELATIONSHIP NAME>
SYNTAX ERROR -- "\", "END", OR "ATTRIBUTES" EXPECTED
SYNTAX ERROR -- "SPECIFIC", "PD", OR "(" EXPECTED
SYNTAX ERROR -- INVALID <PARTICIPATION NAME>
SYNTAX ERROR -- "," EXPECTED
SYNTAX ERROR -- ")" EXPECTED
SYNTAX ERROR -- "MIN", "MAX", "\", "END", ",", "ATTRIBUTES",
                <CATEGORY NAME> OR <ENTITY TYPE NAME>
                EXPECTED
```

```
SYNTAX ERROR -- "\",  "END",  ",",  <CATEGORY NAME>,
                "ATTRIBUTES", OR <ENTITY TYPE NAME> EXPECTED
SYNTAX ERROR -- "=" EXPECTED
SYNTAX ERROR -- <SCHEMA NAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- <VALUE SET NAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- <ATTRIBUTE NAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- <CLASS NAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- <ENTITY TYPE NAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- <CATEGORY NAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- <RELATIONSHIP NAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- <PARTICIPATION NAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- <CATEGORY NAME> OR <ENTITY TYPE NAME> LONGER
                THAN 16 CHARS
SYNTAX ERROR -- INVALID <FILENAME>
SYNTAX ERROR -- <FILENAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- "TYPE" EXPECTED
SYNTAX ERROR -- "INDEX", "REL", OR "SEQ" EXPECTED
SYNTAX ERROR -- "CORRESPONDS" EXPECTED
SYNTAX ERROR -- INVALID <CLASS NAME>
SYNTAX ERROR -- "FIELDS" EXPECTED
SYNTAX ERROR -- <FIELD NAME> LONGER THAN 16 CHARS
SYNTAX ERROR -- INVALID <STARTING LOCATION>, <INTEGER>
                EXPECTED
SYNTAX ERROR -- "I", "R", OR "S" EXPECTED
SYNTAX ERROR -- INVALID <LENGTH>, <INTEGER> EXPECTED
SYNTAX ERROR -- INVALID <KEYNO>, <INTEGER> EXPECTED
SYNTAX ERROR -- "CORRESPONDS", "REFERS", "CATEGORY", ",",
                "\", OR "END" EXPECTED
SYNTAX ERROR -- ",", "END", OR "\" EXPECTED
SYNTAX ERROR -- "OF" EXPECTED
SYNTAX ERROR -- INVALID <FIELD NAME>
SYNTAX ERROR -- "FOR", ",", "\", OR "END" EXPECTED
SYNTAX ERROR -- "Y" OR "N" EXPECTED
SYNTAX ERROR -- INVALID <CLASS NAME> OR <ATTRIBUTE NAME>
```

Error messages for DBD ERROR

```
------------------------------------------------------------

DBD ERROR -- VALUESET NOT DEFINED
DBD ERROR -- ATTRIBUTE MIN SHOULD >= 0
DBD ERROR -- ATTRIBUTE MAX SHOULD >= 1
DBD ERROR -- RELATIONSHIP NOT DEFINED
DBD ERROR -- DEFINED_ON_CLASS OF CATEGORY IS NEITHER
             CATEGORY NOR ENTITYTYPE
```

```
DBD ERROR -- ATTRIBUTE NOT DEFINED
DBD ERROR -- PARTICIPATION <CATEGORY NAME> OR
             <ENTITYTYPE NAME> NOT DEFINED
DBD ERROR -- PARTICIPATION MIN SHOULD >= 0
DBD ERROR -- PARTICIPATION MAX SHOULD >= 1
DBD ERROR -- <ATTRIBUTENAME> AND <CLASS NAME> COMBINATION
             NOT FOUND
DBD ERROR -- <CATEGORY NAME> NOT DEFINED
DBD ERROR -- <CLASS NAME> OR <ATTRIBUTE NAME> NOT DEFINED
DBD ERROR -- <CLASS NAME> NOT DEFINED
DBD ERROR -- <PARTICIPATION NAME> NOT DEFINED
DBD ERROR -- SCHEMA ALREADY EXISTED
```

IV. Program for creating data dictionary procedures and functions.

(A) Description of program 'DICTIONARY'.

Program 'DICTIONARY' contains 34 procedures and functions for users or other components of the ECR DBMS system to access the information stored in the file dictionary ('FILES.DIC'), 'SCHEMA.DIC' file, and data dictionary files for a particular ECR schema. The detailed procedures and functions are listed in the following.

Data structure :

```
  SCHEMANAMETYPE = VARYING [16] OF CHAR;
  PTR_SCHEMAREC = ^SCHEMAREC;
  SCHEMAREC = RECORD
                 SCHEMANAME : SCHEMANAMETYPE;
                 NEXT : PTR_SCHEMAREC;
               END;
  PTR _FIELDNAMEREC = ^FIELDNAMEREC;
  FILENAMEREC = RECORD
                   FIELDNAME : STRINGTYPE;
                   FTYPE : CHAR;  (* 'C' or 'F' *)
                   NEXT : PTR_FIELDNAMEREC;
                 END;
  PTR_UNIQFIELDS = ^UNIQFIELDREC;
```

```
UNIQFIELDREC = RECORD
                 UNIQFIELD : STRINGTYPE;
                 NEXT : PTR_UNIQFIELDS;
               END;
PTR_OBJECTREC = ^OBJECTREC;
OBJECTREC = RECORD
              OBJECT_NAME : STRINGTYPE;
              OBJECT_TYPE : INTEGER;
              NEXT : PTR_OBJECTREC;
            END;
PTR_CLASSNAME = ^CLASSNAME_NODE;
CLASSNAME_NODE = RECORD
                   CLASSNAME : STRINGTYPE;
                   VERSION : INTEGER;
                   NEXT : PTR_CLASSNAME;
                 END;
STR32 = PACKED ARRAY [1..32] OF CHAR;
DATA_VALUETYPE = STR32;
PTR_VALUESETRECORD = ^VALUESETDBDRECORD;
VALUESETDBDRECORD = RECORD
                      VALUESETNAME : [KEY(0)]STRINGTYPE;
                      REF : BOOLEAN;
                      ATTRIBUTENAME,CLASSNAME :STRINGTYPE;
                      CASE VALUESETTYPE : CHAR OF
                        'S' : (LEN : INTEGER);
                        'E' : (COUNT : INTEGER);
                        'I' : (MINI, MAXI : INTEGER);
                        'R' : (MINR, MAXR : REAL)
                    END;
VSTR32 = VARYING [32] OF CHAR;
PTR_EXPLICIT = ^EXPLICIT_UNIT;
EXPLICIT_UNIT = RECORD
                  DATA_VALUE : VSTR32;
                  NEXT : PTR_EXPLICIT;
                END;
UNIQKEYLIST = ^UNIQKEYREC;
UNIQKEYREC = RECORD
               ATTRIBUTENAME : STRINGTYPE;
               CORRES_FILE : STRINGTYPE;
               CORRES_FIELD : STRINGTYPE;
               NEXT : UNIQKEYLIST;
             END;
PTR_ATTRIBUTEREC = ^ATTRIBUTEREC;
ATTRIBUTEREC = RECORD
                 ATTRIBUTENAME : STRINGTYPE;
                 NEXT : PTR_ATTRIBUTEREC;
               END;
ATTR_NODE_PTR = ^FULL_ATTR_NODE;
```

```
TRN_CLASS_PTR = ^TRN_CLASS_NODE;
TRN_CLASS_NODE = RECORD
                   CL_NAME : STRINGTYPE;
                   CL_VERSION : INTEGER
                 END;
FULL_ATTR_NODE = RECORD
                   ATTRIBUTE : STRINGTYPE;
                   CLASS : TRN_CLASS_PTR;
                   OLD_FIELD : STRINGTYPE;
                   FIELD : STRINGTYPE;
                   NEW_FIELD : STRINGTYPE;
                   FTYPE : CHAR;
                   DTYPE : CHAR;
                   DLEN : INTEGER;
                   NEXT_NODE : ATTR_NODE_PTR;
                 END;
PTR_RELATED = ^RELATEDREC;
RELATEDREC = RECORD
               RELATED_TO : STRINGTYPE;
               PARTICIP_NAME1 : STRINGTYPE;
               PARTICIP_NAME2 : STRINGTYPE;
               NEXT : PTR_RELATED;
             END;
CTREE_PTR = ^CNODETYPE;
CNODETYPE = RECORD
              PARENT_PTR : CTREE_PTR;
              DEFINING_TREE : CTREE_PTR;
              DEFINEDON_TREE : CTREE_PTR;
              CLASSNAME : STRINGTYPE;
              CTYPE : CHAR;  (* 'R', 'S' or 'G' *)
              LEVEL : INTEGER;
              NEXT_PTR : CTREE_PTR;
              NEXT_QPTR : CTREE_PTR;
            END;
CLISTPTR = ^NAME_NODE;
NAME_NODE = RECORD
              CTYPE : CHAR;  (* 'S' or 'G' *)
              CLASSNAME : STRINGTYPE;
              LEVEL : INTEGER;
              NEXT : CLISTPTR;
            END;
PTR_DEFINEDON = ^DEFINEDREC;
DEFINEDREC = RECORD
               DEFINE : STRINGTYPE;
               NEXT : PTR_DEFINEDON;
             END;
PTR_DEFINING = PTR_DEFINEDON;
DEFININGREC = DEFINEDREC;
```

```
  PTR_MVREC = ^MV_REC;
  MV_REC = RECORD
              MV_FILENAME, MV_FIELDNAME : STRINGTYPE;
              NEXT : PTR_MVREC;
           END;
```

(1) PROCEDURE SET_SCHEMA (SCHEMA : SCHEMANAMETYPE);

PURPOSE : Set the current schema to the given schema name.

(2) PROCEDURE DELETE_DATABASE (SCHEMANAME : SCHEMANAMETYPE);

PURPOSE : Delete a database (an ECR schema).

(3) FUNCTION GET_SCHEMA : PTR_SCHEMAREC;

PURPOSE : Get all schema names in the ECR DBMS.

```
(4) FUNCTION FIELD_INFO (FILENAME, FIELDNAME : STRINGTYPE) :
                        PTR_FIELDDEF;
```

PURPOSE : Get field definition for a field, given a file name and a field name.

REMARK : This function returns NIL, if no such field is found.

```
(5) FUNCTION FILE_FIELD_INFOS (FILENAME : STRINGTYPE) :
                              PTR_FIELDDEF
```

PURPOSE : Get all field definitions for fields in a file, given a filename.

REMARK : This function returns NIL, if no such file is found.

```
(6) PROCEDURE FILE_FIELD_NAMES (DATAFILENAME : STRINGTYPE;
                   VAR FIELDNAMERECPTR : PTR_FIELDNAMEREC);
```

PURPOSE : Get all field names and indicate all combination key fields, given a file name.

REMARK : FTYPE -- 'C' for combination key field.

'F' for noncombinational field.

(7) FUNCTION UNIQUE_FIELDS (FILENMAME : STRINGTYPE) : PTR_UNIQFIELDS;

PURPOSE : Get all unique field names, given a file name.

REMARK : This function returns NIL, if no unique fields are found.

(8) FUNCTION OBJECT_TYPE (ONAME : STRINGTYPE) : INTEGER;

PURPOSE : Get the object type, given an object name of the current schema.

(9) FUNCTION GET_OBJECT_INFO : PTR_OBJECTREC;

PURPOSE : Get all object names and object types of the current schema (not including value sets).

(10) FUNCTION ALL_CLASSNAMES : PTR_CLASSNAME;

PURPOSE : Get all class names (entity type names, category names and relationship names) of the current schema.

(11) PROCEDURE VALUESET_INFO (ATTR_NAME, CLASSNAME : STRINGTYPE; VAR VALPTR : PTR_VALUESETRECORD);

PURPOSE : Get all information about a value set, given an attribute name and a class name.

REMARK : VALPTR is set to NIL, if no such attribute name and class name are found.

(12) PROCEDURE EXPLICIT_VALUE (VNAME : STRINGTYPE; VAR EXPTR : PTR_EXPLICIT);

PURPOSE : Get all explicit data values for an explicit value set, given a value set name.

REMARK : EXPTR is set to NIL, if no such explicit value set is found.

(13) FUNCTION GET_CLASS_UNIQKEYS (CLASSNAME : STRINGTYPE) : UNIQKEYLIST;

PURPOSE : Get attribute names, corresponding file names, corresponding field names for all unique attributes in a given class.

(14) FUNCTION BELONG_TO (ATTR_NAME, CLASSNAME : STRINGTYPE) : BOOLEAN;

PURPOSE : Check whether an attribute belongs to a class.

(15) PROCEDURE ATTRIBUTE_NAME (CLASSNAME : STRINGTYPE; VAR ATTPTR : PTR_ATTRIBUTEREC);

PURPOSE : Get all attribute names which belong to a class.

REMARK : ATTRPTR is set to NIL, if no attributes belong to this class.

(16) FUNCTION ATTRIBUTE_UNIQUE (ATTR_NAME, CLASSNAME : STRINGTYPE) : BOOLEAN;

PURPOSE : Check whether an attribute of a class is unique.

(17) FUNCTION CHECK_MV_ATTR (ATTR_NAME, CLASSNAME : STRINGTYPE) : BOOLEAN;

PURPOSE : Check whether an attribute of a class is a multi-valued attribute.

(18) FUNCTION GET_FIELD_ATTR_CLASS (FILENAME : STRINGTYPE) : ATTR_NODE_PTR;

PURPOSE : Get all corresponding attribute names, class names, field names, field type, field length, and combination field indicator, given a file name.

REMARK : FTYPE -- 'C' for combinational key field.

'F' for noncombinational field.

DTYPE -- data field type, 'I' for integer,

'R' for real, 'S' for character string.

```
(19) PROCEDURE ENTITY_RELATE_NAME (ENTITYTYPENAME :
                        STRINGTYPE; VAR RELPTR : PTR_RELATED);
```

PURPOSE : Get all class names (entity type names or category names) and participation names related to a given entity type.

REMARK : RELPTR is set to NIL, if no related class names are found.

```
(20) PROCEDURE CHECK_CONNECTION (CLASSNAME, CNAME :
                           STRINGTYPE; VAR RNAME, CLASSNAME2 :
                           STRINGTYPE; VAR YESNO : BOOLEAN);
```

PURPOSE : Get relationship name, related class name (category name or entity type name), given a class name (entity type name or category name) and a connection name.

```
(21) FUNCTION CREATE_CTREE (CLASSNAME : STRINGTYPE) :
                            CTREE_PTR;
```

PURPOSE : Get a C tree (category tree) which contains information about all defining categories and defined on classes (entity types or categories) related to the given class (as root of the C tree).

```
(22) FUNCTION GET_DEFINE_LIST (CLASSNAME : STRINGTYPE) :
                              CLISTPTR;
```

PURPOSE : Get all class names (entity type names or category names), category types and category levels related to the given class (entity type or category) in the order of ascending level value.

```
(23) FUNCTION DEFINED_ON (CLASSNAME, DEFINED_NAME :
                         STRINGTYPE) : BOOLEAN;
```

PURPOSE : Check whether a class name is defined on the given defined name.

Remark : This function works for both subclass and generalization categories.

```
(24) PROCEDURE REPEAT_DEFINEDON_SUBCLASS (CATEGORYNAME :
                 STRINGTYPE; VAR DEFINEDPTR : PTR_DEFINEDON);
```

PURPOSE : Get all level superclass class names of a given subclass category.

REMARK : (a) DEFINEDPTR is set to NIL, if no defined on superclasses are found.

(b) This procedure works for subclass category only.

```
(25) PROCEDURE DEFINING_SUBCLASS (CLASSNAME : STRINGTYPE;
                            VAR DEFININGPTR : PTR_DEFINING);
```

PURPOSE : Get all first level subclass defining class names of a given class.

REMARK : (a) DEFININGPTR is set to NIL, if no defining subclasses are found.

(b) This procedure works for subclass category only.

```
(26) PROCEDURE REPEAT_DEFINING_SUBCLASS (CLASSNAME :
                STRINGTYPE; VAR DEFININGPTR : PTR_DEFINING);
```

PURPOSE : Get all level defining subclass class names of a given class.

REMARK : (a) DEFININGPTR is set to NIL, if no defining subclasses are found.

(b) This procedure works for subclass category only.

```
(27) PROCEDURE CATEGORY_RELATE_NAME (CATEGORYNAME :
                     STRINGTYPE; VAR RELPTR : PTR_RELATED);
```

PURPOSE : Get all class names (entity type names or category names) and participation names related to a category.

REMARK : RELPTR is set to NIL, if no related class names are found.

```
(28) PROCEDURE GET_CONN_MIN_MAX (CLASSNAME, CNAME :
                         STRINGTYPE; VAR MIN, MAX : INTEGER);
```

PURPOSE : Get MIN and MAX structural constraints for one end of participation in a relationship, given a class name (entity type name or category name) and a connection name.

```
(29) PROCEDURE CHECK_RELATIONSHIP (CLASSNAME, CNAME :
           STRINGTYPE; VAR RNAME, CNAME2, CLASSNAME2 :
           STRINGTYPE; VAR MIN, MAX, MIN2, MAX2 : INTEGER;
           VAR YESNO : BOOLEAN);
```

PURPOSE : Get relationship name, related class name (entity type name or category name), connection name for the related class, and MIN, MAX structural constraints for both ends of participations in the relationship, given a class name (entity type name or category name) and a connection name.

```
(30) PROCEDURE GET_JOIN_ATTRIBUTE (CLASSNAME, CNAME :
               STRINGTYPE; VAR FILE1, JOINFIELD1, FILE2,
               JOINFIELD2, FILE3, JOINFIELD3, FILE4,
               JOINFIELD4 : STRINGTYPE; VAR MTON : BOOLEAN);
```

PURPOSE : Get join information of a relationship, given a class name (entity type name or category name) and a connection name.

```
(31) PROCEDURE CORRES_FILES (CLASSNAME : STRINGTYPE; VAR
                 PRIM_FILENAME : STRINGTYPE; VAR MV_RECPTR :
                 PTR_MVREC);
```

PURPOSE : Get a corresponding data file name and corresponding multi-valued file names and multi-valued field names, given a class name.

REMARK : (a) PRIM_FILENAME is set to blanks, if no such class name is found.

(b) MV_RECPTR is set to NIL, if no multi-valued fields are found.

```
(32) PROCEDURE GET_MV_JOININFO (CLASSNAME, ATTRIBUTENAME :
                  STRINGTYPE ; VAR MV_FILE, MV_JOINFIELD,
                  FILE2, JOINFIELD2, MV_FIELD : STRINGTYPE);
```

PURPOSE : Get multi-valued join information, given an multi-valued attribute name and a class name.

REMARK : This procedure sets all output variables to blanks, if no such multi-valued attribute is found.

```
(33) PROCEDURE GET_GENCAT_INFO (GEN_CATEGORY, CLASSNAME2 :
                    STRINGTYPE; VAR GEN_FILE, GEN_JOINFIELD,
                    FILE2, JOINFIELD2 : STRINGTYPE);
```

PURPOSE : Get join information of a generalization category, given a generalization category name and name of a class which takes part in the generalization.

```
(34) FUNCTION GET_SURROGATE_KEY (CLASSNAME : STRINGTYPE):
                                 UNIQKEYREC;
```

PURPOSE : Get surrogate key (if any) and corresponding file name, for a given class.

REMARK : If no surrogate key exists for the given class, then surrogate key name is set to blanks.

(B) How to use the 'DICTIONARY' program.

Any program calling procedures and functions in the program 'DICTIONARY' should do the following steps first.

(1) Put [INHERIT ('DICTIONARY','SCHEMA_FILEDEF', 'SCANNER', 'FILESYS')] in the first line of the program.

(2) Call the following procedures in the beginning of the program block.

```
INIT_FILESYS;
SET_SCHEMA(SCHEMA_NAME);
```

(3) Link program with 'DICTIONARY', 'SCHEMA', 'MOD1', and 'FILESYS'.

REFERENCES

[Bach 77] C. Bachman and M. Daya, "The Role Concept in Database Models", VLDB77 Proceedings, IEEE, Tokyo, Japan, September 1977, pp. 464-476.

[Back 78] J. Backus, "A functional style and its algebra of programs", Commun. ACM, Vol. 21, No. 8, 1978, pp. 613-641.

[Bune 77] P. Buneman and R. E. Frankel, "FQL - A functional query language", Proc. ACM SIGMOD, Int. Conf. Management of Data, Boston, May 30 - June 1, ACM, New York, 1979, pp. 52-57.

[Bune 82] P. Buneman, R. E. Frankel and R. Nikhil, "An implementation technique for database query languages", ACM Trans. Database Systems, Vol. 7, No. 2, 1982, PP. 164-186.

[Cham 76] D. D. Chamberlin, et al. "SEQUEL 2 : A Unified Approach to Data Definition, Manipulation and Control", in IBM Journal of Research and Development, Vol 20, No. 6, November 1976.

[Chan 83] K. Chang, "A Data Dictionary System for a High-Level Data Model", Master's Thesis, Computer Science Department, University of Houston, August 1983.

[Chen 76] P. P. Chen, "The Entity-relationship model: towards a unified view of data", ACM Trans. Database Systems, Vol. 1, No. 1, 1976, pp 9-36.

[Chen 83] P. P. Chen (Ed.), "Entity-Relationship Approach to Information Modeling and Analysis", North-Holland, Amsterdam 1983.

[CODA 71] CODASYL (Committee On Data Systems Languages), Data Base Task Group Report, ACM, New York, 1971.

[CODA 78] CODASYL (Committee On Data Systems Languages), Data Base Task Group Revised Report, ACM, New York, 1978

[Codd 70] E. F. Codd, "A Relational Model for Large Shared Data Banks", Communications of the ACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[Codd 71] E. F. Codd, "A Data Base Sublanguage Founded on the Relational Calculus", Proc. 1971 ACM SIGFIDET Workshop on Data Description, Access and Control, pp. 35-68.

[Codd 72] E. F. Codd, "Relational Completeness of Data Base Sublanguage", Data Base Systems, Courant Computer Science Symposia Series, Vol. 6, Printice Hall, 1972.

[Codd 72a] E. F. Codd, "Further Normalization of the Data Base Relational Model", Data Base Systems, Courant Computer Science Symposia Series, Vol 6, Printice Hall, 1972.

[Date 74] C. J. Date and E. F. Codd, "The Relational and Network Approaches: Comparison of the Application Programming Interfaces", Proc. 1974 ACM SIGMOD Workshop.

[Date 81] C. J. DATE, "An Introduction to Database Systems". Addison-Wesley Publishing Company, 3rd Edition, Vol. 1, 1981.

[Dec 83] Digital Equipment Corporation, "Programming in VAX-11 PASCAL", July, 1983.

[Dec 84] Digital Equipment Corporation, "VAX-11 Record Management Services Reference Manual", "VAX-11 Record Management Services Utilities Reference Manual", "VAX-11 Record Management Services Tuning Guide", "VAX/VMS File Definition Language Facility Reference Manual", September, 1984.

[Elma 80] R. Elmasri, "Semantic Integrity in DDTS (Distributed Database Test System)", Honeywell CCSC, Technical Report HR-80-274 : 17-38, Bloomington, Minnesota.

[Elma 81] R. Elmasri, "GORDAS: A Data Definition Query and Update Language for the Entity-Category-Relationship Model of Data",Honeywell CCSC, Technical Report HR-81-250 : 17-38, Bloomington, Minnesota.

[Elma 83] R. Elmasri and G. Wiederhold, "GORDAS: A formal high-level query language for the entity-relationship model", in [Chan 83], pp. 49-72.

[Elma 84] R. Elmasri, "A DBMS Based On an Extended ER Model", Technical Report #UH-CS-84-5, Department of Computer Science, University of Houston, May 1984.

[Elma 85] R. Elmasri, J. Weeldreyer and A. Hevner, "The category concept: An extension to the entity-relationship model", Data & knowledge Engineering, Vol. 1, No. 1, 1985, pp 75-116.

[Klug 77] A. Klug and D. Tsichritzis (Eds.), "The ANSI/X3/SPARC Report of the Study Group on Data Base Management Systems", AFIPS press, 1977.

[Lin 87] X. Lin, "Extensions of the ECRDBS High Level Database System", Master Thesis, Dept. of Computer science, University of Houston, 1987, in preparation.

[Mins 73] M. Minsky, "Computer Science and Representation of Knowledge", Proceeding of the National Computer Conference, AFIPS, Vol. 42, 1973.

[Ship 81] D. W. Shipman, "The functional data model and the data language DAPLEX", ACM Trans. Database System, Vol. 6, No. 1, 1981, pp. 140-173.

[Smit 77] J. Smith and D. Smith, "Database Abstractions : Aggregation and Generalization", ACM Transactions on Database Systems, Vol. 2, No. 2, June 1977, pp. 105-133.