EXTRACTING FUNCTIONAL DEPENDENCIES AND SYNONYMS FROM RELATIONAL DATABASES

by

Xiaobing Chen

B.S.E.E. Tsinghua University, Beijing, China, 1986

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE

in the School
of
Computing Science

SIMON FRASER UNIVERSITY
November 1992

APPROVAL

Name:	Xiaobing Chen
Degree:	Master of Science
Title of thesis:	Extracting Functional Dependencies and Synonyms from Relational Databases

Examining Committee: Dr. F. David Fracchia, Chairman

Dr. Nick Cercone, Senior Supervisor

Dr. Jiawei Han, Supervisor

Mr. Gary Hall, Examiner

Date Approved: 18 November 1992

ABSTRACT

To build a natural language interface that accesses relational databases, it is important to analyze the underlying databases in order to provide a semantic representation of the relations and attributes in them, so that the natural language interface has the knowledge about the semantic structures of the databases. We need to make clear many kinds of relationships among attributes of relations, so that when forming a relational query corresponding to a natural language query, we can connect attributes and relations correctly and systematically. Among those kinds of relationships between attributes, functional dependencies and the synonym relationship of attributes are most important and have direct impact on matching natural language expressions to relational queries.

In this thesis, we study different strategies and methods to extract such knowledge and information from relational databases. Algorithms are designed and presented to extract functional dependencies and synonyms from unnormalized relations. The algorithms use information retrieved from data dictionaries, and learn from the data. Extracting these relationships is useful for discovering semantic connections among attributes and relations so that a natural language interface will have the knowledge about the structure of the underlying databases it requires to interpret its input. Our algorithms discover those functional dependencies that organize attributes within a relation, as well as the synonymity among attributes which correlates different relations. Two algorithms for functional dependency extraction and synonym matching of attributes were implemented and the results of testing and analysis of the performance of these algorithms are presented.

ACKNOWLEDGMENTS

I would like to thank my senior supervisor, Dr. Nick Cercone, for supervising my M.Sc. study and this thesis. He has been patient with my work and given me sincere help whenever I had any problems. I would like to thank my supervisor, Dr. Jiawei Han, who has made valuable suggestions to my work and helped me to clarify the presentation of the thesis. I am indebted to Mr. Gary Hall, who as my thesis examiner, has carefully examined the thesis draft and helped me to solve several confused problems, and provided testing data for my experiments. I would like to thank Dr. David Fracchia. Being the chairman of my supervisory committee, he has made my thesis defence enjoyable and memorable to me.

I would like to acknowledge the financial support from the School of Computing Science and the Center for Systems Sciences of Simon Fraser University, and the Natural Sciences and Engineering Research Council of Canada.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

Much research has been carried out in the area of querying relational databases with natural languages. To build a natural language interface to a relational database, we first require a parser to translate a natural language query into an internal form of representation in terms of the schema of relations in the database and relational operations such as JOIN, SELECT, PROJECT, etc. We then use the internal representation to query the database after suitable translation into the language of the DBMS, and transform the query result into natural language expressions[1, 5, 11, 3]. In translation from natural language expressions to relation based internal forms, one important item is to analyze the schema of the relations, or more precisely, we must determine the structures of the relations, the relationship among attributes inside a relation and among different relations, so that the natural language interfaces have the domain knowledge and know what kinds of internal structure they are going to translate.

Relational databases are organized into relations with attributes defined in each relation. Usually, a relation represents a real world entity, and its attributes represent the characteristics of the entity. There are some relationships among attributes in a relation and between different relations in a database. These relationships represent how components of the real world entities are correlated. The problem we need to solve is to extract the relationships between attributes in relational databases in order to form a semantic structure for their relations.

1.1 The Task

Many relationships exist among attributes within a relation and among attributes belonging to different relations in a relational database. A relational database is designed on the basis of functional dependencies. A *functional dependency* (FD) is a restriction on a relation; a formal definition is given in Appendix A. Research in [13] showed that knowledge of FDs in a database is important for interpreting natural language expressions which refer to the relationships represented by those FDs. So one important relationship between attributes we need to extract is FDs.

The Synonymity of attributes belonging to different relations is the key factor permitting the natural join of the relations. We define that two attributes are synonyms if they can be logically compared in a natural join condition. In a relational query which specifies a natural join, the attributes in the join condition are synonyms; this synonym relationship between attributes is explicitly specified. It is the user's responsibility to know that the attributes are synonyms in order to correctly specify the *join* operation. For a natural language query, the natural language interface must form a relational query automatically according to the intent of natural language expression. It is the system's responsibility to determine which relations should be involved and, if a join has to be formed, on which attribute pair the join should be applied. Thus we must determine the synonyms for attributes. The synonym relation is another type of relationship we need to extract.

One may assume that these relationships should be specified by the database designer. In addition to the design work, the database designer may also provide documentations of his products. This provide us two ways to get the relationship information when we need it: ask the original database designer or refer to the product documentation.

Sometimes the above access methods are not available. Usually database products were often designed for performance, not purity of functional specifications, thus resulted in poor design; a database could deteriorate with modification; many databases are designed with no explicit FD specifications; and, knowledge we require is not always available in system documentations. Thus we would need to do some analysis on relations and data in the database. We would like to have some automatic utilities extract useful information from the data to help discover the required information. An older, well used database usually has large amount of and persistent data. Analysis of the database will provide interesting and useful information, if we do not make use of them for our task, we waste resources available to us. For this reason, we will work on large, ofter older, poorly documented databases.

From the theory of relational databases, we know relations should be designed and normalized to Third Normal Form (3NF) or Boyce/Codd Normal Form (BCNF) or higher [10, 30], in order to remove redundancy and update anomalies. If a relation was normalized to 3NF or higher, its only functional dependencies are those from its keys to non-key attributes. This aspect is recorded in the data dictionary of the database management system and is accessible on-line even if there is no written documentation for it. But for actual relational databases in use, relations usually were not normalized to 3NF. From our experience and observation of designing relational databases, designers sometimes conveniently put attributes into a relation without worrying about normalizations for the sake of better performance, or even do not think about functional dependencies other than to identify the keys. For a mature relational database, it may not have been constructed and decomposed into 3NF or higher. Especially for older applications, the relation may be specified in First Normal Form (1NF) and the FDs may not have been considered in the original design, or the FDs are not clear; implicit FDs are only expressed in the use of the database by convention. When new relations are added to the database, not all relationships between the new and the old relations can be found or organized, so attributes with the same meaning may have different names.

To extract proper relationships among attributes from these databases is more demanding. They require new views of themselves for correct and efficient use, because it was poorly structured. The large amount of relatively static data also provide us a better environment to extract more information. While it is tedious for humans to analyze such data, an automatic information extraction or knowledge discovery mechanism will help. For our task of extracting FDs and synonym relationship from relational databases, we will consider relations in First Normal Form. No documentation or semantical information resources are assumed available except for the data in the relations and the data dictionary, which exists in the RDBMS. No name conventions are assumed, i.e., attributes with the same name are not definitely synonyms, nor are synonyms necessarily named the same.

1.2 The Thesis Structure

This thesis is organized into five chapters. We introduce in Chapter 2 areas related to our topic, such as the natural language interface for relational databases known as SystemX[3], strategies used for knowledge discovery, and data analysis techniques. In chapter 3, we discuss methods for extracting FDs from relational databases, then we provide a synonym matching algorithm in chapter 4. The algorithms implementation description and testing results are given in the Appendixes. We provide concluding remarks, summonizing our accomplishments, and propose further work in chapter 5.

CHAPTER 2

RELATED WORK

In this chapter we present background information related to our work. We describe SystemX in Section 2.1, for which our extraction system is intended. In the next section, we study the area of knowledge discovery in relational databases since our problem belongs to this category.

2.1 SystemX

SystemX [3, 6] is a natural language interface to relational databases under development at Simon Fraser University. At present, SystemX translates English into the *de facto* standard relational database manipulation language SQL. Figure 2.1 (from [3]) gives a graphical representation of the system.

The system consists of a set of modules that create a canonical query representation according to the input natural language expression. A second set of modules then

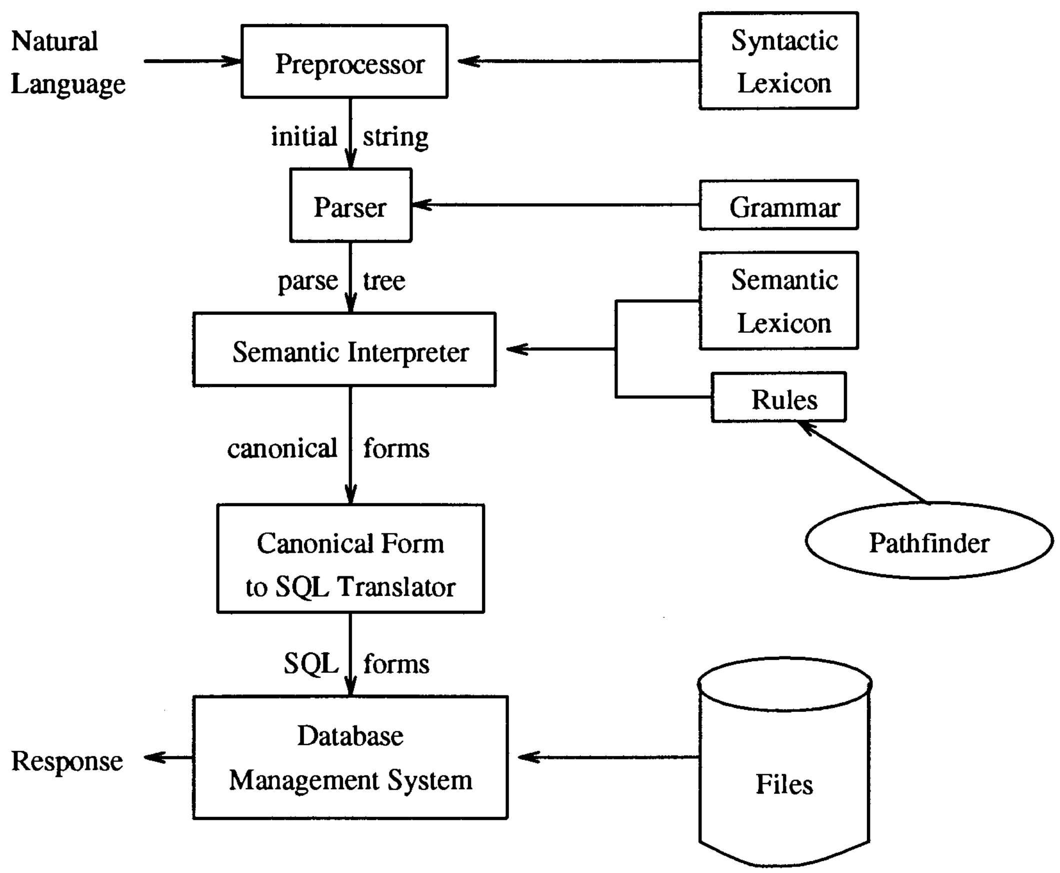

Figure 2.1: A graphical representation of SystemX

translate the canonical form into a logical form and then into SQL. The SQL form is used to query the database. The canonical query representation represents terms in the natural language expression by objects in the underlying database queried and relational operations, as well as quantifiers.

A natural language expression is first transformed into a parse tree organized according to the language syntax structure of the expression. The semantic interpreter transforms the parse tree to the canonical form in terms of database structures and relational operations. Thus it needs domain knowledge pertaining to the underlying database schema. In addition to the domain knowledge built in the semantic lexicon, SystemX requires the Pathfinder[13] subsystem to assist in making appropriate transformations. When the semantic interpreter receives a parse tree as its input, it matches nodes of the parse tree to the database. The leaf nodes are matched to the names of the database relations and attributes. The interpretation of a nonleaf node often requires establishing the relationship between attributes corresponding to the heads of its subtrees. This relationship corresponds to an access path to the database. Thus, an access path to connect its subtrees must be found. Usually many possible access paths exist. Pathfinder is used to select one that contains the most cohesive relationship between the given set of attributes using a semantic model known as a *join graph* whose construction is based almost exclusively on FDs.

Pathfinder [13] generates access paths from the *join graph* whose nodes represent database objects such as relations and attributes, and whose represent relationships among those objects. The relationships have significant matching pattern in natural language expressions because they model real world relationships to which those expressions refer. As stated in [3], *"a database scheme represents relationships between* *entities in the world because database designers represent dependencies that exist between entities by dependencies between the database attributes that denote those entities"*. Expressions in natural language queries refer to these real world dependencies.

[13] defined five types of database dependencies for Pathfinder. The first and the most significant one is the functional dependency, as stated in [13], *"The relationships between entities in the world which correspond to functional dependencies in the database are most likely to be referred to by simple natural language expressions", p.44.* Other types of data dependencies like co-dependency and coincidental dependency are derived from functional dependencies, thus the FD relationship between attributes is the most important relationship for Pathfinder.

Another goal of SystemX is that it be easily portable among different databases. Each natural language interface is built to access a specific database because it needs domain knowledge about the application database. Some automated knowledge discovery utility should be incorporated to SystemX. This thesis discusses methods and algorithms to extract knowledge from relational databases, which facilitate the migration of natural language interfaces to new database applications.

2.2 Knowledge Discovery in Databases

Our work is closely related to knowledge discovery in relational databases. In this section we discuss knowledge discovery issues in databases.

2.2.1 Overview

[12] gives a thorough overview of the area of knowledge discovery in databases. According to this article, knowledge discovery is the nontrivial extraction of implicit, previously unknown, and potentially useful information from data. It has four main characteristics: discovered knowledge is represented with a high-level language; discoveries are accurate or measured with certainty; results are interesting according to user-defined biases; and the discovery process is efficient. For the last characteristic, it noted that most knowledge discovery problems are intractable (NP-hard), so problem domain constraints are specified, or solutions apply only to problems in special cases.

Knowledge discovery in relational databases include techniques to extract information from data. Data in relations is analyzed for classification, pattern identification, summarization, and discrimination, etc. Interesting regularities are found and described as results if they are related to the interests of users. When the amount of data is huge, statistical methods should be used to select appropriate data. Statistics are also used to measure the certainty of discovery.

An RDBMS can provide us with useful utilities in knowledge discovery, such as the data dictionary. The data dictionary is used to store descriptions about relations and attributes in a database. Discovery processes use the data dictionary to retrieve domain knowledge about the problem.

Most knowledge discovery systems extract regularities from data in databases. Some knowledge discovery systems work vertically on data in a relation, such as for discovering data distributions or classifications for attributes. Some systems work horizontally to extract relationships among attributes. In the latter case, vertical data analysis is always required. In some systems, tuples in a relation are clustered and rules for each cluster are discovered. In more sophisticated systems, general rules for attributes, or rules for the rules in each cluster are generalized. FD-extraction requires a system to find generalized rules for attributes because an FD restriction applies to not only a portion or cluster of tuples, but all values of attributes involved.

Some successful knowledge discovery systems include FORTY-NINER[34], INLEN[17], DBLearn[9, 2], as well as systems described in [32, 27, 7, 35].

2.2.2 Existing FD-extraction systems

The FD represents the relationship between attributes in a relation, the data in the relation conforms to the FD constraints. Thus vertical data analysis is required to extract the relationships. Horizontal analysis of the extraction result is also required because the FD is an attribute relationship.

Few systems exist for FD-extraction. [23] described a knowledge discovery system which can be used to detect the FD relationship between attributes. This system constructs a lattice based on tuples and attribute names. A lattice is an acyclic directed graph in which every pair of nodes has a least common superior and a greatest common subordinate and these are necessarily unique. All values appearing in tuples and all tuples are constructed into the lattice. For example, a relation $r(A, B, C, D, E)$ with two tuples and their values as $u(a, b, c, d, e)$ and $v(a', b, c, d', e')$, its lattice is shown in Figure 2.2. In this way a lattice is created for a relation. Tuples with the same values for some of their attributes are connected with intermediate nodes, so relationships reflected in the values can be extracted by further analysis on the lattice.

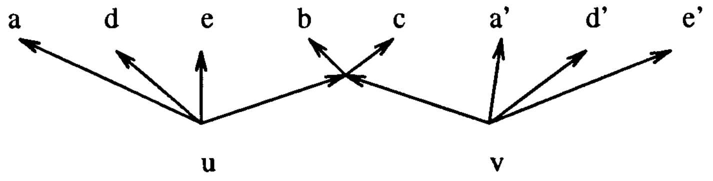

Figure 2.2: Lattice for a sample relation

Although the lattice analysis approach is not originally designed for the goal of FD-extraction, we can see that the FD relationships are represented in a lattice for a relation. In the example in Figure 2.2, any attributes of A, D, or E cannot be functionally determined by B or C because value b and c formed an intermediate node in the lattice. The problems with this method are that it uses all domain values and tuples in a relation to construct the lattice, which is not practical for large relations; and the relationship between tuples or attributes represented in the lattice structure is purely based on values, without any prior guidance in choosing the data, this will make the lattice complicated to analyze for FD relationships.

There is another system for assisting database design processes, for which a subsystem is dedicated to extracting functional dependencies, as described in [18]. The main task of the subsystem is to extract FDs from testing data, the extraction results are used as feedback for database design. It focuses on updating the set of existing FDs when new testing tuples are added. The existing FD set is created by checking the data according to FD definition. No semantic analysis is provided.

In our FD-extraction system to be discussed, we will use the similar method as in [18] to extract FDs implied by data in a relation, and do semantic analysis on the implied FDs.

2.2.3 Data clustering

Some knowledge discovery systems discover knowledge using data analysis to extract interesting patterns or regularities from the data. Some use statistical approaches to sample data for analysis, and the regularities or rules found bear probabilities; some systems classify data into clusters, and check for regularities for each cluster. Various data analysis approaches are described in [4, 7, 8, 9, 14, 16, 23, 28, 32, 35]. Here we describe a data clustering method introduced in [15], which also organizes clusters into a hierarchical structure. Based on this idea we will create attribute hierarchies according to FDs implied in a relation, thus get a well organized FD set.

The data clustering method tries to cluster a set of nodes; each step clusters several nodes together. The intermediate result consists of a level of a hierarchy. The final result will be a hierarchical structure of the set of nodes. For a set of nodes with a metric representing distance values between each pair of nodes, we want to organize the nodes into hierarchical clusters according to the distances. At first, each node is a cluster and the set of original clusters constructs the lowest level in the hierarchy, then clusters close to each other are grouped into new clusters to form another level in the hierarchy. The process is repeated until all nodes are included in a final top level cluster.

Take an example in [15], a set of nodes $(1, 2, 3, 4, 5, 6)$ with its distance matrix as shown in Table 2.1. We use $d(m, n)$ to denote the distance between nodes or clusters m and n. Thus, from Table 2.1, we have $d(1, 2) = .31, d(3, 5) = .04$, etc..

The first cluster is (3 5) because with $d(3, 5) = .04$, they are the pair closest to each other. Then the distance between the new cluster and other nodes should be

	1	2	3	4	5	6
1	0	.31	.23	.31	.23	.23
2	.31	0	.31	.23	.31	.31
3	.23	.31	0	.31	.04	.07
4	.31	.23	.31	0	.31	.31
5	.23	.31	.04	.31	0	.07
6	.23	.31	.07	.31	.07	0

Table 2.1: Distance matrix for a set of nodes

	1	(3 5 6)	2	4
1	0	.23	.31	.31
(3 5 6)	.23	0	.31	.31
2	.31	.31	0	.23
4	.31	.31	.23	0

Table 2.2: Distance matrix after two steps of clustering

adjusted. Fortunately, node 3 and node 5 have the same distance to all other nodes, so the distance value from the new cluster to others are not altered and the semantic meaning of the distance is still preserved.

Then we found that the closest clusters are (3 5) and 6, with $d((3\ 5), 6) = .07$, so we get another cluster (3 5 6). We also see that distances between this new cluster and remaining nodes need not be altered. Now we have Table 2.2.

From Table 2.2, we see that (3 5 6) should be clustered with node 1, and node 2 and 4 should be clustered together at the same time. Finally, we merge the last two clusters (1 3 5 6) and (2 4) together. We have the cluster hierarchy in Figure 2.3.

The key to the process we just described is being able to replace two (or more) objects by a cluster, and still being able to define the distance between such clusters and other objects. The semantics of such a hierarchy depends on applications, and the metric used and the function for distance adjustment after creating a new cluster

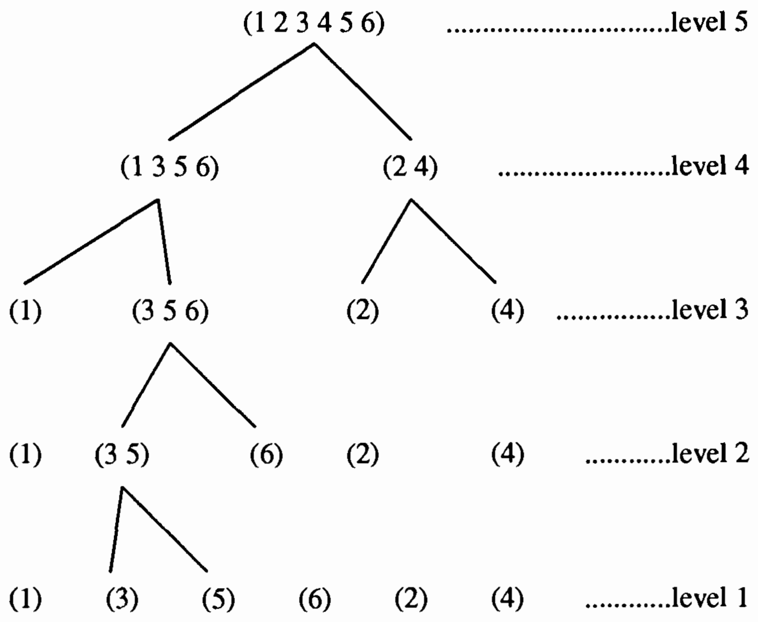

Figure 2.3: A hierarchical clustering scheme

should conform to the semantics. Different functions exist for distance adjustment depending on different applications, such as

$$d((x\ y), z) = max(d(x, z), d(y, z))$$

or

$$d((x\ y), z) = min(d(x, z), d(y, z))$$

for adjusting the distance between object z and a new cluster $(x\ y)$. The semantics for the two functions are different.

When we apply this method in data analysis or synthesis, we should select a meaningful metric and distance calculation function, as mentioned in [8], or the semantics of the cluster hierarchy is not clear.

CHAPTER 3

EXTRACTING FDS FROM 1NF

3.1 Extracting Implied FDs from Data

Definition 3.1 *For a relation R and two attribute sets A and B of R, if the functional dependency $A \rightarrow B$ is satisfied by the current tuples in R, this relationship between A and B is called an* **implied FD** *for A and B and R with current tuples.*

Since most of the time the FD-extraction algorithm we discuss deal with implied FDs, we will still use the symbol $\rightarrow$ to denote implied FDs, i.e., $A \rightarrow B$ should be considered as an implied FD unless stated otherwise.

An implied FD may not be a true FD for the relation. That is, a relation which satisfies the FD condition for two sets of attributes does not mean that the FD is defined for the relation. But every FD defined for a relation must be an implied FD for that relation at any moment. Notice that all characteristics of FDs expressed by axioms in Appendix A also apply to implied FDs.

To extract implied FDs, we just need to check the data by the FD definition. However, we have to check all tuples in the relation before we can make a decision.

For a set of attributes A and two attributes X and Y, $A \rightarrow XY$ implies $A \rightarrow X$ and $A \rightarrow Y$, and vice versa. To simplify the combinations of the determinee for an FD, we consider FDs in the form that the determinee contains only one attribute. In order to simplify the determinants of FDs, we will consider only full FDs as well as full implied FDs, thus $AB \rightarrow C$ will not be extracted if we already have $A \rightarrow C$. Trivial FDs like $XY \rightarrow X$ are always held by any relation, thus we eliminate trivial FDs in our extraction.

3.1.1 Sorting method

The basic method for extracting implied FDs from data is the sorting method. For a given relation with U as its set of attributes, we try to find all determinees for a set of attributes $A \subset U$. First we record $U - A$ as the candidate set of determinees, and sort the tuples in the relation according to the values for A. Then we go through the sorted relation as follows: for an attribute X in the candidate set, if we find two consecutive tuples with the same values for A but not the same values for X, we remove X from the candidate set. After we reach the end of the data, any attribute Y left in the candidate set will be recorded as an implied FD $A \rightarrow Y$.

In a naive algorithm based on the sorting method, we need to consider all possible combinations of attributes in U except for U itself to form the possible determinant A. Because we consider the implied FDs in a form in which the determinee is a single

Input: Relation R with n tuples, and the set of attributes U for R.

Output: The set of implied FDs.

Method:

```
    $C$ = the set of all combinations of attributes from $U$ except $U$;
    for each $A \in C$
        sort the tuples in $R$ according to values for $A$;
        implied($A$) = $U - A$;
        for $i = 2$ to $n$ do
            for any $X \in$ implied($A$) do
                if (tuple[$i-1$]($A$) = tuple[$i$]($A$)) and
                  (tuple[$i-1$]($X$) $\neq$ tuple[$i$]($X$))
                  implied($A$) = implied($A$) - $X$;
            if implied($A$) is empty
                break;
        for each $X$ left in implied($A$)
            record $A \rightarrow X$ as an implied FD;

end.
```

Figure 3.1: A naive algorithm for FD-extraction

attribute, we do not worry about combinations for the determinees.

The naive algorithm based on the sorting method is outlined in Figure 3.1.

In the naive algorithm, tuples are labeled from 1 to n as in an array, *tuple*[i](A) represents the value of ith tuple in the relation for attribute set A. For example, in the following relation:

W	X	Y	Z
5	10	15	20
8	10	15	14
8	9	3	20

Currently, $tuple[1](W, X) = (5, 10)$, $tuple[3](Z) = 20$. We say "currently" because the labeling for tuples may change after we sort on them and swap some tuples. Some of the implied FDs are:

$$(W, X) \to Y, \qquad (Y, Z) \to W$$

3.1.2 Complexity of the naive algorithm

The naive sorting algorithm is only useful for time complexity analysis.

Suppose we have m attributes and n tuples for a relation R. Consider the combinations of attributes, there are C_m^1 combinations with one attribute, C_m^2 combinations with two attributes, ... Thus the total number of combinations are $\sum_{i=1}^{m-1} C_m^i$. For combination A with p attributes, there are $m - p$ attributes originally in A's determinee candidate set, or in *implied*(A) in the naive algorithm of Figure 3.1, each, in the worst case, needs to compare $n - 1$ tuples. So for all combinations with p attributes (C_m^p combinations all together), we need

$$C_m^p \cdot (m - p)(n - 1)$$

comparisons.

Each combination of attributes requires sorting on it, we need to perform sorting

$\sum_{i=1}^{m-1} C_m^i$ times, each sorting needs $O(n \log n)$ in time. The total sorting is

$$\sum_{i=1}^{m-1} C_m^i \cdot O(n \log n)$$

We can calculate the time complexity T for the naive algorithm as:

$$\begin{aligned} T &= \sum_{i=1}^{m-1} C_m^i \cdot O(n \log n) + \sum_{i=1}^{m-1} C_m^i \cdot (m-i)(n-1) \\ &= (2^m - 2) \cdot O(n \log n) + (n-1) \cdot \sum_{i=1}^{m-1} C_m^i \cdot (m-i) \qquad (3.1) \end{aligned}$$

To calculate the last factor $\sum_{i=1}^{m-1} C_m^i \cdot (m-i)$ in (3.1), let

$$\begin{aligned} S &= \sum_{i=1}^{m-1} C_m^i \cdot (m-i) \\ &= C_m^1(m-1) + C_m^2(m-2) + \ldots + C_m^{m-1}(m-(m-1)) \end{aligned}$$

reverse the right side of the equation and add both sides to themselves:

$$\begin{aligned} 2S &= C_m^1(m-1) + C_m^2(m-2) + \ldots + C_m^{m-1}(m-(m-1)) \\ &+ C_m^{m-1}(m-(m-1)) + C_m^{m-2}(m-(m-2)) + \ldots + C_m^1(m-1) \end{aligned}$$

because $C_m^n = C_m^{m-n}$

$$\begin{aligned} 2S &= C_m^1(m-1) + C_m^2(m-2) + \ldots + C_m^{m-1}(m-(m-1)) \\ &+ C_m^1(m-(m-1)) + C_m^2(m-(m-2)) + \ldots + C_m^{m-1}(m-1) \\ &= m(C_m^1 + C_m^2 + \ldots + C_m^{m-1}) \\ &= m(\sum_{i=0}^{m} C_m^i - C_m^0 - C_m^m) \\ &= m(2^m - 2) \end{aligned}$$

or

$$S = \frac{m}{2}(2^m - 2)$$

Substitute this into (3.1):

$$T = (2^m - 2) \cdot O(n \log n) + (n-1) \cdot \frac{m}{2}(2^m - 2)$$

We determine the time complexity as:

$$T = O(n \cdot (\log n + m/2) \cdot 2^m) \tag{3.2}$$

We see that the algorithm will need exponential time with respect to the number of attributes, m, in the relation. This is not a big problem if we work on relations with small number of attributes. In practice, the number of attributes in a relation is usually limited (10 or 15 attributes in a relation is usually considered as too many) so that it will not cause major problems to use the algorithm.

We are usually interested in only full functional dependencies. To repeat, a full FD $A \rightarrow B$ for a relation R is that $A \rightarrow B$ holds on R but $(A-C) \rightarrow B$ does not hold on R for any $C \subset A$. Consider full FDs for a relation, the number of attributes in the determinants of any FD is limited. That means if we can estimate the maximum possible number of attributes that any full FD can have on its left side, we can use that number to limit the number of combinations of attributes for checking. This will reduce the factor 2^m in (3.2) above.

Suppose we estimate k as the maximum number of attributes that any full FD can have as determinants, the naive algorithm will only consider $\sum_{i=1}^{k} C_m^i$ combinations. Get it back to (3.1), the time required by the naive algorithm will be:

$$T = \sum_{i=1}^{k} C_m^i \cdot O(n \log n) + \sum_{i=1}^{k} C_m^i \cdot (m-i)(n-1)$$

or

$$T = O(n \log n) \cdot \sum_{i=1}^{k} C_m^i + (n-1) \cdot \sum_{i=1}^{k} C_m^i (m-i) \tag{3.3}$$

We calculate the last factor of (3.3).

Let

$$\begin{aligned} S &= \sum_{i=1}^{k} C_m^i (m-i) \\ &= \sum_{i=1}^{k} C_m^{i+1}(i+1) \\ &= \sum_{i=2}^{k+1} i \cdot C_m^i \end{aligned}$$

Let

$$G(x) = \sum_{i=2}^{k+1} i \cdot C_m^i \cdot x^i \tag{3.4}$$

divide x from both side of (3.4)

$$\frac{G(x)}{x} = \sum_{i=2}^{k+1} i \cdot C_m^i \cdot x^{i-1} \tag{3.5}$$

(3.5) - (3.4):

$$\begin{aligned} \frac{G(x)}{x} - G(x) &= \sum_{i=2}^{k+1} i \cdot C_m^i \cdot x^{i-1} - \sum_{i=2}^{k+1} i \cdot C_m^i \cdot x^i \\ &= 2C_m^2 x + \sum_{i=3}^{k+1} i \cdot C_m^i \cdot x^{i-1} - \sum_{i=3}^{k+1} (i-1) C_m^{i-1} \cdot x^{i-1} - (k+1) C_m^{k+1} \cdot x^{k+1} \\ &= 2C_m^2 x - (k+1) C_m^{k+1} \cdot x^{k+1} + \sum_{i=3}^{k+1} (i C_m^i - (i-1) C_m^{i-1}) \cdot x^{i-1} \\ &= 2C_m^2 x - (k+1) C_m^{k+1} \cdot x^{k+1} + \sum_{i=3}^{k+1} ((m-(i-1)) C_m^{i-1} - (i-1) C_m^{i-1}) \cdot x^{i-1} \\ &= 2C_m^2 x - (k+1) C_m^{k+1} \cdot x^{k+1} + m \sum_{i=3}^{k+1} C_m^{i-1} \cdot x^{i-1} - 2 \sum_{i=3}^{k+1} (i-1) C_m^{i-1} \cdot x^{i-1} \\ &= 2C_m^2 x - (k+1) C_m^{k+1} \cdot x^{k+1} + m \sum_{i=3}^{k+1} C_m^{i-1} \cdot x^{i-1} - 2 \sum_{i=2}^{k} i C_m^i \cdot x^i \\ &= 2C_m^2 x + (k+1) C_m^{k+1} \cdot x^{k+1} + m \sum_{i=2}^{k} C_m^i \cdot x^i - 2 \sum_{i=2}^{k+1} i C_m^i \cdot x^i \end{aligned}$$

the last factor is $2G(x)$. So we have

$$\frac{G(x)}{x} - G(x) = 2C_m^2 x + (k+1)C_m^{k+1} \cdot x^{k+1} + m\sum_{i=2}^{k} C_m^i \cdot x^i - 2G(x)$$

or

$$G(x) = (\frac{x}{x+1}) \cdot (2C_m^2 x + (k+1)C_m^{k+1} \cdot x^{k+1} + m\sum_{i=2}^{k} C_m^i \cdot x^i)$$

We can see that $G(x)$ is S when $x = 1$. So we have

$$\begin{aligned} S &= G(1) \\ &= \frac{m(m-1)}{2} + \frac{m-k}{2}C_m^k + \frac{m}{2}\sum_{i=2}^{k} C_m^i \end{aligned} \tag{3.6}$$

because

$$\frac{m(m-1)}{2} < \frac{m}{2}C_m^1$$

and from $k \leq m-1$, we have

$$\frac{m-k}{2}C_m^k < \frac{m}{2}C_m^{k+1}$$

Combine the above with (3.6), we have

$$S < \frac{m}{2}C_m^1 + \frac{m}{2}\sum_{i=2}^{k} C_m^i + \frac{m}{2}C_m^{k+1}$$

or

$$S < \frac{m}{2}\sum_{i=1}^{k+1} C_m^i$$

Now we substitute S into (3.3)

$$T < O(n\log n) \cdot \sum_{i=1}^{k} C_m^i + (n-1) \cdot \frac{m}{2}\sum_{i=1}^{k+1} C_m^i$$

or the time complexity is:

$$O(n \cdot (\log n + m/2) \cdot \sum_{i=1}^{k+1} C_m^i) \tag{3.7}$$

From (3.7), we can see that a small change of k would cause greater change of the result, and the change happens drastically when k is around $m/2$. That means if we think k is around $m/2$, we should estimate k more carefully because, at that point, a small decrease of k will save a large amount of time for the algorithm.

In the time complexity of (3.2) or (3.7), n is also a factor which cannot be ignored. Because we work on large databases, the number of tuples in a relation may be huge. If a relation contains 1,000 tuples, even a polynomial time algorithm of $O(n^2)$ will cause a million of steps of computation. In the naive algorithm, because the number of attributes, or m, is limited in practice, it could have better performance compared to an $O(n^2)$ algorithm when n is large. In (3.2), if m is limited and n is large, 2^m would not increase as fast as n, so n^2 would be larger than $2^m \cdot n$ when n reaches some value. This means, n becomes a deterministic factor when the relation has large number of tuples.

If we work on very large relations, the time consideration forces us to work not on all tuples, but part of them. We would select tuples by random method, or with some well-defined criteria. Determining such criteria requires further study. By this, we cannot guarantee that all resulted FDs are implied FDs for the relation, but we can be sure that all correct implied FDs are included in the result.

3.1.3 A modified algorithm for extracting implied FDs

Starting from the naive algorithm, we can make some modifications to reduce the time of computation. We will not create all combination of attributes initially, but compute one combination at a time, then check the data for it, then create another combination, and go on.

First we select attributes which must be checked out as candidate determinants. Those single attributes defined explicitly as keys are removed for consideration. But if multiple attributes together are defined as a key, we should still include them individually because each of them could determine other attributes.

For those selected attributes, we order them into a sequence either using the sequence which are found in the relation schema or a sequence that place one with more variation of values. We check data in rounds. For the first round, we take each single attribute as a candidate determinant, and check the data to see if the candidate functionally determines any other attributes. The next round, we form new candidates by adding a single attribute to the candidates already checked out. Each following round adds one more attribute to form new candidates, until we reach a threshold. The threshold is the maximum number of attributes permitted in the determinants of any full FDs. For example, suppose we have five attributes, A, B, C, D, and E, in a relation, and we have decided that any full FD would have at most three attributes in the determinant. The order of sequence is taken as their natural appearance, i.e., A is the first, B is the second, and so on. For each round, we have possible combinations of attributes as follow:

round 1: A, B, C, D, E
round 2: $AB, AC, AD, AE, BC, BD, BE, CD, CE, DE$
round 3: $ABC, ABC, ABE, ACD, ACE, ADE, BCE, BCE, BDE, CDE$

We stop at round 3 because the next round will exceed the threshold.

At each round, we check the candidate determinants from left to right in the sequence. Each candidate determinant will have a candidate set of dependents. Initially, for the above example, A will have $\{B, C, D, E\}$ as the candidate set, and B will have $\{A, C, D, E\}, \ldots$.

For each round, we don't have to check all possible determinees in a candidate set for a possible determinant, and we don't have to check all the possible determinants. For example, if by checking determinant A, we found $A \rightarrow CD$ is an implied FD, then in checking combination B, we found $B \rightarrow A$, then we don't have to check $B \rightarrow CD$ in the data because $A \rightarrow CD$ and $B \rightarrow A$ imply $B \rightarrow CD$. This lets us remove CD from B's candidate set. In general, we create an ordered sequence on the attributes, and check the determinants in order. For each determinant, we divide its candidate set into two parts, the *former* part contains those attributes positioned before all attributes in the combination with respect to the order, and the rest is the *latter* part. For example, for combination BD, its candidate set could be ACE, then the *former* part would be A because A proceeds B and D in the order, and the *latter* part would be CE. We check the *former* part first, then see if any attribute can be removed from the *latter* part. This is because when we check determinants in order, the determinants in the *former* part have been checked before we start checking the present combination, so after we checked the *former* part, we can use the transitivity characteristic of FDs to infer some determinees in the *latter* part and those determinees need not be checked in data.

For example, in the first round, if the present determinant is B, its candidate set is $ACDE$, its *former* part is A and *latter* part is CDE. When checking against its former part, we may find $B \rightarrow A$. Because the determinants of its *former* part have been checked, we may already have $A \rightarrow C$ and $A \rightarrow D$, so we can use it to reduce the *latter* part to E, and we need only to check if $B \rightarrow E$ holds on the data.

We can do something to reduce candidate determinants too. In the first round, all single attribute candidate determinants are checked, these candidate determinants cannot be removed because each single attribute could functionally determine some other attribute set. In the first round, we try to find all implied FDs for each single attribute determinant. After this step we divide the candidate set for each known determinant into two parts. We call the part that is functionally determined by the determinant the *determined* part and the others the *remaining* part. Now we should make candidate determinants with one more attribute added to an already precessed candidate determinant. The attribute we select to combine with the processed determinant is from the original candidate set of determinees for that determinant because initially, in the first round, all attributes in the relation except the combination attribute is in the candidate set. The following corollary tells us that we can select only from the remaining part of the candidate set without losing any implied FD to be found.

Corollary 3.1 *Given a relation R with attribute set U, with A as a subset of U. For any attribute $B \in A^+$, we have $(AB)^+ \equiv A^+$.*

A^+ is the FD closure of A, i.e., the set of attributes in U that functionally determined by A. Proof of the corollary is simple. Because $A \rightarrow C \Rightarrow AB \rightarrow C$, which means $A^+ \subseteq (AB)^+$, and for any $AB \rightarrow C$, we have:

$$A \rightarrow B \qquad (B \in A^+)$$
$$AA \rightarrow AB \quad (\text{ Augmentation })$$

$$\text{so} \quad A \rightarrow C \quad (\text{ Transitivity })$$

So $(AB)^+ \subseteq A^+$. Thus $(AB)^+ \equiv A^+$.

Back to our algorithm, we're going to make a new candidate determinant by selecting one attribute from the candidate set and combining it with the original determinant. Because in the candidate set, the *determined* part is actually the FD closure of the original combination, selecting attributes from that part will make a determinant whose determinees are already known. So we can remove the *determined* part and choose attributes only from the *remaining* part for new combinations.

For example, for a relation with attribute set $\{A, B, C, D, E, F, G\}$, $\{A\}$ is a combination in the first round, its candidate set is $\{B, C, D, E, F, G\}$. After checking the data, we found implied FDs $A \rightarrow BEF$, then $\{B, E, F\}$ is the *determined* part of $\{B, C, D, E, F, G\}$ and $\{C, D, G\}$ is the *remaining* part. Then we know combinations like $\{A, B\}$, $\{A, E\}$, or $\{A, F\}$ will functionally determine nothing more than attributes in $\{B, E, F\} \cup \{A\}$, the A^+. In order to find new implied FDs, we don't need to include such combinations for further data checking. So we will create combinations $\{A, C\}$, $\{A, D\}$, and $\{A, G\}$ for the next round consideration.

The above phenomenon is not confined only to the first round, because in the axiom, A is any set of attributes, not just a one-attribute set. That is, for any determinant and its candidate set, it is not necessary to check the candidate determinee formed by combining the attribute with the original determinant. The precondition is, of course, we should first find all implied FDs for the original combinations.

In conclusion, to each combination is attached a candidate set consisting of attributes which could be determinees of the combination. The candidate set is divided into *former* part and *latter* part with respect to the order of sequence defined on attributes, this partition is used to check the data for determinees for the combination. After data checking for the combination, the candidate is divided into *determined* and remaining part, which is used to form new combinations for the next round. The first partition may reduce the time used to check data, the other partition may reduce the number of combinations for further consideration. The preconditions are that an order is defined on attributes and combinations are checked according to the order, and at the first round an attribute should have all other attributes in its candidate set so that it need to select attributes only from its candidate set in order to make new combinations.

Now we give the complete algorithm. In the algorithm below, input T is the maximum number of attributes any full FD for this relation can have in its determinant. The algorithm will output other information as count() and weight() values, which will be explained in later sections.

The algorithm invokes subroutine *check*(X, Y), whose function is to check through the tuples in the relation to find out implied FDs from X to any attributes in Y. The found implied FDs then are recorded in a global list. The method used is similar in the naive algorithm.

The algorithm for the calculation of FD closures is from [30].

Input: Relation R with n tuples, the set of attributes U in R, and a threshold T.

Output: Implied FDs and count() and weight() values.

Method:

```
1. remove attributes that are keys by themselves from $U$, assign a sequence number
   to each attribute in $U$ with its natural appearance order in $R$.
2. initialize C, the set of attribute combinations, and their candidate sets.
    2.1 C = empty;
    2.2 for each $A \in U$
        candidate($\{A\}$) = $U - \{A\}$;
        add $\{A\}$ into $C$;
3. *round* = 1 /* start from the first round */
4. checking data within a round.
    4.1 if *round* $> T$, goto step 7;
    4.2 take an $X$ in $C$
        4.2.1 sort data in $R$ based on $X$;
        4.2.2 calculate *count*($X$) and *weight*($X$)
            if $|X| == 1$, calculate count($X$);
            if $|X| == 2$, calculate weight($X$);
        4.2.3 divide candidate($X$) into its *former* and *latter* part;
        4.2.4 checking implied FDs from $X$ to *former*($X$).
            call *check*($X$, *former*($X$))
            candidate($X$) = candidate($X$) - closure($X$);
            if candidate($X$) == {}, goto step 4.2;
        4.2.5 checking implied FDs from $X$ to *latter*($X$).
```

```
            call $check(X, former(X))$
            candidate($X$) = candidate($X$) - closure($X$);
        4.2.6 goto step 4.2

    5. if (++$round$) == $T$, goto 7

    6. prepare for next round determinant set
        6.1 $C_{tmp} = \{\}$;
        6.2 for each $X \in C$ do {
            6.2.1 make combinations with $X$ and each attribute from the remaining
              part of candidate($X$), decide their candidate sets;
            6.2.2 add those combinations with non-empty candidate sets to $C_{tmp}$
            6.2.3 if $C_{tmp} \neq \{\}$, $C = C_{tmp}$;
                    else goto step 4;

    7. calculate weight() for pairs of attributes not counted;

    8. output implied FDs, and count() and weight() information;

end.
```

3.2 Attribute Hierarchy Based on FDs

The result from the algorithm in last section is a set of unorganized implied FDs, all true FDs are included in the set, but there may be some implied FDs that are not real FDs defined for the relation scheme.

We need some analysis to extract "real" FDs from implied FDs. We must be aware that the truly real FDs cannot be extracted solely by checking in the data. Thus the "real" FDs we are going to extract will be called the *intentional FDs.* However, from our effort and analysis, we will see that the intentional FDs are more likely to be real FDs.

In the next section we will analyze some relations in 1NF. Based on the observation we try to sketch a structure to organize FDs in a relation together, and find some measure to create this structure. We will see that the process of creating the structure is helpful in determining intentional FDs, and it provide us a way to get rid of fake FDs from the implied FD set.

3.2.1 Observations from 1NF relations

Relations in 1NF are simply two-dimensional tables with the only requirement being that the values for attributes are atomic, i.e., an attribute cannot take another table or set as values. FDs in a 1NF relation can be defined between any two set of attributes. Table 3.1 is an example of 1NF relation, which is a variation of an example from [20].

The relation in Table 3.1 represents a schedule of airline company in an airport; (FLIGHT DAY) is the key; FLIGHT $\rightarrow$ GATE means that the same FLIGHT always takes passengers at a certain GATE; GATE $\rightarrow$ GATE_LOCATION means that the gate location in the airport is constant.

There are several data redundancies in this relation which makes manipulations of data inconvenient. Suppose we want to add a tuple like (FLIGHT=112, DAY=June 6, PILOT=Bosley, GATE=8, GATE_LOCATION=east). This will make the relation *flight* relation

FLIGHT	DAY	PILOT	GATE	GATE_LOCATION
112	June 6	Bosley	7	east
112	June 7	Brooks	7	east
125	June 10	Mark	7	east
203	June 9	Bosley	12	south
204	June 6	Bruce	15	south

FDs: (FLIGHT DAY) → PILOT GATE GATE_LOCATION
FLIGHT → GATE
GATE → GATE_LOCATION

Table 3.1: A Flight relation in 1NF

passign

FLIGHT	DAY	PILOT
112	June 6	Bosley
112	June 7	Brooks
125	June 10	Mark
203	June 9	Bosley
204	June 6	Bruce

FDs: (FLIGHT DAY) → PILOT

gassign

FLIGHT	GATE	GATE_LOCATION
112	7	east
125	7	east
203	12	south
204	15	south

FDs: FLIGHT → GATE
GATE → GATE_LOCATION

Table 3.2: The flight relations in 2NF

violate the FD FLIGHT → GATE. Assume this addition is valid and we are required to change existing inconsistent tuples. Then we have to search the relation and, for those tuples with FLIGHT=112, change their corresponding GATE value to 8.

To avoid this kind of problem, we can decompose the relation into two relations as in Table 3.2.

In Table 3.2 we put GATE_LOCATION in *gassign* relation because the attribute is related with GATE by an FD GATE → GATE_LOCATION. If we put it in *passign* *flight_gate*

FLIGHT	GATE
112	7
125	7
203	12
204	15

FDs: FLIGHT → GATE

glocation

GATE	GATE_LOCATION
7	east
12	south
15	south

FDs: GATE → GATE_LOCATION

Table 3.3: The flight relations in 3NF

relation, the FD would get lost.

In *gassign*, there is still some data anomaly problems. We can solve these problems by further decomposing *gassign* to relations in Table 3.3. Relations in Table 3.3 and *passign* in Table 3.2 have the property that all attributes are directly functionally dependent on their keys, there is no transitive FDs in each relation, these relations are in 3NF. 3NF relations remove some data redundancies and avoid most potential anomaly problems.

From the above example, we can find some interesting phenomena about relations in 1NF or 2NF. The first is that attributes in such relations can be organized into a hierarchy structure according to FDs, and the decomposition process of relations is the process to create the hierarchy. The attributes in Table 3.1 have the structure of Figure 3.2.

The second observation is that relations in 1NF may have many data redundancies which can cause data manipulation anomalies. Mostly the redundancies exist because of the FD relationship. If $A \rightarrow B$ is an FD in a relation and if in several tuples the value for A is repeated, B will have to repeat its value in these tuples.

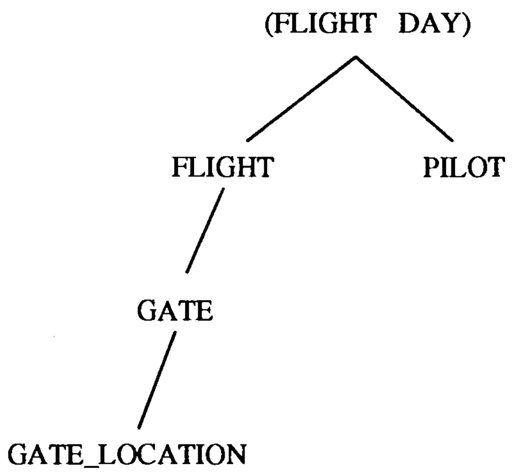

Figure 3.2: Attributes hierarchy for relation *flight*

From these observations we can see that in a 1NF relation, FD restrictions may cause data redundancy. This gives us a hint in finding some intentional FDs from the set of implied fDs. That is, if for a set of attributes there are a lot of redundancies for their data in the relation, then the implied FDs among them are more likely to be intentional FDs.

Another hint is that FDs in a relation can organize attributes into a hierarchy, In the process of creating the hierarchy, we can determine intentional FDs.

3.2.2 A hierarchy structure of attributes and its metric

First we can define a hierarchy structure for attributes in a 1NF relations.

The hierarchy can be represented as a directed graph with attributes or set of attributes as nodes and implied FDs between nodes as arcs. A direct (non-transitive) FD contributes to the hierarchy in the following way: its determinant is located in a particular layer and its determinee is located at the next layer, they are connected by a directed arc from the determinant to the determinee. A hierarchy for a relation will have its top layer comprised of keys and bottom layer comprised, but not confined to, attributes which are not determinants in any FD. The hierarchy is connected and contains all attributes in the relation, because each relation must have a key which connects all attributes in the relation.

Figure 3.2 is an example of attribute hierarchy for the relation in Table 3.1.

The hierarchy can be created bottom up. Suppose we have found all implied FDs for a relation, we can select several attributes to form a cluster. The selection is based on a measure which include attributes involved in direct FDs. The direct FDs in the cluster are used to create two layers of the hierarchy: the determinees are in the lower layer, the determinants are in the higher layer; then the determinants are clustered with other attributes to create other higher layers thereafter. Go on with this process and we will get the keys of the relation as the highest level.

One thing to remember, however, we are not going to get such a hierarchy for a relation. We loosely defined the hierarchy because we do not care what the hierarchy would look like if created, such as whether it can have repeated nodes. We are only interested in the process of creating the hierarchy, because this is the process to confirm implied FDs. Our goal is to extract intentional FDs, and the process of creating hierarchies can help us selecting intentional FDs. Thus during the process, we will record the FDs used for the hierarchy but we will not create the hierarchy.

There are some graph representations of FDs introduced in [20, 33], which are similar to our hierarchy structure of attributes. However, the hierarchy structure is different from those FD graphs in several ways:

- The hierarchy is created according to the implied FDs and a way to cluster attributes, not solely to FDs defined for a relation scheme. Attributes are first clustered, then the implied FDs contained in the cluster are used to construct one level of the hierarchy, then the determinants in this level are clustered with other attributes for other levels.

- The hierarchy structure is a hierarchy of attributes based on FDs, not a structure to represent all FDs, some FDs may not present in the hierarchy because they are not confirmed as intentional FDs. Because our goal is to create connections among attributes based on FDs, we will be satisfied when all attributes are organized together. If we wanted just FDs, we would prompt users with the implied FDs in order to eliminate some false FDs and get the result, in which most of the work is done after we get those implied FDs. In this hierarchy work, our goal is to create a systematic organization of attributes based on intentional FDs

There are several methods to create the FD hierarchy or the FD graph, such as by symbolic analysis on a set of FDs, transitive FDs are extracted and the FD hierarchy is constructed. The method doesn't apply to our task for several reasons. At first, we are working on a set of implied FDs, some of them may not be true, but the symbolic analysis method is intended for real FDs. Second, our goal is to extract intentional FDs, we find that the process to create an FD hierarchy is helpful in extracting the intentional FDs, thus whether the final hierarchy is created or not is not important, the thing really matters is the process in creating the hierarchy. The symbolic analysis method concerns more about creating the hierarchy but less in determining intentional FDs.

We will introduce a hierarchy creation method which provides helpful information in determining intentional FDs. In creating the hierarchy, with some measure, several attributes are selected to be clustered first and only FDs between them are considered. We need to give a method for selecting which attributes should be clustered together. We select those attributes that are related by direct FDs. According to the observation in Section **3.3**.1, when two attributes in a relation in 1NF are involved in a FD relationship, there may be some data redundancy in tuple values for these attributes. Thus the amount of data redundancy evidenced by implied FDs can be used to measure the likelyhood that the FD is intentional. The data redundancy can be measured with the number of values repeated. If two attributes repeat their corresponding values in some tuples, then the more such tuples exist, the more likely it is that these attributes are involved in an FD. Thus we use this frequency of repetition to measure the probability that an implied FD is an intentional FD and use it to cluster attributes.

In the algorithm in Section 3.1.3, the function *weight()* serves for the purpose of calculating repeatness. For two attributes A and B in a relation R, the weight(A, B) is calculated as follows: initially weight(A, B) is set to 0; after the tuples in R are sorted on (A, B), we go through them, if we find two consecutive tuples with values for A and B not changed, we increase weight(A, B) by 1; after we finished checking with all tuples, we get the value for weight(A, B) in this relation.

For example, in the relation in Table 3.1, we have:

weight(FLIGHT,GATE_LOCATION) = 1

weight(FLIGHT,GATE) =1

weight(GATE,GATE_LOCATION) = 2

The weight value differences in the example is small because we have a small number of tuples in the *flight* relation. But because the data represented all FD and non-FDs, the weight values expressed direct FD relationship among attributes and the level of transitive FDs. That is, GATE → GATE_LOCATION is at the lower level while FLIGHT → GATE is at the higher level for the transitive FDs among FLIGHT, GATE, and GATE_LOCATION.

Weight values always involves two objects. These objects may be attributes, but we can also define weight values between clusters. The algorithm in Section 3.1.3 calculates weight values for all pairs of attributes. The determination of weight values between clusters attributes is discussed in the next section.

In the algorithm in Section 3.1.3, *count()* is calculated for each single attribute, this is the number of distinct values for the attribute. This information will be used to help select intentional FDs from implied FDs. This aspect is discussed in Section 3.2.5.

3.2.3 Clustering method

As discussed in the last section, attributes are clustered and implied FDs in the cluster are used to construct a level of the hierarchy, then attributes which are only determinees are removed from further consideration, while determinants are used to cluster with other attributes. Because we create the hierarchy from bottom up, determinees attributes contribute nothing to upper layers, so they can be removed.

The cluster method is based on the discussion in Section 2.3.3, in which we described a method to create a hierarchical structure for a set of nodes with distance values between pairs of nodes. In the working space in our problem, initial nodes are single attributes, the distance measures between nodes are the weight values for pairs of attributes. We cluster attributes with largest weight values, then create a level of the hierarchy if there is any implied FD exists in the cluster that are confirmed by users as an intentional FD. If some attributes are only determinees in the cluster, we remove them, reset the working space—the initial working space with some attributes removed, and cluster from this space again; if no attributes can be removed in a cluster, perhaps because that there is no real FD in it, we adjust the weight value between the cluster and other attributes or clusters, and try to make new clusters according to the adjusted weight value.

We use weight values as the measure of distances between attributes because a weight value is a closeness measurement of attributes with respect to FDs, as described in previous sections. The greater the weight value between two attributes, the more likely that a real direct FD exists between these attributes. If we cluster attributes according to weight values, we get pairs of attributes with direct implied FDs which are more promising to be real.

There is a problem with adjusting weight values. A weight value between two attributes represents the closeness of the attributes with respect to FD relationship. The adjusted weight value should preserve this semantics. For example, consider a relation of 100 tuples with attributes A, B, and C, and weight values for (A, C) and (B, C) of 70 and 50 respectively. Assume A and B are first clustered but no implied FD relationship between them. We need to calculate the weight value between cluster (A, B) and C for further clustering. The weight value should express the semantics that, when (A, B) is considered as a single attribute X, the weight value is the repetition value between attribute X and C as evidenced by the tuples in the relation.

One method to calculate this weight is to do it directly from the data in the algorithm in Section 3.1.3. But this actually requires that all combinations of attributes in a relation be considered which results in an NP-Complete procedure with respect to the number of attributes in the relation. We take another approach to estimate the adjusted weight value.

In the above example, with 100 tuples in the relation and weight(A, C)=70, weight(B, C)=50, the maximum possible repetition value between (A, B) and C is 50 because the repetition values cannot be more than the smaller of weight(A, C) and weight(B, C), i.e.

$$\text{weight}((A, B), C) \leq \min(\text{weight}(A, C), \text{weight}(B, C)).$$

weight(A, C)=70 means that there are 70 tuples with their values for A and C repeated, and weight(B, C)=50 means that there are 50 tuples with their values for B and C repeated, if no tuples with values for A, B, and C repeated at the same time, we would have at least 70+50 = 120 tuples in the relation. Since there are 100 tuples all together, we are sure that there are at least 120-100 = 20 tuples with values for A, B, and C repeated at the same time, thus weight$((A, B), C)$ should be larger or equal to 20.

In general, for attributes A, B, and C in a relation with n tuples, we have

$$\text{weight}((A, B), C) \geq \max(0, \text{weight}(A, C) + \text{weight}(B, C) - n) \tag{3.8}$$

$$\text{weight}((A, B), C) \leq \min(\text{weight}(A, C), \text{weight}(B, C)) \tag{3.9}$$

We should select the adjusted value in this range. In our algorithm, we select the upper bound of the range for the adjusted weight value, because usually n, the number of attributes, is large, which makes the lower bound be zero. Thus the lower bounds for most adjusted values are always the same, while the upper bound can show the difference which makes the adjusted values comparable.

In the same manner, weight values between clusters are also calculated, if we take A, B, and C for clusters as well as attributes.

During the clustering process, we always select the pairs with greatest weight value to combine them. This will preserve transitive FDs in the hierarchy. In transitive FDs like $A \rightarrow B$, $B \rightarrow C$, weight(B, C) is guaranteed to be greater than weight(A, C) and weight(A, B). Because in tuples with values for A and B repeated, the value for B and C will have to repeat. But in tuples with values for B and C repeated, the value for A may not repeat. Thus weight$(B, C) \geq$ weight(A, B) and weight$(B, C) \geq$ weight(A, C). When we first cluster attributes with greater weight values, we cluster B and C together, then we remove C for further consideration, in the next step, A and B are clustered together. This makes the hierarchy in (a) of Figure 3.3. If we cluster A and B together first, we would remove B after we create the A-B link in the hierarchy, and the next step we cluster A and C together, which finally yields the hierarchy in (b) of Figure 3.3, in which the FD $B \rightarrow C$ is lost.

Based on the discussion in above paragraph, we have some corollaries for weight values

Corollary 3.2 *If $A \rightarrow B$, $B \rightarrow C$, then weight$(B, C) \geq$ weight(A, C) and weight$(B, C) \geq$ weight(A, B).*

Corollary 3.3 *If $A \rightarrow B$, $B \rightarrow C$, $C \rightarrow A$, then weight(A, B) = weight(B, C) =*

$$\begin{tikzcd}
A \arrow[d] \\
B \arrow[d] \\
C
\end{tikzcd}$$

(a) the hierarchy when cluster B C first

$$\begin{tikzcd}
 & A \arrow[dl] \arrow[dr] & \\
B & & C
\end{tikzcd}$$

(b) the hierarchy when cluster A B first

Figure 3.3: Two hierarchies with different clustering criteria

weight(C, A).

3.2.4 Non-FD deduction

The process of creating the hierarchy is a process of clustering attributes, breaking clusters, and reclustering attributes. Each time a cluster is formed, the implied FDs among attributes in the cluster are used to create a level of the hierarchy. As long as one implied FD is confirmed as a real FD, this FD will contribute to the hierarchy creation. If some implied FDs are rejected as real FDs by users, these FDs should be recorded and can be used in conducting further hierarchy creation.

As the above process continues, for a cluster with some implied FDs confirmed and some implied FDs rejected as real FDs, a level of hierarchy is created for attributes involved in the real FDs, then some attributes are removed from working space and the cluster is broken, we start clustering again from scratch. Because with some attributes removed, the previously clustered attributes can group with other attributes. Next time in a cluster, the previously rejected implied FDs may appear again in the cluster, when we select implied FDs in the cluster for user confirmation, we thus will not prompt those rejected implied FDs, because we already know that they have been rejected and we have recorded the information.

For a set of implied FDs, if we find that one FD is false, we will see that some other FDs have to be false because of the first false FD. The following corollaries will provide some cases and proofs that how a false FD affects other FDs. Based on the corollaries, we can design a utility to deduce other false FDs from a set of implied FDs when one false FD appears. This utility can be used in the attribute hierarchy creation process to obtain more false FDs when one is rejected by users, which helps to remove false FDs from the implied FD set automatically.

In the following corollaries, we will use $\not\rightarrow$ to denote the non-FD relationship, i.e., $A \not\rightarrow B$ means that A does not functionally determine B.

Corollary 3.4 *If $X \not\rightarrow Y$, X is a set of attributes, then for any subset $Z \subset X$, $Z \not\rightarrow Y$.*

Proof: assume $Z \rightarrow Y$, because Z is a subset of X, $X - Z$ is meanful, thus we have $(Z \cup (X - Z)) \rightarrow Y$ by the augmentation property of functional dependencies, or we have $X \rightarrow Y$, which contradict to the precondition. Thus we must have $Z \not\rightarrow Y$.

Corollary 3.5 *If $X \not\rightarrow Z$, $Y \rightarrow Z$, Y is a single attribute, then $X \not\rightarrow Y$.*

Proof: Assume $X \rightarrow Y$, because $Y \rightarrow Z$, we have $X \rightarrow Z$ by the transitivity property of functional dependencies, which contradicts the precondition. Thus we must have $X \not\rightarrow Y$.

In the second corollary, Y is a single attribute. If Y is a set of attributes, $X \not\rightarrow Y$ will mean that X does not functionally determines any attribute in Y, which may not be true. For example, if Y is $\{A, B\}$ and we have $X \rightarrow A$ and $X \not\rightarrow B$, we see that $X \not\rightarrow Y$ is correct, but we will lose $X \rightarrow A$ if we record this as a non-FD. Thus, we take Y as a single attribute in the corollary.

Based on the above corollaries, we can design utilities to deduce more false FDs when we find one false FD. Generally, we have a set of implied FDs, and we find that one FD $X \rightarrow Y$ in the assumed set is not true, from here we deduce some other FDs in the set of implied FDs that should also be false; and we can find more attributes that previously determined to be dependent on X but now are known not to be dependents of X.

The first corollary is not useful in our system because the implied FDs we extracted are all full implied FDs. That is, if $X \rightarrow Y$ is extracted and stored as a implied FD, we are sure no subset of X can functionally determine Y.

Using the second corollary, we can find more attributes for the right-hand side of an non-FD. For a set of assumed FDs, if we find $X \rightarrow Z$ in the set is actually false, we should record that X does not determines Z, then we check X^+, the FD closure of X. If any single attribute in X^+ determines Z, we will add this attribute to X's non-FD set. This is because we already know $X \not\rightarrow Z$, and for any attribute $Y \in X^+$, if $Y \rightarrow Z$, we should remove Y from X's FD closure according to the second corollary. We can sketch the algorithm for this function as below:

Input: A set of implied FDs, and $X \not\rightarrow Y$.

Output: nfd(X) — attributes not functionally depend on X.

Method:

```
    nfd($X$) = (set of all attributes) - closure($X$) - $Y$;
    For any $A \in (\text{closure}(X) - X)$ do
        if (closure($A$) $\cap$ nfd($X$)) not empty
          add $\{A\}$ into nfd($X$) = nfd(X)
```

end.

3.2.5 Criteria for choosing multiple FDs in a cluster

As we stated, the final decision about FDs is from the user confirmation. When attributes are combined into a cluster, implied FDs in the cluster are prompted to users for confirmation. To make the confirmation process more informative, we give each implied FD a credit so that the higher the credit, the more likely that it is a true FD.

The credit is a relative measure of FDs. Some measurements are based on human experiences and habits in designing and using databases, which are not necessarily true for any relations involved. But the credit calculation does bear some favorable probabilities in determining intentional FDs.

We assign the credit according to the information from the data dictionary and from the data. At first each attribute has an attribute credit according to some properties of the attribute, then for each implied FD, its credit is the sum of credits for attributes from its determinant, combined with some value according to the property of the determinant.

If one attribute is defined as the key of the relation, it will have a greatest attribute credit which is some multiple of the number of tuples in the relation, Otherwise it can take its number of distinct values as its credit, or its *count()* as calculated by the algorithm in Section 3.1.3.

Then the credit for the attribute can be increased if the attribute is defined as NOT NULL.

Another credit adjustment is according to its position defined in the definition of the relation. Although in relational databases theory there is no sequence order for attributes in a relation scheme, in implementations, the RDBMS will record the sequence of attributes when users specify the relation definition, thus in the definition specification, the attribute specified first will have a position 1 in the relation, and the secondly specified attribute will have position 2 in the relation, ... All attributes will have a position in the relation.

In specifying a relation definition, users prefer to specify the key attributes first, or for attributes with FD relations, they usually specify the determinant attributes prior to determinee attributes. Thus when we adjust the credit for an attribute according to its position, we will give more credit to it if the attribute appeared at the front of the position list. Consider an example, we have $A \rightarrow C$ and $C \rightarrow A$ as two implied FDs, if we know that A was defined prior to C in the relation definition, we would assume that A is more likely a determinant than C, thus giving A more credit than C is quite reasonable.

The above information for an attribute, as whether it is defined as a key, or it is restricted to take NOT NULL values, or about its position in the relation scheme, can be extracted from the data dictionary.

From the way we assign credit to attributes, we see that the higher the credit for an attribute, the more likely that it can be a determinant component in some FDs.

For an implied FD, at first its credit is the sum of attribute credits from its determinant attributes, then the implied FD credit is adjusted according to its determinant as a whole. If the determinant has a unique index on it, this implied FD should be added with much more credit. The unique index information can be extracted from the data dictionary.

For implied FDs in a cluster, we prompt them with their credits for user confirmation, so that the users can be more informative about the implied FDs, they will know that the more credit an implied FD has, the more likely that it is a real FD according to information from the relation scheme and data.

The actual credit value is relative, some factors in credit calculation can be adjusted by users. In our implemented system for FD-extraction, we used a credit calculation method to calculate the credit for an attribute as follow:

For an attribute A,

1. if A is defined as key, $credit(A) = maximum$
   else $credit(A) = \#$ of its distinct values.
2. if A is not nullable, $credit(A)$ increases 50%.
3. $credit(A)$ increases 5% if defined as the second last in definition sequence, 10% if defined as the third last, ..., and so on.

The credit for an implied FD is the sum of credits of its determinant attributes.

3.2.6 Combining attributes into groups

Consider the decomposition process of 1NF relations to 3NF. We see that some attributes are decomposed into different relations, that there are only a few attributes are duplicated across relations, and that FDs are localized in the decomposed relations. For example, in the relation in Table 3.1, the FDs FLIGHT $\rightarrow$ GATE and GATE $\rightarrow$ GATE_LOCATION are localized to *gassign* relation in Table 3.2.

From this observation, we see that a particular FD is confined to a group of attributes. We would like to extract FDs in a group of attributes and then, take the group as a single entity to extract its relationship with other attributes or groups.

There are several methods to group attributes. We wish to group attributes which would form an individual relation if the original 1NF relation is decomposed to higher normal forms. This means that the attribute group represent an independent concept in the problem domain.

We think that attributes defined in an index should be grouped together. Users create indexes on a set of attributes because these attributes together define an entity which can be used for indexing other values of tuples, to speed queries or other purposes.

In addition to this, we wish users to provide group information for attributes. The clustering process is also a process to ask for user confirmation of intentional FDs. The users intervention is necessary and makes the extraction mechanism to give better result.

When some attributes are grouped, we try to create the hierarchy structures inside each group, then connect these sub-hierarchies together via the FD relationship.

The whole process is like the following: first confirm and record intentional FDs in each group during the process of sub-hierarchy creation. Then we work with the whole relation. Because intentional FDs in each group are already confirmed, we will not re-extract those FDs when we work with the whole relation, thus some attributes limited to those FDs are excluded in the last attempt of hierarchy creation.

3.3 Implementation

Based on the discussion in above sections, we implemented a FD-extraction system called FUND which works with databases implemented on ORACLE RDBMS.

The system works in two phases. In the first phase, it implements the algorithm in Section 3.1.3; it searches data to extract implied FDs. Results such as implied FDs and count() and weight() information, as well as information from the data dictionary, are stored into an intermediate file using a defined format.

In the second phase, the system reads input from the intermediate file, and use the approaches discussed in above sections to construct attribute hierarchies. At first, the user may specify grouping information in the intermediate file, in order to guide the FD-extraction process. Then hierarchies are built in each group of attributes. In creating a hierarchy, weight values are used to cluster attributes or clusters. If there are some implied FDs in a cluster, they are prompted with credit values for user confirmation. If some implied FDs are confirmed as real FDs, they are recorded, some attributes are removed from the working space, the cluster is broken and a new round of clustering starts. If no FDs are confirmed, the rejected implied FDs are stored and some other false FDs can be inferred, the stored non-FD record is used to narrow the searching space of FDs; after the record and deduction of non-FDs, weight value between the cluster and other attributes or clusters are adjusted, and we form another cluster for further consideration.

After each group of attributes are considered, we may have some sub-hierarchies for each group. Then we apply the cluster method to attributes in the relation. This time, attributes from different sub-hierarchies may be clustered together, in this case, we need to consider not only to cluster two attributes but connect the sub-hierarchies together. The method to connect sub-hierarchies are discussed in last section.

The final result will be confirmed FDs which can be used to create a hierarchy structure of attributes. These FDs are more likely to be real because they come from the hierarchy creation process.

FUND implemented an interactive interface for user confirmation. In this interactive environment, users can browse database information in addition to FD confirmation.

The components of FUND system and their functions, relationship between each other, are outlined in Figure 3.4.

In Appendix A, we give test result of this system.

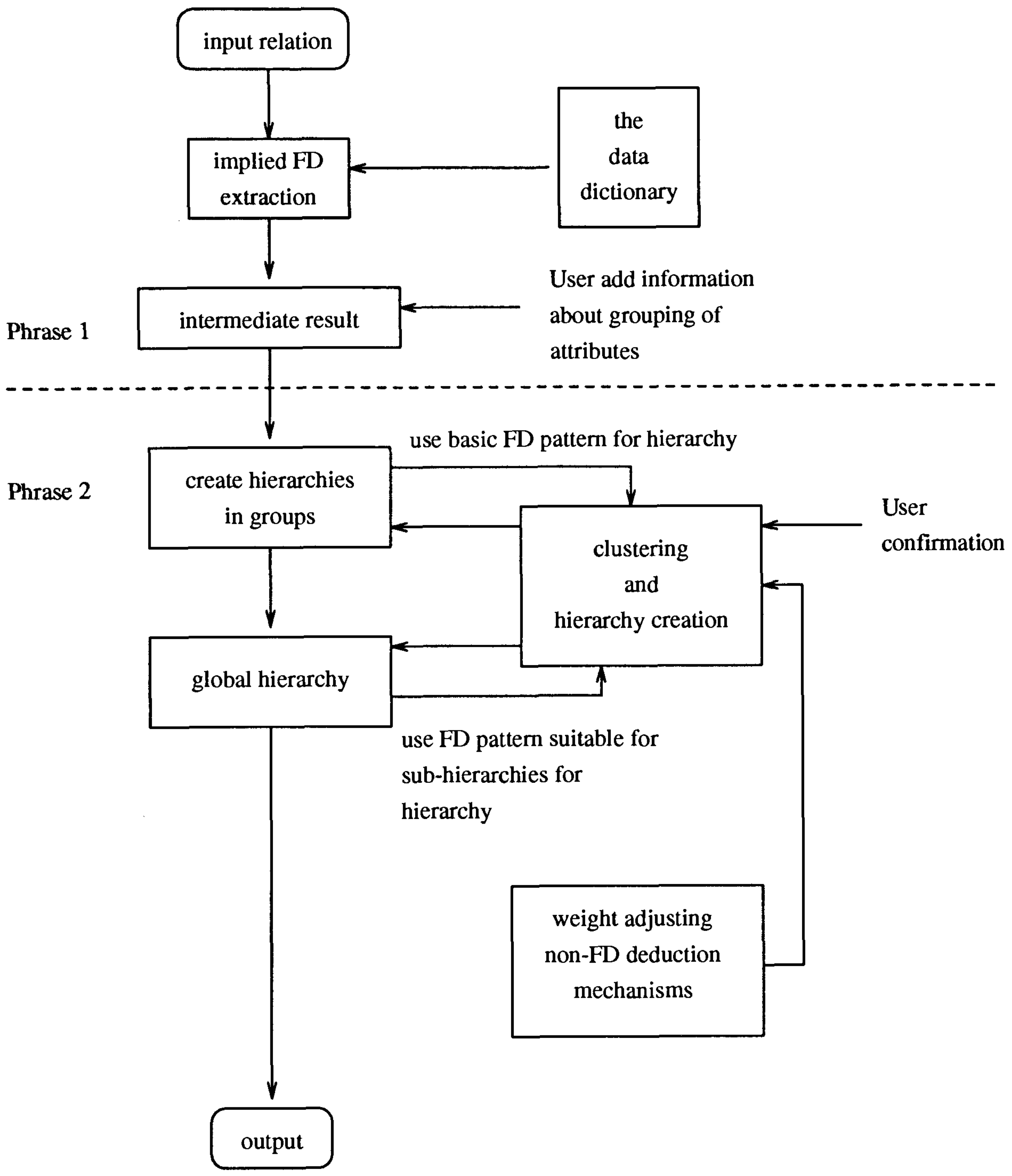
input relation
implied FD extraction
the data dictionary
intermediate result
User add information about grouping of attributes
Phrase 1
Phrase 2
use basic FD pattern for hierarchy
create hierarchies in groups
clustering and hierarchy creation
User confirmation
global hierarchy
use FD pattern suitable for sub-hierarchies for hierarchy
weight adjusting non-FD deduction mechanisms
output

Figure 3.4: Structure of FUND, an FD-extraction system

CHAPTER 4

THE SYNONYM MATCHING ALGORITHM

As stated in Chapter 1, we define the synonymity of attributes as: two attributes from different relations are synonyms if they can be logically compared in a natural join condition. The comparison must be logically correct because only such comparisons will make sense in real queries.

Synonyms are usually used in join conditions for comparison, thus it is necessary to know synonym attributes when forming a query across relations. Relations in a database is related via synonym attributes. In discovering the structure of a database for a natural language interface, it is important to find synonym attributes from different relations.

The synonym matching problem is simpler when compared with the FD-extraction problem. For two relations in a relational database, we take one attribute from a relation and try to determine if there is any attribute from the other relation that could be a synonym for the attribute. We assume that attributes with the same name, or naming that has the same linguistic meaning, does not suggest that they are synonyms.

For example, in a database describing a product's development and distribution, there is a relation for the product's implementation and another relation for sales statistics. In the first relation, there is an attribute named *company* which represents who developed a product; in the second relation, there is an attribute also named *company* (or *corporation*), but it denotes to the organization to which the product is sold. Although the two attributes have the same name (*company*), or the names have the same linguistic meaning (*company* and *corporation*), they are not synonyms.

Synonyms from different relations may have different names, especially in a database to which more relations are added later. Different names may be used for the same object, possibly because relations are designed by different groups of people and archive management for the database development is not organized properly. In the above example, the second relation may need a field to denote a product's developers and *developer* was used for the attribute name. This *developer* is a synonym for *company* in the first relation. Remember that, as in the FD-extraction problem, we are dealing with poorly documented, mature databases; there could be many pairs of attributes which are synonyms but we cannot derive the synonym relationship from their names. Thus we make the no-name-convention assumption for our synonym-matching problem.

In one relation, different attributes may have similar meaning although they represent different aspects of an entity. For example, in a management relation scheme *department(employee_id, name, ..., manager_id)*, attribute *employee_id* and *name* denote an employee's id and name, attribute *manager_id* denotes the manager for the department in which the employee works. A manager's id in one tuple may appear in the *employee_id* column because the manager belongs to another, higher-level department. Although *employee_id* and *manager_id* denote different objects in the relation, they could be assumed as synonyms in some queries. For example, to find an employee who is also a manager, we would issue an SQL query like:

```
SELECT name FROM department
WHERE employee_id IN
( SELECT manager_id FROM department )
```

In this query, *employee_id* and *manager_id* are compared as synonyms. In other situations, queries relating two relations may compare one attribute in a relation with two attributes in the other relation, thus the attribute in the first relation may have two or more synonyms in the other relation.

In most cases, synonym attributes in one relation do not have the same value for the same tuple. In a query which considers two attributes in one relation as synonyms, although the query works on one relation, it is equivalent to say that the query works on two identical relations and *joins* them together. Thus synonym attributes from one relation can be considered synonyms from different relations with respect to the *join* operation.

The purpose of our synonym matching system is to extract synonyms for Pathfinder[13] in SystemX[3] so that Pathfinder could form the *join graph* to connect different relations. From this point and from the discussions in above paragraphs, we would like to have our system to match synonyms from one relation to another, and there could be more than one synonym in a relation for an attribute in the other. While the system requires two relations as input, attributes from one relation are matched with attributes from the other relation, but not to the relation to which it belongs. For the purpose of extracting synonyms from a single relation, this relation should be used for both inputs.

There are many works which find synonyms from the linguistic point of view, especially in natural language understanding[21, 3]. Starting from our assumptions, those methods do not apply to our problem domain. What we can do is to study the information from the data dictionary and from data in the database, providing suggestions of synonym pairs, and as most knowledge discovery systems do, expect the final confirmation from users.

4.1 Exploit Information in Data Dictionary

The data dictionary provides information about the structures of tables defined in a database and the definitions of attributes for relations. For the types of tables described in a data dictionary, there are relations, indexes, clusters, views, and others. The descriptions for relations and indexes are confined to single relations, while a cluster or a view may include several relations. Since we need to find synonyms of attributes from different relations, we may consider cluster and view definitions for our task.

4.1.1 Checking the cluster definition

A cluster[29] is a physical organization of data. Clustering permit several related tables to share the same extents of disk space. In addition to economizing space, clusters improve the performance of *join* queries because tuples that are joined are stored together.

To be clustered, a group of tables must share at least one column with the same type, length, and meaning, i.e. they must share at least on synonym. One effect of clustering is that the rows from all of the tables that have the same value in their cluster columns are stored in the same disk page(s). The cluster columns are stored only once and are shared by each of the shared tables.

To create clustered tables, you must first create a cluster; then you must create the tables and specify that they are to be members of the cluster.

A cluster is created by specifying a set of cluster attributes. Then we can create tables in it by specifying which of their attributes correspond to the cluster attributes. The matching attributes from each table will share the same storage. Thus they must be defined with the same data type and length.

For example, we can create a cluster with attributes M and N. Then we create a table $R_1(A_1, A_2, A_3)$ with A_1, A_2 matched to M, N, and another table $R_2(B_1, B_2, B_3, B_4)$ with B_2, B_4 also matched to M, N. In this way the values for A_1 and B_2 will be stored as one. The values for A_2 and B_4 will also be stored together.

From the definition and organization of clusters, we see that, for any two relations clustered together, their corresponding attributes in the cluster must be synonyms.

We can study the cluster and relation definitions to find those synonym pairs. We do not require user confirmation for them, thus in our system we perform cluster checking before other kinds of checking. Because one attribute may have multiple synonyms in another relation, the synonyms found by cluster checking can be used as feedback for further extraction. For example, suppose attribute A was as a synonym of attribute B from another relation R via cluster checking. Later we find that attribute C in R could also be a synonym for A via a method which requires user confirmation. Then we would prompt the user thusly: "A already matched B. Will A also match C?". This provides more information for the user to make decisions.

4.1.2 Checking the *view* definition

view is a reorganization of the schema of relations. A view is a pseudo relation with some attributes and data from underlying relations, but the data is not copied from the underlying relations. There are many reasons for providing views for relations. For security reasons, an ordinary person may only be allowed to see a portion of a relation and that portion can be defined as a view and the access authority can be restricted to that view. A view also provides the user with a different appearance of the underlying relations, which makes for convenient user access. For example, if a view is defined across relations, the user can use the view with simple queries instead of specifying queries with various join conditions across many relations.

When a view is defined across relations, it takes portions of underlying relations to form pseudo a table as if new a table has been created. In most cases, it connects relations by join operations. A view is defined according to a SELECT query, and only the definition is stored in the data dictionary. When querying a view, the view definition query is executed and the underlying relations are accessed.

We give an example for view definition. Assume there is a relation *teach(instructor, course, ...)* representing that *instructor* teaches *course*, and another relation *take(student, course, ...)* representing that *student* takes *course*, we can define a view for "students taking courses taught by instructors" as

SELECT *student instructor* FROM *take teach*
WHERE *take.course* = *teach.course*

This view takes two relations and uses the join operation to connect them.

When a view is defined on more than one relation, we can check the view definition for synonym matching purposes. We actually analyze the defining SELECT query. The SELECT query may involve join operations, the synonym information may be extracted from the join conditions.

For a SQL SELECT statement, it may have the form

SELECT $attr_1 \dots attr_m$ FROM $rel_1 \dots rel_n$
WHERE *join_condition*

where the *join_condition* is NOT, AND, or OR of comparisons like

$rel_1.attr_1 = rel_2.attr_1$
$rel_2.attr_2 < 100$

Operators in the comparison can be $=, <, \leq, >, \geq$, and $\neq$, corresponding to *equal, less* *than*, etc., operations. Two attributes compared as above in a view definition will be considered as synonyms.

From this analysis, we can see that any SELECT query can be analyzed for synonym matching, not just the *view* definition query. If we have transaction records for queries applied to the databases, we can study every SELECT query to obtain additional information.

4.1.3 Attribute definition

An attribute is defined in a relation with data type, data length, data precision, etc.. In relational databases, attributes are defined to be of some basic data types provided by the RDBMS. Generally, the basic data types are INTEGER, FLOAT, STRING, DATE, etc.. Associated with each type, there may be data length, data precision, etc.. Attributes with the same data type can have different lengths, such as different lengths of strings. In addition to the above data type properties, some RDBMSs have data dictionary tables which provide COMMENTS field to store textual descriptions about the defined attributes and relations.

The ORACLE RDBMS provides only simple data types, an attribute must be defined with one of them. No user defined data types or abstract data types such as the RECORD type in PASCAL are allowed. Thus, there is not very much information to glean the synonymity of attributes according to their type definition. But synonymous attributes must be defined with the same basic data type. The precise description of their type definition may not be exact, for example, they may have different data lengths due to a design mismatch or other design considerations for the database.

When we check a pair of attributes for synonym matching, we should check if they are of the same basic data type. If they also equal on data length and data precision, etc., more credibility is given to our assumption.

If the RDBMS also provides a COMMENTS field for attribute definition, we can prompt the comments for users in the confirmation process. Analyzing the text comments for semantic equivalence appears difficult and is beyond our objectives.

4.2 Data Analysis

In addition to examining the data dictionary, we need to apply data analysis methods to the data in the database to discover synonyms.

If two attributes from different relations in a database are synonyms, they have many values in common. So, for two attributes under consideration, the number of values that are common between them can be used as a metric. We define a *d-value* for a pair of attributes as the ratio of the number of common values to the sum of the distinct values in each attribute.

Definition 4.1 *For two attributes A and B from different relations, let* **c-value** *c(A, B) be:*

$$c(A, B) = (\textit{number of distinct values equal on } A \textit{ and } B)$$

then the **d-value** *d(A, B) is*

$$d(A, B) = 100 \times c(A, B)/(\textit{ number of distinct } A + \textit{ number of distinct } B)$$

For example, consider the following two relations:

R_1	A	B
	5	10
	5	20
	8	10
	7	10

R_2	C	D
	8	10
	2	10
	20	8
	7	20

Using *dis(X)* for the number of distinct values for attribute X, we have

$$\text{dis}(A) = 3, \qquad \text{dis}(B) = 2, \qquad \text{dis}(C) = 4, \qquad \text{dis}(D) = 3$$

We can obtain

$$c(A, C) = 2, \qquad c(A, D) = 1, \qquad c(B, C) = 1, \qquad c(B, D) = 2$$

Thus, the *d-value* for those pairs are:

$$d(A, C) = 200/7, \qquad d(A, D) = 50/3, \qquad d(B, C) = 50/3, \qquad d(B, D) = 40$$

The *d-value* for a pair of attributes expresses the closeness of the pair in value. It is equivalent to measuring the relative size of the intersection of two sets. Figure 4.1 represents two sets A, B, and their intersection C, the *d-value* is equivalent to the ratio of the shaded area and (area A + area B), or

$$\textit{d-value} = \frac{|C|}{|A| + |B|} \times 100$$

Thus for two attributes with larger *d-value*, it expresses that they have more common values, and more likely that they are synonyms.

In the calculation of *d-values*, we take attribute definitions into consideration. Only a pair of attributes defined of the same type and similar length, precision, etc. will have its *d-value* calculated.

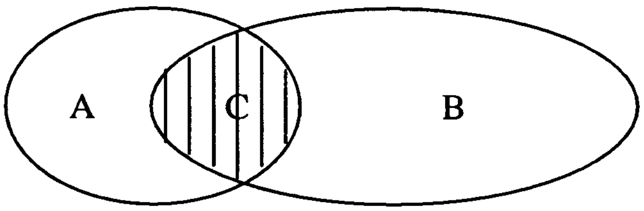

Figure 4.1: Set representation of *d-value*

We can then pick attribute pairs in descending order of their *d-values* and prompt for user confirmation of them in synonym matching. From the above example, we order the pairs according to their *d-values* as $((B, D), (A, C), (A, D), (B, C))$, thus, B and C will be first selected for confirmation, because this pair has the largest *d-value.*

4.3 The Algorithm and Implementation

We have designed an algorithm to extract synonyms based on cluster checking, *view* checking, and *d-value* calculations. A program called SYNONYM based on the algorithm was implemented on the ORACLE RDBMS platform.

The program first performs cluster checking. The approach is straightforward. We need to check cluster definitions in the data dictionary to see if the two relations under consideration are involved in one cluster. If they are, the corresponding attributes matched into the cluster are recorded as synonyms and stored in each others synonym lists.

The program then checks *view* definitions. In ORACLE, *view* definitions are stored as text strings in the data dictionary, which are in the format of SQL SELECT queries. So the *view*-checking portion of the program has implemented a parser to extract join conditions from the definition queries. Attributes from different relations involved in comparisons of *join* conditions are recorded as synonyms.

Then for each pair of attributes from different relations which have similar attribute definition, its *d-value* is calculated.

For complexity analysis, we assume a naive method to calculate the number of common values for two attributes, or the *c-value* defined in Definition 4.1. Assume two relations with m_1 attributes and n_1 tuples in one relation and m_2 attributes and n_2 attributes in the other. For a pair of attributes A and B from each relation respectively, each value of A will be compared with all values of B, thus to get $c(A, B)$ we need $n_1 n_2$ comparisons. There are $m_1 m_2$ different pairs of attributes from the two relation, thus to determine the *c-value* for all pairs, the time required is $O(m_1 m_2 n_1 n_2)$.

In the implementation, we use relational operations provided by ORACLE to calculate the *d-value* for all pairs of attributes that have not been found as synonyms by cluster or *view* checking, and are defined with the same data type, length, etc. To calculate the number of distinct values for an attribute A from relation R, we issue an SQL query:

SELECT COUNT (DISTINCT A) FROM R.

To calculate the *c-value* for attribute A and B from relations R_1 and R_2 respectively, we issue an SQL query:

SELECT COUNT (DISTINCT A) FROM R_1 R_2
WHERE $A = B$

COUNT and DISTINCT are SQL functions The query does a *join* operation on attribute A and B. Because ORACLE has its standard methods to implement these functions and operations, and it can consult other resources such as indexes to enhance their performance, we can calculate the *c-value* efficiently.

The last step is the confirmation, which is implemented as an interactive program with other functions in addition to requesting confirmation. It selects a pair of attributes with the highest *d-value*, prompts with other information about the pair, such as the list of synonyms each attribute already has, and the comment about an attribute stored in the data dictionary (if there is any), then asks the user to confirm whether or not they are synonyms.

Commands provided by the interactive environment are:

help — list the commands provided

table *tablename* — display the scheme of *tablename*

current — display the current pair of attributes for confirmation

yes/no — claim/disclaim synonym relationship of the current pair of attributes

display *table.attr* — display the descriptions for *table.attr*, including its current synonyms list.

add/remove *table1.attr1 table2.attr2* — force addition/deletion of the synonym relationship between *table1.attr1* and *table2.attr2*

quit —- finish up

The interactive environment enables us not only to confirm an assumption of synonyms, but also to change the synonym list for each attribute dynamically. This gives users freedom to judge the candidate pairs according to the attribute information provided by the program and their own biases. A completely automatic extraction process can also be implemented in that the system checks the list of pairs of attributes in descending order of their *d-values*.

In the Appendix we will provide results of the implementation.

CHAPTER 5

CONCLUSIONS AND FUTURE RESEARCH

5.1 Conclusion

In this thesis we studied methods discover functional dependencies and attribute synonyms in relational databases. This information is useful for natural language interfaces to relational databases. We reviewed various systems in the area of knowledge discovery in databases, and applied several data analysis methods to design and implement two systems for discovery of FDs and synonyms in relational databases respectively. These systems make use of the data dictionary and information resulting from analysis of the data both.

The system for the FD discovery is called FUND. It operates on ORACLE databases with first normal form relations. FUND first checks the data exhaustively to extract implied FDs, which are possible FDs suggested by the amount data in the relation. Any true FDs must be included in these implied FDs. Mechanisms based on the theory of FDs are employed to control the search of the data, avoiding a completely trivial search of all combinations of attributes. The implied FDs are analyzed based on a sophisticated data analysis method and information from the data dictionary to create a hierarchy of FDs. FDs for the relation are organized and connected into a hierarchy which represents their transitivity relations. The hierarchy also provide an organization of attributes in a relation in the way that attributes are related via FDs. This structure of the attribute organization is necessary for constructing the join graph in Pathfinder, which provides the representation of relation schema used for a natural language interface.

The other system, SYNONYM, in an ORACLE RDBMS discovers synonyms for attributes from different relations. The synonyms are formed by analyzing the data dictionary and the data in the relations. The data analysis method is different from that for FD discovery.

Both systems provide interactive interfaces to enable users to confirm or reject intermediate results thus providing guidance for further discovery.

We have tested both systems on test databases. The results showed that they performed to our expectations.

5.2 Future Research

Most knowledge discovery approaches deal with intractable (NP-hard) problems. Thus, either problem domains need to be restricted or statistical methods are used to analyze data and produce results bearing probabilities as certainty measures. Our systems will have problems when the amount of data is huge. In the worst case, the discovery processes are time expensive. Fortunately, the time-consuming parts are restricted to earlier stages of the processes and do not require user intervention, thus the earlier discovery can be executed as background tasks.

One method of dealing with this problem is to use statistical methods to sample data for analysis. Research into selecting an appropriate data set is necessary. Statistics and probabilistic theory also provide mechanisms to predict future events based on past observations[25, 26]. The theory can conduct us in sampling proper data set for consideration.

There are some other problems with the FD-extraction system. The first concerns the metric used to calculate the distance or weight values in the stage of clustering raw-FDs. For two attributes, their weight value is measured as the number of points in the table where they change value at the same time. This represents the closeness of the two attributes with respect to functional dependencies, i.e. the weight value measures the possibility that a non-transitive FD may exists between the pair of attributes. But when two attributes are grouped as a cluster, the weight value between the cluster and other attributes somehow, loses the semantic of closeness to some extent. To preserve the semantics of the measure, we should treat attributes in a cluster as one attribute and calculate the weight value between this new attribute and others by going through the data, but this requires all combinations of attributes be considered which is NP-Complete. Another method is to use other metrics for the measurement which represent the characteristics of functional dependencies.

Another problem is that the discovery result depends on the sequence of FD confirmation. Although a user confirmation mechanism can reduce the search space, sometimes a weak FD confirmation will hide a strong FD which is more appropriate in the hierarchy representation. Such case happened to the test in Appendix B, where when attributes are not first grouped, a direct (non-transitive) FD is missed in the result. This is also a major problem in knowledge discovery and artificial intelligence, where search for a global solution is prohibited and only local optimistic result is obtained instead.

For the synonym extraction problem, we can exploit more information from the data dictionary. Some data dictionaries store integrity rules for data, such as to restrict positive values for attributes representing age, or to restrict that only positive numbers less than or equal to 1 can be used for an attributes representing probabilities. This kind of information can be used to specify the value domain of attributes. And attributes that are synonyms must have the same interest domain.

If the above kinds of information are stored in the data dictionary. We would extract this information to see if there are similar constraints or integrity rules for the values for two attributes from different relations, and such attribute pairs will be assumed as synonyms.

Appendix A

Relational databases Theory and the Data Dictionary

We present a brief introduction to the design theory of relational databases and functional dependency. [30, 10] provide a thorough introduction to them.

A.1 Functional Dependency

[30] defines FD as:

Definition A.1 *Let $R(A_1, A_2, \ldots, A_n)$ be a relation scheme, and let X and Y be subsets of $A_1, A_2, \ldots, A_n$. We say $X \to Y$, read "X* **functionally determines** *Y" or "Y* **functionally depends on** *X" if, whatever relation r is the current value for R, it is not possible that r has two tuples that agree in the components for all attributes in the set X yet disagree in one or more components for attributes in the set Y. X*

is called the **determinant** *and Y is the* **determinee** *for the functional dependency.*

Definition A.2 *Let $X \rightarrow Y$ be a functional dependency on scheme R, X and Y are subsets of attributes of R. $X \rightarrow Y$ is a* **full functional dependency** *if, for any attribute $A \in X, (X - A) \rightarrow Y$ is not a functional dependency for R.*

For a relational scheme R, every relation r in R must conform to every FD. But conversely, the fact that an FD apparently holds for an instance of a relation R, does not mean that the FD is defined for R.

We cannot decide FDs by checking in the data of a particular relation, and we can study the characteristics of FDs without worrying about any real relation. Given a set of FDs, we can infer new FDs by a set of rules called *Armstrong's axioms.* First we excerpt some concept definitions about FDs from [30].

Definition A.3 *Let F be a set of functional dependencies for relation scheme R, and let $X \rightarrow Y$ be a functional dependency. We say F* **logically implies** *$X \rightarrow Y$, written $F \models X \rightarrow Y$, if every relation r for R that satisfies the dependencies in F also satisfies $X \rightarrow Y$.*

Definition A.4 *We define F^+, the* **closure** *of F, to be the set of functional dependencies that are logically implied by F; i.e., $F^+ = \{X \rightarrow Y \mid F \models X \rightarrow Y\}$*

In the following axioms, F denotes a set of FDs defined on a relational scheme R; U is the set of all attributes for R; X, Y, and Z are subsets of attributes of U; and XY is a shorthand for $X \cup Y$.

Axiom A.1 (Reflexivity) *If $Y \subseteq X$, then $X \rightarrow Y$ is logically implied by F. This kind of dependencies are called the* **trivial dependencies.**

Axiom A.2 (Augmentation) $X \rightarrow Y \models XZ \rightarrow YZ$

Axiom A.3 (Transitivity) *If $X \rightarrow Y$ and $Y \rightarrow Z$ holds, then $X \rightarrow Z$ holds.*

Armstrong's axioms are sound and complete, which means that all the inferred FDs using Armstrong's axioms are correct, and we can infer all possible FDs implied by F using only Armstrong's axioms.

In the next chapter, we will use these axioms to specify corollaries for our FD-extraction algorithm.

[33] provides a graph representation of FDs for a relational scheme, which provides some hints in organizing FDs for subsequent analysis in later chapters.

A.2 Design of relational databases

Functional dependencies are the design basis for relational databases. A database is a representation of a real world model, its relations represent entities of the model and attributes of relations represents characteristics of the entities. No arbitrary combinations of values for attributes are valid; there are some dependency relationships existing among the data. FD is one of the most important types of dependencies for the relational representation of real world models.

A relational scheme should incorporate its FD restrictions in order to map a real world model correctly. A badly designed scheme, however, may lead to anomalies in the data. If the design allows a particular data value of the determinant of an FD to be repeated in a relation, the value for the determinee of the FD will also repeat, which results in data redundancy. The redundancy may cause update, insertion, and deletion anomalies. The reader is referred to [10, 30] for exposition to the concepts and examples.

The anomaly problems can be solved by refining a poorly designed scheme to a good design via decomposition in the level of normal forms (NFs). Relations in 3NF or higher have less data redundancy and are thus considered good design, while relations in lower normal forms have more data redundancy and will likely cause anomalies.

A.3 The Data Dictionary

The data dictionary is an important resource in knowledge discovery because it stores metadata—the information about information—for relational databases. Concepts of the data dictionary are introduced in [22, 31].

The data dictionary is both a tool and a resource. As a tool, the dictionary permits us to document, organize, and control the design, development, and use of databases. As a resource, the dictionary is an organized repository of information describing the source and the content of data.

In an RDBMS, definitions for relations, views, attributes, indexes, clusters, etc., are all stored in the data dictionary. The data dictionary has descriptions about attributes such as their data types, lengths, and whether NULL values are permitted. By checking the dictionary we could decide which attribute (or a set of attributes) is the key (if defined) for a relation, or assume that an attribute could be a determinant for some FD because it does not permit NULL values. In extracting synonym relationships between attributes, two attribute are more likely to be synonyms if they are defined as the same data type, data length, and take values in the same domain. This type of information can be found from the data dictionary.

In the ORACLE RDBMS, the data dictionary is presented as several sets of tables (relations), users use the same language (SQL) and methods to access the dictionary as in accessing ordinary relations in application databases. Some sets of tables are used by the database administrator so that the dictionary acts as a tool for maintenance and management of the RDBMS; some tables store the descriptions about relations and attributes, etc., of database applications and can be accessed by ordinary users. All dictionary tables are described in [24]. We select some of them which are useful to our task. The following format:

table_name *description of the table (or relation)*

column_name *description of the column (or attribute) in* **table_name**

is used to describe those tables.

ACCESSIBLE_COLUMNS columns of all tables, views, and clusters

TABLE_NAME the name of the table

COLUMN_NAME the name for this column or attribute

DATA_TYPE data type of the attribute

DATA_LENGTH number of bytes for the data of the attribute

NULLABLE the type, length, and nullable definitions are the characteristics of the attribute

COLUMN_ID can be used to decide the significance of attributes

ALL_COL_COMMENTS comments on columns of tables or views; can be used to prompt user for decisions

ALL_INDEXES description of indexes on tables

TABLE_NAME on which table the index was created

UNIQUENESS whether it is a unique index; an unique index is used to check if a set of attribute(s) is the key

ALL_IND_COLUMNS on which columns the index was created

ALL_VIEWS view definition

TEXT the view definition text as an SQL query

USER_CLUSTERS cluster definition; clusters can be used to check for synonyms

The above components are part of the tables which we will use in the implementation of a knowledge discovery system. In the next two chapters, we describe two systems which extract functional dependencies and synonym relationships respectively. They use combinations of techniques and approaches discussed in this chapter and approaches pertinent to the specific problems.

Appendix B

Test Result of FUND

To illustrate the functionality of FUND, we create a 1NF relation by joining several relations and running FUND on the joined relation to see if it can recover the structures of the underlying relations.

Below are several relations in a UNIVERSITY database. The relations are in 3NF and their corresponding keys are underlined.

- *STUDENT(<u>STUD#</u>, SNAME, SSEX)*
- *CLASS(<u>CLID</u>, CNAME, UNITS, FAC#)*
- *FACULTY(<u>FAC#</u>, FNAME, FSEX)*
- *GRADES(<u>STUD#, CLID</u>, GRADE)*

By joining them together, we created a *UNIVERSITY* relation with attributes:

UNIVERSITY(STUD#, SNAME, SSEX, CLID, CNAME, UNITS, GRADE, FAC#, FNAME, FSEX)

There should be following functional dependencies in *UNIVERSITY*:

(STUD#) $\rightarrow$ *(SNAME, SSEX)*
(CLID) $\rightarrow$ *(CNAME, UNITS, FAC#)*
(FAC#) $\rightarrow$ *(FNAME, FSEX)*

key: *(STUD#, CLID)*

When FUND works on *UNIVERSITY* relation, it first extracts raw FDs for the relation, as well as *weight* values for pairs of attributes. When in the analysis phase, it gives FD assumptions with their credits for user to confirm. The process is illustrated below:

1. *(FNAME)* $\rightarrow$ *(FAC#) : 8*
(FAC#) $\rightarrow$ *(FNAME) : 10*

With a larger credit, *(FAC#)* $\rightarrow$ *(FNAME)* is confirmed as an FD.

2. *(CNAME)* $\rightarrow$ *(CLID) : 19*
(CLID) $\rightarrow$ *(CNAME) : 22*

Similarly, *(CLID)* $\rightarrow$ *(CNAME)* is confirmed.

3. *(CLID)* $\rightarrow$ *(FAC#) : 22*

No alternatives and this FD seems correct.

4. *(STUD#) → (SNAME) : 33*

Confirmed.

5. *(CLID) → (UNITS FSEX) : 22*

Confirmed.

6. *(STUD#) → (SSEX) : 33*

Confirmed.

7. *(STUD# CLID) → (GRADE) : 55*
(CLID GRADE) → (STUD#) : 37
(STUD# GRADE) → (CLID) : 48

According to the credit values, FUND favors *(STUD# CLID) → (GRADE)* to be confirmed.

The result yields FDs:

- *(STUD# CLID) → (GRADE)*
- *(STUD#) → (SNAME SSEX)*
- *(CLID) → (CNAME UNITS FAC# FSEX)*
- *(FAC#) → (FNAME)*

In the result, the FD *(FAC#) → (FSEX)* is lost compared to the original FDs. This is because from step 3 above, *(CLID) → (FAC#)* is deduced prior to *(FAC#) → (FSEX)*, which causes *FAC#* removed from further consideration. But when there were more tuples in the relation, it would be more possible that *(FAC#)* → *(FSEX)* be induced first because *(CLID)* → *(FSEX)* is a transitive FD.

Another way to polish the result is using grouping. If we group *FAC#*, *FNAME*, and *FSEX* together in the intermediate result from first phase, we will finally get exactly the FDs from the original relations, as:

- *(STUD# CLID)* → *(GRADE)*
- *(STUD#)* → *(SNAME SSEX)*
- *(CLID)* → *(CNAME UNITS FAC#)*
- *(FAC#)* → *(FNAME FSEX)*

The other test used an NSERC database with large amount of data (almost 10000 tuples). The relation scheme is:

D_GRANT(DEPT, GRANT_CODE, GRANT_TITLE, GRANT_ORDER)

with FDs:

key: *(DEPT, GRANT_CODE)*
GRANT_CODE → *(GRANT_TITLE, GRANT_ORDER)*

The first phase of FUND calculated the weight between *GRANT_TITLE* and *GRANT_ORDER* as the highest (1647), and it happened that there are implied FDs between them. Thus in the second phase, FUND prompt these two attributes first for FD confirmation. When user rejected that there is any FD exist between them, it worked right all the way to give the result the same as the defined FDs. This test shows that user intervention is important in excluding irrelevant results.

Appendix C

Test result of SYNONYM

The two relations shown here are from a real database about NSERC award information.

One relation *AREA* has the schema:

AREA(AREA_CODE, AREA_TITLE, AREA_TITRE)

In *AREA*, *AREA_CODE* is defined as NUMBER.

In the other relation, *AWARD*, it has NUMBER typed attributes as:

AWARD(ORG_CODE, FISCAL_YR, COMP_YR, AMOUNT, CTEE_CODE, AREA_CODE, DISC_CODE, CNT2)

We illustrate the result as SYNONYM looking for synonyms from *AWARD* for attribute *AREA_CODE* in *AREA*. Because SYNONYM matches synonyms only for attributes defined of the same data type, we listed only attributes in *AWARD* that are defined as NUMBER.

In the following table, *AREA_CODE* in *AREA* is matched with each attributes in *AWARD* listed above:

Attribute from AWARD	*d-value*
ORG_CODE	10
FISCAL_YR	0
COMP_YR	0
AMOUNT	4
CTEE_CODE	7
AREA_CODE	190
DISC_CODE	2
CNT2	0

The *d-value* is in the range of 0 to 200.

In this example, SYNONYM takes *AREA_CODE* in *AWARD* as the synonym for *AREA_CODE* in *AREA*, according to the *d-value*s calculated, which is obvious.

Appendix D

Program Listing

Source code for FUND and SYNONYM are listed here. The systems are implemented using PC—a SQL embedded C—and C, on the ORACLE RDBMS.

The first three programs belong to FUND. **search.pc** performs the functions of the first phase in FUND, which is to extract implied FDs given a relation; **fund.c** performs the second phase, which determines intentional FDs during the process of attribute hierarchy creation; **fund.h** is the header file included by both **search.pc** and **fund.c**.

The last listing is SYNONYM. Program **synonym.pc** is in PC language.

search.pc

```
/*====================================================================
 *
 *   @(#) search.pc, last modified June 4, 1992
 *
 *   This program accesses a relation in a relational database, checks the
 *   data dictionary and the data in the relation, extracts information
 *   about attributes, indexes, and gets all FDs which are correctly held
 *   by the data in the relation. The result is output to a file.
 *
 ====================================================================*/

#include "fund.h"

#define USER "xiaobing/chen@nsc"

#define TUPLE 10000     /* Max # tuples a relation can have */

typedef char store_type[TUPLE+1][FIELD_LENGTH+1];


EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR uid[30];

    /* the size for the following VARCHAR arrays should be the same as
       NAME_LENGTH
    */
    VARCHAR table_name[30];
    VARCHAR column_name[30];
    VARCHAR index_name[30];
    char nullable;
    int column_id;
    char unique[10];

    VARCHAR holder[65];         /* the array size should be as FIELD_LENGTH */
    char query[100];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;


attr_type attr[MAX_NUM_ATTR];   /* to hold attribute specification   */
int num_attr;                   /* number of attributes in the table */

index_type index[INDEX_NUM];    /* structure to hold index specification  */
int num_index;                  /* number of indexes on the table         */

int select_list[MAX_NUM_ATTR];
static int num_selected;

store_type *s;
int inds[TUPLE+1];
int n;

int left_size;          /* Max # attributes in determinants of FDs */

struct fd_stru *fds;

struct weight_stru {
        unsigned long pair;
        int value;
} * weight;
int num_weight;

static void load(), fd_checking(), store_fd(), distinct_val(), weight_val(),
            quick_sort();
```

```
static unsigned long check_in_store(), closure();
static int name2num(), cmp();

main(argc, argv)
int argc;
char **argv;
{
    int i, j, m, round;
    long int all;
    unsigned long l, a_left, a_right;
    struct fd_stru *cp, *cq, *cr, *check, *pick;

    if (MAX_NUM_ATTR > 8*sizeof(int)) {
        printf("Too many attributes permitted!\n");
        exit(-1);
    }

/* login to ORACLE, use userid and password provided by USER */
    strcpy(uid.arr, USER);
    uid.len = strlen(uid.arr);
    EXEC SQL CONNECT :uid;
    if (sqlca.sqlcode != 0) {
        printf("Connection problem.\n");
        exit(-1);
    }

    EXEC SQL WHENEVER SQLERROR GOTO errrpt;

/* get the table name and check if it is a TABLE */
    if (argc == 2) strcpy(table_name.arr, argv[1]);
    else {
        printf("give me the table name: ");
        scanf("%s", table_name.arr);
    }
    table_name.len = strlen(table_name.arr);
    EXEC SQL SELECT TABLE_TYPE INTO :column_name FROM ALL_CATALOG
        WHERE TABLE_NAME = :table_name;
    if ( strncmp(column_name.arr, "TABLE", 5) != 0) {
        printf("\nIt's not a table.\n");
        exit(-1);
    }

/* query for columns in the table */
    EXEC SQL DECLARE C1 CURSOR FOR
        SELECT COLUMN_NAME, NULLABLE, COLUMN_ID FROM ALL_TAB_COLUMNS
        WHERE TABLE_NAME = :table_name;
    EXEC SQL OPEN C1;
    for (i=0; i<MAX_NUM_ATTR; i++) {
        EXEC SQL FETCH C1 INTO :column_name, :nullable, :column_id;
        if (sqlca.sqlcode == 1403) break;              /* last row selected */
        column_name.arr[column_name.len] = '\0';
        j = column_id - 1;
        strcpy(attr[j].name, column_name.arr);
        if (nullable == 'Y') attr[j].nullable = 1;
        else attr[i].nullable = 0;
        attr[i].key = 0;
        attr[i].count = 0;
    }
    num_attr = i;
    EXEC SQL CLOSE C1;

/* query for indexes for the table */
    EXEC SQL DECLARE C2 CURSOR FOR
        SELECT INDEX_NAME, UNIQUENESS FROM ALL_INDEXES
        WHERE TABLE_NAME = :table_name;
    EXEC SQL OPEN C2;
    for (i=0; i<INDEX_NUM; i++) (
        EXEC SQL FETCH C2 INTO :index_name, :unique;
        if (sqlca.sqlcode == 1403) break;
        index_name.arr[index_name.len] = '\0';
        strcpy(index[i].name, index_name.arr);
        if ( unique[0] == 'U' ) index[i].unique = 1;
        else index[i].unique = 0;
        index[i].on = 0;
    )
    num_index = i;
    EXEC SQL CLOSE C2;


/* query for columns on which indexes were created */

/* prepare the SQL query, with index_name unfilled */
    EXEC SQL DECLARE C3 CURSOR FOR
        SELECT COLUMN_NAME FROM ALL_IND_COLUMNS
        WHERE TABLE_NAME = :table_name AND INDEX_NAME = :index_name;

/* fill in index_name for each index, and find the columns for the index */
    for (i=0; i<num_index; i++) (
        strcpy(index_name.arr, index[i].name);
        index_name.len = strlen(index_name.arr);
        EXEC SQL OPEN C3;

        for (j=0; j<num_attr; j++) (
            EXEC SQL FETCH C3 INTO :column_name;
            if (sqlca.sqlcode == 1403) break;
            column_name.arr[column_name.len] = '\0';
            m = name2num(column_name.arr);
            index[i].on |= (0x01 << m);
        )
        EXEC SQL CLOSE C3;
        if ( j == 1 && index[i].unique )  attr[m].key = 1;
    )

/* select only non-key attributes for data checking */
    num_selected = 0;
    for (i=0; i<num_attr; i++)
        if ( !attr[i].key ) select_list[num_selected++] = i;
    if (num_selected == 0) (
        printf("\nEvery single attribute in the relation is a key\n");
        exit(1);
    )

    printf("\nTable definition read\n");

    s = (store_type *)calloc(num_selected, sizeof(store_type));
    num_weight = (num_selected * (num_selected-1))/2;
    weight = (struct weight_stru *)calloc(num_weight,
                sizeof(struct weight_stru));
    if (s==NULL || weight==NULL) (
        printf("\nmemory allocation failed.\n");
        exit(-1);
    )

    printf("\nmax # of fields the left side of any fully FD can have: ");
    scanf("%d", &left_size);
    if (left_size > num_selected) left_size = num_selected;

    printf("\nLoading data...");
```

```
    load();

    printf("\nData loading successful\n");

    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;

    for (i=0; i<num_selected; i++) s[i][n][0] = '\0';
    for (i=0; i<=n; i++) inds[i] = i;

    m = 0;
    for (i=0; i<num_selected-1; i++) (
        all = 0x01 << i;                /* all represents attribute i */
        for (j=i+1; j<num_selected; j++) (
            /* the pair for attributes i and j */
            weight(m].pair = ((unsigned long)0x01 << j) | all;
            weight[m].value = -1;
            m++;
        )
    )

/* initialize checking list */
    all = 0x01;
    for (i=1; i<num_selected; i++)  all = (all << 1) | 0x01;
    check = NULL;
    for (i=num_selected-1; i>=0; i--) (
        cp = (struct fd_stru *)malloc(sizeof(struct fd_stru));
        cp->left = (unsigned long)0x01 << i;
        cp->right = all & ~cp->left;
        cp->next = check;
        check = cp;
    )

    fds = NULL;

    for (round=1; round<=left_size; round++) (
        pick = check;
        while ( pick != NULL ) (

            printf("checking: ");
            outputname(stdout, pick->left);
            putchar('\n');

            quick_sort(pick->left, 0, n-1);
            fd_checking(pick);
            if (round == 1)  distinct_val(pick->left);
            if (round == 2)  weight_val(pick->left);    /* if two fields as
                                                            the left, we calculate
                                                            their weight
                                                         */

            pick = pick ->next;
        )

        if (round == left_size) break;

/* build checking list for next round */
        cp = NULL;    /* the new checking list will be temporally linked to cp */
        pick = check;
        while (pick != NULL) (
            if (pick->right != 0) (
                all = (long int)0x01 << (sizeof(long int)*8-1);
                while (!(all & pick->left)) all >>= 1;
                all &=  pick->right;
                l = 0x01;
                for (i=0; i<num_selected; i++) {
                    if (l & all) {
                        a_left = pick->left | l;
                        a_right = closure(a_left);
                        a_right = pick->right & ~(a_right);
                        if (a_right != 0) {
                            cq = (struct fd_stru *)
                                        malloc(sizeof(struct fd_stru));
                            cq->left = a_left;
                            cq->right = a_right;
                            cq->next = NULL;
                            if (cp == NULL)  {
                                cp = cq;
                                cr = cp;
                            }
                            else {
                                cr->next = cq;
                                cr = cq;
                            }
                        }
                    }
                    l <<= 1;
                }
            }
            pick = pick->next;
        }

/* free the memory allocated for check_head */
        while (check != NULL) {
            cq = check->next;
            free(check);
            check = cq;
        }
        check = cp;
    }

/* calculate weight values for any pair left uncalculated */
    for (i=0; i<num_weight; i++)
        if (weight[i].value == -1) {
            quick_sort(weight[i].pair, 0, n-1);
            weight_val(weight[i].pair);
        }


    output();

    exit(0);

errrpt:
    printf("\n %.70s (%d)\n", sqlca.sqlerrm.sqlerrmc, -sqlca.sqlcode);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(-1);
}


/************************************************************************
*
*   Synopsis
*       load()
*
*   Description
*       loads data into global array s.
*
************************************************************************/
static void
load()
{
    int i;

    for (i=0; i<num_selected; i++) {
        sprintf(query, "SELECT %s FROM %s ORDER BY ROWNUM ",
                attr[select_list[i]].name, table_name.arr);

        EXEC SQL PREPARE S FROM :query;
        EXEC SQL DECLARE CL CURSOR FOR S;
        EXEC SQL OPEN CL;

        for (n=0; n<TUPLE; n++) {
            EXEC SQL FETCH CL INTO :holder;
            if (sqlca.sqlcode == 1403) break;
            holder.arr[holder.len] = '\0';
            strcpy(s[i][n], holder.arr);
        }
        EXEC SQL CLOSE CL;
    }
}


/************************************************************************
*
*   Synopsis
*       fd_checking(p)
*           struct fd_stru *p;
*
*   Description
*       it checks the FDs from p->left to any attributes represented in
*       p->right. The check takes on the data in array s. s should be
*       sorted according to p->left before calling this function.
*       The result is stored in the list fds. p->right is updated to
*       contain fields cannot be determined by p->left.
*
************************************************************************/

static void
fd_checking(p)
struct fd_stru * p;
{
    unsigned long m, r, former;
    long int latter;
    int i, j;

    r = p->right;    /* r is the set of field for checking as the right side */
    i = 0;

/* check tuples with NULL values for p->left. Because s is sorted according
   to p->left,
   all NULL values for p->left are at the begining of s.
   s[][n] represents a tuple nulled in all fields.
*/
    while (i<n && cmp(p->left, i, n)==0) {
        m = 0x01;
        for (j=0; j<num_selected; j++) {
            if ( m & r && s[j][inds[i]][0] != '\0' )
                r &= ~m;
            m <<= 1;
        }
        i++;
    }

/* check the former part of r */
    /* latter with its sign bit set */
    latter = (long int)0x01 << (sizeof(long int)*8-1);

    while (!(latter & p->left))  latter >>= 1;
    former = r & (~latter);                  /* former is the former part of r */
    m = check_in_store(p->left, former, i); /* m is the right side
                                               determinants after checking */
    if (m != 0) {
        store_fd(p->left, m);    /* store result */
        m = closure(p->left);    /* calculate the current closure of p->left */
        p->right &= ~m;          /* remove the closure from p->right  */
        r &= ~m;                 /* also remove the closure from r  */
    }
    latter &= r;                 /* latter now is the latter part of r */
    m = check_in_store(p->left, latter, i);
    if (m != 0) {
        store_fd(p->left, m);
        p->right &= ~m;
    }
}


/*************************************************************************
*
•   Synopsis
•       unsigned long check_in_store(l, r, w)
•           unsigned long l, r;
•           int w;
*
*   Description
*       it goes through the data stored in the globle array s, from w to
•       the end, and checks if l determines any attributes represented
•       in r. It returns those checking-confirmed right-sides as an
•       unsigned long int.
•       The store s must be sorted according to l before calling this function.
•
*************************************************************************/

static unsigned long
check_in_store(l, r, w)
unsigned long l, r;
int w;
{
    unsigned long m;
    int i, j;

    for ( i=w+1; r!=0 && i<n; i++)
        if ( cmp(l, i, i-1)==0 ) {       /* two tuples with l fields equal */
            m = 0x01;
            for (j=0; r!=0 && j<num_selected; j++) {
                if (m & r && cmp(m, i, i-1)!=0)  r &= ~m;
                m <<= 1;
            }
        }
    return(r);
}
```

```
/*************************************************************************
*
•   Synopsis
•       store_fd(l, r)
•           unsigned long l, r;
•
•   Description
•       it stores the FD  l --> r  into fds.
•       r must be non-zero.
•
*************************************************************************/

static void
store_fd(l, r)
unsigned long l, r;
{
    struct fd_stru *v;

    v = fds;
    while (v != NULL  &&  v->left != l)  v = v->next;
    if (v != NULL)  v->right |= r;
    else {
        v = (struct fd_stru *)malloc(sizeof(struct fd_stru));
        v->left = l;
        v->right = r;
        v->next = fds;
        fds = v;
    }
}


/*************************************************************************
•
•   Synopsis
*       unsigned long closure(x)
•           unsigned long x;
*
*   Description
•       it takes a set of attributes represented in x, and derive the
•       FD closure for x according to FDs in the globle fds list.
•       It returns the closure represented in an unsigned long int.
•
*************************************************************************/

static unsigned long
closure(x)
unsigned long x;
{
    unsigned long result;
    struct fd_stru *p;

    do {
        result = x;
        p = fds;
        while (p != NULL) {
            if ((x & p->left) == p->left) x |= p->right;
            p = p->next;
        }
    } while (result != x);
    return(result);
}
```

```
/*********************************************************************
*
*   Synopsis
*       distinct_val(node)
*           unsigned long node;
*
*   Description
*       it counts the number of distinct values for attribute node.
*
*********************************************************************/

static void
distinct_val(node)
unsigned long node;
(
    int i, j, m;

    for (i=0; i<num_attr; i++)
        if (node & (0x01 << i))  break;
    m = 1;
    for (j=1; j<n; j++)
        if (cmp(node, j-1, j) != 0)  m++;
    attr[i].count = m;
)


/***********************************************************************
*
*   Synopsis
*       weight_val(pair)
*           unsigned long pair;
*
*   Description
*       it takes a pair of attributes represented in pair, calculates
*       its weight value from the globle store s, and record the value
*       into the globle array weight at the entry for pair.
*       s must be sorted on pair prior to the calling of this function
*
************************************************************************/

static void
weight_val(pair)
unsigned long pair;
(
    int i, j, value;

    for (i=0; i<num_weight; i++)
        if (weight[i].pair == pair) break;
    if (i == num_weight) return;
    value = 0;
    for (j=1; j<n; j++)
        if (cmp(pair, j-1, j) == 0) value++;
    weight[i].value = value;
)


/**************************************************************************
*
*   Synopsis
*       int name2num(c)
*           char *c;
*
*   Description
*       for an attribute name in string c, it returns the cardinal of the
*       attribute in array attr.
*
**************************************************************************/

static int
name2num(c)
char *c;
(
    int i;

    for (i=0; i<num_attr; i++)
        if (strcmp(attr[i].name, c) == 0) break;
    return(i);
)


/**********************************************************************
*
*   Synopsis
*       quick_sort(base, from, to)
*           unsigned long base;
*           int from, to;
*
*   Description
*       it sorts the data in array s from 'from' to 'to', based on 'base'.
*       the data in s is actually not moved by sorting but with an index
*       array inds[] changed.
*
*************************************************************************/

static void
quick_sort(base, from, to)
unsigned long base;
int from, to;
(
    int i, j, tmp;

    if (from >= to ) return;

    i = from +1;
    j = to;
    while (i<=j) (
        while ( i<=j && (cmp(base, i, from) <= 0) ) i++;
        while ( i<=j && (cmp(base, from, j) <= 0) ) j--;
        if (i<j) (
            tmp = inds[i];
            inds[i] = inds[j];
            inds[j] = tmp;
            i++;
            j--;
        ]
    )
    tmp = inds[from];
    inds[from] = inds[j];
    inds[j] = tmp;
    quick_sort(base, from, j-1);
    quick_sort(base, j+1, to);
    return;
)


/**************************************************************************
*
*   Synopsis
*       int cmp(base, c1, c2)
*           unsigned long base;
*           int c1, c2;
*
*   Return values
*       same as strcmp(s1, s2).
*
*   Description
*       it compares two tuples at positions c1 and c2 according to the fields
*       represented by base.
*       c1 and c2 are referenced through index inds[].
*
*****************************************************************************/

static int
cmp(base, c1, c2)
unsigned long base;
int c1, c2;
{
    int i, result;

    for (i=0; i<num_selected; i++) {
        if (0x01 & base) {
            result = strncmp(s[i][inds[c1]], s[i][inds[c2]], FIELD_LENGTH);
            if (result != 0) return(result);
        }
        base >>= 1;
    }
    return(0);
}


output()
{
    int i, j;
    unsigned long m;
    FILE *fp;
    char buf[NAME_LENGTH + 4];
    struct fd_stru *p;

    strcpy(buf, table_name.arr);
    strcat(buf, ".db");
    printf("\n\n...data searching completed...\n");
    printf("results will be in file: %s\n", buf);
    if ((fp = fopen(buf, "w")) == NULL) {
        printf("open %s for writing result failed\n", buf);
        printf("output to stdout\n\n");
        fp = stdout;
    }
    fprintf(fp, "TABLE %s\n\n", table_name.arr);
    fprintf(fp, "MAX_LEFT_FD %d\n\n", left_size);

    for (i=0; i<num_attr; i++)
        fprintf(fp, "FIELD %s %d %d %d\n", attr[i].name, attr[i].nullable,
                        attr[i].key, attr[i].count);
    fputc('\n', fp);
    for (i=0; i<num_index; i++) {
        fprintf(fp, "INDEX %s %d", index[i].name, index[i].unique);
        m = 0x01;
        for (j=0; j<num_attr; j++) {
            if (m & index[i].on)  fprintf(fp, "  %s", attr[j].name);
            m <<= 1;
        }
        fputc('\n', fp);
    }

    for (i=0; i<num_weight; i++)
      if (weight[i].value != 0) {
        fprintf(fp, "WEIGHT");
        outputname(fp, weight[i].pair);
        fprintf(fp, "     %d\n", weight[i].value);
      }
    fputc('\n', fp);

    p = fds;
    while (p != NULL) {
        fprintf(fp, "PFD  (");
        outputname(fp, p->left);
        fprintf(fp, "  ) --> (");
        outputname(fp, p->right);
        fprintf(fp, "  )\n");
        p = p->next;
    }

    fprintf(fp, "\nEND\n\n");
    fclose(fp);
}


outputname(f, node)
FILE *f;
int node;
{
    int i;
    unsigned long m;

    m = 0x01;
    for (i=0; i<num_selected; i++) {
        if (m & node) fprintf(f, "  %s", attr[select_list[i]].name);
        m <<= 1;
    }
}
```

fund.c

```
/*=========================================================================
 *
 *   @(#) fund.c, last modified June 4, 1992
 *
 *   This program analyze the result from program 'search'. It enhances the
 *   FD-extraction result.
 *
==========================================================================*/

#include "fund.h"

#define MAX_LINE 200    /* the max # chars per line in the input file */

struct working_board {
    unsigned long first, second;
    int value;
};

struct working_board  origin[MAX_NUM_ATTR*(MAX_NUM_ATTR-1)/2],
                      board[MAX_NUM_ATTR*(MAX_NUM_ATTR-1)/2];
int origin_num, board_num;

char table_name[NAME_LENGTH+1];

attr_type attr[MAX_NUM_ATTR];
int attr_num;

index_type group[GROUP_NUM];
int group_num;

struct fd_stru *pfd, *cfd, *nfd;        /* presumed and confirmed FDs */

unsigned long all;      /* all attributes */

int left_size;  /* the max # of attributes at the left side of any FD,
                   default to 5 */

unsigned long total_count;      /* the sum of attr[i].count */

static int name2num();
static void readnames(), output(), analysis();

main(argc, argv)
int argc;
char **argv;
{
    char buffer[NAME_LENGTH+4];
    char one_line[MAX_LINE+1];
    int i, j, k, l;
    FILE *fp;
    struct fd_stru *p;

    if (argc == 1) {
        printf("Input the .db file: ");
        scanf("%s", buffer);
    }
    else  strcpy(buffer, argv[1]);

    if (strstr(buffer, ".db")==NULL)
        strcat(buffer, ".db");

    if ((fp = fopen(buffer, "r"))==NULL) {
        printf("Can't open %s.\n", buffer);
        exit(-1);
    }

    attr_num = group_num = origin_num = 0;
    pfd = NULL;
    total_count = 0;
    left_size = 0;

/* read the attribute or field names */
    while (!feof(fp)) {
        fgets(one_line, MAX_LINE, fp);
        if (strncmp(one_line, "FIELD", 5)==0) {
            sscanf(one_line, "FIELD %s%d%d%d", attr[attr_num].name, &j, &k, &l);
            attr[attr_num].nullable = j;
            attr[attr_num].key = k;
            attr[attr_num].count = 1;
            total_count += 1;
            attr_num++;
        }
    }
    if (attr_num == 0) {
        printf("No attribute in the relation\n\n");
        exit(1);
    }

    rewind(fp);
    while (!feof(fp)) {
        fgets(one_line, MAX_LINE, fp);
        if (strncmp(one_line, "TABLE", 5)==0) {
            sscanf(one_line, "TABLE %s", table_name);
            continue;
        }

        if (strncmp(one_line, "MAX_LEFT_FD", 11)==0) {
            sscanf(one_line, "MAX_LEFT_FD %d", &left_size);
            continue;
        }

        if (strncmp(one_line, "INDEX", 5)==0) {
            sscanf(one_line, "INDEX %s%d", group[group_num].name, &k);
            group[group_num].unique = k;

/* the input line should have the format
        INDEX index_name uniqueness(0/1) on_attr ...
*/
            j = 7;      /* the smallest position subscript that the blank
                           before [0/1) can occur
                        */
            while (one_line[j] != ' ')  j++;
            while (!isdigit(one_line[++j])) ;
            j++;
            readnames(&(one_line[j]), &(group[group_num].on));
            if (group[group_num].on != 0)  group_num++;
            continue;
        }

        if (strncmp(one_line, "GROUP", 5)==0) {
            sprintf(group[group_num].name, "$GROUP_%d", group_num);
            group[group_num].unique = 0;
            readnames(&(one_line[6]), &(group[group_num].on));
            if (group[group_num].on != 0)  group_num++;
            continue;
        }

        if (strncmp(one_line, "WEIGHT", 6)==0) {
```

```
        char name1[NAME_LENGTH+1], name2[NAME_LENGTH+1];

        sscanf(one_line, "WEIGHT %s%s%d", name1, name2, &k);
        origin[origin_num].value = k;
        k = name2num(name1);
        if (k == -1)  continue;
        origin[origin_num].first = (unsigned long)0x01 << k;
        k = name2num(name2);
        if (k == -1)  continue;
        origin[origin_num].second = (unsigned long)0x01 << k;
        origin_num++;
        continue;
    }

    if (strncmp(one_line, "PFD", 3)==0) {
        p = (struct fd_stru *)malloc(sizeof(struct fd_stru));
        j = 4;
        while (one_line[j] != '(')  j++;
        readnames(&(one_line[j+1]), &(p->left));
        while (one_line[j] != '>' )  j++;
        while (one_line[j] != '(' )  j++;
        readnames(&(one_line[j+1]), &(p->right));
        p->next = pfd;
        pfd = p;
    }
}

fclose(fp);

while (pfd != NULL && (pfd->left == 0 || pfd->right == 0)) {
    cfd = pfd;
    pfd = pfd->next;
    free(cfd);
}
if (pfd == NULL) {
    printf("no presumed FDs provided, this means no FD exists\n");
    exit(0);
}

cfd = pfd;
while (cfd->next != NULL) {
    nfd = cfd->next;
    if (nfd->left == 0  ||  nfd->right == 0) {
        cfd->next = nfd->next;
        free(nfd);
    }
    else
        cfd = cfd->next;
}

if (left_size == 0)  left_size = 5;

all = 0;
for (i=0; i<attr_num; i++)
    all = (all << 1) | 0x01;

cfd = nfd = NULL;

for (i=0; i<group_num; i++)
    analysis(group[i].on);

analysis(all);
```

```
    printf("\n\n\nAnalysis complete!\n\n");
    printf("filename to save the result ( '-' for stdout): ");
    scanf("%s", buffer);
    if (strcmp(buffer, "-") != 0)
        if ((fp = fopen(buffer, "w")) == NULL)
            printf("\nCan't open %s\n", buffer);
        else
            output(fp);

    output(stdout);
}


/***********************************************************************
*
*   Synopsis
*       int name2num(c)
*           char *c;
*
*   Return values
*       i if attr[i].name == c;
*       -1 otherwise.
*
***********************************************************************/

static int
name2num(c)
char *c;
{
    int i;

    for (i=0; i<attr_num; i++)
        if (strcmp(attr[i].name, c)==0) break;
    if (i == attr_num)  return(-1);
    return(i);
}


/***********************************************************************
*
*   Synopsis
*       readnames(s, x)
*           char *s;
*           unsigned long *x;
*
*   Description
*       searches a string s to find words, which should an attribute name,
*       convert the name to its sequencial number, and store it at as bit
*       positions in *x.
*       it will convert all words in the string until ')' or eoln is
*       encountered
*       if the attribute does not exist, the bit in *x is not set.
*       (so *x may be zero on returning, calling function should be awared
*        of it)
*
***********************************************************************/

static void
readnames(s, x)
char *s;
unsigned long *x;
{
    int i, j, k;
```

fund.c

```
    char name[NAME_LENGTH+1];

    *x = 0;
    i = 0;
    while (1)
        switch (s[i]) {
            case ')':
            case '\0':
            case '\n':
                return;
            case ' ':
                i++;
                break;
            default:
                j = 0;
                do {
                    name[j++] = s[i++];
                } while (s[i]!=' ' && s[i]!='\0' && s[i]!=')' && s[i]!='\n');
                name[j] = '\0';
                k = name2num(name);
                if (k != -1)
                    *x |= (unsigned long)0x01 << k;
        }
}


/**********************************************************************
*
*   Synopsis
*       merge(i, x, y)
*           int i;
*           unsigned long x, y;
*
*   Description
*       at board[i] is a pair (y z), this procedure searches board from i,
*       to find a pair (x, z), make an entry at i as for pair ((x y) z),
*       remove the pair (x, z), and update board_num.
*
**********************************************************************/

static void
merge(i, x, y)
int i;
unsigned long x, y;
{
    int j;
    unsigned long z;

    if (board[i].first == y)  z = board[i].second;
    else z = board[i].first;

    for (j=i+1; j<board_num; j++)
        if (board[j].first == x && board[j].second == z  ||
            board[j].first == z && board[j].second == x)
            break;

    board[i].first = x | y;
    board[i].second = z;
    board[i].value =
        ((board[i].value <= board[j].value) ? board[i].value : board[j].value);
    board_num--;
    if (j != board_num)
            board[j] = board[board_num];
}


/**********************************************************************
*
*   Synopsis
*       adjust_board(a, b)
*           unsigned long a, b;
*
*   Description
*       it adjusts the weight values between the nodes in board and (a b).
*       a and b are merged together.
*
**********************************************************************/

static void
adjust_board(a, b)
unsigned long a, b;
{
    int i;

    i = 0;
    while (i < board_num) {
        if (board[i].first == a && board[i].second == b  ||
            board[i].first == b && board[i].second == a ) {
            board_num--;
            if (board_num != i) {
                board[i] = board[board_num];
                i--;
            }
        }
        else if (board[i].first == a || board[i].second == a)
            merge(i, b, a);
        else if (board[i].first == b || board[i].second == b)
            merge(i, a, b);
        i++;
    }
}


/**********************************************************************
*
*   Synopsis
*       fill_board(x)
*           unsigned long x;
*
*   Description
*       copy weight values from origin[] to board[], for nodes in x.
*       if no weight values for a pair in x, the weight for it is zero.
*
**********************************************************************/

static void
fill_board(x)
unsigned long x;
{
    int i, j;
    unsigned long m, n;

    board_num = 0;
    m = 0x01;
    j = 0;
    for (i=0; i<attr_num; i++) (
        if (m & x) (
            j++;
            if (j == 2) break;
        )
        m = m << 1;
    )
    if (j < 2)  return;

    for (i=0; i<origin_num; i++)
        if (origin[i].first & x  &&  origin[i].second & x) (
            board[board_num] = origin[i];
            board_num++;
        )
    j = board_num;

    m = 0x01 << (attr_num-1);
    while (!(m & x))  m = m >> 1;
    while (m != 1) (
        if (m & x) (
            n = m >> 1;
            while (n != 0) (
                if (n & x) (
                    for (i=0; i<j; i++)
                        if ((board[i].first == m && board[i].second == n) ||
                            (board[i].first == n && board[i].second == m) )
                            break;
                        if (i == j) (
                            board[board_num].first = m;
                            board[board_num].second = n;
                            board[board_num].value = 0;
                            board_num++;
                        )
                )
                n = n >> 1;
            )
        )
        m = m >> 1;
    )
)


/*********************************************************************
*
*   Synopsis
*       unsigned long closure(p, l)
*           struct fd_stru *p;
*           unsigned long l;
*
*   Description
*       it calculates the FD-closure of l determined by p.
*
*********************************************************************/

static unsigned long
closure(p, l)
struct fd_stru *p;
unsigned long l;
(
    unsigned long result;
    struct fd_stru *q;

    do (
        result = l;
        q = p;
        while (q != NULL) (
            if ((l & q->left) == q->left)  l |= q->right;
            q = q->next;
        )
    ) while (result != l);
    return(result);
)


/*********************************************************************
*
*   Synopsis
*       non_fd(x, y)
*           unsigned long x, y;
*
*   Description
*       updates nfd that x |-> y.
*
*********************************************************************/

static void
non_fd(x, y)
unsigned long x, y;
(
    struct fd_stru *p;
    unsigned long m, n;
    int i;

    p = nfd;
    m = closure(pfd, x);
    while (p != NULL && p->left != x)  p = p->next;
    if (p == NULL) (
        p = (struct fd_stru *)malloc(sizeof(struct fd_stru));
        p->left = x;
        p->next = nfd;
        p->right = all & ~m;
        nfd = p;
    )

    p->right |= y;

    if (p->right != (all & ~x)) (
        n = 0x01;
        m = m & ~x;
        for (i=0; i<attr_num; i++) (
            if ((n & m) && !(p->right & n))
                if (p->right & closure(pfd, n))
                    p->right |= n;
            n = n << 1;
        )
    )
)


/*********************************************************************
*
*   Synopsis
```

fund.c

```
•       release(p)
•           struct fd_stru *p;
•
•   Description
•       release a fd_stru list pointed by p.
•
*********************************************************************/

static void
release(p)
struct fd_stru *p;
(
    if (p != NULL) (
        release(p->next);
        free(p);
    )
)


/*********************************************************************
*
•   Synopsis
•       evaluate(q)
•           struct fd_stru *q;
•
•   Description
•       assigns credits to the presumed FD pointed by q;
*       q->right is not zero, and q->credit has been initialized.
•
*********************************************************************/

static void
evaluate(q)
struct fd_stru *q;
(
    int i;
    unsigned long m;

    m = 0x01;
    for (i=0; i<attr_num; i++) (
        if (m & q->left) (
            if (attr[i].key)
                q->credit += total_count;
            else
                q->credit += attr[i].count;
            if (!attr[i].nullable)
                q->credit = q->credit + 0.5 * attr[i].count;
            q->credit = q->credit + (attr_num-i)*0.05*total_count;
        )
        m = m << 1;
    )

    for (i=0; i<group_num; i++)
        if (q->left == group[i].on  &&  group[i].unique)
            q->credit = q->credit * 1.1;
)


/*********************************************************************
•
•   Synopsis
•       outputnames(fp, x)
•           FILE *fp;
•           unsigned long x;
•
•   Description
•       outputs to fp the attribute names stored in x, in format of:
*               attr1 attr2 ...
•
*********************************************************************/

static void
outputnames(fp, x)
FILE *fp;
unsigned long x;
(
    int i;
    unsigned long m;

    m = 0x01;
    for (i=0; i<attr_num; i++) (
        if (m & x)  fprintf(fp, "%s ", attr[i].name);
        m <<= 1;
    )
)


/*********************************************************************
*
•   Synopsis
•       output_an_fd(fp, left, right)
*           FILE *fp;
•           unsigned long left, right;
•
•   Description
*       outputs to fp an FD left --> right, in format of:
•        ( attl1 attl2 ... ) --> ( attr1 attr2 ... )
•
*********************************************************************/

static void
output_an_fd(fp, left, right)
FILE *fp;
unsigned long left, right;
(
    fprintf(fp, "( ");
    outputnames(fp, left);
    fprintf(fp, ") --> ( ");
    outputnames(fp, right);
    fputc(')', fp);
)


/*********************************************************************
*
•   Synopsis
•       output(fp)
•           FILE *fp;
•
•   Description
•       outputs the database information to fp.
*
*********************************************************************/

static void
```

```
output(fp)
FILE *fp;
(
    int i;
    struct fd_stru *p;

    fprintf(fp, "TABLE  %s\n\n", table_name);
    fprintf(fp, "MAX_LEFT_FD %d\n\n", left_size);

    for (i=0; i<attr_num; i++)
        fprintf(fp, "FIELD  %s %d %d %d\n", attr[i].name, attr[i].nullable,
                attr[i].key, attr[i].count);
    fputc('\n', fp);

    for (i=0; i<group_num; i++) (
        fprintf(fp, "GROUP  %s %d  ", group[i].name, group[i].unique);
        outputnames(fp, group[i].on);
        fputc('\n', fp);
    )

    fputc('\n', fp);
    for (i=0; i<origin_num; i++) (
        fprintf(fp, "WEIGHT  ");
        outputnames(fp, origin[i].first | origin[i].second);
        fprintf(fp, "   %d\n", origin[i].value);
    )

    fputc('\n', fp);
    p = pfd;
    while (p != NULL) (
        fprintf(fp, "PFD  ");
        output_an_fd(fp, p->left, p->right);
        fputc('\n', fp);
        p = p->next;
    )

    fputc('\n', fp);
    p = nfd;
    while (p != NULL) (
        fprintf(fp, "NFD  ");
        output_an_fd(fp, p->left, p->right);
        fputc('\n', fp);
        p = p->next;
    )

    fputc('\n', fp);
    p = cfd;
    while (p != NULL) (
        fprintf(fp, "CFD  ");
        output_an_fd(fp, p->left, p->right);
        fputc('\n', fp);
        p = p->next;
    )

    fprintf(fp, "\n\n\nEND\n");
)


/*********************************************************************
*
*   Synopsis
*       browse(h, n)
*           struct fd_stru h[];
*           int n;
*
*   Description
*       output to stdout the FDs in array h[] with their credits.
*       n is the size of the array.
*
*********************************************************************/

static void
browse(h, n)
struct fd_stru h[];
int n;
(
    int i;

    for (i=0; i<n; i++)
        if (h[i].left != 0) (
            printf("sequential number: %d\n", i+1);
            output_an_fd(stdout, h[i].left, h[i].right);
            printf("\ncredit for its left side: %d\n\n", h[i].credit);
        )
)


/*********************************************************************
*
*   Synopsis
*       static int confirmation(p)
*           struct fd_stru *p;
*
*   Return values
*       1       there're confirmed FDs;
*       0       otherwise
*
*   Description
*       confirmation process about a list of presumed FDs pointed by p.
*       the list may contain FDs with empty right side.
*       After the confirmation, for each entry in p, its right field
*       contains the confirmed FDs and its credit field contains the
*       confirmed non-FDs. (So all entries should be updated)
*
*       it provides an interactive environment, with the following functions:
*
*            0  quit confirmation process
*           -1  help, print a list of commands
*           -2  browse the presumed list
*           -3  print the database information
*           -4  print a selected attribute information
*           -5  print confirmed FDs
*           -6  print confirmed non-FDs
*
*********************************************************************/

static int
confirmation(p)
struct fd_stru *p;
(
    struct fd_stru *r, *head;
    int i, n, sel;
    unsigned long m;
    char s[NAME_LENGTH+1];
```

fund.c

```
    i = 0;
    r = p;
    while (r != NULL ) {
        if (r->right != 0)  i++;
        r = r->next;
    }
    head = (struct fd_stru *)calloc(i, sizeof(struct fd_stru));
    r = p;
    n = 0;
    while (r != NULL) {
        if (r->right != 0) {
            head[n].left = r->left;
            head[n].right = r->right;
            head[n].credit = r->credit;
            head[n].next = r;
            n++;
        }
        r = r->next;
    }

    printf("\n\nConfirmation session\n\n");

    browse(head, n);

    printf("\nselect one FD for confirmation\n");
    printf("You can:\n");
    printf("\tinput the sequential number,\n");
    printf("\tor negative numbers for other functions (-1 for help).\n");

    while (1) {
        for (i=0; i<n; i++)
            if (head[i].left != 0)  break;
        if (i == n)
            sel = 0;
        else {
            printf("\ninput an integer: ");
            scanf("%d", &sel);
        }
        switch (sel) {
          case 0:
            printf("\nThe FDs not confirmed will be considered as non-FDs\n");
            printf("Are you sure to quit ( 0 -- no; 1 -- quit) : ");
            scanf("%d", &sel);
            if (sel == 1) {
                m = 0;
                for (i=0; i<n; i++) {
                    r = head[i].next;
                    if (head[i].left == 0) {
                        r->credit = r->right & ~head[i].right;
                        r->right = head[i].right;
                    }
                    else {
                        r->credit = r->right;
                        r->right = 0;
                    }
                    if (r->right != 0)  m = 1;
                }
                printf("\n\nEnd of current confirmation session\n");
                if (m) return(1);
                return(0);
            }
            break;

          case -1:
            printf("\nYou can input the sequential number for an FD\n");
            printf("or the following commands:");
            printf("\t 0         quit the confirmation process\n");
            printf("\t-1         help, print this list of commands\n");
            printf("\t-2         browse the list of FDs for consideration\n");
            printf("\t-3         print the database information\n");
            break;

          case -2:
            browse(head, n);
            break;

          case -3:
            output(stdout);
            break;

          default:
            if (sel > 0   &&  sel <= n) {
                sel--;
                if (head[sel].left == 0) {
                    printf("this FD has just been considered\n");
                    break;
                }
                output_an_fd(stdout, head[sel].left, head[sel].right);
                printf("\nthe above FD for confirmation\n");
                printf("to decide which attribute on the right side is a \
dependent, you may input: \n");
                printf("\tthe name of the attribute,\n");
                printf("\tor @ to select all attributes\n");
                printf("input $ to finish\n");
                m = 0;
                while (m != head[sel].right) {
                    printf("\n> ");
                    scanf("%s", s);

                    if (strcmp(s, "@") == 0) {
                        m = head[sel].right;
                        break;
                    }

                    if (strcmp(s, "$") == 0)  break;

                    i = name2num(s);
                    if (i == -1 ||
                        (((unsigned long)0x01 << i) & head[sel].right) == 0)
                        printf("no such attribute on the right\n");
                    else
                        m = m | ((unsigned long)0x01 << i);
                }
                head[sel].right = m;
                head[sel].left = 0;
            }
            else
                printf("input out of range\n");
        }
    }
}


/**********************************************************************
```

fund.c

```
*
*   Synopsis
*       analysis(x)
*           unsigned long x;
*
*   Description
*       it analyzes the FDs in a group of x.
*
*************************************************************************/

static void
analysis(x)
unsigned long x;
{
    int i, j, m, round;
    unsigned long candidate, y, z;
    struct fd_stru *p, *q, *r, *r1;
    short int done, need;

    fill_board(x);
    while (board_num > 0) {
        m = board[0].value;
        j = 0;
        for (i=1; i<board_num; i++)
            if (m < board[i].value) {
                m = board[i].value;
                j = i;
            }
        candidate = board[j].first | board[j].second;
        round = 1;
        p = NULL;
        done = 0;
        while (round <= left_size) {
            if (p == NULL) {
                y = 0x01;
                for (i=0; i<attr_num; i++) {
                    if (y & candidate) {
                        q = (struct fd_stru *)malloc(sizeof(struct fd_stru));
                        q->left = y;
                        q->next = p;
                        p = q;
                    }
                    y = y << 1;
                }
            }
            else {
                r = p;
                r1 = NULL;
                while (r != NULL) {
                    y = 0x01;
                    while ((y & r->left)==0)  y = (y << 1) | 0x01;
                    y = (y >> 1) & candidate;
                    if (y != 0) {
                        z = 0x01;
                        for (i=0; i<attr_num; i++) {
                            if (z & y) {
                                q = (struct fd_stru *)
                                        malloc(sizeof(struct fd_stru));
                                q->left = r->left | z;
                                q->next = r1;
                                r1 = q;
                            }
                            z = z << 1;
                        }
                    }
                    r = r->next;
                }
                release(p);
                p = r1;
            }
            if (p->next == NULL)  break;

            q = p;
            need = 0;
            while (q != NULL) {
                q->right = closure(pfd, q->left) & candidate
                                & ~(closure(cfd, q->left));
                if (q->right) {
                    r = nfd;
                    while (r != NULL  &&  r->left != q->left)  r = r->next;
                    if (r != NULL)
                        q->right = q->right & ~r->right;
                }
                q->credit = 0;
                if (q->right) {
                    evaluate(q);
                    need = 1;
                }
                q = q->next;
            }

            if (need) {
                done = confirmation(p);
                q = p;
                while (q != NULL) {
                    if (q->credit != 0)
                        non_fd(q->left, q->credit);
                    q = q->next;
                }
            }
            if (done)  break;

            round++;
        }

        if (done) {
            q = p;
            y = 0;
            while (q != NULL) {
                if (q->right) {
                    r = cfd;
                    while (r != NULL && r->left != q->left)  r = r->next;
                    if (r != NULL)
                        r->right |= q->right;
                    else {
                        r = (struct fd_stru *)malloc(sizeof(struct fd_stru));
                        r->left = q->left;
                        r->right = q->right;
                        r->next = cfd;
                        cfd = r;
                    }

                    y |= q->right;
                }
                q = q->next;
            }

            q = p;
            while (q != NULL) {
                if (q->right)  y = y & ~q->left;
                q = q->next;
            }
            x = x & ~y;
            fill_board(x);
        }
        else
            adjust_board(board[j].first, board[j].second);

        release(p);
    } /* while (board_num > 1) */
}
```

fund.h

```
/*=========================================================================
*
*   @(#) fund.h  last update June 4, 1992
*
*   Definition header file, used by search.pc and fund.c
*
==========================================================================*/

#include <stdio.h>
#include <string.h>

#define INDEX_NUM 5             /* Max # indexes a relation has */
#define GROUP_NUM INDEX_NUM + 5 /* Max # groups */

#define FIELD_LENGTH 42

#define NAME_LENGTH 30

#define MAX_NUM_ATTR 10         /* Maximum as 8 x sizeof(long int) */

/* structure for attributes description */
typedef struct {
        char name[NAME_LENGTH+1];
        short int nullable;     /* 1 if nullable, 0 otherwise */
        short int key;          /* 1 if key, 0 otherwise */
        int count;              /* # distinct values */
} attr_type;

/* structure for index description */
typedef struct {
        char name[NAME_LENGTH+1];       /* index name */
        short int unique;               /* uniqueness */
        unsigned long on;               /* attributes the index created on */
} index_type;

/* structure for FD description */
struct fd_stru {
        unsigned long left, right;
        unsigned long credit;
        struct fd_stru * next;
};
```

synonym.pc

```
/*=========================================================================
 *
 * @(#)synonym  May 5, 1992
 *
 * This is a program which does synonym matching. Giving two relations with no
 * naming conventions, i.e., fields with the same name do not suggest that they
 * are synonyms, the program tries to find attributes that are synonyms.
 *
 * One attribute can have more than one attributes in the other relation as
 * synonyms.
 *
 * For two fields A and B in different relations, we define d(A, B) as:
 *      c(A,B) = 100*(# distinct tuples equals on A and B)
 *      d(A,B) = c(A,B)/(distinct A) + c(A,B)/(distinct B)
 * this d-value will be used to judge the closeness of A and B.
 *
 *=========================================================================*/

#include <stdio.h>
#include <string.h>

/* login onto ORACLE */
#define USER "xiaobing/chen@aaa"

/* the d-value below which should not be considered */
#define LOW_BOUND 0

/* maximum number of attributes in each relation */
#define MAX_NUM_ATTR 20

/* length of table names, attribute names, etc., in bytes */
#define NAME_LENGTH 30


/* for parsing of view_text */
#define END   0
#define DOT   1
#define EQUAL 2
#define VAR   3
#define WHERE 4
#define AND   5
#define OR    6
#define NOT   7
#define OTHER 8


/* list to store synonyms for each attribute */
struct syn_stru {
        int field;                      /* attribute number */
        struct syn_stru * next;
};

/* structure for attribute specification */
struct attr_stru {
        char name[NAME_LENGTH];         /* attribute name */
        char type[8];                   /* data type  */
        int  length;                    /* data length  */
        int  precision;
        int  scale;
        char comments[200];             /* comments on the field */
        int n_syn;                      /* # synonyms already matched to it */
        struct syn_stru * syn;          /* points to its list of synonyms */
};

/* array for the attributes of two relations */
struct attr_stru a1[MAX_NUM_ATTR], a2[MAX_NUM_ATTR];

int n1, n2;             /* # of actural attributes in each relation */


/* array to store d value */
static int d[MAX_NUM_ATTR][MAX_NUM_ATTR];

/* maximum number of synonyms an attribute can have*/
int max_syn;

/* position indicator for view text analysis */
int pos;

EXEC SQL BEGIN DECLARE SECTION;
    VARCHAR uid[30];

    VARCHAR table1[30];         /* table names */
    VARCHAR table2[30];
    VARCHAR col_name[30];
    VARCHAR col_name1[30];
    VARCHAR col_type[8];
    int     col_length;
    int     col_precision;
    short int pre_ind;
    int     col_scale;
    short int scale_ind;
    VARCHAR col_comments[200];
    short int com_ind;

    VARCHAR buf[40];
    int count;
    char query[260];
    VARCHAR view_text[300];
EXEC SQL END DECLARE SECTION;
EXEC SQL INCLUDE SQLCA;



static void cluster_checking(), view_checking(), data_checking(),
            confirmation(), output();

main(argc, argv)
int argc;
char **argv;
{
    int i;

/* login to ORACLE, use userid and password provided by USER */
    strcpy(uid.arr, USER);
    uid.len = strlen(uid.arr);
    EXEC SQL CONNECT :uid;
    if (sqlca.sqlcode != 0) {
        printf("Connection problem.\n");
        exit(1);
    }

    EXEC SQL WHENEVER SQLERROR GOTO errrpt;

    if (argc == 3) {                    /* arguments in the command line are
                                           names for the two tables */
        strcpy(table1.arr, argv[1]);
        strcpy(table2.arr, argv[2]);
    }
    else {
        printf("The two table names for synonym matching: ");
        scanf("%s%s", table1.arr, table2.arr);
    }
    table1.len = strlen(table1.arr);
    table2.len = strlen(table2.arr);

/* check if the two tables exist in the database */
    EXEC SQL SELECT TABLE_TYPE INTO :buf FROM ALL_CATALOG
        WHERE TABLE_NAME = :table1;
    if (strncmp(buf.arr, "TABLE", 5) != 0) {
        printf("\n%s is not a table\n", table1.arr);
        exit(1);
    }
    EXEC SQL SELECT TABLE_TYPE INTO :buf FROM ALL_CATALOG
        WHERE TABLE_NAME = :table2;
    if (strncmp(buf.arr, "TABLE", 5) != 0) {
        printf("\n%s is not a table\n", table2.arr);
        exit(1);
    }

    printf("\nthe max # of synonyms an attribute can have (usually 1): ");
    scanf("%d", &max_syn);
    if (max_syn < 1) max_syn = 1;

/* SQL query to get attribute descriptions from data dictionary.
   The table queried is   ALL_TAB_COLUMNS
*/
    EXEC SQL DECLARE C1 CURSOR FOR
        SELECT COLUMN_NAME, DATA_TYPE, DATA_LENGTH, DATA_PRECISION,
               DATA_SCALE FROM ALL_TAB_COLUMNS
        WHERE TABLE_NAME = :buf;

/* query for table 1 */
    strcpy(buf.arr, table1.arr);
    buf.len = strlen(buf.arr);
    EXEC SQL OPEN C1;

    for (i=0; i<MAX_NUM_ATTR; i++) {
        EXEC SQL FETCH C1 INTO  :col_name, :col_type, :col_length,
                                :col_precision:pre_ind, :col_scale:scale_ind;
        if (sqlca.sqlcode == 1403) break;       /* end of fetch */

        col_name.arr[col_name.len] = '\0';
        col_type.arr[col_type.len] = '\0';
        strcpy(a1[i].name, col_name.arr);
        strcpy(a1[i].type, col_type.arr);
        a1[i].length = col_length;
        a1[i].precision = (pre_ind == -1)? -1 : col_precision;
        a1[i].scale = (scale_ind == -1)? -1 : col_scale;

        a1[i].comments[0] = '\0';       /* these two lines are initialization */
        a1[i].n_syn = 0;
        a1[i].syn = NULL;
    }
    n1 = i;                       /* number of attributes in table 1 */
    EXEC SQL CLOSE C1;

/* query for table 2 */
    strcpy(buf.arr, table2.arr);
    buf.len = strlen(buf.arr);
    EXEC SQL OPEN C1;

    for (i=0; i<MAX_NUM_ATTR; i++) {
        EXEC SQL FETCH C1 INTO  :col_name, :col_type, :col_length,
                                :col_precision:pre_ind, :col_scale:scale_ind;
        if (sqlca.sqlcode == 1403) break;
        col_name.arr[col_name.len] = '\0';
        col_type.arr[col_type.len] = '\0';
        strcpy(a2[i].name, col_name.arr);
        strcpy(a2[i].type, col_type.arr);
        a2[i].length = col_length;
        a2[i].precision = (pre_ind == -1)? -1 : col_precision;
        a2[i].scale = (scale_ind == -1)? -1 : col_scale;

        a2[i].comments[0] = '\0';
        a2[i].n_syn = 0;
        a2[i].syn = NULL;
    }
    n2 = i;
    EXEC SQL CLOSE C1;


/* get comments for each attributes.
   The table queried is  ALL_COL_COMMENTS
*/
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    for (i=0; i<n1; i++) {
        strcpy(col_name.arr, a1[i].name);
        col_name.len = strlen(col_name.arr);

        EXEC SQL SELECT COMMENTS INTO :col_comments:com_ind
                FROM ALL_COL_COMMENTS
        WHERE TABLE_NAME = :table1 AND COLUMN_NAME = :col_name;
        if (sqlca.sqlcode < 0  &&  sqlca.sqlcode != -1406)  goto errrpt;
        if (com_ind == -1)
            a1[i].comments[0] = '\0';
        else {
            col_comments.arr[col_comments.len] = '\0';
            strcpy(a1[i].comments, col_comments.arr);
        }
    }

    for (i=0; i<n2; i++) {
        strcpy(col_name.arr, a2[i].name);
        col_name.len = strlen(col_name.arr);

        EXEC SQL SELECT COMMENTS INTO :col_comments:com_ind
                FROM ALL_COL_COMMENTS
        WHERE TABLE_NAME = :table2 AND COLUMN_NAME = :col_name;
        if (sqlca.sqlcode < 0  &&  sqlca.sqlcode != -1406)  goto errrpt;
        if (com_ind == -1)
            a2[i].comments[0] = '\0';
        else {
            col_comments.arr[col_comments.len] = '\0';
            strcpy(a2[i].comments, col_comments.arr);
        }
    }

    EXEC SQL WHENEVER SQLERROR GOTO errrpt;

    cluster_checking();

    EXEC SQL WHENEVER SQLERROR CONTINUE;

    view_checking();
    EXEC SQL WHENEVER SQLERROR GOTO errrpt;

    data_checking();

    confirmation();

    output();

    exit(0);

errrpt:
    printf("\n %.70s (%d)\n", sqlca.sqlerrm.sqlerrmc, -sqlca.sqlcode);
    EXEC SQL WHENEVER SQLERROR CONTINUE;
    EXEC SQL ROLLBACK WORK RELEASE;
    exit(1);
}



/************************************************************************
 *
 *   Synopsis
 *       int synonyms(x, y)
 *           int x, y;
 *
 *   Arguments
 *       int x, y        attributes in table1 and table2 respectively.
 *
 *   Return values
 *       see description below.
 *
 *   Description
 *       checks if x and y are synonyms presently recorded in one's attribute
 *       description structure.
 *       It returns 1 if they are, and 0 if they're not.
 *
 ************************************************************************/
static int
synonyms(x, y)
int x, y;
{
    struct syn_stru *p;

    p = a1[x].syn;
    while (p != NULL  &&  p->field != y)  p = p->next;
    if (p == NULL) return(0);
    else return(1);
}



/************************************************************************
 *
 *   Synopsis
 *       add_synonym(x, y)
 *           int x, y;
 *
 *   Arguments
 *       int x, y        attributes in table1 and table2 respectively.
 *
 *   Description
 *       record the fact that x and y are synonyms.
 *
 ************************************************************************/

static void
add_synonym(x, y)
int x, y;
{
    struct syn_stru *p;

    if (a1[x].n_syn >= max_syn  ||  a2[y].n_syn >= max_syn) {
        printf("Too many synonyms for one field, synonyms not added\n\n");
        return;
    }
    if (synonyms(x, y) == 0) {
        p = (struct syn_stru *)malloc(sizeof(struct syn_stru));
        p->field = y;
        p->next = a1[x].syn;
        a1[x].syn = p;
        a1[x].n_syn++;

        p = (struct syn_stru *)malloc(sizeof(struct syn_stru));
        p->field = x;
        p->next = a2[y].syn;
        a2[y].syn = p;
        a2[y].n_syn++;

        d[x][y] = 0;
        printf("%s.%s and %s.%s added as synonyms.\n\n", table1.arr,
                a1[x].name, table2.arr, a2[y].name);
    }
}



/************************************************************************
 *
 *   Synopsis
 *       rem(x, y)
 *           int x, y;
 *
 *   Arguments
 *       int x, y        attributes in table1 and table2 respectively.
 *
 *   Description
 *       remove the record  that x and y are synonyms.
 *
 ************************************************************************/

static void
rem(x, y)
int x, y;
{
    struct syn_stru *p, *q;

    p = a1[x].syn;
    while (p != NULL  &&  p->field != y) {
        q = p;
        p = p->next;
    }
    if (p == NULL)
```

```
        printf("they're not synonyms.\n\n");
    else {
        if (p == a1[x].syn) a1[x].syn = p->next;
        else  q->next = p->next;
        free(p);
        a1[x].n_syn--;

        p = a2[y].syn;
        while (p != NULL  &&  p->field != x) {
            q = p;
            p = p->next;
        }
        if (p == a2[y].syn)  a2[y].syn = p->next;
        else  q->next = p->next;
        free(p);
        a2[y].n_syn--;
        printf("synonyms removed.\n\n");
    }
}



/***********************************************************************
*
*   Synopsis
*       data_checking()
*
*   Description
*       checking data in the database to calculate d-values. The d-values
*       will be used in the synonym-matching procedure in confirmaion().
*
***********************************************************************/

static void
data_checking()
{
    int c1[MAX_NUM_ATTR], c2[MAX_NUM_ATTR];
    int i, j, m;
    struct syn_stru *p;

/* get the number of distinct tuples for each attributes in table 1 */
    for (i=0; i<n1; i++) {
        sprintf(query, "SELECT COUNT(DISTINCT %s) FROM %s ", a1[i].name,
                table1.arr);
        EXEC SQL PREPARE DS1 FROM :query;
        EXEC SQL DECLARE DC1 CURSOR FOR DS1;
        EXEC SQL OPEN DC1;
        EXEC SQL FETCH DC1 INTO :count;
        EXEC SQL CLOSE DC1;

        if (count == 0) {
            printf("\ntable with no data yet\n");
            return;
        }
        c1[i] = count;
    }

/* get the number of distinct tuples for each attributes in table 1 */
    for (i=0; i<n2; i++) {
        sprintf(query, "SELECT COUNT(DISTINCT %s) FROM %s ", a2[i].name,
                table2.arr);
        EXEC SQL PREPARE DS2 FROM :query;
        EXEC SQL DECLARE DC2 CURSOR FOR DS2;
        EXEC SQL OPEN DC2;
        EXEC SQL FETCH DC2 INTO :count;
        EXEC SQL CLOSE DC2;

        c2[i] = count;
    }

/* calculating d-values. only pairs of the same data type and length are
   calculated
*/
    for (i=0; i<n1; i++)
        for (j=0; j<n2; j++)
          if (strcmp(a1[i].type, a2[j].type)==0 && a1[i].length==a2[j].length
            && a1[i].precision==a2[j].precision && a1[i].scale==a2[j].scale &&
                 !synonyms(i, j)) {
  sprintf(query,"SELECT COUNT(DISTINCT %s) FROM %s, %s WHERE %s.%s =%s.%s",
                 a1[i].name, table1.arr, table2.arr, table1.arr, a1[i].name,
                 table2.arr, a2[j].name);
                     EXEC SQL PREPARE DS3 FROM :query;
                     EXEC SQL DECLARE DC3 CURSOR FOR DS3;
                     EXEC SQL OPEN DC3;
                     EXEC SQL FETCH DC3 INTO :count;
                     EXEC SQL CLOSE DC3;

                     d[i][j] = 100*count/c1[i] + 100*count/c2[j];
          }
}



/***********************************************************************
*
**  Synopsis
**      int name2num(t, s)
**          int t;      taking two values (1/2), representing table1 or
**                      table2.
**          char *s;    attribute name string.
**
**  Return values
*       the cardinal of s if s found in table t,
**      or -1 if not found.
**
***********************************************************************/

static int
name2num(t, s)
int t;
char ** s;
{
    struct attr_stru *a;
    int n;
    int i;

    if (t != 1  && t!= 2)  return(-1);
    if (t == 1) {
        a = a1;
        n = n1;
    }
    else {
        a = a2;
        n = n2;
    )
    for (i=0; i<n; i++)
        if (strcmp(s, a[i].name) == 0)
            return(i);
    if (i == n)  return(-1);
)



/***********************************************************************
*
*   Synopsis
*       cluster_checking()
*
*   Description
*       it checks if two attributes from the two tables are defined
*       on the same cluster column of a cluster. If it is, they are
*       recorded as synonyms.
*       data dictionary table checked for cluster:
*               USER_CLU_COLUMNS
*
***********************************************************************/

static void
cluster_checking()
(
    int i, j;
    struct syn_stru * p;

    EXEC SQL DECLARE CC CURSOR FOR
        SELECT T1.TAB_COLUMN_NAME, T2.TAB_COLUMN_NAME, T1.CLUSTER_NAME
        FROM USER_CLU_COLUMNS T1, USER_CLU_COLUMNS T2
        WHERE T1.CLUSTER_NAME = T2.CLUSTER_NAME
          AND T1.CLU_COLUMN_NAME = T2.CLU_COLUMN_NAME
          AND T1.TABLE_NAME = :table1
          AND T2.TABLE_NAME = :table2;
    EXEC SQL OPEN CC;
    while (1) (
        EXEC SQL FETCH CC INTO :col_name, :col_name1, :buf;
        if (sqlca.sqlcode == 1403) break;
        col_name.arr[col_name.len] = '\0';
        col_name1.arr[col_name1.len] = '\0';
        buf.arr[buf.len] = '\0';

/* col_name col_name1 are synonyms, they are converted to attribute number */
        i = name2num(1, col_name.arr);
        j = name2num(2, col_name1.arr);

        if (i >=0  &&  j >= 0)  (
            printf("\nby checking CLUSTER %s...\n", buf.arr);
            add_synonym(i, j);
        )
    )
    EXEC SQL CLOSE CC;
)


/***********************************************************************
*
*   Synopsis
```

```
*       int get_word(s)
*           char *s;
*
*   Description
*       to get a word from view_text, returns symbolic constant.
*       invoked only by view_syn()
*
***********************************************************************/

static int
get_word(s)
char *s;
(
    char c;
    int i;

    i = 0;
    while (view_text.arr[pos] == ' ')  pos++;
    c = view_text.arr[pos];
    if (c == '\0')   return(END);
    pos++;
    if (c == '.')  return(DOT);
    if (c == '=')  return(EQUAL);
    if (c >='A'  &&  c <= 'Z') (
        while ((c >= 'A' && c <= 'Z') || (c >= '0' && c <= '9') ||
               (c == '_') || (c == '#') || (c == '$')) (
            s[i++] = c;
            c = view_text.arr[pos++];
        )
        pos--;
        s[i] = '\0';
        if (strcmp(s, "WHERE") == 0) return(WHERE);
        if (strcmp(s, "AND") == 0)   return(AND);
        if (strcmp(s, "OR") == 0)    return(OR);
        if (strcmp(s, "NOT") == 0)    return(NOT);
        return(VAR);
    )
    return(OTHER);
)


/***********************************************************************
*
*   Synopsis
*       view_syn()
*
*   Description
*       analizes a view_text to extract synonym informations.
*       Presently it checks if two attributes from the two tables
*       are involved in a EQUAL condition, then they are recorded
*       as synonyms.
*
***********************************************************************/

static void
view_syn()
(
    char symbol[NAME_LENGTH+1], hold[NAME_LENGTH+1];
    int token;
    int t1, a1, t2, a2;

    pos = 0;
    while ((token = get_word(symbol)) != WHERE && token != END) ;
```

synonym.pc

```
    while (token != END) {
        t1 = t2 = a1 = a2 = -1;
        while ((token = get_word(symbol)) != VAR  &&  token != END) ;
        if (token != END) {
            token = get_word(hold);
            if (token == DOT) {
                token = get_word(hold);
                if (strcmp(symbol, table1.arr) == 0)  t1 = 1;
                if (strcmp(symbol, table2.arr) == 0)  t1 = 2;
                a1 = name2num(t1, hold);
                token = get_word(symbol);
            }
            else
                if ((a1 = name2num(1, symbol)) != -1)
                    t1 = 1;
                else {
                    a1 = name2num(2, symbol);
                    t1 = 2;
                }
            if (a1 != -1  &&  token == EQUAL) {
                token = get_word(symbol);
                if (token == VAR) {
                    token = get_word(hold);
                    if (t1 == 1) t2 =2;
                    else t2 =1;
                    if (token == DOT) {
                        token = get_word(hold);
                        if (t2 == 1 && strcmp(symbol, table1.arr) == 0 ||
                            t2 == 2 && strcmp(symbol, table2.arr) == 0  )
                                a2 = name2num(t2, hold);
                    }
                    else  a2 = name2num(t2, symbol);
                    if (a2 != -1) {
                        printf("\nby checking VIEW %s...\n", col_name.arr);
                        if (t1 == 1) add_synonym(a1, a2);
                        else add_synonym(a2, a1);
                    }
                }
            }
        }
    }
}

/*******************************************************************
*
*   Synopsis
*       view_checking()
*
*   Description
*       taking the text definitions for VIEWs which contains table1
*       and table2, calling view_syn() to analyze the texts.
*
********************************************************************/

static void
view_checking()
{
    EXEC SQL DECLARE CV CURSOR FOR
        SELECT VIEW_NAME, TEXT FROM USER_VIEWS;
    EXEC SQL OPEN CV;

    while (1) {
        EXEC SQL FETCH CV INTO :col_name, :view_text;
        if (sqlca.sqlcode == 1403)  break;
        if (sqlca.sqlcode < 0  &&  sqlca.sqlcode != -1406)  continue;
        col_name.arr[col_name.len] = '\0';
        if (view_text.len == 300)  view_text.len = 299;
        view_text.arr[view_text.len] = '\0';

        if (strstr(view_text.arr, table1.arr) != NULL  &&
                strstr(view_text.arr, table2.arr) != NULL)
            view_syn();
    }
    EXEC SQL CLOSE CV;
}


/*******************************************************************
*
*   Synopsis
*       get_current(x, y)
*           int *x, *y;
*
*   Arguments
*       int *x, *y;     the two int pointers will be used to return the
*                       selection result; if no pair of attributes can
*                       be selected, *x will bring back value -1.
*
*   Description
*       gets one pair of attributes which are most likely synonyms
*       based on the d-value array. Pairs with d-value less than
*       LOW_BOUND is not selectable.
*       The pair is not previously confirmed as synonyms or non-synonyms.
*
********************************************************************/

static void
get_current(x, y)
int *x, *y;
{
    int i, j, m;

    m = LOW_BOUND;
    for (i=0; i<n1; i++)
      for (j=0; j<n2; j++)
        if (a1[i].n_syn < max_syn && a2[j].n_syn < max_syn && m < d[i][j]) {
            m = d[i][j];
            *x = i;
            *y = j;
        }
    if (m == LOW_BOUND)  *x = -1;
    return;
}


/*******************************************************************
*
*   Synopsis
*       show_current(x, y)
*           int x, y;
*
*   Description
*       prints the current pair of attributes for synonym confirmation.
```

synonym.pc

```
*
************************************************************************/

static void
show_current(x, y)
int x, y;
(
    if (x == -1)  printf("\nNo more pair can be synonyms\n");
    else (
        printf("\n%s.%s\n", table1.arr, a1(x].name);
        printf("comments: %s\n", a1[x].comments);

        printf("\n%s.%s\n", table2.arr, a2[y].name);
        printf("comments: %s\n", a2[y].comments);

        printf("\ntheir being synonyms need to be confirmed\n");
    )
)



/************************************************************************
*
•   Synopsis
•       get_attr(s, t, a)
•           char *s;
*           int *t, *a;
*
*   Arguments
*       char *s;        a input string for an attribute name, can be a single
*                       attribute name or prefixed by 'tablename.'.
*       int *t;         will return the table the attribute is in.
*                       if the attribute is in both table1 and table2, return 0;
*                       if the attribute is in only one table, return 1 or 2;
*                       if the attribute is not in any table, return -1.
*       int *a:         the cardinal of the attribute, valid only when *t does
*                       not return -1.
*
•   Description
•       given a string in s representing the (full) name of an attribute, it
•       tries to locate what table the attribute is in and the cardinal of
•       the attribute if it's in one of the table.
•
*************************************************************************/

static void
get_attr(s, t, a)
char *s;
int *t, *a;
(
    char c[NAME_LENGTH];
    int i, j;

    i = 0;
    while (s[i] != '\0'  &&  s[i] != '.')  c[i] = s[i++];
    c[i] = '\0';
    if (s[i] == '.') (
        if (strcmp(c, table1.arr) == 0)  *t = 1;
        else if (strcmp(c, table2.arr) == 0)  *t = 2;
        else (
            *t = -1;
            return;
        )
        j = 0;
        do (
            c[j++] = s[++i];
        ) while (s[i] != '\0');
        *a = name2num(*t, c);
        if (*a == -1) *t = -1;
        return;
    )
    else (
        *a = name2num(1, c);
        if (*a >= 0)
            if (name2num(2, c) == -1)  *t = 1;
            else *t = 0;
        else
            if ((*a = name2num(2, c)) >= 0)  *t = 2;
            else  *t = -1;
        return;
    )
)



/************************************************************************
*
*   Synopsis
•       putattr(t, i)
*           int t, i;
*
•   Arguments
•       int t;  table t, takes values of 1 or 2;
•       int i;  the i's attribute in the attribute structure array.
•
*   Description
•       outputs the description of attribute a in table t.
*       t and a must be correct for locating an attribute in a1[] or a2[].
•
*************************************************************************/

static void
putattr(t, i)
int t, i;
(
    struct attr_stru *a, *aa;
    struct syn_stru *p;

    if (t == 1) (
        a = a1;
        aa = a2;
    )
    else (
        a = a2;
        aa = a1;
    )
    printf("name: %s\ntype: %s\t\tlength: %d\n", a[i].name, a[i].type,
                a[i].length);
    printf("comments: %s\n", a[i].comments);
    printf("synonyms in the other table:\n");
    p = a[i].syn;
    while (p != NULL) [
        printf("%s  ", aa[p->field].name);
        p = p->next;
    )
    printf("\n\n");
}
```

```
/**********************************************************************
*
*   Synopsis
*       show_table(t)
*           int t;
*
*   Argument
*       int t;  takes values 1 or 2, indicates which table.
*
*   Description
*       it prints out the information for all attributes in a table t.
*
***********************************************************************/

static void
show_table(t)
int t;
(
    int i, n;
    struct syn_stru *p;

    if (t == 1) (
        n = n1;
        printf("\n\nTABLE %s\n\n\n", table1.arr);
    )
    else (
        n = n2;
        printf("\n\nTABLE %s\n\n\n", table2.arr);
    )
    for (i=0; i<n; i++)  putattr(t, i);
]


/**************************************************************************
*
*   Synopsis
*       confirmation()
*
*   Description
*       it is an interactive environment to prompt a pair of attributes
*       for user confirmation as synonyms. It is based on the d-value
*       calculated previously. It provides the following command:
*
*       help                    help message.
*       table tablename         the schema of tablename
*       current                 display current selection for confirmation.
*       yes                     claim of synonym after two attributes provided.
*       no                      disclaim of synonym.
*       display table.attr      display the description of attr# in table# and
*                               its current synonyms.
*       add table1.attr1 table2.attr2
*                               add table1#.attr1# and table#.attr2# as
*                               synonyms.
*       remove table1.attr1 table2.attr2
*                               remove these two attributes as synonyms.
*       quit                    quit the confirmation environment.
*
**************************************************************************/

static void
```

```
confirmation()
(
    int i, j;
    struct syn_stru *p;
    char command[10], line[50];
    char t1[2*NAME_LENGTH+1], t2[2*NAME_LENGTH+1];
    int cur1, cur2, it1, it2, at1, at2;

    printf("\n\n\t\t\tConfirmation Process\n\n");
    printf("type help for HELP\n\n");

    get_current(&cur1, &cur2);
    show_current(cur1, cur2);
    do (
        do (
            putchar('>');
            gets(line);
        ] while (strlen(line] == 0);
        i = 0;
        while (line[i] != '\0') (
            if (line[i] >= 'a' && line[i] <= 'z')
                line[i] = line[i] - 'a' + 'A';
            i++;
        ]
        if (line[0] == 'H') (
            printf("\n\n Command available:\n");
            printf("H[elp]\t\t\tthis help message.\n");
            printf("T[able] tablename\tdisplay the schema of tablename");
            printf("C[urrent]\t\t\tshow current pair asking for \
confirmation\n");
            printf("Y[es]\t\t\tclaim of synonym for the current pair.\n");
            printf("N[o]\t\t\tdisclaim of the current pair as synonyms.\n");
            printf("D[isplay] [table.]attr\n");
            printf("\t\t\tdescribe table.attr and its current synonyms.\n");
            printf("A[dd] [table1.]attr1 [table2.]attr2\n");
            printf("\t\t\tadd table1.attr1 and table2.attr2 as synonyms.\n");
            printf("R[emove] [table1.]attr1 [table2.attr2]\n");
            printf("\t\t\tremove these two attributes as synonyms.\n");
            printf("Q[uit]\t\t\tquit the confirmation environment.\n");
        )
        else if (line[0] == 'T') (
            if (sscanf(line, "%s%s", command, t1) != 2)
                printf("a tablename should follow this command.\n\n");
            else if (strcmp(t1, table1.arr) == 0) show_table(1);
                 else if (strcmp(t1, table2.arr) == 0) show_table(2);
                      else printf("wrong table name.\n\n");
        )
        else if (line[0] == 'C')  show_current(cur1, cur2);
        else if (line[0] == 'Y') (
            if (cur1 != -1) (
                add_synonym(cur1, cur2);
                get_current(&cur1, &cur2);
            )
            show_current(cur1, cur2);
        )
        else if (line[0] == 'N') (
            if (cur1 != -1) (
                d[cur1][cur2] = 0;
                get_current(&cur1, &cur2);
            )
            show_current(cur1, cur2);
        )
        else if (line[0] == 'D') (
            if (sscanf(line, "%s%s", command, t1) != 2)
```

```
                printf("a string representing a field should follow the \
command.\n");
            else {
                get_attr(t1, &it1, &at1);
                switch (it1) {
                  case -1:  printf("table or attribute name not found\n\n");
                            break;
                  case  0:  printf("field name not unique. table name as \
prefix should be provided\n\n");
                            break;
                  default:  putattr(it1, at1);
                }
            }
        }
        else if (line[0] == 'A') {
            if (sscanf(line, "%s%s%s", command, t1, t2) != 3)
                printf("Two strings representing two fields should follow \
this command.\n");
            else {
                get_attr(t1, &it1, &at1);
                get_attr(t2, &it2, &at2);
                if (it1 == -1  ||  it2 == -1)
                    printf("table or attribute name not found\n\n");
                else if(it1 == 0  ||  it2 == 0)
                        printf("field name not unique. table name as prefix \
should be provided\n\n");
                else if (it1 == it2)
                        printf("identical table selected.\n\n");
                else {
                    if (it1 == 1)  add_synonym(at1, at2);
                    else  add_synonym(at2, at1);
                    get_current(&cur1, &cur2);
                    show_current(cur1, cur2);
                }
            }
        }
        else if (line[0] == 'R') {
            if (sscanf(line, "%s%s%s", command, t1, t2) != 3)
                printf("Two strings representing two fields should follow \
this command.\n");
            else {
                get_attr(t1, &it1, &at1);
                get_attr(t2, &it2, &at2);
                if (it1 == -1  ||  it2 == -1)
                    printf("table or attribute name not found\n\n");
                else if(it1 == 0  ||  it2 == 0)
                        printf("field name not unique. table name as prefix \
should be provided\n\n");
                else if (it1 == it2)
                        printf("identical table selected\n\n.");
                else {
                    if (it1 == 1)  rem(at1, at2);
                    else  rem(at2, at1);
                    get_current(&cur1, &cur2);
                    show_current(cur1, cur2);
                }
            }
        }
    } while (strcmp(line, "QUIT") != 0);
}
```

```
/*******************************************************************
*
•   Synopsis
•       output()
*
*   Description
•       outputs the final result, currently only to the screen, not
•       saved into any file.
•
*******************************************************************/

static void
output()
{
    show_table(1);
    printf("\n\n");
    show_table(2);
}
```

REFERENCES

[1] M. Bates, & R. Bobrow **Natural Language Interfaces: What's here, What's coming, and Who Needs It**, *Artificial Intelligence Applications for Business* (W. Reitman, ed.), pp.179-194. Norwood, NJ:Ablex Publishing Company, 1984

[2] Y. Cai, N. Cercone, & J. Han **Attribute-Oriented Induction in Relational Databases**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by G. Piatetsky-shapiro and W. Frawley, pp.213-228, AAAI/MIT Press, 1991

[3] N. Cercone, G. Hall, S. Joseph, M. Kao, W.S. Luk, P. McFetridge, & G. McCalla **Natural Language Interfaces: Introducing SystemX**, *Advances in Artificial Intelligence in Software Engineering*, JAI Press Inc., Vol.1, pp.169-250, 1990

[4] M. Chen & L. McNamee **Summary Data Estimation Using Decision Trees**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by G. Piatetsky-shapiro & W. Frawley, pp.309-323, AAAI/MIT Press, 1991

[5] N. Cercone & G. McCalla **Accessing Knowledge Through Natural Language**, In **Advances in Computers**, 25th Anniversary Issue(M.C. Yovits, ed.), pp.1-99, New York: Academic Press, 1986

[6] N. Cercone, P. McFetridge, G. Hall, C. Groenboer **An Unnatural Natural Language Interface**, *Presented at the combined 16th International ALLC Conference & 9th International Conference on Computers and the Humanities*, June 5-10, 1989, University of Toronto

[7] K. Chan & A. Wong **A Statistical Technique for Extracting Classificatory Knowledge from Databases**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by G. Piatetsky-shapiro and W. Frawley, pp.107-123, AAAI/MIT Press, 1991

[8] D. Chiu, A. Wong, & B. Cheung **Information Discovery through Hierarchical Maximum Entropy Discretization and Synthesis**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by G. Piatetsky-shapiro and W. Frawley, pp.125-140, AAAI/MIT Press, 1991

[9] Y. Cai **Attribute-Oriented Induction in Relational Databases**, *MSc Thesis, Computing Science, Simon Fraser Univ.*, 1989.

[10] C. J. Date **An Introduction to Database Systems**, Vol.1, Fourth Edition, Chapter 17, Addison-Wesley, 1986

[11] J. Davidson **Natural Language Access to a Database: User Modelling and Focus**, *Proceedings of the 4th National Conference of the CSCSI/SCEIO, Saskatoon, Saskatchewan*, 1982, pp.204-211

[12] W. Frawley, G. Piatetsky-Shapiro & C. Matheus **Knowledge Discovery in Databases: An Overview**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by G. Piatetsky-shapiro and W. Frawley, pp.1-27, AAAI/MIT Press, 1991

[13] G. Hall **Querying Cyclic Databases in Natural Language**, *MSc Thesis, Computing Science, Simon Fraser Univ.*, 1986.

[14] D. Jensen **Knowledge Discovery Through Induction with Randomization Testing, Knowledge Discovery in Databases Workshop 1991**, AAAI-91, pp.148-159

[15] S. Johnson **Hierarchical Clustering Schemes**, *PSYCHOMETRIKA*, vol. 32, No. 3, Sept. 1967, pp.241-254

[16] M. Jafer & J. Żytkow **Cashing in on the Regularities Discovered in a Database, Knowledge Discovery in Databases Workshop 1991**, AAAI-91, pp.133-147

[17] K. Kaufman, R. Michalski, & L. Kerschberg **Mining for Knowledge in Databases: Goals and General Description of the INLEN System**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by G. Piatetsky-shapiro & W. Frawley, pp.449-462, AAAI/MIT Press, 1991

[18] M. Kantola, H. Mannila, K. Räihä, & H. Siirtola **Discovering Functioal Dependencies in Relational Databases**, *Knowledge Discovery in Databases Workshop 1991*, AAAI-91, pp.179-190

[19] W. Kim **Object-Oriented Databases: Definition and Research Directions**, *IEEE Trans. on Knowledge and Data Engineering*, Vol.2, No.3, pp.327-341, Sept. 1990.

[20] D. Maier **The Theory of Relational Databases**, Computer Science Press, 1983, pp.51-62, pp.98-101

[21] **Natural Language Processing**, *proceedings EAIA '90, 2nd Advanced School in Artificial Intelligence*, M. Filgueiras, *et al.* eds, Guarda, Portugal, Oct. 8-12, 1990

[22] R. Narayan **DATA DICTIONARY: IMPLEMENTATION, USE, AND MAINTENANCE**, Prentice Hall, Mainfrase Software Series, 1988, pp.1-9, pp.33-54

[23] G. Oosthuizen **Lattice-Based Knowledge Discovery**, *Knowledge Discovery in Databases Workshop 1991*, AAAI-91, pp.221-235

[24] **ORACLE RDBMS DATABASE ADMINISTRATOR'S GUIDE**, Version 6.0, Copyright ©Oracle Corporation, 1987

[25] E. Pednault **Minimal-Length Encoding and Inductive Inference**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by Piatetsky-shapiro, G. and Frawley, W., pp.71-92, AAAI/MIT Press, 1991

[26] E. Pednault **Inferring Probabilistic Theories from Data**, *Proceedings of the Seventh National Conference on Artificial Intelligence*, pp.624-628, Menlo Park, Calif: American Association for Artificial Intelligence, 1988

[27] C. Schaffer **On Evaluation of Domain-Independent Scientific Function-Finding Systems**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by Piatetsky-shapiro, G. and Frawley, W., pp.93-104, AAAI/MIT Press, 1991

[28] J. Schlimmer **Learning Determinations and Checking Databases, Knowledge Discovery in Databases Workshop 1991**, AAAI-91, pp.64-76

[29] **SQL*Plus User's Guide**, Version 2.0, Copyright ©Oracle Corporation, 1987

[30] J.D. Ullman **PRINCIPLES OF DATABASE AND KNOWLEDGE-BASE SYSTEMS**, vol.1, Chapter 7, Computer Science Press, 1988

[31] C. Wertz **The DATA Dictionary Concepts and Uses**, 2nd Edition, QED Information Sciences, Inc., 1989

[32] Y. Wu & S. Wang **Discovering Functional Relationships from Observational Data**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by G. Piatetsky-shapiro and W. Frawley, pp.55-70, AAAI/MIT Press, 1991

[33] C. Yang **Relational Databases**, Prentice-Hall, 1986, pp.97-112

[34] J. Żytkow & J. Baker Interactive Mining of Regularities in Databases, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by G. Piatetsky-shapiro and W. Frawley, pp.31-53, AAAI/MIT Press, 1991

[35] W. Ziarko **The Discovery, Analysis, and Representation of Data Dependencies in Databases**, *KNOWLEDGE DISCOVERY IN DATABASES*, edited by G. Piatetsky-shapiro and W. Frawley, pp.195-209, AAAI/MIT Press, 1991