13

AUTOMATIC REVERSE ENGINEERING
EXTRACTION OF ERC+ FROM
RELATIONAL DATABASES

By

Raji F. Abou Rakabeh

June 1998

AUTOMATIC REVERSE ENGINEERING

EXTRACTION OF ERC+ FROM

RELATIONAL DATABASES

RF
44
C.1

By
Raji F. Abou Rakabeh
B.Sc., Lebanese American University, June 1995

PROJECT

Submitted in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science
at the Lebanese American University
June 1998

Signatures Redacted

Dr. Issam Moghrabi (Advisor)
Assistant Professor of Computer Science
Lebanese American University

Signatures Redacted

Dr. Nashat Mansour
Associate Professor of Computer Science
Lebanese American University

To my parents

Table of Contents

	Table of Figures	vi
	Acknowledgments	viii
	Abstract	ix
Chapter 1	Introduction	1
	1.1 Background	1
	1.2 Reverse Engineering	2
	1.3 Reverse Engineering and Entity Relationship Model	3
	1.4 Organization of this Report	4
Chapter 2	ERC+ Extraction Method	5
	2.1 An Overview of the Entity Relationship Complex Plus Model (ERC+)	5
	2.1.1 Features of ERC+	7
	2.1.2 Examples of Possible ERC+ Schemas	9
	2.2 Reasons for Using ERC+ Model	10
	2.3 Extraction Method's Main Tasks	12
Chapter 3	Information Extraction	14
	3.1 Tables Extraction	15
	3.2 Extracting "Select" Statements	15
	3.2.1 Source Code Features	16
	3.2.2 Syntax of the General "Select" Statement	16
	3.2.3 Storing Queries	20
	3.3 Creation of a Connection Diagram	22
	3.4 Drawing Conclusions about Keys	24
	3.4.1 Tables with a Single Attribute	25
	3.4.2 Queries with Key-Selection	26
	3.4.3 Anchors with Cyclic Join Links	26
	3.4.4 Anchors in the GROUP BY Clause	27
	3.4.5 Elimination by Candidate Key	27
	3.4.6 Candidate Key Deduction	29

3.4.7	Candidate Key Uniqueness Test	31
3.4.8	Relations with Undetermined Candidate Key	33
3.4.9	Primary Key Selection	34
3.5	Determining Link Type	36
3.6	Deleting Non-Structural Links	40
Chapter 4	Translation Rules	44
4.1	Node Rules	45
4.2	Link Rules	49
4.3	Refinement Rules	51
Chapter 5	Experimental Results	60
5.1	Example 1	60
5.2	Example 2	70
5.3	Execution Time	74
Chapter 6	Conclusion and Further Work	76
References		78
Appendix A	Creating Subsets of a Set	80
Appendix B	Breadth-First Search Technique	81

Table of Figures

Chapter 2	Figure 2.1	A diagram of hypothetical ERC+ schema	9
	Figure 2.2	An ERC+ diagram for a complex entity type	10
	Figure 2.3	An ERC+ generalization diagram	10
	Figure 2.4	Data flow chart of the method	13
Chapter 3	Figure 3.1	Data flow chart for Information Extraction	14
	Figure 3.2	Connection Diagram showing join conditions, anchors and links	24
	Figure 3.3	Connection Diagram formed from queries 1 and 2	28
	Figure 3.4	Connection Diagram with keys and inclusion dependencies	37
	Figure 3.5	Connection Diagram with links 3 and 4 classified as non-structural	40
	Figure 3.6	Connection Diagram with only structural links	41
	Figure 3.7	Example queries of an application	41
	Figure 3.8	Connection Diagram for queries in figure 3.7	42
	Figure 3.9	Connection Diagram with an induced link 4	42
Chapter 4	Figure 4.1	Data flow chart for Translation Rules	44
	Figure 4.2	A Connection Diagram	46
	Figure 4.3	Another Connection Diagram	47
	Figure 4.4	Table defining the translation of nodes	48
	Figure 4.5	Table defining the translation of links	50
	Figure 4.6	Example of Connection Diagram	51
	Figure 4.7	ERC+ diagram when L has “undetermined” or “outgoing” property for figure 4.6	51
	Figure 4.8	Shared property <i>ENGINE</i>	52
	Figure 4.9	Generic type G with shared property <i>ENGINE</i>	53
	Figure 4.10	Connection Diagram with a dependent and relationship type	54
	Figure 4.11	ERC+ diagram when L has “outgoing” or “undetermined” property for the connection Diagram in figure 4.10	54
	Figure 4.12	Connection Diagram with dependent and entity type	55

	Figure 4.13	ERC+ diagram when L has “outgoing” or “undetermined” property for the Connection Diagram in figure 4.12	55
	Figure 4.14	Connection Diagram with relationship and entity type objects	56
	Figure 4.15	ERC+ diagram when L has “outgoing” or “undetermined” property for the Connection Diagram in figure 4.14	56
	Figure 4.16	Connection Diagram with two entity objects	57
	Figure 4.17	ERC+ diagram when L has “outgoing” or “undetermined” property for the Connection Diagram in figure 4.16	57
	Figure 4.18	Entity type with no “isa” link	58
	Figure 4.19	Connection Diagram with an “isa” link	58
	Figure 4.20	ERC+ diagram showing attributes with an “isa” link for Connection Diagram in figure 4.19	58
	Figure 4.21	Connection Diagram with a relationship and two entity objects	59
	Figure 4.22	ERC+ diagram for the Connection Diagram in figure 4.21	59
Chapter 5	Figure 5.1	Connection Diagram for Example 1	62
	Figure 5.2	Connection Diagram with only structural links for Example 1	67
	Figure 5.3	ERC+ diagram for example 1	68
	Figure 5.4	ERC+ as displayed by our tool for Example 1	69
	Figure 5.5	ERC+ as printed by our tool for Example 1	70
	Figure 5.6	Connection Diagram for Example 2	71
	Figure 5.7	ERC+ diagram for Example 2	72
	Figure 5.8	ERC+ for the entity EMPLOYEE of Example 2	72
	Figure 5.9	ERC+ for the entity DEPARTMENT of Example 2	73
	Figure 5.10	ERC+ as printed by our tool for Example 2	73

Acknowledgements

I would like to express my deepest thanks to Dr. Issam Moghrabi for his continuous support, motivation and invaluable suggestions that were essential to finish this work, in addition to being one of the best professors I have ever met. I also wish to express my sincere appreciation to Dr. Nashat Mansour for his encouragement and support, and for being an excellent reader.

Special thanks go to Dr. Ahmad Kabbani and Dr. Layla Khoury for their moral support. Also I would like to thank Mr. Vatche Papazian, Dr. May Abboud and Dr. Samer Habre for being among the greatest professors I have encountered at the Lebanese American University. My deepest appreciation also goes to Mrs. Siham El-Zein, my director, for being such a kind and understanding boss, who is a strong supporter of higher education and knowledge.

I would like to especially thank Ms. Rola Sharrouf for extensively proofreading this manuscript. I am also grateful to Mr. Ahmad Solh, Mr. Abbas Tarhini, Mr. Tarek Dana and Mr. Ali Alaywan for their constant backing and encouragement. Also a warm thank you to Ms. Rania Bedran and Ms. Suzan Abu Shakra. Thanks to Mr. Ziad Abou Tarieh, Mr. Wajdi Jabr, Mr. Hadi Majed and Mr. Farid Massoud. I have to thank also many of my friends, especially Mrs. Zeina Bedran Dana, who provided opportunities for healthy breaks from work by arranging many plans as going to theatres, restaurants, coffee shops and cinemas.

Finally, the greatest appreciation goes to my family for their support, patience and understanding.

Abstract

A reverse engineering method is presented and implemented in this work to extract a conceptual schema from a relational database. In this work we extend the extraction method presented in Andersson's paper entitled "Extracting an entity relationship schema from a relational database through reverse engineering". Such a method depends on the analysis of the database extension as well as the analysis of data manipulating statements, SQL statements, in the source code of a database application that uses a Relational Database Management System. Attributes representing references between tables in the relational schema are determined by an analysis of join conditions in all queries of the database. Candidate keys are determined with the help of the referencing attributes in their corresponding relations. The analysis of the database extension will determine whether the proposed keys are actually primary keys or not. Also, database extension helps in determining the inclusion dependencies between tables. When ambiguity occurs concerning candidate keys, users can interact with the system to clarify such ambiguity. This approach makes it possible to efficiently construct a conceptual schema, ERC+, from only elementary information.

Chapter 1

Introduction

1.1 Background

Year after year, any business application will continuously face new needs and requirements in its Information System (IS) [11]. Such change in the IS will make the business application suffer unless it is properly adapted to satisfy the new needs and requirements. The above scenario is too common; thus, the application is subjected to correction, adaptation and enhancements many times without being documented properly. If we look closely at the modified application, we find it different from the original application and the current documentation does not reflect the application in hand. Such development will make the application unstable. Non-stability does not mean that the system is not working, but every time a change is attempted, unexpected and serious side effects occur.

These properties, lack of documentation and non-uniformity, resulting from numerous changes and modification lead to an inflexible systems and continuous high maintenance costs. The advisable solution to this non-stability and inflexibility is to reengineer the system and produce a new application. This new application will satisfy all needs and requirements of the current IS, producing an up-to-date documentation. As we know, a database application usually has its structure chart, data flow diagram as well as its conceptual schema. In this paper, we present a

method to extract a conceptual schema from a relational database using Reverse Engineering.

1.2 Reverse Engineering

The term “reverse engineering” originated in the hardware world [11]. A company takes a competitive hardware product from the market and attempts to break it down in order to understand the secrets of its design and manufacturing. Such secrets could easily be understood if specifications of the competitor’s design and manufacturing were obtained. However, such documentation is not available to the company doing reverse engineering. Basically, reverse engineering succeeds at coming up with one or more design and manufacturing specifications for a product simply by examining an actual model of the product.

Reverse engineering for software works quite in the same way; yet in most cases, the program undergoing reverse engineering is not a competitor’s but rather, it is the company’s own work. Therefore, reverse engineering for software is the process of analyzing a program (source code and data structures) in an effort to create a representation of the program at a higher level of abstraction than source code and data structures. Reverse engineering tools extract data, architecture, and procedural information from an existing program [11].

Reverse engineering could be resembled to a “magic slot,”[11] whereby undocumented source listing and its data structures (database) are fed into the slot at

one end and full documentation for the computer program comes out the other end. Yet unfortunately, the magic slot does not exist in reality. Despite the fact that reverse engineering can extract design information from the source code and data structures, yet the level of abstraction and completeness of the documentation vary to a high extent. The level of abstraction deduced must be very high. The reverse engineering process should be capable of deriving procedural design representation (a low level of abstraction); program and data structure information (a somewhat higher level of abstraction); data and control flow models (a high level of abstraction). As the abstraction level increases, the software engineer is provided with information that will allow easier understanding of the program. In this work, we will produce the entity relationship model (a high level of abstraction) of a database application using Reverse Engineering.

1.3 Reverse Engineering and Entity Relationship Model

Despite its physical structure and logical organization, a database permits the definition of data objects and supports a number of methods for forming relationships among the objects. Thus, in order to reengineer one database schema into another, it is important to understand the nature of existing objects and their relations.

The work presented here represents a data reverse engineering that extracts an Entity Relationship Model (high level of data abstraction) from a relational database using both the source code of the application, as well the extension of the database.

Let us consider the reverse engineering process to be a box fed with the source code and the database of a given database application. This box will produce a high level of data abstraction in the form of Entity Relationship Model. The model used in the paper is an Entity Relationship Complex Plus (defined in chapter two) [12]. From the source code, the reverse engineering process will inspect all “select” statements to extract the join conditions between tables. These join conditions will help in finding the relationship between the database tables. With the help of the database extension, and in a selective manner, some of the join conditions are tested to first determine the nature of the relationships between tables, and second to propose a key for each table in the database. Database extension will also help in checking whether the proposed keys are primary keys or not.

1.4 Organization of this Report

This project is organized as follows. Chapter 2, entitled “ERC+ Extraction Method,” includes an overview of the Entity Relationship Complex Plus Model (ERC+), reasons for using such model and main tasks of Andersson’s extraction method.

Chapter 3, “Information Extraction” presents the first task of our extraction method. It includes tables extraction, extracting “select” statements from the source code of the application, creation of a Connection Diagram, drawing conclusions about keys. Chapter 4 “Translation Rules” will complete the work done on chapter 3 by applying translation rules to the Connection Diagram. In chapter 5, “Experimental Results,” we show the implementation of our extraction method applied to some examples. Chapter 6 includes the conclusion and the suggested further work.

Chapter 2

ERC+ Extraction Method

2.1 An overview of the Entity Relation Complex Plus Model (ERC+)

Nowadays, several new domains of applications (namely software engineering, office automation etc...) require objects with multivalued attributes and multi-instantiation (generalization concepts). Such objects are derived from database technology facilities, which model and manage objects of arbitrary complexity, usually referred to as "Complex Objects" [10]. These facilities should allow for:

1. Retrieval and modification of any complex object as a whole, as seen as a logical unit of information from the application's viewpoint, and
2. Retrieval and modification of a complex object's components may be considered as objects themselves, which independent from the parent object(s) with which they participate.

For instance, to retrieve the text of a document includes the retrieval of all of its chapters, sections, paragraphs etc... .On the other hand, it should be possible to change the text of a specific paragraph without being forced to manipulate the whole test.

In this section, we shall introduce the concept of an enhanced entity relationship model referred as “ERC+” (for entity relationship complex plus generalization). The development of ERC+ data model was started in 1983 at the University of Burgundy (Dijon, France) as part of a project on heterogeneous distributed databases systems [12]. One of the major problems was proper integration of heterogeneous schemas. Clearly, this task required accurate knowledge of the semantics of existing schemas, beyond capabilities of traditional data models. A richer semantics model was needed. Therefore, besides technical qualities (expressive power), friendliness was assigned as additional requirement for the data model.

ERC+ is an extended entity-relationship model and can be viewed as a big step from traditional data models in the direction of object-oriented models [8]. ERC+ model permits the display of the structure of a database schema in diagrams with an increased readability and with the emphasis that tends towards the user perspective (the semantics of things) rather than the data model perspective [10]. These facilities are certainly the reasons that allowed the success of this approach as a database modeling technique.

Later on, in 1987, the concept of Complex Object is added to the ERC+ model. Complex object modeling is in need of a very basic facility, namely the ability to indicate the representation of an object by a collection of information, which can be decomposed into its components [6]. In turn, each of these components may be represented by a collection of information. It is important to emphasize the fact that supporting complex object does not contradict the basic distinction between entity and attribute. The distinction is based on semantics consideration (indicating the primary

objects of interest) and not on syntactic properties (whether or not they are atomic in nature). From this we can conclude that ERC+ model is a new conceptual modeling representation.

Such a model specifically allows ERC+ diagrams to have a number of levels. Moreover, at any node in the tree representing the structure of the object, values may be atomic, or form a set or a multiset.

2.1.1 Features of ERC+

The ERC+ model fully uses the three basic concepts: entity, relationship and attribute [12]. An entity represents a self-standing object (one which exists per self). A relationship represents an association among entities. An attribute conveys the representation of a property (of an entity, a relationship or an attribute). An attribute is merely a vehicle to attach a value to its parent object.

The major features of the ERC+ model allow for complex structures within the attributes and also allow attributes, entities and relationships to be valued in a multiset. ERC+ features are as follows:

1. Entity types are made up of one or more attributes.
2. Relationship types may connect any number of participating entity types; they are said to be cyclic if the same entity type participates more than once in the relationship type.

3. A role is associated to each participating entity, relationship types and attributes. The participation is characterized by its cardinalities in the form of min:max [9]. Min could take either 0 or 1, while max could take either 1 or n (n means many/ ∞). Possible cardinalities are 0:1, 1:1, 0:n and 1:n.
4. Attributes represent characteristics of entities or of relationships. Attributes may be:
 - a. Either optional or mandatory: an instance of an optional attribute may be empty, no value (min = 0); for mandatory attribute a value must be defined in each instance of the attribute (min = 1).
 - b. Either multivalued or monovalued: an instance of a multivalued attribute may include several values (max = n = ∞), while an instance of a monovalued attribute is made up of a single value (max = 1).
 - c. Either atomic or complex: if atomic, the attribute is non decomposable; if complex, the attribute is made up of a set of other attributes, which are said to be the components of that attribute. Component attributes may be atomic or complex. This nesting can proceed to any number of levels.
5. Two generalization (multi-instantiations) are supported, the “maybe” and the “isa” [13]. “Maybe” is used to express that object instances *can be* part of the extension of several object types. “Isa” means that the extension of one object type *is* included in the extension of another object type. These operators can be defined in modal logic, using the operator M for possibility and the operator N for necessity [1]. Let X and Y be object types and x and y be the population of the respective object type. The following two propositions hold.

$$X \text{ maybe } Y \equiv M(x \cap y \neq \emptyset)$$

$$X \text{ isa } Y \equiv N(x \subseteq y)$$

2.1.2 Examples of possible ERC+ schemas

Diagrammatic conventions:

Rectangles represent entity types.

Diamond boxes represent relationship types.

Attributes are represented as names attached to parent object by a line.

A dotted arrow represents generalization in the form of “maybe”.

A continuous arrow represents generalization in the form of “isa”.

A single dotted line represents a 0:1 link (optional monovalued).

A single continuous line represents a 1:1 link (mandatory monovalued).

A double dotted line represents a 0:n link (optional multivalued).

A double line (once dotted, once continuous) represents a 1:n link (mandatory multivalued).

Figure 2.1: a diagram of hypothetical ERC+ schema

Note that the links (with its associated cardinalities) are read from entity type to relationship or attribute type. Also, links are read from relationship type to attribute type.

Figure 2.2: an ERC+ diagram for a complex entity type

Figure 2.3: an ERC+ generalization diagram

2.2 Reasons for using ERC+ Model

One may pose the question of why to use Entity Relationship Complex Plus (ERC+) model instead of any other traditional models. The answer is simply that ERC+ has

proven to be the most suitable model that can satisfy most of the new applications domains (software engineering, office automation...).

As we have seen in the previous section, modern applications require from database technology an entity relationship model allowing us to represent complex objects, to be more readable and to emphasize the user perspective (the semantics of things) rather than data model perspective. ERC+ model, with all of its features, can satisfy such requirements. On the other hand, traditional data models fail to respond to such new requirements. For instance, the structural constraint inherited by the classical relational model (normalization rules, and in particular, the first normal form) force the designer to split the representation of a complex object into a set of relations, each one usually recording a single fact about the original object type. A first relation will describe the single-valued properties of the object type; additional relations are needed to describe the set-valued properties. Finally, other relations will be added to record any $n \times m$ link between the object type and other object types. As a result of such constraint, the perception of the whole thing as a single object is lost. Hence, such data model will no longer be capable of representing a complex object as one unit; it becomes less readable, emphasizing the data model perspective rather than the user perspective (the semantics of things). Besides, ERC+ model approach is intended to extend its scope to all users activities during database life cycle, not only schema definition. Such intent is very important in the reengineering process.

2.3 Extraction Method's Main Tasks

Any database application typically includes queries in the form of “select” statements embedded in its source code. Forms, reports, macros and procedures/functions represent the source code. These “select” statements, which completely reflect the manipulation of a database, necessarily possess all structural meaning Connections between the tables of the database. An analysis based on the properties of the “select” statements is complete and captures the application semantics as seen by the schema designer and the query programmer. Therefore, all we need in the extraction method as input are the source code and the database extension of the application.

The extraction method used in this work is divided into two main tasks [1] (figure 2.4). The first task, “Information Extraction Stage,” is to retrieve all necessary information using the source code and database extension as input. The outcome of this task will be a preliminary Connection Diagram (to be defined in the next chapter). The second task, “Translation Rules Stage,” is to apply different translation rules to the preliminary Connection Diagram so that it is transformed to a final form. This final form of the Connection Diagram will hold all necessary information enabling us to draw the corresponding ERC+ diagram. It is worth noting that the extraction method presented in this work builds on Andersson’s paper as a background, simply because no implementation details and data structures were suggested. Besides, Andersson’s method proposed in his paper was vague and not complete in certain subjects. As mentioned in section 5 of the same paper dated 1994, Marten Andersson stated that a preliminary study has been made on the applicability of the method to a real system used to administrate the students at a university. In a step ahead of his

paper, some ideas have been added to his method, so that it can be more useful and more practical.

Figure 2.4: data flow chart of the method

Chapter 3

Information Extraction

Figure 3.1: data flow chart for Information Extraction

This chapter aims at retrieving all necessary information from both the source code and the database extension of an application. This information is depicted in a Connection Diagram. The Connection Diagram is studied, analyzed and enhanced using the database extension. Retrieving meaningful information involves many steps (figure 3.1), which shall be explained in details throughout this chapter.

3.1 Tables Extraction

As we have mentioned earlier, a database application consists of one database and a source code (forms, reports, etc...). Any database is logically formed of a number of tables. These tables are either embedded in the database itself, as in Access, or scattered as files in a directory representing the database, as in Foxpro. In our package, we have catered for both implementations of tables.

The first step to be done in the extraction method is to retrieve the tables from the database of the application. A list of these tables is formed, whereby each table is associated with the fields forming it. Such a list is called the relation list.

3.2 Extracting “Select” Statements

After we extract the database tables of an application, we then extract “select” statements from the source code of the application. Source code includes forms, reports, macros and procedures/functions in a text file formats. This step in the information extraction stages traces all “select” statements from the source code

listings and creates a list of queries holding the traced “select” statements. Such a list is called the query list.

3.2.1 Source Code Features

Our extraction method does not put any constraints on the programmer. The programmer can use any personal programming style he chooses. A programming style can make use of several spaces and tabs, as many as the programmer desires. In addition, a line of code can have one or more programming statements, “select” statement clauses, etc.... In this project, a programmer has the freedom to use one of the following commenting directives:

1. `/* comment */` or `{ comment }`: all statements between “`/*`” and “`*/`”, or “`{`” and “`}`” are ignored when parsing the source code.
2. `// comment`: once “`//`” is encountered, the remaining text on the line is ignored.

Such flexibility while parsing the source code allows the project to operate on the most of the database application’s source code.

3.2.2 Syntax of the General “Select” Statement

This project can trace and parse a general format of a “select” statement as described below. Such a general format is similar to that most of SQL servers in the market uses.

Syntax Conventions:

- <> Angle brackets indicate information that the programmer provides.
- [] Square brackets denote that the information is optional.
- | A vertical bar separating options indicates that you can use one of the stated options.
- ... An ellipsis indicates continuity in a similar fashion.

Syntax of the “select” statement:

```

SELECT      [all | * | distinct]
            [<alias>.<select_item> [AS <column_name>]
            [, [<alias>.<select_item> [AS <column_name>]...]]

FROM        <table> [<alias>] [, <table> [<alias>]...]

[INTO       <destination> <dest_item>]

[WHERE      <joincondition> [AND<joincondition>...]
            [AND|OR <filtercondition> [AND|OR <filtercondition>...]]]

[GROUP BY   <groupcolumn> [, <groupcolumn>...]]

[HAVING     <filtercondition>[AND|OR <filtercondition>...]]

[ORDER BY   <order_item> [ASC|DESC]
            [, <order_item> [ASC|DESC]...]]

```

SELECT clause:

- “ALL” & “*” appear alone in a SELECT clause
- DISTINCT appear once in a SELECT clause
- <select_item> can be one of the following:
 - a. The name of a field from a table in the FROM clause

- b. A constant
- c. SQL field function like: average, count, min, max and sum

FROM clause

- Any number of <table> can appear
- <alias> is optional

INTO clause

Destination can be ARRAY, CURSOR, DBF or TABLE. <dest_item> is a variable name.

WHERE clause

- <joincondition>: can take this form

<field1> <comparison> <field2>

where:
 - a. field1 & field2 have the format [<alias>.]field_name
 - b. comparison operator can be: =, <, !=, >, >=, <=
e.g. company.id = payment.id
- <filtercondition>: can take one of the following forms
 1. <field> <comparison> <constant>

constant can be either a numeric constant or a memory variable.

e.g. student.id = 999111
 2. <field> [NOT] BETWEEN <start_range> AND <end_range>

e.g. customer.taxrate BETWEEN 5 and 6
 3. [NOT] EXISTS (<subquery>)

This form allows nested queries.

e.g. ... Exists (SELECT * FROM invoices i WHERE customer.zip = i.zip)

4. <field> [NOT] IN (<subquery>)

Also this allows nested queries.

e.g. ... student.id IN (SELECT id FROM courses c WHERE c.c# = 'cs1')

5. <field> [NOT] LIKE <string>

e.g. customer.state LIKE 'LA'

GROUP BY clause

<groupcolumn> can take one of the following:

- a. name of a table field
- b. field that include an SQL field function

HAVING clause

<filtercondition>: similar to the previous definition, except that sub-queries forms are not allowed.

ORDER BY clause

<order_item>: is a field in a table from the FROM clause that is also an item in the SELECT clause.

In our project, one constraint is applied to the WHERE clause. This constraint is as follows:

Nested parentheses are not allowed in the <joincondition> and in the <filtercondition> of any select statement.

Yet, if we investigate clearly any database application, it is very common and natural not to find nested parentheses. This is true because simple `<joincondition>` and `<filtercondition>` could easily retrieve any kind of data in a “select” statement

3.2.3 Storing Queries

In our extraction method, not all clauses of a “select” statement hold all the necessary information. For this reason, only valuable information will be stored in our data structure. Hence, a query in our work is represented by the SELECT, FROM, WHERE, and GROUP BY clauses. From the source code listing, we trace all queries as well as nested ones and store them in a list called query-list. Each query stored will have properties taken from the “select” statement clauses as indicated below.

From the SELECT clause, two information pieces can be retrieved. First, we can find out whether or not the keyword DISTINCT exists. Second, a list of the select clause fields is formed. Each node in that list will hold the field name and the optional table/alias if it exists. All SQL field functions are ignored.

A list of tables is formed from the FROM clause where each node in that list will hold a table name and the optional alias if it exists.

From the WHERE clause all join and filter conditions will be represented in a list. Each node will hold both fields of join condition with their table/alias of each, and the comparison operator. In the case of filter condition, one field, a constant and the

comparison operator are stored with table/alias of the field encountered. Filter conditions with BETWEEN or LIKE operators are ignored.

From the GROUP BY clause, a list of grouped items is formed. Each node will hold a column name and the table/alias if it exists.

After the list of queries is constructed, it is passed through an enhancement operator so that ambiguity is resolved. Let us look at this example:

```
SELECT * FROM student, enrollment WHERE id = id
```

As we can see, the join condition contains the field “id” on both sides of the join condition. To resolve this ambiguity, table “student” is attached to the first field, and the table “enrollment” is then attached to the second one. The query will look like this:

```
SELECT * FROM student, enrollment WHERE student.id = enrollment.id
```

So, every field in every query should appear with its corresponding table.

Let’s look at another example with nested queries:

```
SELECT * FROM student s
```

```
WHERE exists (SELECT * FROM enrollment e WHERE e.id = s.id)
```

If we look at the representation of the nested query in our project, we find that “e.id=s.id” is considered as a join condition; however, “s.id” is a constant value passed as parameter to the nested query. Hence, and to convert “e.id=s.id” to a filter condition, the alias “s” in “s.id” should be deleted. By deleting “s”, “id” becomes without a table/alias and hence is considered as a constant.

3.3 Creation of a Connection Diagram

The extraction method aims at building a Connection Diagram. This diagram is subject to different operations after which it will contain all necessary data for constructing ERC+ diagram.

Semantically related attributes in different relation schemes are identified through an analysis of join clauses in every “select” statement of the application; on other words, in every query in the query-list. When attributes are compared with an equal sign “=”, we call it an equi-join condition. In this work, we only consider equi-joins because what we are looking for are attributes that represent references between relation schemes in the database. From these references we can retrieve information about the relationships between tables. Comparing references with anything other than equality is a meaningless operation.

The information resulting from the analysis of each query is represented in a Connection Diagram with nodes representing relation schemes. An edge between two nodes R_i and R_j implies that there is a join condition involving attributes in R_i and R_j . A composite key in the relation schema results in several links, one for each attribute. Let us introduce the notion of anchor that replaces the links of a composite key with one single link in the Connection Diagram. Let the relation scheme R_i includes the set of attributes X_{i_k} , and R_j include the set of attributes Y_{j_l} . If there is at least one query with a join condition including X_{i_k} and Y_{j_l} , we say that the anchors X_{i_k} and Y_{j_l} link

the relation schemes R_i and R_j . Attributes of anchors are denoted in brackets: $\{a_1, \dots, a_i, \dots, a_n\}$, where a_i is an attribute.

To sum up, a Connection Diagram contains a node for each table in the database. Each node contains a list of queries that uses the current table in its FROM clause, and it also has a list of anchors that participate with another anchor to form a link. This link can first be between anchors of different relations. Second, it can be between two different anchors in the same relation. Finally, it can be between the same anchor of the same relation (cyclic join). With every anchor we associate a list of queries that uses this anchor in its join condition, we call it anchor-query list. We give an example below to illustrate the use of anchors.

Given a database with two relation schemes PERSON and STUDENT as follows:

PERSON(name, address, tel)

STUDENT(name, address, parentname, parentaddress)

The source code of the corresponding application contains this query:

```
SELECT Student.name, Student.address
FROM Student , Person P1, Person P2
WHERE Student.name = P1.name
and Student.address = P1.address
and Student.parentname = P2.name
and Student.parentaddress = P2.address
and P2.tel = P1.tel
```

Query 1

The Connection Diagram for the given database and query is:

Figure 3.2: Connection Diagram showing join conditions, anchors and links

As we can see in figure 3.2, there are four anchors as follows:

1. {Person.name, Person.address}
2. {Person.tel}
3. {Student.name, Student.address}
4. {Student.parentname, Student.parentaddress}

Also from the Connection Diagram, we can see that the join conditions induce three links that are numbered 1, 2 and 3.

3.4 Drawing Conclusions about Keys

Given our initial assumption that we don't have information about keys available to aid the reverse engineering process, we must find sources other than the data dictionary to determine the key. One possible solution is to study the data manipulating statements, "select" statements with the help of the database extension

of the database. The Data Dictionary of a database is excluded from our analysis due to the following reasons:

1. It is possible that the unique key constraint is imposed explicitly from within the source code of the application rather than implicitly by the Relational Database Management Systems (RDBMS).
2. It is possible that the database is imported to our method without its data dictionary.
3. We want our method to be as independent as possible from a Data Dictionary.

All we need in drawing conclusions about relation's primary keys are: the Connection Diagram, the query-list and the database extension. The following sections will show the steps through which our analysis passes so that a primary key is determined. Note that a list of candidate keys and another of non-keys are maintained with every node of the Connection Diagram. Candidate keys list represents anchors that can be primary keys, while non-keys list represents anchors that can not be primary keys.

3.4.1 Tables with a Single Attribute

For every node in the Connection Diagram, we check if the number of attributes of the current relation scheme is one or not. If only one attribute exists, then this attribute is the trivial primary key.

3.4.2 Queries with Key-Selection

A common situation in a relational database application is that the primary key of a relation is used to retrieve the tuple that matches a key-value. We can call this a key-selection. Key-selection check is applied to queries with only one relation scheme in its FROM clause (non-join queries). So, for every node in the Connection Diagram and from the list of queries associated with this node, we check for key-selection only with non-join queries. If key-selection exists in a query with the absence of the DISTINCT keyword, we can then propose attributes participating in the key-selection as a candidate key. If the non-join query doesn't include the keyword DISTINCT, then it will indicate that the result is expected to be a single tuple and that the attribute or attributes in the selection criteria is the primary key of the relation scheme. If DISTINCT exists, we add the attribute(s) to the non-keys list.

3.4.3 Anchors with Cyclic Join Links

Cyclic joins can be used to draw conclusions about keys. A cyclic join is a link with one anchor. Such an anchor can not be a candidate key because any join condition between two relation schemes aims at producing new information other than the existing ones. For key cyclic join, no new data is produced and hence, programmers will not use such joins. All cyclic joins used in any database application are non-key cyclic joins and thus such an anchor can't be a key. For every node in the Connection Diagram, we check the anchor list and if a cyclic join occurs, we add its anchor to the non-keys list.

3.4.4 Anchors in the GROUP BY clause

In any “select” statement, the use of the GROUP BY clause aims at forming groups of tuples. These groups are formed according to the values of attributes forming the GROUP BY clause. Every tuple in a certain group will have the same value for the GROUP BY clause attributes. From this, we can conclude that the GROUP BY clause attributes have duplicate values in different tuples when used with non-join queries. Hence, GROUP BY clause attributes can not be candidate key. Thus for every node in the Connection Diagram, we analyze all non-join queries in the query list associated with the node and if a GROUP BY clause exists, we add its GROUP BY clause attributes to the non-keys list of the current node.

3.4.5 Elimination by Candidate Key

When tuples from the same relation are compared, the candidate key is used to exclude tuples that match themselves. This is because any join, between two or more relations, aims at producing new data. When a tuple is joined with itself, the candidate key value will be the same on both tuples, which represent the same original tuple. As a result of this join, no new data will be produced. For this reason, every anchor participating in a cyclic join link is tested against this rule, in every node of the Connection Diagram. When such an anchor is found, we check for key elimination in every query of the attached anchor-query list. Key elimination is detected when the WHERE clause of such queries exclude tuples having the same values for certain fields. These fields are then added to the candidate keys list of the corresponding relation.

Let us add the following query to the application given in section 3.3.

```
SELECT DISTINCT s1.name, s1.address
FROM student s1, student s2
WHERE s1.parentname = s2.parentname AND
s1.parentaddress = s2.parentaddress AND
NOT (s1.name = s2.name AND s1.address = s2.address)
```

Query 2

The Connection Diagram in figure 3.3 corresponds to the new given is as follows:

Figure 3.3: Connection Diagram formed from queries 1 and 2

The link numbered 4 is a cyclic join link. {parentname, parentaddress} is the only anchor that participates in link 4. Query 2 produces this link and at the same time, the anchor {name, address} holds the attributes that are used to exclude tuples that match themselves. So, we can conclude that the anchor {name, address} is possibly a candidate key and hence, is added to the candidate keys list associated with the relation scheme STUDENT.

3.4.6 Candidate Key Deduction

In this work, we tend to find as many candidate keys as we can, so that we can later check each one to find out whether or not the uniqueness constraint of any relation in a database is satisfied or not. In case more than one candidate satisfies the uniqueness constraint, the user is then involved in choosing the primary key in the corresponding relation. In case no candidate key satisfies the uniqueness constraint, we are obliged to form all possible candidate keys from the list of table fields, which is rather time consuming. Any possible candidate key is a subset of the table fields. As an enhancement to Andersson's paper, we propose an analysis to be performed on the Connection Diagram. The analysis is based on some rules that can be used in order to propose candidate keys.

This analysis is based on the study of the anchors of a node in the Connection Diagram. Anchors form references to other relations through links. This means that both anchors of any link are semantically related. This leads us to conclude that any anchor could be one of the following:

Case 1: Anchors could, by themselves, be candidate keys of a relation and then could be chosen to be primary key.

Case2: Anchors could be part of a candidate key; e.g. as in the case of the foreign key.

A foreign key consists of attributes that are primary keys in other relations. A foreign key value in a relation can't exist unless it has a matching entry in the mother relation.

Case 3: Anchors are not necessarily candidate keys in the corresponding relation.

The following rules will be the basis of our analysis in deducing candidate keys:

When:

- (1) the number of anchors of a certain relation in the Connection Diagram is one, and
- (2) if this anchor is found on the non-keys list, and
- (3) if the fields of a relation is composed of this anchor with an additional one attribute,

we can then propose the fields of the table to be one of its candidate keys. This situation represents an all-key relation, so we add all fields of the relation to the list of candidate keys list. This represents case 2 stated above.

When:

- (1) the number of anchors of a certain relation in the Connection Diagram is one, and
- (2) if this anchor doesn't belong to the list of non-keys,

we can then propose this anchor to be a candidate key and add it to the list of candidate keys. This represents case 1 mentioned above.

When:

- (1) an anchor is not on the non-keys list, and
- (2) if this anchor participates in two links whose other anchors, one anchor for each link, are not identical,

we can then conclude that such an anchor could be a candidate key and is added to the list of candidate keys. This is true because it indicate that values of such an anchor in different tuples are distinct for it may represent an object identity. Object identities (primary keys) are usually distributed over many relations.

The final key deduction rule will apply only with relations whose candidate keys list is empty. The proposed candidate key will be composed of attributes from all the anchors of the corresponding relation.

3.4.7 Candidate Key Uniqueness Test

After the analysis proposed in section 3.4.6 is completed, it is time to prove whether or not the proposed candidate keys are really so. Candidate keys are characterized by the uniqueness property [4]; i.e. no two tuples in a relation can have the same key value. In order to check for this property, we need to scan the database extension.

Every node in the Connection Diagram is associated with a list of candidate keys. Each of these keys should be tested against the uniqueness property. This is done by invoking an SQL SELECT statement as follows:

Let “T” represent the table name and “KEY” represent the proposed candidate key.

Syntax of the required query:

```
SELECT count(*)  
FROM <T>  
GROUP BY <KEY>  
HAVING count(*) > 1
```

Semantics of the required query:

We apply the grouping operator to the involved query. The grouping items are the proposed candidate key attributes. Only groups with a number of tuples (count)

greater than ONE will satisfy the HAVING clause condition and hence, the number of tuples in such a group ($\text{count} > 1$) will be found in the table produced by the invoked query. If the table produced by the query contains tuples, then it means that there is a group in the involved relation whose number of tuples (count) is greater than ONE. Thus, we deduce that the proposed candidate key doesn't satisfy the uniqueness property, so we delete it from the proposed list of candidate keys. Once the test is completed, we rename the proposed candidate keys list to be the candidate keys list of the corresponding relation.

E.g. Let STUDENT be the involved relation, and {NAME, ADDRESS} be the proposed candidate key.

```
SELECT count(*)  
FROM STUDENT  
GROUP BY NAME, ADDRESS  
HAVING count(*) > 1
```

If the result of running this query is a table with at least one tuple, then the proposed candidate key {NAME, ADDRESS} is not a candidate key and will be deleted from the proposed candidate key; otherwise, it is a candidate key.

The time needed for executing the proposed query will depend on the sorting technique used by the SQL server (in this project it is the Borland Database Engine) through which we execute that query. Any GROUP BY operator is implemented by sorting the relation involved according to the fields' values found in the GROUP BY clause.

Let n be the number of tuples in the involved relation. The time consumed by the worst sorting techniques, e.g. bubble sort, is $O(n^2)$; whereas the time consumed by the best sorting techniques, e.g. quick sort, is $O(n \log(n))$ [2].

3.4.8 Relations with Undetermined Candidate Key

Following the analysis done on anchors, and after the uniqueness validation test of candidate keys is performed, it is possible to end up with a relation having no candidate keys. In this situation, the user should decide on one of the following:

1. An automatic generation of all subsets of the relation's fields can be formed. Each of these subsets will be tested for the uniqueness property, as described in section 3.4.7. Any subset that passes the uniqueness test will be added to the list of candidate keys.

Let f be the number of fields in a certain relation, then the number of subsets will be $2^f - 1$.

Given the set $\{0,1,2\}$, then f will be 3.

All possible subsets will be: $\{0\}$, $\{1\}$, $\{2\}$, $\{0,1\}$, $\{0,2\}$, $\{1,2\}$, $\{0,1,2\}$.

The cardinality of the subset is 7, which is $2^3 - 1 = 7$.

So, in order to check for the uniqueness property, a time in $O(2^f n^2)$ will be consumed for worst sorting techniques, and $O(2^f n \log(n))$ for best sorting techniques. However, both cases involve a huge consumption of time.

2. A manual generation of the primary key can be formed. At this stage, the user has to interact with our system in order to propose the candidate key. The proposed key should be formed from the list of fields of the involved relation. Once the user proposes the key, we check for the uniqueness property, as described in section 3.4.7. In case the proposed key passes the test, we add it to the candidate key list; otherwise, the user will continuously propose different keys until the uniqueness test succeeds. The time taken for this action will be $O(n^2)$ for the worst sorting techniques, and $O(n \log(n))$ for the best sorting techniques.

The user will be responsible for the decision to be taken when a relation is found without a candidate key. If the user doesn't have any knowledge concerning the database application whose ERC+ diagram we are trying to extract, then he should choose the first option, automatic generation. In such a case, automatic key generation will save a lot of time. The wasted time would be spent by the user in forming, manually, a random primary key, as well as waiting for the uniqueness test to be completed. Such an act could be repeated many times without reaching any result. On the other hand, if the user has some knowledge of the database application, it is advisable to directly form the primary key using the manual key generation, thereby saving the time required by the automatic key generation.

3.4.9 Primary Key selection

Every relation should only have one primary key. So, if a relation has one candidate key in its candidate keys list, then this key is considered to be its primary key. However, if more than one candidate key is found, then the user, here again, has to

interact with our system to select one primary key out of the candidate keys list. Such a decision should be taken by a person having some knowledge about the database application whose ERC+ diagram we are attempting to extract. At this point, every node in the Connection Diagram will have its primary key determined.

With the information depicted in the Connection Diagram of figure 3.3, we should determine the primary key / object identifier (OId) of relations STUDENT and PERSON. Node PERSON has two anchors {name, address} and {tel}. The anchor {tel} participates in a cyclic link (link 3), hence {tel} can't be a key (as described in section 3.4.3). For this reason, {tel} is added to the non-key list of the node PERSON. The anchor {name, address} participates in two links (1 and 2), so we can add {name, address} to the list of candidate keys of node PERSON (as described in section 3.4.6). The second node in the Connection Diagram corresponds to the relation STUDENT. The anchor {parentname, parentaddress} participate in a cyclic link (link 4), thus {parentname, parentaddress} can't be a key (as described in section 3.4.3), and is added to the non-keys list of the node STUDENT. The anchor {name, address} is proposed to be a candidate key and is added to the proposed candidate keys list. This is true because when tuples from the same relation are compared (query 2, section 3.4.5), the anchor's {name, address} value is used to exclude tuples that match themselves (as described in section 3.4.6). To complete the information extraction stage, we have to check for uniqueness property. As mentioned in section 3.4.7, a test will be done for {name, address} of STUDENT and {name, address} of PERSON. Assume the test succeeded, hence both candidate keys will be announced as primary keys.

3.5 Determining Link Type

Any link between two anchors is a bi-directional link; i.e. if a link exists between two anchors X_{i_k} and Y_{j_l} of relations R_i and R_j , then the edge from X_{i_k} to Y_{j_l} and from Y_{j_l} to X_{i_k} should have a link type. The type of a link could be one of the following: “incoming”, “outgoing”, “undetermined” and “overlapping”. These link types are defined according to the following criteria:

An inclusion dependency $X_{i_k} \subseteq Y_{j_l}$ is defined on the anchor X_{i_k} of relation R_i and the anchor Y_{j_l} of relation R_j iff for every tuple in R_i where $X_{i_k} = v$, there is a tuple in R_j where $Y_{j_l} = v$. We call the edge from X_{i_k} to Y_{j_l} an “outgoing” link, and the anchor X_{i_k} will hold the “outgoing” link type. The edge from Y_{j_l} to X_{i_k} is called an “incoming” link, and the anchor Y_{j_l} will hold the “outgoing” link type.

If $X_{i_k} = Y_{j_l}$ ($X_{i_k} \subseteq Y_{j_l}$ and $Y_{j_l} \subseteq X_{i_k}$), then the edge from X_{i_k} to Y_{j_l} will have both “incoming” and “outgoing” link types. We name such a link as an “undetermined” link. The same applies for the edge from Y_{j_l} to X_{i_k} ; whereby it is also called “undetermined” link. Both X_{i_k} and Y_{j_l} hold the “undetermined” link type.

When $X_{i_k} - Y_{j_l} \neq \emptyset$ and $Y_{j_l} - X_{i_k} \neq \emptyset$ and $X_{i_k} \cap Y_{j_l} \neq \emptyset$, then the link is overlapping. In other words, if at least one tuple in R_i doesn't exist in R_j , if at least one tuple in R_j doesn't exist in R_i , and if at least one tuple in R_i exists in R_j , then we

call the link an “overlapping” link. Both X_{i_k} and Y_{j_l} will hold the “overlapping” property.

A referencing anchor is an anchor that is the source as of outgoing link. From the above definitions, we recognize that the database extension ought to be analyzed in order to determine the type of the link.

Figure 3.4: Connection Diagram with keys and inclusion dependencies

In figure 3.4, underlined anchors represent primary keys/object identifiers (OIds). Assume that after we analyzed the database extension of STUDENT and PERSON, we find out that there is an inclusion dependency between $\{name, address\}$ of STUDENT and $\{name, address\}$ of PERSON, link numbered 1. This means that the edge from $\{name, address\}$ of STUDENT to $\{name, address\}$ of PERSON is an “outgoing” link; therefore, anchor $\{name, address\}$ of STUDENT will hold the “outgoing” property. On the other hand, the two anchors $\{name, address\}$ of PERSON will hold the “incoming” property.

In order to determine the type of each link in the Connection Diagram, we should execute two queries. Each query will determine the type of each link direction. If a

link exists between X_{i_k} of R_i and Y_{j_l} of R_j , then the first query will test the inclusion dependency of the edge from X_{i_k} to Y_{j_l} . Based on the output of this query, X_{i_k} and Y_{j_l} anchors will hold the appropriate link type. The second query will test for the inclusion dependency of the edge from Y_{j_l} to X_{i_k} . Also according to the output of the query, X_{i_k} and Y_{j_l} anchors' link types will be updated. After the execution of both queries, an anchor can hold either the "incoming", "outgoing", "undetermined" or "null" property. Null indicates that the type of link is not determined yet.

Let T1 and T2 represent relations name, and A1 and A2 represent anchors participating in a link.

Query 1:

```

SELECT count(*) as C FROM <T1>
WHERE EXISTS (SELECT "" FROM <T2>
WHERE <T2.A2 = T1.A1>)
  
```

The interpretation of this query is as follows:

For every tuple in relation T1, the nested "select" statement will be executed. If a tuple in relation T2 whose A2 attributes' values are the same as A1 attributes' values, then the current tuple of relation T1 will be represented in the resulting table of the query by incrementing the count.

If the count value in the table resulting from executing query 1 is the same as the number of tuples in T1, then there is an inclusion dependency from A1 of T1 to A2 of T2. A1 holds the “outgoing” link property and A2 holds the “incoming” link property. If the count value in the table resulting from executing query 1 is less than the number of tuples in T1, then, at this point, the type of such a link ($A1 \rightarrow A2$) can not be determined (the null property).

Query 2:

```
SELECT count(*) as c FROM <T2>
      WHERE EXISTS (SELECT "" FROM <T1>
                    WHERE <T1.A1 = T2.A2>)
```

Query 2 is executed after query 1. If the count value in the table resulting from executing query 2 is the same as the number of tuples in T2, then there is an inclusion dependency from A2 of T2 to A1 of T1. If the link from A1 to A2 possesses the inclusion dependency from query 1, then this link is an “undetermined” link. If the count value in the table resulting from executing query 2 is less than the number of tuples in T2, then, at this point, the type of such a link ($A2 \rightarrow A1$) can not be determined (the null property).

When this case occurs (null property), we check for the count value in resulting table of either queries (1 or 2), and if this value is not zero, we set the link property to be an “overlapping” link. In case the count value in the resulting table is zero, we delete such a link for it doesn’t hold any necessary information, as reflected by the database extension.

Let n be the number of tuples in a table. The times needed to execute a simple non-join query is in $O(n)$. Each of queries 1 and 2 will consume time in $O(n^2)$.

3.6 Deleting Non-Structural Links

Queries can exist including comparisons of attributes that don't represent references. In general, these links should be removed for they don't hold any structural information. For this purpose we define structural links as follows:

A structural link $L(X_{i_k}, Y_{j_l})$ is any link where at least one of the anchors is a primary key/object identifier (Old).

Figure 3.5: Connection Diagram with links 3 and 4 classified as non-structural

According to the definition of *Structural Links*, links 3 and 4 in figure 3.5 are classified as non-structural and in the diagram in figure 3.6, they are removed.

Figure 3.6: Connection Diagram with only structural links

In some cases, the criterion in the definition of structural links proves too strong; i.e. some links that contain structural information are classified as non-structural links.

Consider figure 3.8 with the Connection Diagram resulting from the queries in figure 3.7.

```

SELECT DISTINCT carno
  FROM WORKS, BUILD
 WHERE eno=1001 and
        WORKS.depno=BUILD.depno

SELECT depno
  FROM DEPARTMENT, BUILD
 WHERE BUILD.depno=DEPARTMENT.depno and
        Carno=76

SELECT eno
  FROM EMPLOYEE, WORKS
 WHERE WORKS.eno=EMPLOYEE.eno and
        Depno=2
  
```

Figure 3.7: Example queries of an application

Figure 3.8: Connection Diagram for queries in figure 3.7

Link 2 in figure 3.8 is a non-structural link according to the definition of structural links. However, since the target of link 2, BUILD.depno, is linked to DEPARTMENT.depno, the transitive property of the inclusion dependency can be used to defer the non-structural link DEPARTMENT.depno and create link 4 as in figure 3.9. Link 2 is removed. If the inclusion dependency of link 3 has been in the opposite direction, link 2 couldn't have been recovered to link 4.

Figure 3.9: Connection Diagram with an induced link 4

To check whether or not we can recover a non-structural link to a structural link, we can apply a transitive-link procedure. This procedure aims at finding a chain of links such that the target anchor of one link is the source anchor of the next link in the chain. All links in the chain should have an outgoing or undetermined link. Once the target anchor is found to be an Old, we terminate the transitive-link procedure and we create a new link. The new induced link has its source anchor to be the source anchor of the first link in the chain, and the target anchor will be the target anchor of the last link in the chain. The link type of the new induced link will be “undetermined” if all links in the chain have an “undetermined” link type. If one of the links in the chain has an outgoing link, then the link type of this new induced link will be “outgoing”. On the other hand, if the transitive-link procedure reached a dead end without having the target anchor of the last link in the chain as an Old, then we announce the first link in the chain to be a non-structural link; thus, deleting it from the Connection Diagram. We reach a dead end situation because the transitive-link procedure insists at forming a non-cyclic chain.

To implement such a transitive-link procedure, we have to use the Breadth-First Search technique. Breadth-First Search algorithm tries to find the next link in the chain such that its target anchor is an Old. In other words, if the source anchor of the new link to be formed participates in many links (sibling links), we select the link whose target anchor is an Old. If the target anchors of all sibling links don't end with an Old, then the Breadth-First Search algorithm will select in turn these siblings to be the new next link. It is important to mention that it is possible to recover a non-structural link to one or more structural links. The Breadth Search First technique will produce an induced link from the chain with the least number of links.

Chapter 4

Translation Rules

Now that we have completed the Information Extraction step, it is time to apply some translation rules to the Connection Diagram. The Connection Diagram is translated to ERC+ diagram by applying a set of Translation rules divided into three groups (figure 4.1).

Figure 4.1: data flow chart for Translation Rules

1. Node rules produce translations of nodes in the Connection Diagram based on correspondences between keys and anchors in the relation scheme corresponding to the node.
2. Link rules draw conclusions about links with respect to their nodes.
3. Refinement rules apply to the result from the two previous groups to generate the final translation.

The node and link rules create intermediate types that are translated by refinement rules. Every join condition induces a link and every structural link is introduced in the ERC+ scheme, which means that the rules are complete in the sense that they capture the application semantics as seen by the query programmer.

4.1 Node Rules

The node rules use the id-independency property of the object types represented in the relation schemes to generate solution. Id-independency is determined by an analysis of correspondences between the referencing anchors and the key of an object type.

A primary key K is id-independent iff no referencing anchor (anchor with an “outgoing” property) defined on a proper subset of attributes in K is an Old in another relation scheme. An object type is id-independent iff the key that represents its Old is id-independent. A key or an object type that is not id-independent is id-dependent.

To be an id-independent, one of the following cases should occur:

- (1) When there are no referencing anchors.
- (2) When all referencing anchors are defined on the entire Old of the object type.
- (3) When all referencing anchors are defined on attributes not included in the Old.
- (4) When all referencing anchors are defined on both the entire Old and on attributes not included in the Old.

Figure 4.2: A Connection Diagram

Consider the Connection Diagram in figure 4.2. This Connection Diagram has two nodes. These nodes represent STUDENT and PERSON objects. The key {name, address} of relation PERSON is an id-independent because no referencing anchors exist. Since the key is id-independent, PERSON is id-independent. The key {name, address} of relation STUDENT is also id-independent because there is no referencing anchor defined on the proper subset of attributes in {name, address}. Since the key {name, address} is id-independent, STUDENT is id-independent.

Figure 4.3: Another Connection Diagram

Consider the Connection Diagram in figure 4.3. The Connection Diagram has four nodes. These nodes represent EMPLOYEE, WORKS, BUILD and DEPARTMENT objects. EMPLOYEE is id-independent because no referencing anchor exists. WORKS is id-dependent since the referencing anchor {eno} is a proper subset of the key {eno, depno}. BUILD is id-dependent for the referencing anchor {depno} is a proper subset of the key {depno, carno}. DEPARTMENT is id-independent because no referencing anchor exists.

Node Rule 1

An id-independent object type is translated to *entity* type.

Node Rule 2

An id-dependent object type is translated to a *relationship* type if it has more than one referencing anchor with attributes that form a proper subset of the attributes of the OId of the object type.

Node Rule 3

An id-dependent object type is translated to a *dependent* type when there is only one referencing anchor with attributes that form a proper subset of the attributes of the OId of the object type.

Dependent type is introduced as a generalization of a multi-valued attribute and weak entity type. A weak entity type can be distinguished from a multi-valued attribute as it is either linked to a relationship or it is involved in a multi-instantiation. This is handled by a refinement rule in section 4.3.

<i>Node Rule</i>	<i>Object Type Before Translation</i>	<i>Condition</i>	<i>Object Type after Translation</i>
1	Id-independent	No referencing anchor defined on OId	Entity type
2	Id-dependent	More than one referencing anchor defined on OId	Relationship type
3	Id-dependent	One referencing anchor defined on OId	Dependent type

Figure 4.4: Table defining the translation of nodes

Consider the Connection Diagram in figure 4.2 with its two relations. Since the two relations STUDENT and PERSON are id-independent, these relations are translated to entity types (Node Rule 1).

The Connection Diagram shown in figure 4.3, has four relations. EMPLOYEE and DEPARTMENT are id-independent, so they are translated to entity types (Node Rule 1). WORKS is id-dependent with two referencing anchors, {eno} and {depno}, defined on proper subset of its key {eno, depno}, so WORKS is translated to

relationship type (Node Rule 2). BUILD is id-dependent with one referencing {depno} anchor defined on a proper subset of its Old {depno, carno}; hence, it is translated to a dependent type (Node Rule 3).

4.2 Link Rules

The link rules determine the meaning of the links in the Connection Diagram with respect to properties of the anchors. We introduce the concept of property as generalization characteristics to be used as an intermediate type during translation.

Link Rule 1

If there is a link L between X_{i_k} in table R_i and Y_{j_l} in table R_j such that X_{i_k} is not a key in R_i , and Y_{j_l} is a key in relation R_j , and there is an inclusion dependency $X_{i_k} \subseteq Y_{j_l}$, then L is a “property” link.

Link Rule 2

A link between two primary keys indicates a multi-instantiation link. The inclusion dependency of the link determines the type of multi-instantiation (“isa” or “maybe” link type).

1. If the link has an inclusion dependency, then it is translated to an “isa” link.
2. If the link is overlapping, then it is translated to a “maybe” link.
3. If the link is undetermined, then it is translated to a “one-to-one” relationship type (cardinality is 1:1).

<i>Link Rule</i>	<i>Number of OIds</i>	<i>Link Type Before translation</i>	<i>Link Type After Translation</i>
<i>1</i>	One Old		Property type
<i>2</i>	Two OIds	Inclusion Dependency	isa type
<i>"</i>	<i>"</i>	Overlapping	maybe type
<i>"</i>	<i>"</i>	Undetermined	1-1 type

Figure 4.5: Table defining the translation of links

Consider again the Connection Diagram in figure 4.2. The referencing anchor (link 2) defined from the anchor {parentname, parentaddress} of relation STUDENT to {name, address} of relation PERSON is a “property” link. This is true because {parentname, parentaddress} of relation STUDENT is not an Old, while {name, address} of relation PERSON is an Old (Link Rule 1). Link 1 defined from {name, address} of relation STUDENT to {name, address} of relation PERSON is a multi-instantiation link because both anchors are OIds. Since the anchor {name, address} of STUDENT has an “outgoing” property (inclusion dependency), the link 1 is translated to an “isa” link type (Link Rule 2).

Links 1, 3 and 4 of the Connection Diagram in figure 4.3 are “property” links (Link Rule 1).

4.3 Refinement Rules

The refinement rules combine the information given by the two previous sets of rules to produce the final translation of the Connection Diagram.

Any property link $L(X_{i_k}, Y_{j_l})$ will have a cardinality of 1:1 for the edge from X_{i_k} to Y_{j_l} . If L has an “outgoing” property, then the edge from Y_{j_l} to X_{i_k} will have a cardinality of 0:n. If L has an “undetermined” property, then the edge from Y_{j_l} to X_{i_k} will have a cardinality of 1:n.

Refinement Rule 1

A property link $L(X_{i_k}, Y_{j_l})$ defined on two different anchors in the same table induces a relationship type. The table is translated to an entity type.

Figure 4.6: Example of Connection Diagram

Figure 4.7: ERC+ diagram when L has “undetermined” or “outgoing” property for figure 4.6

Any property link can have either an “outgoing” or “undetermined” property embedded in X_{i_k} . If X_{i_k} has an “outgoing” or “undetermined” property, then the cardinality of the induced link defined from X_{i_k} to the induced relation is 1:1. When X_{i_k} has an “outgoing” or “undetermined” property, then the cardinality of the induced link defined from Y_{j_l} to the induced relation is 0:n or 1:n correspondingly (figure 4.7).

Refinement Rule 2

A property shared between relation schemes translated to entity types induces a new entity type carrying the common property, and with a super type relation to the entity types sharing the property.

Figure 4.8: Shared property ENGINE

Consider the example in figure 4.8 that illustrates refinement rule 2. The diagram includes three relation schemes where the values of ENGINE.engineno are distributed over MOTORCYCLE.engineno and CAR.engineno.

In figure 4.9, a new entity type *G* has been created that carries the property *engine*, and that is a generic type of *MOTORCYCLE* and *CAR*. This means that both links 1 and 2, in figure 4.8, will be redirected so that they will be an “isa” link toward the induced generic entity type *G*. Object *ENGINE* will become an attribute to the generic entity type *G* with a cardinality 1:1 on both directions.

*Figure 4.9: Generic type *G* with shared property *ENGINE**

Refinement Rule 3

A dependent link, which is linked to a relationship type or by a multi-instantiation (“isa” or “maybe”) link, is a weak entity type; otherwise it is a multivalued attribute. Weak entities are entities in the sense that they represent objects in the real world, yet they are either dependent to other entities or they are a super type of other entities. Multivalued attributes represent attributes such that different values are associated with other attributes.

Consider the example in figure 4.10. *T1* is a dependent object for it has a dependent key {*A1*, *A2*, *A3*} (see section 4.1).

Assume that T2 is a relationship type. Refinement rule 3 applies to this case for a dependent (T1) is linked to a relationship type (T2). Therefore, T1 is translated to a weak entity type.

Figure 4.10: Connection Diagram with a dependent and relationship type

Figure 4.11: ERC+ diagram where L has "outgoing" or "undetermined" property for the Connection Diagram in figure 4.10

As shown in figure 4.11, the cardinality of the edge from $\{A1, A2\}$ to $\{B1, B2\}$ is 1:1, while the edge from $\{B1, B2\}$ to $\{A1, A2\}$ is 0:n if L has an "outgoing" property or 1:n if L has an "undetermined" property.

Consider the Connection Diagram in figure 4.12. T1 is dependent for its Old $\{A1, A2\}$ is id-dependent (see section 4.1). Let T2 be an entity type with $\{B1\}$ as its Old.

Since T1 is not linked to a relation type or by a multi-instantiation link (“isa” or “maybe”), it is translated to a multivalued attribute.

Figure 4.12: Connection Diagram with dependent and entity type

Figure 4.13: ERC+ diagram where L has “outgoing” or “undetermined” property for the Connection Diagram in figure 4.12

As shown in figure 4.13, the cardinality of the edge from entity T2 to attribute T1 is 0:n if L has an “outgoing” property, or 1:n if L has an “undetermined” property.

Refinement Rule 4

A property link between an entity type and a relationship type is translated to a role.

Consider the Connection Diagram in figure 4.14. Let T1 be a relationship and T2 an entity type. L1 is a property link defined from {A1} (part of T1’s Old) to {B1} (Old of relation T2), so it is translated to a role (figure 4.15) (refinement rule4).

Figure 4.14: Connection Diagram with relationship and entity type objects

Figure 4.15: ERC+ diagram where L has “outgoing” or “undetermined” property for the Connection Diagram in figure 4.14

The edge from T2 to T1 will have a cardinality of 0:n if L1 has an “outgoing” property or 1:n if it has an “undetermined” property. The edge from T1 to T2 has a cardinality of 1:1.

Refinement Rule 5

A property between entities induces a new relationship type.

Consider the Connection Diagram in figure 4.16.

T1 and T2 are entities with $\{A1\}$ and $\{B1\}$ as their Olds correspondingly. L is a property link defined from the anchor $\{B2\}$ to $\{A1\}$.

Figure 4.16: Connection Diagram with two entity objects

Figure 4.17: ERC+ diagram where L has “outgoing” or “undetermined” property for the Connection Diagram in figure 4.16

According to the refinement rule 5, a relation $R1$ is induced with two links $L1$ and $L2$. $L1$ is a 1-1 link (figure 4.17). $L2$ is defined from $T1$ to the induced relationship with a cardinality of 0:n if L has an “outgoing” property or 1:n if it has an “undetermined” property.

Refinement rule 5 is the last rule to be applied to a Connection Diagram. Having applied all translation rules, the Connection Diagram will hold all necessary information for the ERC+ diagram. In constructing the ERC+ diagram, one has to pay attention to the following remarks:

1. All attributes will be attached (1:1 cardinality) to its corresponding entity E1(A1, A2) (figure 4.18) except when the entity E1 participates in an “isa” link (figure 4.20) with another entity E2(B1, B2).

Figure 4.18: Entity type with no “isa” link

Figure 4.19: Connection Diagram with an “isa” link

E1 will hold all its attributes except those attributes of the source anchor of the “isa” link L. The reason behind this is that an “isa” link represents an inheritance. Hence, E1 will inherit all attributes of entity E2.

Figure 4.20: ERC+ diagram showing attributes with an “isa” link for Connection Diagram in figure 4.19

2. Relationship, multivalued attribute and weak entity types will be attached by all of their attributes except those composing the source anchors of the entire links

(figure 4.22). This is because attributes of a join condition show the relationship between two objects rather than the characteristics of the object itself.

Figure 4.21: Connection Diagram with a relationship and two entity objects

Attributes B1 and B2 of relationship T2 are not shown on the ERC+ diagram since they are the source anchors of links L1 and L2.

Figure 4.22: ERC+ diagram for the Connection Diagram in figure 4.21

Chapter 5

Experimental Results

In this chapter we present the results of applying our method to some arbitrary database applications. The results are twinned with detailed explanation of the steps through which the extraction method passes. Experiments are done on a Pentium 200 MHz, 32 RAM microcomputer. The implementation is done using the Delphi visual language (version 3.0).

5.1 Example 1

The first test case to be studied consists of the following source code listing:

```
SELECT STUDENT.name, STUDENT.address
FROM STUDENT, PERSON P1, PERSON P2
WHERE STUDENT.name = P1.name
AND STUDENT.address = P1.address
AND STUDENT.parentname = P2.name
AND STUDENT.parentaddress = P2.address
AND P2.tel = P1.tel
```

Query 1

```
SELECT DISTINCT s1.name, s1.address  
  
FROM STUDENT s1, STUDENT s2  
  
WHERE s1.parentname = s2.parentname AND  
  
s1.parentaddress = s2.parentaddress AND  
  
NOT (s1.name = s2.name AND s1.address = s2.address)
```

Query 2

The database for *example 1* will contain two relations PERSON (name, address, tel) and STUDENT (name, address, parentname, parentaddress). The database is of type MS ACCESS (data.mdb), Appendix C.

As in section 3.1 (Table Extraction), the tool will access the database and create the corresponding relation list. Relation list will have two nodes representing relations PERSON and STUDENT. Relation PERSON will be associated with the attributes “name”, “address” and “tel”, whereas relation STUDENT will be associated with the attributes “name”, “address”, “parentname” and “parentaddress”.

The next step is to extract the “select” statements from the source code, as in section 3.2 (Extracting Select Statements). The method will detect two queries and create the corresponding query list.

Then, the Connection Diagram should be created, as in section 3.3 (Creation of Connection Diagram), with the help of the relation list and query list. Since the relation list has two tables, then the Connection Diagram will be composed of two nodes, one for relation PERSON and the other for STUDENT. Query 1 has three join

conditions, so that each of these joins will be represented by a link (1,2 and 3) in the Connection Diagram, figure 5.1.

These join conditions are:

1. Student.name = P1.name AND Student.address = P1.address
2. Student.parentname = P2.name AND Student.parentaddress = P2.address
3. P2.tel = P1.tel

(P1 and P2 are two aliases for relation PERSON)

Figure 5.1: Connection Diagram for Example 1

Similarly for query 2, it has one join condition, thus representing link 4.

4. s1.parentname = s2.parentname AND s1.parentaddress = s2.parentaddress

(S1 and S2 are two aliases of relation STUDENT)

The Connection Diagram in figure 5.1 has four links, two of which are cyclic links (3 and 4).

After creating the Connection Diagram, we have to find the primary key of the relation PERSON, as well as the relation STUDENT, as in section 3.4 (Drawing

Conclusions about Keys). This stage requires many checking steps to be done for both relations PERSON and STUDENT, yet some of these steps can propose candidate keys.

Since link 3 is a cyclic link, the anchor {tel} can't be a candidate key, thereby it is added to the list of non-keys of relation PERSON as in section 3.4.3 (Anchors with Cyclic Join Links). The second anchor for relation PERSON, {name, address}, is proposed to be a candidate key since it participates in two different links with the relation STUDENT as in section 3.4.6 (Candidate Key Deduction). The anchor {name, address} is placed on the candidate keys list of relation PERSON.

For relation STUDENT the anchor {parentname, parentaddress} is placed on the non-keys list for it participates in the cyclic link 4 in figure 5.1. The second anchor {name, address} is added to the list of candidate keys of relation STUDENT for its values are used to exclude tuples that match themselves when tuples from the same relation STUDENT are compared, as in section 3.4.5 (Elimination by Candidate Key).

When all checking tests are done, we end up with relation PERSON having one candidate key {name, address}. The relation STUDENT has also one candidate key {name, address}. To prove whether or not the proposed candidate key for relations PERSON and STUDENT are really candidate keys, we have to do the candidate key uniqueness test as in section 3.4.7. Uniqueness test for {name, address} of relation PERSON is done by executing the following query:


```
SELECT count(*)  
  
FROM PERSON  
  
GROUP BY name, address  
  
HAVING count(*) > 1
```

As per the database extension, figure C.1 and C.2, the result of this query turned out to be a table with no tuples. This means that the proposed candidate key is really so for the relation PERSON. Since only one candidate key {name, address} exists, it is announced to be the primary key/Old for the relation PERSON.

Similarly, the following query has been executed to check if {name, address} is a candidate key or not for the STUDENT relation.

```
SELECT count(*)  
  
FROM STUDENT  
  
GROUP BY name, address  
  
HAVING count(*) > 1
```

The resulting table also has no tuples, hence {name, address} is a candidate key for relation STUDENT. Since one candidate key exists, {name, address} is the primary key/Old of the relation STUDENT.

The next step is to determine the link type of links (1 and 2) in the Connection Diagram of figure 5.1. Note that we will not find the link type of links 3 and 4 since they are cyclic links and will be deleted for being non-structural links.

For link 1:

To determine the link type of the edge defined from {name, address} of relation STUDENT to {name, address} of relation PERSON, we execute the following two queries.

```
SELECT count(*) as c FROM STUDENT
      WHERE EXISTS (SELECT "" FROM PERSON
                    WHERE PERSON.name = STUDENT.name
                    AND PERSON.address = STUDENT.address)
```

The resulting table upon executing the query above has a count value equal to that of relation STUDENT. This means that there is an inclusion dependency, thereby {name, address} of relation STUDENT has an “outgoing” property, while {name, address} of PERSON has an “incoming” property. The second query determined the type of link for the edge defined from {name, address} of relation PERSON to {name, address} of relation STUDENT.

```
SELECT count(*) as c FROM PERSON
      WHERE EXISTS (SELECT "" FROM STUDENT
                    WHERE STUDENT.name = PERSON.name
                    AND STUDENT.address = PERSON.address)
```

The table resulting from the execution of the above query has a count value less than that of relation PERSON. Therefore, no inclusion dependency exists from {name, address} of relation PERSON to {name, address} of relation STUDENT.

Conclusion about link 1:

1. $\{name, address\}$ of relation *PERSON* has an “incoming” property
2. $\{name, address\}$ of relation *STUDENT* has “outgoing” property

For link 2:

To determine the link type of link 2, also two queries are to be executed.

```
SELECT count(*) FROM STUDENT
      WHERE EXISTS (SELECT "" FROM PERSON
                    WHERE PERSON.name = STUDENT.parentname
                    AND PERSON.address = STUDENT.parentaddress)
```

The query listed above determines the type of the edge defined from $\{parentname, parentaddress\}$ of relation *STUDENT* to $\{name, address\}$ of relation *PERSON*. The count value in the resulting table is the same as that of relation *STUDENT*. Therefore, $\{parentname, parentaddress\}$ has an “outgoing” property and $\{name, address\}$ has an “incoming” property.

The next query will determine the type of link for the edge defined from $\{name, address\}$ of relation *PERSON* to $\{parentname, parentaddress\}$ of relation *STUDENT*.

```
SELECT count(*) FROM PERSON
      WHERE EXISTS (SELECT "" FROM STUDENT
                    WHERE STUDENT.parentname = PERSON.name
                    AND STUDENT.parentaddress = PERSON.address)
```

The count value resulting from executing the above query is less than the number of tuples in relation PERSON. Therefore, no inclusion dependency exists between $\{\text{name, address}\}$ of relation PERSON and $\{\text{parentname, parentaddress}\}$ of relation STUDENT.

Conclusion about link 2:

1. $\{\text{name, address}\}$ of relation PERSON has an “incoming” property
2. $\{\text{parentname, parentaddress}\}$ of relation STUDENT has “outgoing” property

According to section 3.6 (Deleting Non-Structural Links), links 3 and 4 are considered non-structural links because $\{\text{tel}\}$ and $\{\text{parentname, parentaddress}\}$ are not OId’s of relation PERSON (in link 3) and relation STUDENT (in link 4) correspondingly. Links 1 and 2 are considered structural links because $\{\text{name, address}\}$ of relation PERSON is a primary key/OId.

Figure 5.2: Connection Diagram with only structural links for Example 1

Figure 5.2 shows $\{\text{name, address}\}$ of relation PERSON and $\{\text{name, address}\}$ of relation STUDENT as their respective OIds. Anchors $\{\text{name, address}\}$ and $\{\text{parentname, parentaddress}\}$ of relation STUDENT are referencing anchors.

In the final stage we apply the translation rules described in chapter 4 to the Connection Diagram in figure 5.2. The Old {name, address} of relation PERSON is id-independent, hence relation PERSON is translated to entity type (Node Rule 1). Similarly for relation STUDENT, it is translated to an entity type since its Old is id-independent (Node Rule 1).

Link 1, in figure 5.2, defined from {name, address} of relation STUDENT to {name, address} of relation PERSON is an “isa” link type because both of its anchors are Old and an inclusion dependency exists on the link (Link Rule 2). Link 1 defined from {parentname, parentaddress} of relation STUDENT to {name, address} of relation PERSON is a property link, since the target anchor {name, address} of relation PERSON is Old (Link Rule 1).

According to the refinement rule 5, relationship R1 is induced because we have a property link between two entities. Link 2 has an “outgoing” property, so the edge from PERSON to R1 is 0:n, whereby the cardinality of the edge from STUDENT to R1 is 1:1.

Figure 5.3 shows the corresponding ERC+ diagram for *Example 1*.

Figure 5.3: ERC+ diagram for example 1

The textual tree representation displayed by our tool for *example 1* is shown in figure 5.4.

Figure 5.4: ERC+ as displayed by our tool for Example 1

The textual tree representation printed by our tool for *example 1* is shown in figure 5.5.

Figure 5.5: ERC+ diagram as printed by our tool for Example 1

5.2 Example 2

The second test case to be studied consists of the following source code listing:

```

SELECT DISTINCT carno
FROM works, build
WHERE eno = 10144
AND works.depno = build.depno

```

Query 1

```
SELECT depno  
  
FROM department, build  
  
WHERE build.depno = department.depno  
  
AND carno = 76
```

Query 2

```
SELECT eno  
  
FROM works, employee  
  
WHERE works.eno = employee.eno  
  
AND depno = 2
```

Query 3

The database for Example 2 consists of the following four relations, Appendix C:

EMPLOYEE (eno, birthdate)

WORKS (eno, depno)

BUILD (depno, carno)

DEPARTMENT (depno, building)

The final form of the Connection Diagram produced by our system after applying all stages of our extraction method is shown in figure 5.6.

Figure 5.6: Connection Diagram for Example 2

Figure 5.7 shows the ERC+ diagram for Example 2.

Figure 5.7: ERC+ diagram for Example 2

The textual tree representation displayed by our tool for *example 2* is shown in figure 5.8 and figure 5.9. Figure 5.8 shows the ERC+ for the entity **EMPLOYEE** as displayed by our tool.

Figure 5.8: ERC+ for the entity **EMPLOYEE** of Example 2

Figure 5.9 shows the ERC+ for the entity DEPARTMENT as displayed by our tool.

Figure 5.9: ERC+ for the entity DEPARTMENT of Example 2

The textual tree representation printed by our tool for *example 2* is shown in figure 5.10.

```
Entity Relationship Complex +
+-EMPLOYEE
| +-0:n R: WORKS (eno = eno)
| | +-1:1 E: DEPARTMENT (depno = depno)
| | | +-0:n A: BUILD (depno = depno)
| | | | +-1:1 A: CARNO
| | | | +-1:1 A: DEPNO
| | | | +-1:1 A: BUILDING
| | | | +-1:1 A: DEPNO
| | +-1:1 A: DEPNO
| | +-1:1 A: ENO
| +-1:1 A: BIRTHDATE
| +-1:1 A: ENO
+-DEPARTMENT
+-0:n R: WORKS (depno = depno)
| +-1:1 E: EMPLOYEE (eno = eno)
| | +-1:1 A: BIRTHDATE
| | +-1:1 A: ENO
| +-1:1 A: DEPNO
| +-1:1 A: ENO
+-0:n A: BUILD (depno = depno)
| +-1:1 A: CARNO
| +-1:1 A: DEPNO
+-1:1 A: BUILDING
+-1:1 A: DEPNO
```

Figure 5.10: ERC+ diagram as printed by our tool for Example 2

5.3 Execution Time

In this section we will concentrate on the execution time of an arbitrary database application rather than on the analysis of the execution.

The database application consists of 700 queries. These queries are duplicates of the following two queries:

Query 1

```
SELECT STUDENT.name, STUDENT.address
FROM STUDENT, PERSON P1, PERSON P2
WHERE STUDENT.name = P1.name
AND STUDENT.address = P1.address
AND STUDENT.parentname = P2.name
AND STUDENT.parentaddress = P2.address
AND P2.tel = P1.tel
```

Query 2

```
SELECT DISTINCT s1.name, s1.address
FROM STUDENT s1, STUDENT s2
WHERE s1.parentname = s2.parentname AND
s1.parentaddress = s2.parentaddress AND
NOT (s1.name = s2.name AND s1.address = s2.address)
```

The database contains two tables: PERSON and STUDENT. Relation PERSON consists of 1,000 tuples, figure C.2, while relation STUDENT consists of 500 tuples, figure C.1.

Our tool will take only 75 seconds in order to produce the ERC+ diagram, figure 5.4, for the corresponding database application.

Chapter 6

Conclusion and Further Work

A method for extracting a conceptual schema of a relational database has been presented so far in this paper. The method is implemented using Reverse Engineering technique and is based on an extended version of Andersson's proposal.

The method is based on the analysis of the database extension as well as the analysis of data manipulation statements, the SQL statement, of a database application. Semantically related attributes, as well as keys, are determined through an analysis of the data manipulating statements. The extracted information is represented in a Connection Diagram.

The analysis of the database extension will determine whether the proposed key is actually a primary key or not. Also it helps in determining the inclusion dependencies between tables. The Connection Diagram is subject to a number of operations before it reaches its final form. These operations include applying translation rules to the Connection Diagram and hence, the Connection Diagram will hold all necessary information enabling us to draw the corresponding Entity Relationship Complex Plus diagram.

Our further development of Andersson's proposal for the implementation of his extraction method, a tool is produced to assist software engineers in their process of reengineering a database application. The application accepts the source code in a

text file format and the database can be in one of the following formats: Access, Foxpro, Dbase, Paradox and Interbase. The tool can be extended to accept any new database formats. The output of our application is a textual tree representation of the ERC+ diagram, which can be printed or saved for later work.

We propose (as further development) to twin the technique developed here with data mining algorithms that exploit our passes over the database queries/contents to elicit interesting attribute correlation. Also, it is possible to add to our package a tool that designs conceptual, fully normalized, relational databases from a flat representation (first normal form) that can support system implementation. Further work can also include a graphical representation of the ERC+ diagram produced by our package.

The work completed here is provisioned to constitute an integral part of a larger package for the development and engineering of database application. The package is intended for use in research and education at LAU.

References

1. Andersson, M. (1994). Extracting an entity relationship schema from a relational database through reverse engineering. In P. Loucopoulos (ed.), Entity-relationship approach-ER'94, Business Modeling and Re-engineering (pp. 403-419). Manchester, U.K.
2. Brassard, G., Bratley, P. (1996). Fundamentals of algorithmic. New Jersey: Prentice-Hall.
3. Coad, P., Yourdan, S. (1991). Object-oriented analysis (2nd ed.). New Jersey: Prentice-Hall.
4. Elmasri, R., Navathe, S. (1994). Fundamentals of database systems (2nd ed.). California: the Benjamin/Cummings Publishing Company.
5. Fisher, A. (1991). Case: using software development tools (2nd ed.). New York: John Wiley and Sons.
6. Goldstein, R., Storey, V. (1990). Some findings on the intuitiveness of entity-relationship construct. In F. Lochovsky (ed.), Entity relationship approach to database design and querying (pp. 9-23). New York: Elsevier Science Publishing Company.
7. Korth, H., Silberschatz, A. (1991). Database systems concepts (2nd ed.). New York: McGraw-Hill.
8. Markowitz, V. Shoshani, A. (1990). Name assignment technique for relationship schemas representing extended entity-relationship schemas. In F. Lochovsky (ed.), Entity relationship approach to database design and querying (pp. 75-93). New York: Elsevier Science Publishing Company.

9. Parent, C., Rolin, H., Yetongnon, K., Spaccapietra (1989). An ER calculus for the entity-relationship complex model. In F. Lochovsky (ed.), Entity relationship approach to database design and querying (pp. 361-384). New York: Elsevier Science Publishing Company.
10. Parent, C., Spaccapietra, S. (1994). About entities, complex objects and objected-oriented data models. INRIA, "Programme de Recherches Coordonnées de troisième génération" (PRC BD3).

Wysiwyg:\\94\\http://lbdwww.epfl.ch/e/publications/
11. Pressman, R. (1997). Software Engineering: A practitioner's approach (4th ed.). New York: McGraw-Hill.
12. Spaccapietra, S., Parent, C. (1992). ERC+: An object based entity relationship approach. In P. Loucopoulos, R. Zicari, Conceptual modeling, database and case: An integrated view of information systems development. John Wiley.
13. Tucherman, L. Gualandi, P. (1990). A proposal for formalizing and extending the generalization and subset abstractions in the entity-relationship model. In F. Lochovsky (ed.), Entity relationship approach to database design and querying (pp. 27-41). New York: Elsevier Science Publishing Company.
14. Yeates, D., Shields, M. Helmy, D. (1994). System analysis and design. London: Pitman Publishing.

Appendix A

Creating Subsets of a Set

If set $A = \{0, 1, 2\}$, then

the set of all subsets is $G = \{ \{0\}, \{1\}, \{2\}, \{0, 1\}, \{0, 2\}, \{1, 2\}, \{0, 1, 2\} \}$

Given $A = \{a_1, a_2, \dots, a_i, \dots, a_{n-1}, a_n\}$, the algorithm for finding all subsets of any set G is as follows:

- Create an empty set G
- Form a list L consisting of the following sets $\{a_1\}, \{a_2\}, \dots, \{a_i\}, \dots, \{a_{n-1}\}$ and add the set $\{a_n\}$ to G
- While the list L is not empty
 - Remove a set S from the list L
 - Create a set B such that $B = A - S$
 - For every element b in B
 - Create a set P such that $P = S + \{b\}$
 - If $a_n \in P$ then add P to G
 - else add P to L
 - Add S to L

The number of subsets in G will be $2^n - 1$, hence the time taken by the above algorithm is in $O(2^n)$

Appendix B

Breadth-First Search Technique

Breadth-First search technique checks all paths of a given length before moving on to any longer paths.

To conduct a breadth-first search, one should apply the following algorithm:

- Form a one-element queue consisting of a zero length path that contains only the root node
- Until the first path in the queue terminates at the goal node or the queue is empty
 - Remove the first path in the queue
 - Create new paths by extending the first path to all the neighbors of the terminal node
 - Reject all new paths with loops
 - Add all new paths, if any, to the back of the queue
- If the queue node is found, announce success; otherwise, announce failure

In our extraction method, the goal node is the anchor that is an Old in its corresponding relation.

If we have n nodes and a branches in a certain graph, then the time needed, by breadth-first search algorithm, to find a path ending with the goal node is in $\theta(\max(a, n))$.