Relational Database Reverse Engineering using the Rigi System

by

Ming Du

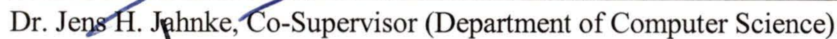
A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming to the requested standard

Dr. Hausi A. Müller, Co-Supervisor (Department of Computer Science)



Dr. Jens H. Jahnke, Co-Supervisor (Department of Computer Science)

Dr. Daniel M. German, Departmental Member (Department of Computer Science)

Dr. Kenny Wong, External Examiner (Department of Computing Science, University of Alberta)

© Ming Du, 2002
University of Victoria

All rights reserved. This work may not be reproduced in whole or in part,
by photocopy or other means, without the permission of the author.

QA76.9
D26D8

Abstract

Data Base Reverse Engineering (DBRE) is a process of recovering or reconstructing the functional and technical specifications of databases. More precisely, it is a process to recover the schemas that are the results of a database design process. It mainly starts from the source code, such as SQL, of the database application program. Recovering these specifications generally helps to re-document, convert, restructure, maintain, migrate, or extend the legacy databases. The problem is extremely complex for old and ill-designed databases, because there is usually no documentation that can be relied upon and the lack of systematic methodologies for designing and maintaining the databases has led to tricky and obscure code. Therefore, DBRE has long been recognized as a complex, tedious and prone-to-failure activity.

Various computer-aided DBRE tools have been developed to reduce the complexity of this task. However, a major limitation of current approaches is that they cannot represent database schemas and database structure simultaneously. In this thesis, we propose an approach that can visualize database structure and schema concurrently. The approach is built on the current version of the Rigi system, a popular reverse engineering tool. The contribution of this thesis is that Rigi is extended to a useful DBRE tool. It can be used to extract and represent database design information.

Examining Committee:

Dr. Hausi A. Müller, Co-Supervisor, Department of Computer Science

Dr. Jens H. Jahnke, Co-Supervisor, Department of Computer Science

Dr. Daniel M. German, Departmental Member, Department of Computer Science

Dr. Kenny Wong, External Examiner, Department of Computing Science, University of Alberta

Contents

Abstract .... **ii**

Contents .... **iv**

List of Figures .... **vi**

Acknowledgements .... **vii**

1 Introduction **1**

1.1 Relational database understanding environment 1

1.2 Information visualization aids human perception and understanding 3

1.3 Problem 6

1.3.1 Database reverse engineering using the Rigi environment 6

1.3.2 Limitations of the existing relational DBRE tools 7

1.4 Approach 9

1.4.1 Structural logical schema analysis and visualization of views 10

1.4.2 Using visualization techniques to support the understanding 11

1.5 Thesis overview 12

2 Implementation environment **14**

2.1 ANSI/SPARC Architecture and Database Design Methodology 14

2.2 Relational databases 17

2.3 SQL–A Relational Database Language 21

2.4 Parser construction technology and Lex & Yacc 22

3 Related work **25**

3.1 Database Reverse Engineering Environment 25
3.2 DB-MAIN 29
3.3 Varlet 35
3.4 SeeData 38
3.5 Rigi project 42
3.6 Summary 45

4 The Rigi DBRE environment 47

4.1 Introduction 47
4.2 The SQL parser of the Rigi DBRE tool 48
4.3 The SQL domain model 49
4.4 Hierarchical structure visualization 50
4.5 Structured logical schema analysis 54
4.6 An example 58
4.7 Extended work to the logical schema 62
4.7.1 Editable logical schema – reengineering ability 62
4.7.2 Displaying external schemas (views) 63
4.7.3 Visualizing data sizes for tables 65

5 A case study 69

5.1 The sample system 69
5.2 The visual representations 70
5.3 Summary 76

6 Conclusions 77

6.1 Summary 78
6.2 Future work 80

Bibliography 84

Appendix 88

List of Figures

Figure 1.1: Relational database reverse engineering using the Rigi environment....... 10
Figure 2.1: Database architecture and database design 17
Figure 2.2: Lex and Yacc system 24
Figure 3.1: Conceptual and Logical schema regeneration in DB-MAIN 32
Figure 3.2: The schema analysis process of Varlet 36
Figure 3.3: Seven views in SeeData 39
Figure 4.1: The hierarchical structure of relational databases 52
Figure 4.2: Visualization of Primary key, Foreign key and NOT NULL 57
Figure 4.3: An example of the structured logical schema analysis 59
Figure 4.4: Structure information of the SHriMP schema 61
Figure 4.5: Visualization of a sample view 65
Figure 4.6: Tables with different sizes 68
Figure 5.1: First-cut view of the MERCK database 71
Figure 5.2: Tables in the MERCK database 73
Figure 5.3: Recovered structure of the MERCK database 74
Figure 5.4: Detailed design information of the ITEM subsystem 75

Acknowledgements

My thanks go first and foremost to my supervisors Dr. Hausi A. Müller and Dr. Jens H. Jahnke. I am very grateful to Dr. Hausi A. Müller for his support, encouragement and guidance throughout my degree program, especially, in preparing this manuscript.

There are no words that can express my thanks to Dr. Jens H. Jahnke for giving me many suggestions on improvements and new ideas to expand on what I had already done. Working through revision after revision of this thesis for me has provided valuable insight in areas that needed improvement and corrections that I would not have found.

Thanks also to other members of my supervisory committee for their generous help.

Thanks to past and present graduate students and researchers in the department along the way. This effort would not have been possible without the help of many. In particular Johannes Martin, Paul W. Swoboda, Tarja Systä and Shohreh Hadian, who assisted me a lot in preparing this thesis.

Chapter 1

Introduction

1.1 Relational database understanding environment

A large legacy database is inherently complex and may become even harder to understand as it ages. Database structures may degrade because of changing requirements, evolving technology, and unplanned enhancements. After years of modification, the scope and size of the database may be far beyond what its originators envisioned. The initial model may become almost unrecognizable, and even its designers may no longer understand the overall structure.

There are two crucial factors for legacy database understanding: experienced database professionals and systematic updated documentation. Unfortunately, the lack of both makes understanding difficult. Industrial database applications often evolve over three or more generations of developers, cover several hundred thousand lines of code, and maintain a vast amount of data. Maintaining consistent documentation for such huge database systems is not an easy task. After years of evolution, database engineers are

faced with insurmountable understanding problems when dealing with a legacy database: experienced database professionals have left and documentation has become obsolete.

Moreover, companies have to adapt or modernize existing legacy database applications in order to keep up with emerging requirements. These requirements may comprise the integration of a legacy database application with other information systems or the migration to new modern database technologies, such as object-oriented databases. As mentioned above, a typical problem that complicates such a task is that the documentation of a legacy database application has become outdated and the people who are familiar with the application have left. Thus, the understanding problem becomes more critical for the companies possessing the legacy database applications.

In summary, inherent complexity, degraded structures, lack of experienced personnel, and poor documentation, all impede the understanding of legacy databases and together make the understanding of legacy databases more difficult. As such, a database administrator often has to fall back on the analysis of source code to understand how the database system works, spending much time and perhaps duplicating the efforts of others.

One way of augmenting the legacy database understanding process is through computer-aided reverse engineering using *Database Reverse Engineering (DBRE) tools.* Tool-support during legacy database understanding activities has become important in decreasing the database maintenance engineers' and the database users' time of manual source code analysis and to help focus on the important database understanding issues. The size and complexity of legacy database systems require automated reverse engineering support to facilitate the generation of graphical and textual reports of the database systems in order to simplify the understanding process.

Making databases easier to understand, maintain and migrate is an important issue today and will become more important as database technologies evolve and legacy data accumulates.

1.2 Information visualization aids human perception and understanding

Because of the characteristics of legacy databases, automated reverse engineering support is required to facilitate the understanding of entire database systems. At the same time, with the explosive growth of networks and the widespread availability of the World Wide Web, huge volumes of information are becoming available in networked relational databases. Networked databases can be accessed from different locations. Due to networks, databases are more popular and useful than ever before and, hence, more users query databases to get needed information. As a result, databases are playing more important roles today than ever before. Moreover, the need for database management is growing steadily. There is another problem: how can the database administrators extract the information in the legacy relational databases effectively and quickly. Information visualization strives to address this problem by presenting the database information in pictorial form and using human recognition capabilities to detect patterns.

Visual information plays an important role in human communication and recognition. In their every day life, people are used to information on tangible media stored visually. They employ a whole set of perception and association techniques to get information from the outside environment. However, for a given legacy business database, the overall structures and the relationships between the objects and some important components of

these are invisible to users. If database administrators want to acquire information of databases in order to make management decisions, they have to analyze the source code used to create the databases. In other words, all of the information needed to understand and administer the databases is invisible. All the clues for visual perception and associations disappear in a database system.

For example, access to a relational database without any graphical interface to provide clues would rely on one's ability to generate a correct query by using the SQL [16] language and the computer's ability to match the user's query generated by SQL to information stored in the relational databases. At best, when interaction is involved for accessing such a relational database, one can modify the query based on the retrieved results of the query. In the real world, this is like searching for a book in a library without light. One can walk around from stack to stack and get a few books each time, then walk out of the library to see if the book is of interest. If it is not, then along with an educated guess based on knowledge about the library and the experience acquired from the last use, another location must be accessed in order to get the right book. Thus, success in finding the book greatly depends on the user's ability to walk to the right place (generating an accurate query) and to adjust locations until the right place is reached (interactive query generation) [3].

This simple example shows that when the information stored in a system (e.g., in a relational database) is invisible to the users, access to that system is difficult and time consuming. The users have to repeat the same process several times (depending on the user's knowledge and experience) to try and obtain the needed information.

Visual information can really ease the processes of understanding, maintenance and access of an information system. Could we turn the light on? Could we make the stored invisible information visible to the users? Furthermore, how accessible does the picture make the particular information which the user needs? These are questions that researchers have been seeking answers for since the early 1960s when the computer was first used for information retrieval. Today, following the rapid advances in computer information technologies, the need of visual information is more urgent than ever. More visualization tools should be built using advanced computer technologies, both hardware and software, to display the information to the users visually.

To learn how a DBRE tool should be built so that it is useful to its users, three existing DBRE tools, *SeeData*, *Varlet* and *DB-MAIN* were studied. These DBRE tools are reviewed in detail in Chapter 3. One result of this study is that graphical views of relational databases, obtained with these DBRE tools, provide orientation cues for navigation and provide navigation facilities for further exploration. Graphical representations of relational databases obtained with these DBRE tools have been recognized as playing an important role in the relational database understanding.

In summary, visualization is more than a method of computing. It is a process enabling the user to observe, digest, and make sense of the information. It involves more than pretty pictures. Recent advances in visualization have brought information visualization to the center stage. In the not-too-distant future, we are going to see an overall increase in the development of information visualization methods, systems and tools.

1.3 Problem

Databases play a critical role in almost all areas where computers are used, including business, engineering, medicine, law, education, and library science, to name just a few.

Legacy relational databases are operational in many companies and are difficult to understand and maintain due to their size, poor documentation and the evolution history. There is a need for reverse engineering tools to support the understanding and maintenance of legacy relational databases.

1.3.1 Database reverse engineering using the Rigi environment

Rigi is a powerful tool for the analysis of large and complex legacy software systems. It is a system for analyzing and understanding evolving software systems by allowing the user to explore and summarize information structures. The main steps of the Rigi approach are to extract artifacts and their interrelationship from the source code, synthesize these artifacts into higher level abstractions and then build the graphical representations of the modeled subject systems. The aim of the Rigi approach to extracting system abstractions out of a subject is to expose its overall structure and recover architectural design information [2]. In other words, Rigi reads the complex source code of an information system and outputs a graphical representation of that system. The reverse engineering environment provided by Rigi makes it an ideal tool for the visualization of relational databases.

So far, Rigi's primary focus has been on code reverse engineering as opposed to data reverse engineering. Because of the importance of database understanding and the important position that Rigi occupies among the reverse engineering tools, we propose an

approach to combine Rigi and DBRE methods to support DBRE. That proposed approach is described briefly in the next section.

1.3.2 Limitations of the existing relational DBRE tools

DBRE tools support database engineers in the process of analyzing and understanding complex database systems. These tools decrease the database engineers' time of manual source code analysis and help focus on important issues of database system understanding. The functionality of such tools varies from editing and browsing capabilities to the generation of textual and graphical reports. There are several different database models: such as relational databases, hierarchical databases, network databases, and object-oriented databases. Because the *relational databases* are dominating the current database market, several existing *relational DBRE tools* such as *Varlet*, *DB-MAIN*, and *SeeData* have been evaluated. Our thorough study of these existing tools revealed several limitations with respect to schema analysis, especially logical schema analysis.

According to current database development, logical database design is the process of constructing a model of the information used in an enterprise based on a specific data model, but independent of a particular database management system (DBMS) and other physical considerations [14]. The logical database design results in the creation of a logical data model of the part of the enterprise that is interested in modeling. A logical database schema is used to represent the designed logical data model.

The logical schema serves an important role during the operational maintenance stage of the database application lifecycle. It is the expression of understanding of the working of the enterprise and the meaning of its data in a selected data model (e.g., the

relational data model). As a result, any useful DBRE tool would choose the logical schema as the initial reconstructed object. In other words, logical schema analysis is a starting point and foundation for a DBRE process. A logical schema that is structurally complete and semantically enriched will provide its users with invaluable help to carry out DBRE more effectively.

The goal of these existing DBRE tools is to aid the understanding of an analyzed database system, although each of them focuses on different phases of the DBRE process, such as logical schema analysis of Varlet and the refinement of both logical and conceptual schemas with DB-MAIN. Various graphical user interfaces are generated via these tools to facilitate the understanding. All of them provide good reverse engineering capabilities in different usage contexts.

However, the generated logical schema views from the existing DBRE tools have limitations: none of them can provide a picture to show the overall structure of the database and the logical schema simultaneously. Tools, such as SeeData, allow users to obtain multiple views that represent rich information for structural visualization. These views provide great help to understand the database structure. Tools such as Varlet and DB-MAIN allow users to obtain complete schemas of the database. These schemas describe the design information about the database at different levels of abstraction. Therefore, users can use these tools to recover the database design process, which is the goal of DBRE. If users want to know the structure of the database, they have to use tools such as SeeData. If users want to obtain schema views of a database, they have to use tools such as Varlet and DB-MAIN.

Is there a tool that can be used to understand both structure and schemas of a database simultaneously? In this thesis, we propose an approach to address this question with the Rigi DBRE environment. Using this tool, users can extract the structure (architectural pattern) and the schemas concurrently.

1.4 Approach

The Rigi system is a suitable environment for DBRE. The Rigi approach concentrates on extracting structural abstractions. The result of Rigi's approach is to creating higher-level abstract representations of a subject system in order to represent layered subsystem structures graphically. It is an ideal environment for representing the overall hierarchical structures of relational databases. Rigi is also end-user programmable and extensible by means of a scripting language and, hence, can be used as a basis for building a new DBRE environment, the Rigi DBRE system. The approach to building Rigi as a DBRE environment is to add an SQL (Structured Query Language) parser for Rigi. Therefore, Rigi can be used to analyze the relational databases that are implemented by SQL. The reason that SQL was chosen to extract the design information is that SQL is a standard language. The functions provided by Rigi to reconstruct the design information of existing, complex legacy software systems can be transferred and applied to analyzing complex legacy relational database systems. The SQL parser is written using the UNIX tools Lex and Yacc, which are lexical analysis scanner and parser generators. Figure 1.1 illustrates this approach.

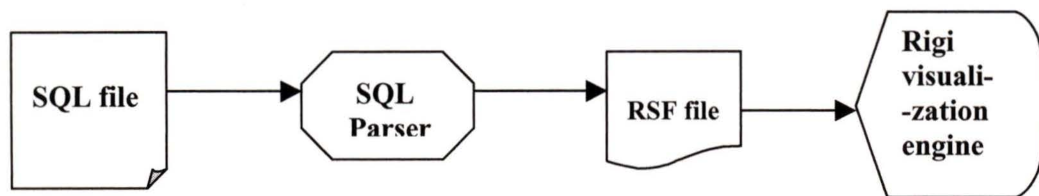


Figure 1.1 Relational database reverse engineering using the Rigi environment

Input. The input of the Rigi DBRE system is the SQL source code including the DDL (Data Definition Language) and the DML (Data Manipulation Language).

Output. The output of the Rigi DBRE system is the graphical views of the structured logical schemas and external schemas as well as the data size for each table.

The SQL parser extracts the needed information from the database description (SQL source code). After the specification of the relational database is extracted, an RSF file is generated. This RSF file can then be loaded into the Rigi system and a visual representation of the parsed result is displayed on the *rigiedit* that is a graph editor. Using *rigiedit* as an interface, visual information that supports the understanding and the management of relational databases will be provided to the database users.

1.4.1 Structured logical schema analysis and visualization of views

The importance of the logical schemas as well as the external schemas (views) in a relational database system is seen in that they are normally the main outputs of DBRE tools. Unlike other tools, the Rigi DBRE tool implements its logical schema analysis based on an overall hierarchical structure of the analyzed database. The DDL text of a relational database is analyzed and parsed to produce the visual representations of the hierarchical structure, logical schema, and the external schemas (views). This process consists of analyzing and parsing the database definition statements included in the SQL

scripts that specify the global database structure and external schemas (views) as well as the detailed information for the tables.

Getting a complete, visual representation of a logical schema is the first and most critical step of the entire DBRE process. It gives the database administrators direct instructions to understand, maintain, re-document and evolve complex database systems.

Some DBRE tools go a step further and allow the user to build conceptual schemas. They transfer the logical schemas to the conceptual schemas (an important high-level component of the database development methodologies), such as entity-relationship diagrams for the relational database model, to recover the conceptual design for the implemented databases. The Rigi DBRE tool also facilitates the redesign of the generated logical schema. In other words, it provides an editable logical schema analysis.

The representations of the external schemas (views) show the related information for the parts of the database that are used to serve particular users. It enables database administrators and database users to focus on the more interesting and peculiar parts. Because the views hide the rest of the database from that user, the unrelated information can be hidden from the overall logical database structure picture in the Rigi DBRE environment. This facility simplifies database management and maintenance for both user groups: database managers or database users.

1.4.2 Using visualization techniques to support the understanding

Color is an important and powerful component of the visualization technique. Domain attributes can be mapped to colors. Quantifiable attributes can be visualized well with different colors [20]. The differences between the analyzed objects can be presented using colors. Hence, colors are used to represent the different objects of the relational

databases in the Rigi DBRE environment, such as, schemas, tables and views as well as the data sizes of each table. As a result, the database administrators can easily recognize the different objects of the database and know the "weight" of each table in order to find out the "central components" for the subject databases. The data sizes of each table can be obtained from the analysis of the DML (Data Manipulation Language) source code.

Filtering is another visualization technique, which allows users to concentrate on features of interest. In hierarchical structures of relational databases, users may wish to see only those nodes (objects of databases) satisfying certain properties, such as internal nodes, (e.g. tables), leaf nodes, (e.g. columns), specified branches of the hierarchy (e.g. schemas or tables). Using the powerful filtering functions provided by Rigi, users may easily focus on the special parts in the hierarchical structures of subject databases.

1.5 Thesis overview

The main thrust of this thesis is to provide a powerful DBRE environment for visualizing the structures, logical schemas, views, and the data sizes of tables in a relational database environment. Chapter 2 provides the background material on the Rigi DBRE environment. A brief explanation of each technique used in the Rigi DBRE approach is presented. Chapter 3 describes the works related to the Rigi DBRE environment. The Rigi system and the existing relational DBRE tools were studied to explore the methods used to extend the Rigi system to a DBRE environment. It features the Rigi system and the three existing DBRE tools: Varlet, SeeData and DB-MAIN. Chapter 4 discusses the details of the implementation of the Rigi DBRE environment. Each section of Chapter 4 details a portion of the Rigi DBRE environment. Some figures present the results of a small example database analyzed using the Rigi DBRE system. A case study is discussed

in Chapter 5. Finally, Chapter 6 summarizes the results and the contributions of this thesis and suggests some directions for future research.

Chapter 2

Implementation environment

2.1 ANSI/SPARC Architecture and Database Design Methodology

Before we present the DBRE process and the existing DBRE tools, it is useful to discuss what the Database Architecture (DA) and the Database Design Methodology (DDM) are.

The architecture of a database system, called *ANSI/SPARC architecture*, is standard terminology and a generic architecture for database systems produced in 1975 by the American National Standards Institute (ANSI) Standards Planning and Requirements Committee (SPARC), ANSI/X3/SPARC (ANSI, 1975) [14]. The ANSI/SPARC architecture is also called *three-level architecture* or *three-schema architecture*. The goal of the three-level architecture is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

1. The *external* or *view level* includes a number of *external schemas* or *user views*. Each external schema describes the part of the database that a particular user or user group

is interested in and hides the rest of the database from that user or user group [16]. In addition, different views may have different representations of the same data [14].

2. The *conceptual level* has a *conceptual schema*. It describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints [16].
3. The *internal level* has an *internal schema*. It describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database [16].

Next, we describe the Database Design Methodology (DDM) which deals with the database design process.

According to Connolly and Begg, a DDM consists of phases that contain steps, which guide the designers in the techniques appropriate at each stage of the project, and also help to plan, manage, control, and evaluate database development projects. Furthermore, it is a structured approach for analyzing and modeling a set of requirements for a database in a standardized and organized manner [14].

In presenting a database design methodology, the database design process can be divided into three main phases: *conceptual*, *logical*, and *physical* database design. Connolly and Begg describe the three phases in their textbook as follows [14]:

1. *Conceptual database design*: to build the conceptual representation of the database. It is the process of constructing a model of the information system, independent of *all* physical considerations. The result is a *conceptual schema*.

2. *Logical database design*: to translate the conceptual representation into the logical structure of the database, which is based on the target data model for the database (for example, the relational data model). Its result is a *logical schema*, which is the description of the data structures as they are implemented on the particular data model. For instance, the logical schema of a relational database describes its tables, columns, primary and foreign keys as well as constraints.
3. *Physical database design*: to describe how the logical structure is to be implemented physically on the target database management system (DBMS). During this phase, the internal storage structure and the file organization for the database are specified. The result is an *internal schema*.

Thus, during the database design process, three schemas are produced and added to the database system. One special schema (which does not appear in the *three-schema architecture*) called logical schema is the result of the Data Model Mapping (DMM). Each DBMS (Data Base Management System) has its own logical data model as well as a logical schema. The logical schema is an essential component of a database system. Figure 2.1 illustrates the correspondence between the ANSI-SPARC architecture and the conceptual, logical, and physical database design.

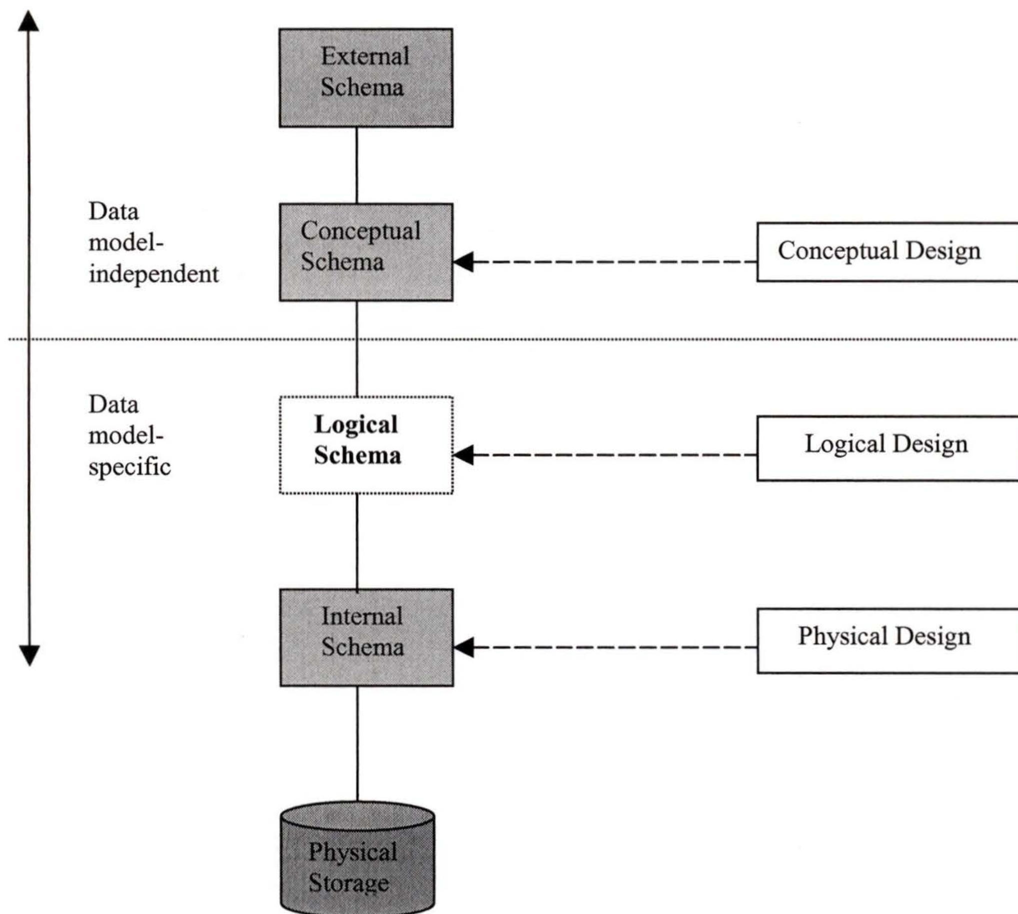


Figure 2.1 Database architecture and database design

2.2 Relational databases

Relational databases have become popular, largely due to their simple data model. The relational database model was conceived by E. F. Codd in 1969, then a researcher at IBM [22]. The model is based on branches of mathematics called *set theory* and *predicate logic*. The basic idea behind the relational model is that a database consists of a series of unordered tables (or relations) that can be manipulated using non-procedural operations that return tables. This model was in vast contrast to the more traditional database

theories of the time that were much more complicated, less flexible and dependent on the physical storage methods of the data [22].

It is commonly thought that the word *relational* in the relational model comes from the fact that you relate together tables in a relational database. Although this is a convenient way to think of the term, it is not accurate. Instead, the word *relational* has its roots in the terminology that Codd used to define the relational model. The table in Codd's writings was actually referred to as a relation (a related set of information). In fact, Codd (and other relational database theorists) used the terms: *relations*, *attributes* and *tuples* where most of us use the more common terms: *tables*, *columns* and *rows*, respectively. The relational model can be applied to both databases and DBMSs [22].

Tables in the relational model are used to represent "things" in the real world. Each table should represent only one set of things. These things (or entities) can be real-world objects or events. For example, a real-world object might be a customer, an inventory item, or an invoice. Examples of events include patient visits, orders, and telephone calls. Tables are made up of rows and columns.

The relational model dictates that each row in a table be unique. If you allow duplicate rows in a table, then there's no way to uniquely address a given row via programming. This creates all sorts of ambiguities and problems that are best avoided.

A *unique key* is used to implement unique constraints. A unique constraint does not allow two different rows to have the same values in the key columns. A *primary key* is used to implement entity integrity constraints. A primary key is a special type of unique key. There can only be one primary key per table, even though several columns or combination of columns may contain unique values. All columns (or combination of

columns) in a table with unique values are also referred to as *candidate keys*, from which the primary key must be drawn. All other candidate key columns are referred to as *alternate keys* [22] [29]. Keys can be simple or composite. A simple key is a key made up of one column, whereas a composite key is made up of two or more columns. The decision as to which candidate key is the primary one rests in your hands—there is no absolute rule as to which candidate key is best. Fabian Pascal, in his book *SQL and Relational Basics*, notes that the decision should be based upon the principles of *minimum* (choose the fewest columns necessary), *stability* (choose a key that seldom changes), and *simplicity/familiarity* (choose a key that is both simple and familiar to users) [23].

Although primary keys are a function of individual tables, if you created databases that consisted solely of independent and unrelated tables, you would have little need for them. Primary keys become essential, however, when you start to create relationships that join together multiple tables in a database. A *foreign key* is used to implement referential integrity constraints. Referential constraints can only reference a primary key or a unique key. The values of a foreign key can only have values defined in the primary key or unique key they are referencing or the NULL value [29]. Foreign keys build the relationships between tables and make the tables dependent and related to each other.

Null is another important concept of the relational data model. A *null* represents a value for an attribute that is currently unknown or is not applicable to a particular tuple [14]. In an SQL CREATE TABLE statement, the *NULL* indicates whether a column is allowed to contain nulls. When *NOT NULL* is specified, the system rejects any attempt to insert a null in the column.

Imagine that a company maintains a database of its employees — there might be a lot of attributes such as age, salary, emergency contacts, appraisal, etc. There may be a need to look at the database for different applications serving different users. The company may need to make available demographic data, for example, to a governmental agency. Only some of the attributes need be supplied – and others ought not to so as to protect privacy.

One of the fundamental benefits of relational databases is the concept of *views*. A *base relation* is a named relation corresponding to an entity in the conceptual schema, whose tuples are physically stored in the database. A *view* is the dynamic result of one or more relational operations operating on the base relations to produce another relation. A *view* is a virtual relation that does not actually exist in the database, but is produced upon request by a particular user at the time of request [14]. The *views* are the way users perceive the data. *Views* provide several advantages:

- *A powerful and flexible security mechanism* — Views can hide parts of the database from certain users. The user has no idea about the existence of any attributes or tuples that are not appearing in the view.
- *A customized way to access data* — The owners of the views can choose the way of accessing the data following their needs, so that different views may have different representations of the same data.
- *A simplified way to* perform *complex operations on the base relations* — The complex operations on the base relations can be transferred to simple operations on the views. For instance, if a view is defined as a join of two base relations, the user may now perform the simpler operations, such as selection, on the view which will be

translated by the DBMS (Data Base Management System) into equivalent operations on the join.

2.3 SQL–A Relational Database Language

SQL is a non-procedural database language that provides facilities for:

- database definition and object creation
- complex queries
- updating
- the handling of security and privacy constraints

SQL is the first and, so far, only standard database language (ISO, 1992) [14] to have gained wide acceptance. It is accepted as the protocol that enables databases built under different DBMSs and/or running on different machine architectures to communicate, forming true distributed systems. Nearly every major current vendor provides database products based on SQL or with an SQL interface, including DB2, Oracle, and Access. SQL statements can be embedded in conventional programming languages such as COBOL or C, and semi-procedural 4GLs.

According to Connolly and Begg, SQL statements fall into three categories [14]:

- Data Manipulation Language (DML): A language that provides a set of operations to support the basic data manipulation operations on the data held in the database.
- Data Definition Language (DDL): A descriptive language that allows the DBA or user to describe and name the entities required for the application and the relationships that may exist among the different entities.
- Embedded SQL: SQL statements are embedded directly into the source code and mixed with the host language statements. This approach allows users to write

programs that access the database directly. A special pre-compiler modifies the source code to replace SQL statements with calls to DBMS routines. The source code can then be compiled and linked in the normal way. The ISO standard specifies embedded support for Ada, C, COBOL, Fortran, Pascal, and PL/1.

2.4 Parser construction technology and Lex & Yacc

Rigi DBRE system uses a lexer and a parser to analyze SQL source code. The terms concerning the SQL parser are now examined.

According to Kiniry and Cheong, *Lexical analysis* (or *scanning*) is the first stage in processing a language, usually for compilation. The source program is fed into a *lexer* (also known as a *scanner*) as a stream of characters. The lexer groups these characters into *lexemes* (or *tokens*), which are word-like elements, such as keywords, identifiers, and punctuation. These elements are the indivisible units of the language.

Next, the *parser*, (or *recognizer*) processes the stream of tokens and determines whether the syntactic structure matches its *grammar*. A grammar is the formal definition of the syntactic structure of a language. The parser can also build an *abstract syntax tree* (AST) as part of its output. This data structure contains the parser's internal representation of the parsed code [24].

Lexers and parsers can be written by hand, but the manual process is complex and time consuming. To solve these problems, today there are a variety of parser/lexer generators to aid the processes, such as Eli, JavaCC, Lex and Yacc, to name just a few. Some of them are used to generate the parser/lexer for a particular language, such as JavaCC which is a Java parser/lexer generator written in Java. Some of them can generate the parsers/lexers for different languages, such as Lex and Yacc and Eli. These generators

take care of the tedious part of the generation process and provide many advantages compared to manual work (e.g., ease in writing, modifying and debugging). Lex and Yacc were chosen as the generators to implement the Rigi DBRE system.

Lex – A Lexical Analyzer Generator and Yacc – A Parser Generator

Lex and Yacc were both developed at Bell Laboratories in the 1970s. Yacc was developed by Stephen C Johnson, and Lex was designed by Mike Lesk and Eric Schmidt to work with Yacc. Both Lex and Yacc have been standard UNIX utilities since the 7th Edition UNIX [1].

Lex and Yacc are tools designed for writers of compilers and interpreters although they are also useful for many applications that will interest the non-compiler writer. Any application that looks for patterns in its input is a good candidate for Lex and Yacc. Furthermore, they allow for rapid application prototyping, easy modification, and simple maintenance.

According to Lesk and Schmidt, Lex is a program generator designed for lexical processing of character input streams. It accepts a high-level, problem-oriented specification for character string matching, and produces a program in a general-purpose language which recognizes regular expressions [25]. Lex source is a table of regular expressions and corresponding program fragments. The user provides the regular expressions and corresponding program fragments in the source specifications given to Lex. The table is translated to a program that reads an input stream, copying it to an output stream and partitioning the input into strings that match the given expressions. The divided units are usually called *tokens*. As each such string is recognized the corresponding program fragment is executed. The recognition of the expressions is

performed by Lex. The program fragments written by the user are executed in the order in which the corresponding regular expressions occur in the input stream.

According to Johnson, Yacc provides a general tool for describing the input to a computer program. The Yacc user specifies the structures (via *grammar*, a grammar specifies which sequences of tokens have a desired structure [17]) of his or her input, together with code to be invoked as each such structure is recognized. Yacc turns such a specification into a subroutine that handles the input process [26].

Lex can be used alone for simple transformations, or for analysis and statistics gathering on a lexical level. Lex can also be used with a parser generator to perform the lexical analysis phase; it is particularly easy to interface Lex and Yacc. Lex programs recognize only regular expressions; Yacc writes parsers that accept a large class of context-free grammars, but require a lower level analyzer to recognize input tokens. Thus, a combination of Lex and Yacc is often appropriate. When used as a preprocessor for a later parser generator, Lex is used to partition the input stream, and the parser generator (Yacc) assigns structure to the resulting pieces [27]. The flow of control in such a case is shown in Figure 2.2

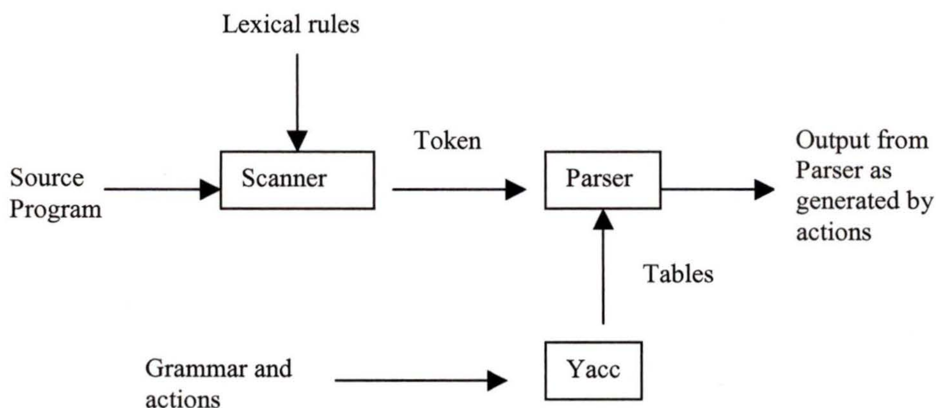


Figure 2.2 Lex and Yacc

Chapter 3

Related work

3.1 Database Reverse Engineering Environments

According to the *Database Design Methodology* (DDM) discussed in Chapter 2, the entire database design process is divided into three phases: *conceptual design*, *logical design*, and *physical design*. During this process, three schemas are generated: the conceptual schema, the logical schema, and the internal schema. These schemas provide useful insights into a database and help in the redesign of databases. But what if one is faced with legacy databases created over time by different development teams for different applications? Reverse engineering is focused on the challenging task of understanding legacy program code without having suitable documentation and most DBRE tools provide just what is needed to atone for these past sins: the elicitation of database schemas.

The DDM shows that the steps of the database *forward engineering* process is from higher-level abstraction and design (conceptual design) to a lower-level abstraction and design (logical design), furthermore, to the lowest-level implementation (physical

design). Database reverse engineering can be seen as the inverse process. It can be characterized as analyzing a software system from physical to logical design. *First,* the system components and their interactions are identified (e.g., some tools have code parsers to extract physical schemas from SQL during this step, like DB-MAIN). S*econd*, the system is represented at a higher level of abstraction (i.e., logical schema). *Finally*, the highest level of abstraction is synthesized (i.e., conceptual schema). In other words, for most DBRE tools, database reverse engineering is divided into two phases: elicitation of the logical schema and elicitation of the conceptual schema [6]. Not all DBRE tools follow this DBRE process. They may only support part of the entire database design process as their DBRE process. They may concentrate on different phases of a database design process. SeeData, for example, focuses on logical schema analysis to recover the logical design information. However, all DBRE tools, even if following different DBRE processes, must work on the different database layers to implement their DBRE: on the external layer to extract the views (external schemas); on the conceptual layer to extract the conceptual schema; on the logical layer to extract the logical schema; and on the physical layer to extract internal schemas.

The *logical schema* is a description of the database structure designed by the database designers, and as application programmers can see and use it. For relational databases, *Logical schema* specifies its tables, columns, primary keys, and foreign keys as well as the constraints to which the data are submitted. A logical schema can be reconstructed by analyzing the specification of the database represented in source code such as DDL.

The *conceptual schema* of a database is an abstract, implementation-independent description of the information that the database stores. Usually, the entity-relationship

model is used to express a conceptual schema of a relational database. The entity-relationship diagram (ER diagram) comprises entity types, relationship types, attributes and various properties and constraints that represent the concepts and structures of the application domain. More often than not, the database administrator is faced with modifying an existing system, or at least with understanding an existing system in order to extract data from it. An invaluable aid in such circumstances is an ER diagram. Once, the logical schema of a relational database has been recovered it can be translated into a conceptual schema. In view of this, building a logical schema is the first step in the DBRE process. The implementation of the Rigi DBRE system, which is the subject of this thesis, focuses on logical schema analysis.

Motivation and objectives of DBRE

DBRE is just one step in the database system life cycle. Indeed, painfully recovering the specifications of a database is not sufficient motivation. It is generally intended to re-document, convert, restructure, maintain or extend legacy database applications. Here we review some of the most frequent objectives of DBRE.

• *Knowledge acquisition in system development* — During the development of a new database system, one of the early phases consists of gathering and formalizing user requirements from various sources, such as user interviews and corporate document analysis. In many cases, a partial implementation of the future database system may exist already; the analysis of such a database system (executing DBRE process) can bring out early useful information.

• *System maintenance* — Fixing bugs and modifying database system functions require understanding of the component involved.

• *System reengineering* — Reengineering a database system is changing its internal architecture or rewriting the code of selected components without modifying the external specifications. The overall goal is to restart with a cleaner implementation that should make further maintenance and evolution easier. Quite obviously, the technical aspects as well as the functional specifications have to be clearly understood.

• *System extension* — This term refers to changing and augmenting the functional goals of a database system (e.g., adding new functions).

• *System migration* — Migrating a database system means replacing one or several of the implementation technologies. For example, moving a centralized database to a distributed database system platform.

• *System integration* — Integrating two or more database systems to yield a unique database system that provides an integrated set of functions operating on the integrated data set.

• *Quality assessment* — Analyzing the code and the data structures of a database system in some detail can bring out useful hints about the quality of this database system and about the way it was developed.

• *Data extraction/conversion* — In some situations, the only component to salvage when abandoning a legacy system is its database. The data has to be converted into another format, which requires some knowledge of its physical and semantic characteristics. On the other hand, most data warehouses are filled with aggregate data that is extracted from corporate databases. This transfer requires a deep understanding of the physical data structures to write the extraction routines and their semantics to interpret them correctly.

• *Data Administration* — DBRE is also required when developing a data administration function that has to know and record the description of all the information resources of the organization.

• *Component reuse* — In emerging system architectures, database reverse engineering allows developers to identify, extract, and wrap legacy functional and data components and integrate them into a new system [21].

The next section reviews three existing relational DBRE tools related to our research: *Varlet*, *SeeData*, and *DB-MAIN*. These tools were chosen because they are excellent examples of successful applications in the relational database reverse engineering domain and occupy prominent positions among the DBRE tools. The Rigi DBRE environment is built upon the strengths and ideas espoused by these tools.

3.2 DB-MAIN

DB-MAIN is a CASE tool used to execute DBRE and program understanding. Its main goal is to support all the database application engineering processes, ranging from database development to database system evolution, migration and integration. In this scope, mastering DBRE is an essential requirement. This tool was developed by the Database Engineering Research Group of the University of Namur, Belgium. It is part of the DB-MAIN project.

Next, we discuss the main aspects and components of DB-MAIN that are related to DBRE activities and are attractive for the Rigi DBRE environment.

DB-MAIN is a DBRE tool that works on the conceptual layer, the logical layer, and the physical layer of a relational database. In other words, DB-MAIN reverses the entire database design process in order to implement the DBRE process. From DB-MAIN, users

can obtain the representation of the schemas for different layers. However, DB-MAIN regenerates the three schemas by different ways.

First, the way that is related to the recovery of the conceptual schema and logical schema is described as follows:

The process can be split into two main phases, which are independent from each other:

1. Retrieve the existing data structures from their DDL script and do further refinement and integration (called *Data Structure Extraction*), and
2. Retrieve a possible conceptual schema that defines the semantics that underlies these data structures (called *Data Structure Conceptualization*)

Data Structure Extraction

This phase recovers the logical schema. Database systems generally supply a description of the schema, such as a DDL script. The first-cut regenerated schema is a rich starting point that can be refined through further analysis.

DDL text analysis — analyzing the data structure declaration statements (in the specific DDL) included in the schema scripts and application programs. The result is a first-cut logical schema. For the *DDL text analysis*, an automatic extractor is used. DB-MAIN has a built-in SQL DDL parser to implement this function.

Schema refinement — non-declarative sources of information are analyzed to detect evidence of additional data structures and integrity constraints. This step includes *program analysis* and *data analysis*.

Schema integration — when more than one information source has been processed, the analyst is provided with several, generally different, extracted (and possibly refined)

schemas. The final logical schema must include the specifications of all these partial views, through the *schema integration* process.

The end product of this Data Structure Extraction process is a complete logical schema.

Data Structure Conceptualization

This second phase addresses the conceptual interpretation of the generated logical schema. It consists of two sub-processes, namely *basic conceptualization* and *conceptual normalization.*

Basic conceptualization — the main objective of this process is to extract all the relevant semantic concepts underlying the logical schema.

Conceptual normalization — this process restructures the basic conceptual schema to give the generated conceptual schema the desired qualities one expects from the final conceptual schema (e.g. expressiveness, simplicity, minimality, readability, generality, and extensibility).

The processes of DB-MAIN to generate the conceptual schema and the logical schema are depicted in Figure 3.1.

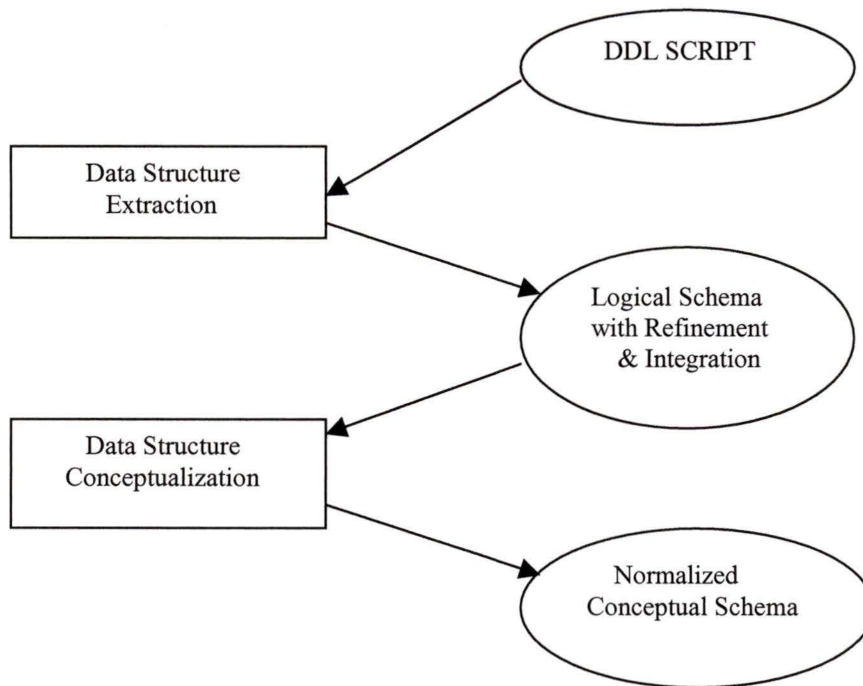


Figure 3.1 Conceptual and Logical schema regeneration in DB-MAIN

As mentioned above, DB-MAIN operates on the physical layer of the analyzed database. DB-MAIN implements this part through a code parser that extracts the physical schema from SQL [18].

In order to understand the schemas generated from the DB-MAIN better, either from the forward engineering (database design) process or the reverse engineering process, it is helpful to know which specification model and which schema transformation approach are used in the DB-MAIN.

The DB-MAIN tool allows analysts and developers to represent and specify information structures, data structures and processing units that make up an information system. These specifications must comply with the so-called *DB-MAIN specification model* that defines the valid objects and their relationships. The model includes a number of concepts: *projects*, *products* (schemas, views and text files), *entity types*, *relationship types*, *attributes*, *domains*, *groups*, *inter-group constraints*, *collections* and *processing*

units. These concepts can be used to describe information systems at different levels of abstraction. The *DB-MAIN specification model* includes the following levels:

Conceptual: entity types, super/subtype hierarchies, relationship types, attributes, identifiers, and constraints.

Logical: record types, fields, referential constraints, redundancy, etc.

Physical: entity collections, access keys, physical data types as well as other implementation details [7].

The desirability of the transformational approach to software engineering is now widely recognized. The process of developing a program can be formalized as a set of transformations [13]. In the same way, the schema transformation approach has been applied in DBRE and the transformational approach is the cornerstone of the DB-MAIN DBRE tool.

A schema transformation consists of deriving a target schema S* from the source schema S by some kind of local or global modification. For example, adding an attribute to an entity type, deleting a relationship, and replacing a relationship type by an equivalent entity type, are three instances of simple schema transformations. Schema transformations define forward and backward mappings between schemas and especially between conceptual schemas and logical schemas.

DB-MAIN proposes a three-level transformation toolkit, namely *elementary transformations*, *global transformations*, *and model-driven transformations.* This toolkit is generic; therefore it can be used in any database engineering process, especially, in DBRE to migrate the logical schema to the conceptual schema.

In summary, DB-MAIN offers the functions necessary to carry out DBRE of large and complex database applications in a really effective way. Especially since, each of its DBRE processes appears as the reverse of a standard database design process. In other words, it works on the three layers of a database system to regenerate the three different schemas. Users can understand a database much better with these schemas. It also builds a generic, wide-spectrum, representation model for conceptual, logical and physical objects; and accepts both entity-based and object-oriented specifications. This specification model is used for either database forward engineering or reverse engineering. Any schemas generated from DB-MAIN must follow this specification model. In other words, for this tool, forward engineering and reverse engineering share a common schema specification model (i. e., meta schema).

Forward engineering and reverse engineering have complementary implementation processes. They utilize the generated schemas for different intentions. For example, for forward engineering, the logical schema is a source of information for the physical design phase. This schema provides the physical database designer with a vehicle for making tradeoffs that are important to the design of an efficient database. Thus, throughout the process of developing a logical schema, the schema should be continually tested and validated against the user requirements. The logical schema must not include data redundancy that can cause update anomalies when implemented. The logical schema must represent the data of the database accurately and efficiently and contain the necessarily detailed information. Conversely, for reverse engineering, one of the important goals of the regeneration of the logical schema is to aid the understanding of the database. The regenerated schema should be readable and not be cluttered, especially

for large legacy database systems that are the main target of DBRE. Thus, a simple, clear picture is most appropriate for the logical schema. As a result, the common specification model of DB-MAIN is not a good choice.

3.3 Varlet

Varlet is a flexible DBRE tool developed originally at the University of Paderborn, Germany (now being developed at the University of Victoria, Canada). As a DBRE tool, Varlet provides both logical schema analysis and conceptual schema analysis.

The general goal of DBRE activities is to recover logical (logical schema) and conceptual design (conceptual schema) information for a subject database. The recovered design information provides high-level abstractions that are a prerequisite to achieve a variety of assessment and maintenance activities. Most DBRE tools assume complete structural and semantic information about the schemas. They do not provide support to earlier analysis activities to obtain the needed information to reconstruct a logical schema that is structurally complete and semantically enriched. Unfortunately, some important structural and semantic information does not exist explicitly in the schema declaration statements of many legacy database systems. If one wants to recover a complete logical schema for a legacy database, he (she) has to deal with implicit information that might exist in schema declaration statements, data, procedural code or the obsolete documentation, like Varlet does.

The main difference between Varlet and other DBRE tools is that Varlet is a flexible DBRE tool. Its schema analysis process is not a strict process and it can accept inconsistent information. In other words, Varlet deals with the variety of application context, uncertain and partial contradicting analysis results as well as the uncertain

assumptions from the analysts during the schema analysis process. Figure 3.2 describes the customizable, semi-automatic schema analysis process of Varlet.

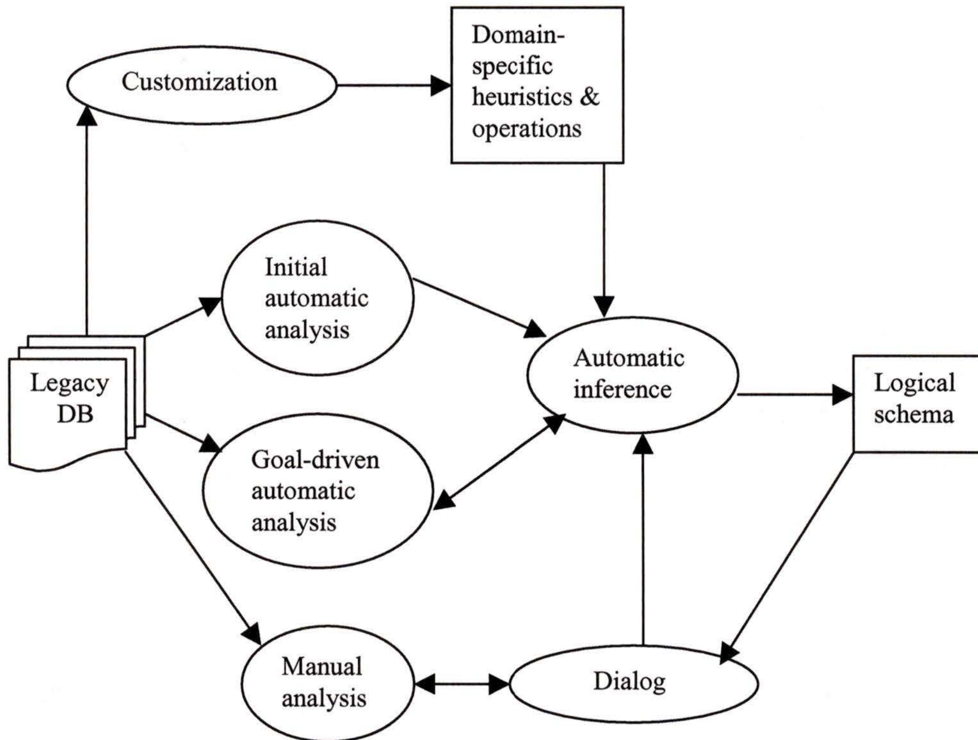


Figure 3.2 The schema analysis process of Varlet

The *Customization Process* investigates the analyzed legacy database to decide the current application context. It results in a set of domain-specific heuristics and operations that is sent to the *Automatic Inference Process*.

The *Initial Automatic Analysis Process* produces a set of (situation-specific) facts about the legacy database. The facts are taken as indicators which are then sent to the *Automatic Inference Process* and combined with domain-specific heuristics generated from the *Customization Process*.

The *Automatic Inference Process* infers new situation-specific knowledge about the legacy database. This result might include definite facts as well as uncertain and inconsistent hypotheses and it is sent to the *Goal-driven Automatic Analysis Process* to check.

The *Goal-driven Automatic Analysis Process* validates intermediate assumptions. The result of this validation is also sent to the *Automatic Inference Process*.

Now, the output of the Automatic Inference Process is an incomplete logical schema that might still partially be inconsistent. This schema is sent to the Dialog Process to do further processing.

The *Dialog Process* checks the controversial information and does further manual investigations to produce new additional hypotheses or definite facts about the legacy database. Again, this generated information is sent to the *Automatic Inference Process*.

The described semi-automatic legacy database schema analysis process is iterated until the engineers get the consistent and complete information about the logical schema of the legacy database.

Industrial experience shows that this tool is well accepted because it does not need a specific predefined sequence of analysis steps to get a consistent and complete logical schema of a legacy database. On the contrary, it supports the DBRE process in an evolutionary process. It helps focus the attention of users on the most controversial or ambiguous parts of the legacy database [5].

3.4 SeeData

SeeData is a visualization system developed by AT&T Bell Laboratories to visualize the structure of large relational databases. This visualization system does not regenerate the entity-relationship diagrams or logical schemas as some other DBRE tools do. Instead, it produces a two-dimensional, multi-view, color representation of the database structure. It also represents the associations among database objects, and the database's relationship to application code. SeeData is a DBRE tool that reproduces either logical design or physical design information. For example, the *Overview* view and the *Association* view of SeeData display information about logical design and physical design. The *Path* view, on the other hand, definitely works only on the physical layer.

SeeData has been described as a relational DBRE tool to display the entire structure of databases without the clutter characteristic of graph-based systems. It does this by replacing the large graphs with a simple representation supported by separate views that show the details. The main function of SeeData is to provide seven linked views. The main view called *Overview* shows an abstract representation of all tables in the database. Around this main view, six additional views give the users different perspectives on the database structure. Each view is shown in a separate window because the views are linked. If one object appears in multiple views, any change to this object in one view will affect all other views in which that object appears. Figure 3.3 displays the relationships between the *Overview* view and the other six additional views.

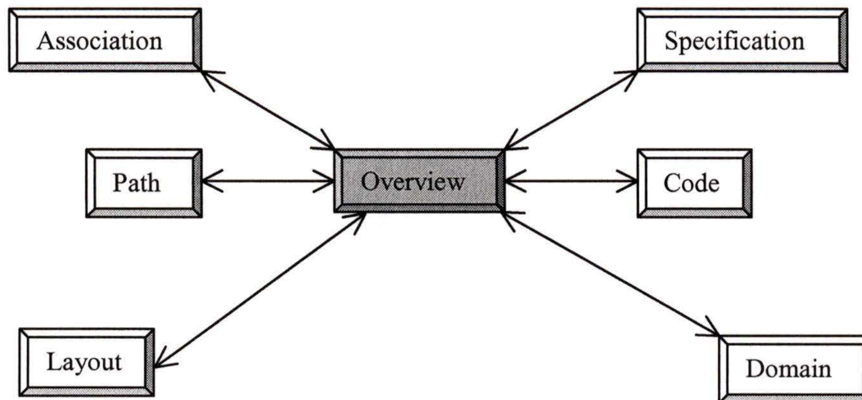


Figure 3.3 Seven views in SeeData

Next, each of the seven views are reviewed.

Overview. The *Overview* gives a high-level abstract representation of database tables and their characteristics. It shows that the SeeData is a DBRE tool working on the database's logical and physical layers.

First, it regenerates the information of the logical design of the analyzed database. It shows all the relations and their attributes in the database. The relations are drawn in a rectangular layout with bars in columns. Each bar represents a single relation. The bar length is used to present the numeric statistics, such as the number of attributes within a relation, the size of a relation tuple, or the maximum number of tuples possible in a relation. Selecting a different statistic causes the bar lengths to change accordingly.

Second, information about the physical design is also provided by this view. The color of each relation encodes a categorical statistic associated with the relation. The user can select the categorical variables, such as the relation access method, how a relation is stored, or how the relation is distributed over processors.

All this information is put on a single screen so that the engineers can understand how a single relation or a set of relations fits into the whole database. In other words, this overall view provides a single picture through which the user can see the entire database and understand database-wide patterns. Thus, it eases the task of understanding the database.

Association. Because of the relationships among relations, a change to a relation might affect several other relations. To acquire such information, SeeData has the *Association* view. This view shows the attributes in a relation and associated relations of this relation. The associations among relations that share data values define a network that database users may follow with joins to retrieve a set of related information.

Specification. The *Specification* view contains the queries predefined in the DML in a scrolling text window. This specification view is linked with the *Association* view. If the user touches something in the *Association* view, SeeData takes him or her to the query defining that association in the *Specification* view.

Path. Sometimes, the user needs data from another relation. To have the needed data, the user can open the *Path* view window. This window represents the shortest paths connecting the two relations of interest. SeeData measures distance in terms of the number of joins needed to traverse the path.

Code. *Code* views are useful to help the user find the application code that references a relation and where in the code that relation is accessed. The *Code* view uses trees to represent the hierarchically structured subsystems, modules, and files for the applications that access the database. Each application has its own tree. Because the *Code* views are linked to the *Overview*, the user can identify all relations accessed from a unit of code.

Layout. The *layout* view is the physical layout in memory for a tuple of a relation and the relative sizes of relation attributes. This function can provide the information about how the larger attribute size will affect memory use.

Domain. Because the user may want to change the domain of an existing attribute, he/she needs to know how domains are used and shared among relations. The *Domain* view shows the domains in the database. Because the *Domain* view and the *Overview* are linked, the user can locate domains used by relations and relations containing a particular domain [9].

In summary, SeeData is a useful DBRE visualization tool to aid the understanding and manipulation of relational database systems. It provides high-level abstract representations of the tables, their attributes, and the relationships between the tables. It also regenerates rich information about the physical layer, such as access methods (via the *Overview* view) and physical layout in memory (via the *Layout* view). SeeData uses the visualization techniques, like colors to present the difference. Colors provide information such as access methods and they are uniform in all views.

One main limitation that was found in SeeData is the information organization. It not only puts the information about the different database layers (or different schemas) in one single picture, but also puts the information that normally is in one single schema into different views. For example, it puts the regenerated information about the logical layer and the regenerated information about the physical layer together in a single screen. It violates the rule which states that three database design phases produce three different schemas and the convention of the DBRE process that is used by database engineers. To do so, the information provided will easily cause confusion to its users. Furthermore, in

SeeData, the information about relations and the relationships among them is put into two separate views: *Overview* and *Association*. This separation makes it difficult for database administrators to recognize all tables and the relationships among them simultaneously.

3.5 Rigi project

Rigi is an interactive, visual tool developed at the University of Victoria for program understanding, software analysis, reverse engineering, and programming in-the-large. Rigi does it through a reverse engineering approach that models the system by extracting artifacts from the information space, organizing them into higher level abstractions, and presenting the model graphically [8].

Reconstructing the design of existing software is especially important for complex legacy systems. To help the understanding of the subject system, documentation has always come to people's minds first. That means that software engineers rely mostly on internal documentation to help them understand programs. But, what should software engineers do with poor and obsolete documentation? Studying source code is a way to understand software systems, but it is not always an easy task, especially for complex legacy software systems. As a result, software engineers have almost no option but to change the direction of understanding and use reverse engineering tools, such as Rigi, to ease the task of understanding large, old, and complex software systems.

Rigi consists of a graph editor and a parsing system with a central repository. The parsing system consists of several subsystems (parsers). Based on a user-selectable option, the Rigi parser invokes a parsing subsystem specific to the application programming language. The Rigi parser is designed to extract entities and relationships among the entities from the source code. For example, the entities that the C parser

extracts from C source code files are *functions*, *data-types*, and *variables*. The relationships that the C parser extracts are *function-calls* among function-entities, *data accesses* among functions and data structure entities, and *variable references* between functions and variables. The C parser also extracts other attributes of the entities, such as their type (either function or data) and optionally the line number in the source code file. The Rigi parser reads programs from standard input and writes the entities, relationships, and attributes it extracts as RSF (Rigi Standard Format) files to standard output. The RSF file consists of a sequence of triples (one triple per line).

The RSF files contain information about the actual software artifacts. For example, the triple: *arcType startNodeName endNodeName* is used to present an arc and its two nodes. Each node represents an entity and the arc shows the relationship between the two entities. For instance, the triple: *call main printf* indicates that there is call between two functions named *main* and *printf.*

The output stream of the parser (RSF file) is loaded into the graph editor, which provides an easy-to-use interface to the users to visualize the initial graph. A language-dependent *domain model* specifies the entity types and relationship of interest for the specific application programming language for the graph editor.

To manage the complexity of the graph, the graph editor allows you to collapse the related artifacts into subsystems automatically (or manually). For example, when a large software system is analyzed, a visual, first-cut representation of the subject system can be displayed on the Rigi screen after loading the generated RSF file. The resulting visual clutter can be confusing. However, by using the Collapse function provided by Rigi, this problem can be solved. The Collapse can encapsulate a data type and its access functions

into a software subsystem, forming an abstract data type. These subsystems typically represent high-level software components (e.g., abstract data types), personnel assignments or other application-specific information [8].

On the other hand, a subsystem node in an overview frame can be opened to reveal more structures. Rigi provides a flexible way (*Rigi views*) to focus attention on important facts of the subject software system. *Rigi views* can be either a high-level overview of the entire structure of a software system or the detailed substructures of specific parts. Thus, the relevant views can be assembled and targeted for different levels of users, such as database administrators who manage the entire database that usually includes several projects, or project managers who administer individual project (schema). As a result, the user can employ Rigi in different ways such as guided tours, overviews, and details.

The discovered structural information is useful for making development and management decisions. The generated information serves as documentation that is up-to-date and current because it is derived from the actual source code. As a result, Rigi helps to understand legacy software systems where the existing documentation may be missing or lacking. Rigi also aids reengineering tasks that need to discover design information in existing software. The Rigi system should be an ideal environment for relational database reverse engineering after extending it with some components and functions that are related to relational databases.

The following section discusses the limitations of these existing tools and the contribution of the Rigi DBRE system to relational database reverse engineering.

3.6 Summary

This chapter has provided an overview of research that is closely related to this thesis. First, the generic methodology for DBRE was described. In particular, three prominent DBRE tools were highlighted as well as the implementation foundation of this thesis, the Rigi system.

The tools discussed here provide a flexible and powerful means of visualization and schema analysis. Varlet puts its focus on logical schema analysis offering solutions for two inherent problems, namely *imperfect knowledge* and *variety of application contexts*. Humans, such as knowledge engineers, application experts and reengineers, actively participating as well as the iterative process are prominent characteristics of the DBRE process of Varlet. In contrast, DB-MAIN focuses on the refinement of the two generated schemas, logical schema and conceptual schema. Both tools recover the schemas, with all the explicit and implicit structure and constraint information. Finally, The SeeData mainly displays visual representations of the structures of large relational databases. It provides an overall view through which the users can see the structure of the entire database, with six kinds of separate views that show different aspects.

It is clear that schema analysis and structure visualization with rich information are basic components of the DBRE tools. From the study of these tools, the basic idea of how to build a DBRE tool was acquired. For example, which phase of a DBRE process should be considered first when the Rigi system is upgraded to a DBRE environment? Moreover, since each of these tools places a different emphasis on the DBRE process, visualization in SeeData and schema analysis for both Varlet and BD-MAIN, some limitations were found among these tools. SeeData supports strong structure visualization technology but

lacks in schema analysis. Varlet and DB-MAIN have powerful schema analysis technology but they lack architecture recovery tools.

The goal of extending the Rigi system to a DBRE system is to provide a DBRE tool that overcomes the limitations found in the existing DBRE tools. The Rigi DBRE system can support schema analysis (logical schema analysis) with structure visualization. As Rigi presents the generated logical schema, it also shows the overall hierarchical structure of the analyzed relational database.

Chapter 4

The Rigi DBRE environment

4.1 Introduction

Before the Rigi DBRE tool is described in detail, a brief overview about this tool is presented here. The Rigi DBRE tool works on the logical layer of a database. In other words, the Rigi DBRE tool puts its focus on logical schema analysis. During its DBRE process, the Rigi DBRE tool presents the overall logical schema as well as the hierarchical structure of a relational database. At this point, the Rigi DBRE tool overcomes the limitation found in existing DBRE tools. Furthermore, the Rigi DBRE tool provides the functions to redesign the generated logical schemas.

Besides the *hierarchical structure visualization*, *logical schema analysis* and *logical schema redesigning*, the Rigi DBRE tool also considers other issues that are critical to manage and maintain a relational database, namely *view visualization* and *data size visualization*. The five main functions implemented by the Rigi DBRE tool are listed as follows:

- Visualization of logical schemas
- Visualization of hierarchical structures

- Reengineering of generated logical schemas
- Visualization of views (external schemas)
- Visualization of data sizes

Each of these functions facilitates an understanding of a particular area of a relational database. Together, they reveal the most important characteristics of a relational database, from the overall structure to the detailed information. Particularly, from the graphical views generated from the listed visualization processes, the users can grasp important information about the database at first sight. On the contrary, it would be a tough task to obtain the same information by browsing the source code. The data size visualization is a typical example of how users can get the information from the visual form easily.

4.2 The SQL parser of the Rigi DBRE tool

As mentioned in the first chapter, the proposed approach in this thesis is to add an SQL parser to the current Rigi system in order to extend the Rigi system to a DBRE environment. UNIX compiler tools, Lex & Yacc were used to implement this parser. Obviously, this parser consists of two components: a Lex scanner and a Yacc parser. The Yacc parser controls the parsing process. It calls the Lex whenever it needs a token from the input that is in the SQL program. In the developed Rigi DBRE environment, the Lex and the Yacc have to agree what the token codes (particularly defined for SQL source code) are. This problem was solved by letting the Yacc parser define the token codes. The tokens in the Rigi SQL parser grammar are the components of SQL sentences such as SCHEMA, CREATE, INSERT, TABLE, and so on. The Lex scans through the SQL stream recognizing tokens. As soon as Lex finds a token which is interest to Yacc, it returns it to Yacc immediately.

Next, specific sequences of tokens are needed to recognize in order to build the relationships among the tokens and perform appropriate actions (C routines) to produce the needed RSF files for input to the Rigi system. The specific sequences of tokens are defined in the grammar of Yacc. The used Lex specification and Yacc grammar are available free from UNNET.

After an RSF file is generated, it is loaded into the Rigi editor to get the visual representation of the analyzed database. But first, before the actions that will be implemented in those C routines of the SQL parser are defined, a domain model of the SQL language must be defined and added to the Rigi system.

4.3 The SQL domain model

The *rigiedit* graph editor can be specialized for particular domains, such as C and C++. A *domain model* defines a domain and determines what node and arc types are possible in the domain. This model is a meta-level description of actual, token-level graph data conforming to the domain. Each domain has a set of proper node and arc types and these aspects are stored in a set of domain files [8].

The Rigi SQL domain model consists of three domain files, namely *Riginode*, *Rigiarc* and *Rigicolor*.

Riginode declares the names of *node types*. These *node types* are the entities. An entity is a distinct object (a person, place, thing, concept, or event) in an organization that is to be represented in the database. Each entity will be displayed in *rigiedit* as a *node*. Together, these nodes with the relationships among them (the relationships will be defined in the *Rigiarc* file) compose the visual representations generated from the Rigi DBRE process. The *Riginode* file of the SQL domain model includes the node types,

such as *Schema*, *Table*, *View*, *Attribute*, *Primarykey*, *Notnull*, *Midtable*, *Heavytable*, etc. Each of these node types is assigned with a unique color. Thus, different objects that are presented with the different colors can be recognized easily in the generated graphical views.

Rigiarc declares the names of arc types that show the relationship between two node types. These two node types are starting node type and ending node type. Given the node types, what are the important relationships between these node types that represent the structures, schemas, and other properties of a relational database? These relationships are the arc types defined in the *Rigiarc* file. The *isDefinedIn* arc represents the relationship between the parents and the children, such as schemas and tables or tables and columns. The *reference* arc connects the foreign key column of the current table and the primary key column of another table. The *selectFrom* arc tells the locations from where the view is selected.

Rigicolor determines the colors of nodes and arcs. To make a distinction between node types as well as the arc types, different colors are used. Users can determine the colors of nodes and arcs according to their own needs and wishes.

4.4 Hierarchical structure visualization

A better way to analyze large amounts of source code is to identify system abstractions first and, in particular, to determine overall structure and recover architectural design information. This recovered information can then be used to break down the source code into understandable and manageable subsystems at various levels of abstraction. These subsystem structures, in turn, can then serve as organizational axes for program understanding as well as risk, change, and impact analyses. Reverse engineering has

emerged as a promising technology for recognizing such architectural features in source text [12].

A software architecture diagram is a high-level view of a software system. There are several common ways to represent software architectures including *hierarchical tree*, *box-and-line drawings*, and *nested boxes* [10]. These diagrams represent the structural abstractions of the underlying software. In other words, they are visualizations of a large, complex software system, such as a legacy database system. Details are abstracted from the source code during the visualization process of a conceptual representation of the software system. [11]

Like other software systems, a relational database system can also be visualized in different ways based on their specific characteristics. In a relational database system, several unrelated projects may co-reside. Each project has its own schema that includes all related tables. Multiple schemas might exist in a database system, each schema might have multiple tables, and each table might have multiple columns. As a result, the objects (such as schemas, tables and columns) of a relational database build up a hierarchy. The top layer is the root representing an entire database and the bottom layer consists of leaves that represent the columns of this database. Figure 4.1 shows the hierarchical structure of relational databases.

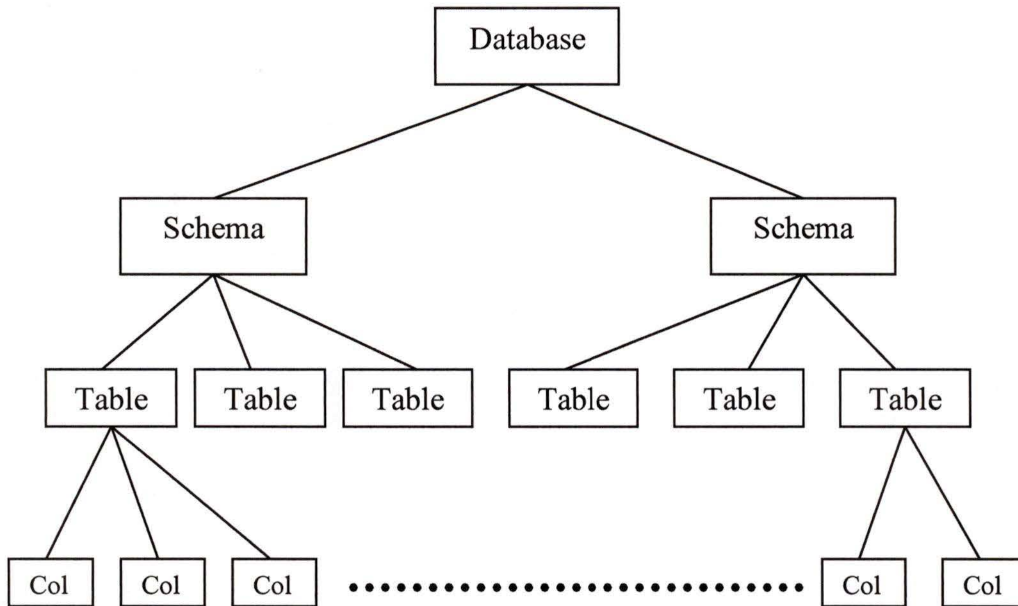


Figure 4.1: The hierarchical structure of relational databases

Figure 4.1 shows that a relational database is more than a collection of tables. Additional structures, on several levels, help to maintain the integrity of the data. A relational database's hierarchical structure provides an overall organization to the tables. In other words, a hierarchical structure is the organizational structure of a collection of related tables.

Database administrators need a quick and easy way to understand the entire hierarchical structure of the databases in order to understand the database objects and the relationships among them. They need database visualization tools that present the needed information in visual ways to support understanding of database architectures. Of course, the database administrators can collect the architectural information from the source code (e.g., the SQL DDL source code). But there is a definite limitation here: performing an abstraction to represent the hierarchical structure of a database can be difficult and time

consuming when working with only source code, especially for the legacy database systems. It needs experienced engineers with a lot of time. Furthermore, because of the human limitations, some important information of the structures might be missed during manual work. Such mistakes can lead the understanding process in a wrong direction.

Now, the implementation process of visualizing the hierarchical structure of a relational database by the Rigi DBRE tool can be explained in detail.

The objects that are located at different layers of the hierarchical structure of a relational database are (from top to bottom): database, schema, table, and column. Each of them is a node type, and these instances are presented as nodes in a Rigi editor window. The relationships between the layers are the same, presented using the arc type *isDefinedIn*.

These four objects are given, as tokens, to Lex and Yacc. The output of the SQL parser is a set of generated triples that are stored in RSF files. Each time, the parser meets one of the four tokens in the parsed SQL program, it should add a triple like *type nodeName nodeType* to the RSF file. As a result, when the generated RSF file is loaded into *rigiedit*, the corresponding nodes can be created in a *rigiedit* window. Whenever the parser meets two different node types, the relationship between the two node types should be considered immediately. Usually, this relationship is one to many, such as the relationships between schemas and tables, and the relationships between the tables and columns. In certain cases, the relationship could be one to one. For example, if there is only one schema existing in a database, the relationship between the database and the schema should be one to one. During the time in which the nodes are created, the

relationships among these nodes are also built. These actions that create nodes and build relationships are implement by C routines in Yacc.

4.5 Structured logical schema analysis

In the previous section, the visualization process of the hierarchical structures was described. Graphical representations of the database hierarchical structures can be generated through that visualization process. In this section, the focus is on the second step of the Rigi DBRE process: *structured logical schema analysis*.

The general goal of the DBRE process is to recover conceptual design information, a complete conceptual schema diagram for an implemented database system. For the relational database, it is the *entity-relationship diagram*. The conceptual diagrams display the entities of interest explicitly as well as the relationships (associations) between them. However, the conceptual schema is the highest level of abstraction for a relational database. In order to obtain such a diagram, a series of DBRE activities must be done first. Logical schema analysis is the most important of these activities. The conceptual schema is generated from the logical schema through a conceptualization process. Since, logical schema analysis is the foundation of conceptual schema analysis, the reasonable deduction is that logical schema analysis is the basis of the entire DBRE process.

Like some existing DBRE tools, the Rigi DBRE tool is dedicated to the logical schema analysis. However, the logical schema analysis process of the Rigi DBRE tool is different from some existing DBRE tools. Actually, the logical schema analysis of the Rigi DBRE tool is implemented by adding the logical schema information to the generated hierarchical structure. Such a logical schema analysis is called: *structured logical schema analysis*.

The reason for utilizing the hierarchical tree as the Rigi DBRE approach to represent the logical schema of relational databases is that users can view the logical schema and the hierarchical structure of a database simultaneously. Furthermore, such structural visualization leaves out some details of definitions of tables, so that the entire database structure and the selected components of tables can be displayed with effectively.

A relational logical schema comprises tables made up of columns, primary keys, unique keys and foreign keys as well as constraints to which the data are submitted. During the first phase of the Rigi DBRE process, a hierarchical structure is generated. This generated structure view represents the tables and columns that are needed for a logical schema. Thus, for the logical schema analysis phase of the Rigi DBRE tool, the focus is on visualizing primary keys, unique keys and foreign keys as well as one selected table constraint.

As mentioned in Chapter 2, primary key and foreign key are two important components in the tables in a relational database. Primary key uniquely identifies a row in a table and foreign key connects the related tables in a database. Both of them need to be represented in the logical schema during the Rigi DBRE process.

The process of visualizing unique keys is left out here because it is exactly the same as the process of visualizing primary keys that is described in detail next.

Another important component that needs to be represented in the Rigi DBRE tool's logical schemas is the constraint: *NOT NULL*. Constraints are rules that determine what values the attributes of a table can assume. By applying constraints to a column, you can prevent the entering of invalid data into that column. The characteristics of a table

column, plus the constraints that apply to that column, decide the column's domain that is the set of all values the column can contain.

Traditionally, the application program that uses the database applies any constraints to a database. The most recent DBMS products, however, enable the users to apply constraints directly to the database. This approach has several advantages. If a database is accessed by multiple applications, the constraints need only be applied once to this database rather than multiple times. Additionally, adding constraints at the database level is usually simpler than adding them to an application. In many cases, only a clause is tacked onto your CREATE statement in the SQL program. Such an approach provides a chance for the Rigi DBRE tool to present the constraints in a visual form through the analysis of the SQL DDL program.

There are many constraints that you can add to the columns or tables in a database by way of SQL. There are three kinds of constraints that can be applied to the tables: *column constraints*, *table constraints*, and *assertions*. A *column constraint* is a condition that is imposed on a column in a table. A *table constraint* is a constraint that is applied on an entire table. An *assertion* is a constraint that can affect more than one table. Among so many constraints, *NOT NULL* was chosen as the representative constraint to be visualized by the Rigi DBRE tool. Certainly, more constraint visualization functions would be developed for the Rigi DBRE tool in the future and are discussed at the end of this thesis in the conclusions Chapter.

Next, the approaches used to visualize primary keys, foreign keys and the NOT NULL constraint are described. If the effect of a constraint applies within a table, *different colors* will be utilized to distinguish the columns that have this constraint from

those columns that do not have this constraint. There are three such column types: the *PRIMARY KEY* columns, the columns with the *NOT NULL* constraint, and the columns without any additional constraints.

If the effect of a constraint goes beyond one table (e.g., the *FOREIGN KEY*), arcs are used to represent the relationships among those columns that have the *FOREIGN KEY* constraint among the related tables. One end of such an arc is linked to a primary key node and the another end of this arc is linked to a foreign key node. Figure 4.2 sketches out the basic idea of our approach to visualize primary key, foreign key and the NOT NULL constraint.

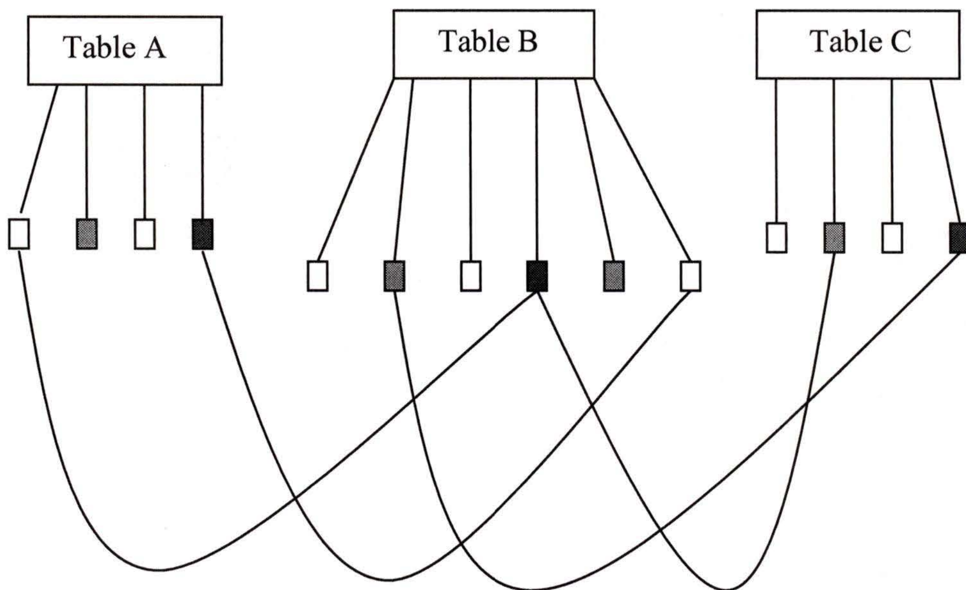


Figure 4.2 Visualization of primary key, foreign key and NOT NULL

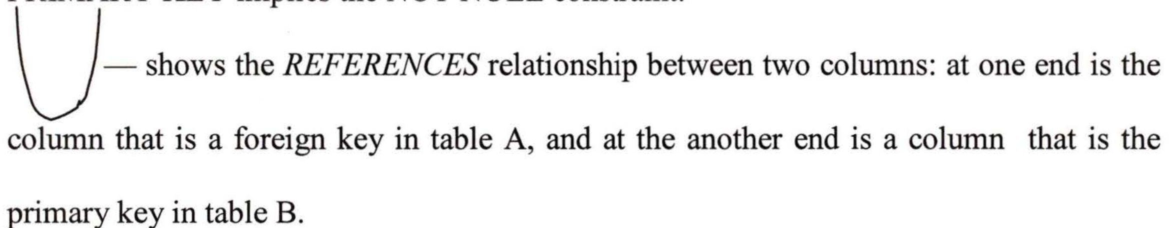
Here, the different gray degrees are used to indicate the columns with different characteristics:

□ — columns that do not have any additional attributes.

▩ — columns that have the *NOT NULL* constraint.

■ — columns that are the PRIMARY KEY columns.

In a real *rigiedit* window, the different colors are used to represent the different attributes of the columns. Because a primary key can not have a null value, the PRIMARY KEY implies the NOT NULL constraint.



— shows the *REFERENCES* relationship between two columns: at one end is the column that is a foreign key in table A, and at the another end is a column that is the primary key in table B.

Different situations for the primary keys and foreign keys are considered during the Rigi DBRE process. For example, a table might have multiple columns with the FOREIGN KEY constraint with references to different tables; a PRIMARY KEY column might correspond to multiple FOREIGN KEY columns in different tables.

4.6 An example

To show the graphical representation of the structured logical schema analysis generated with the Rigi DBRE tool, a small example of a relational database is used here. The SQL DDL script of this database is used as the input to the Rigi SQL parser.

This database is built based on the information of a small research group. This database, named *RevEng*, has two schemas, named *Rigi* and *SHriMP*. Each schema, considered as an independent project, has several tables that describe the detailed information about the research project. For example, the *People* table describes the staffing information; whereas *Offices* table and *Hardware* table each provide partial information about the subject schema. As in the real world, every entity has a set of associated attributes; so each of these tables has a set of columns that embody a single characteristic of the table.

First, the hierarchical structure of this small example database is shown in Figure 4.3. As mentioned above, there are four different kinds of node type: *database*, *schema*, *table*, and *column* that represent the different layers of the hierarchical structure. Each of these node types has a different color. Because the relationships among these nodes are the same, all of the arcs that connect the different layers in this picture have the same color.

Second, information of the logical schema, such as PRIMARY KEY columns, FOREIGN KEY columns and columns with the NOT NULL constraint, is also added to the hierarchical structure in Figure 4.3, as discussed above.

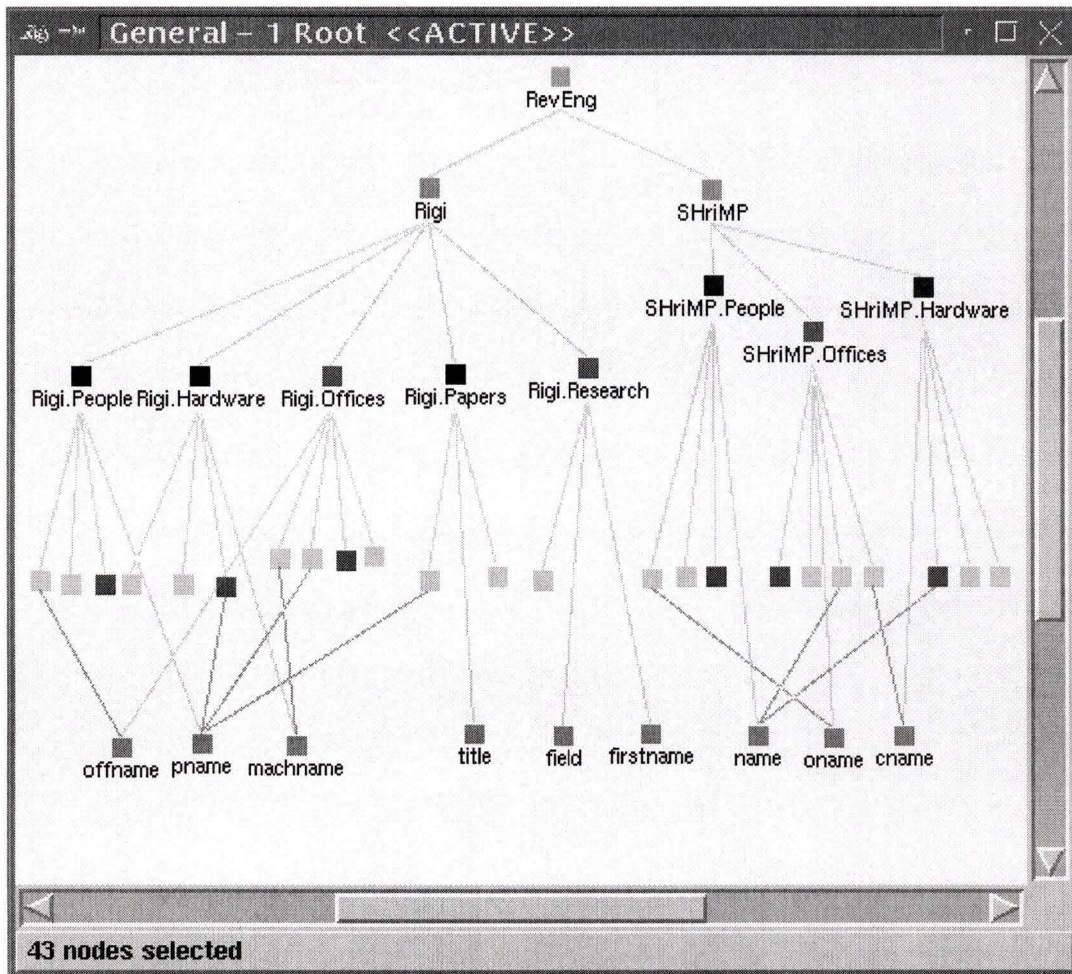


Figure 4.3 An example of the structured logical schema analysis

To sum up, through the first phase of the Rigi DBRE process, a hierarchical structure is generated. Secondly, through the logical schema analysis phase, several important components of a relational database are added to the hierarchical structure: primary keys, foreign keys, and NOT NULL constraints. The final graphical view provides either a logical schema or a database structure of a database to users.

Next, the Rigi *view* function that was mentioned in Chapter 3 is explored to display a part of the small sample database, a project schema – SHriMP in a the Rigi editor window (Figure 4.4).

Figure 4.4 shows that the Rigi *views* are useful for the analysis and understanding of a relational database. Looking at Figure 4.3, we see two schemas. Each schema is related to an independent project included in the RevEng database. The database administrator might want to see the overall picture of this database as well as the individual picture for each schema. On the other hand, project managers might have no need to know the entire database as well as other projects. As such, the only structure of a specific project is needed and it can be carried out through the *view* function of Rigi. Figure 4.4 displays only the structure information of the SHriMP schema, one of the two schemas in the *RevEng* database mentioned above.

Usually, as people design database systems, each project has its own associated schema, which one can distinguish from other schemas by name. It is possible that different schemas have tables with the same names even though these tables have different contents. If there exists a chance of a naming ambiguity, a table name is qualified by using its schema name as well to ensure that no one accidentally mixes in tables from one project with those of another. This is especially important in the Rigi

DBRE tool because the same name tables will create duplicate triples: *type nodeName nodeType*. As a result, when the generated RSF is loaded into *rigiedit* later, all of the tables with the same name will only correspond to one node in the *rigiedit* window, which is obviously incorrect.

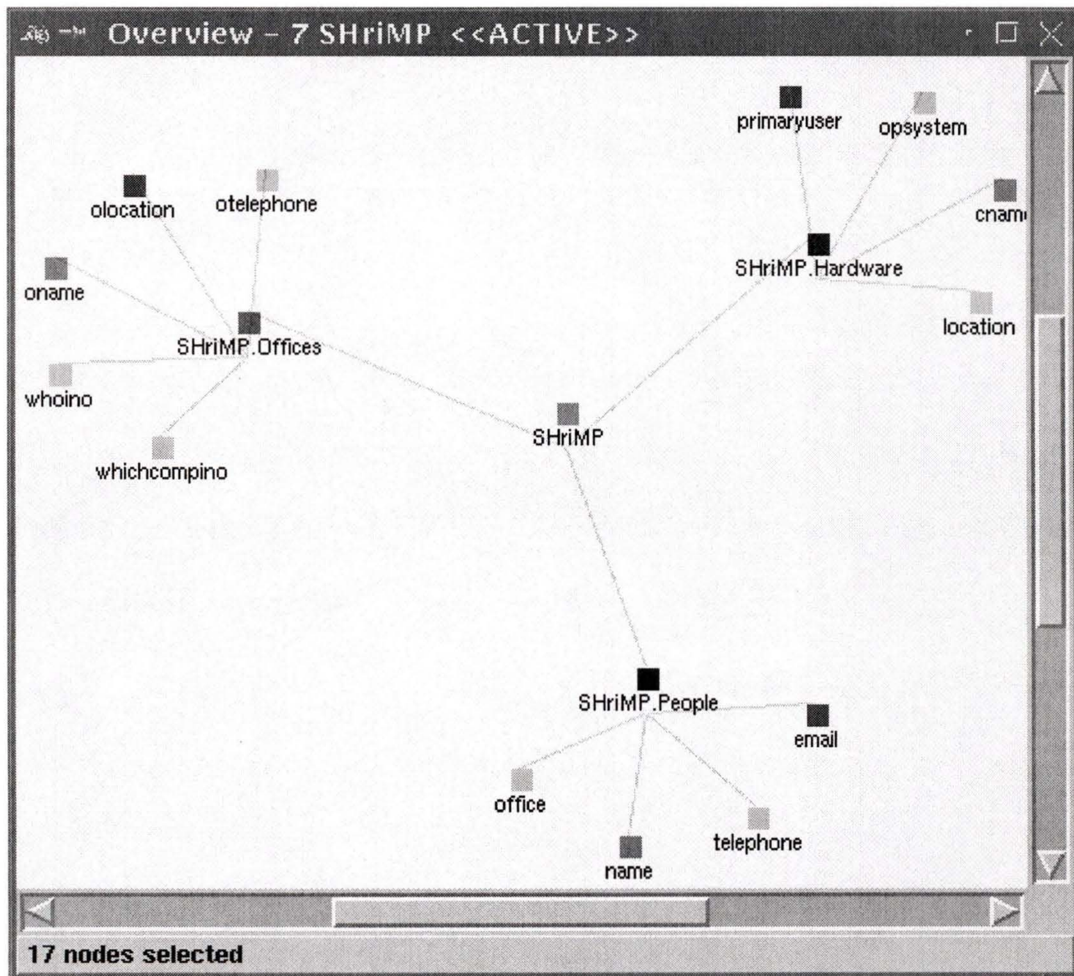


Figure 4.4 Structure information of the SHriMP schema

4.7 Extended work to logical schema

4.7.1 Editable logical schema – reengineering ability

To this point we have concentrated on the reverse engineering ability of the Rigi DBRE tool. During its DBRE process, the Rigi DBRE tool first extracts the hierarchical structure from the DDL source code. Based on the generated hierarchical structure, a logical schema, including Primary keys, Foreign keys and Constraint NOT NULL is extracted from the parsed DDL code.

With the logical schema picture, database administrators, who have the responsibility for database design (usually, they are not the original designers for the analyzed legacy databases), can easily recognize the logical design information.

> "It has been repeatedly shown that no matter how hard designers and programmers try to anticipate and provide for users' needs, the effort will always fall short. It is not the fault of the designers and programmers; in general, it is impossible to know in advance all that will be needed. Noone can foresee all the situations their systems and applications will encounter; customizations inevitably become necessary." [20]

After they acquire the logical design information for the structured logical schema analysis, experienced database administrators might find that some changes are needed to the current logical schema in order to improve the performance of the analyzed database. Such actions might include adding Primary keys, Foreign keys or the NOT NULL constraint to a column. Such tasks would require the Rigi DBRE tool to have the function of redesigning the generated logical schema. As a result, an editable logical schema analysis (reverse engineering + reengineering) is needed.

Based on the reengineering ability of the Rigi system, there are two ways to implement the needed reengineering function: first, using the "Changing the type of a

node" function of the Rigi system, one can change the node type of a particular node. For example, if a node type is changed from "attribute" (column without any constraints) to "Notnull" (column with the NOT NULL constraint), the NOT NULL constraint will be added to this specific column. Second, using the "Deleting a node" function and the "Creating a new node" function of the Rigi system together, also can achieve the goal of changing the type of a node. There are also functions in the Rigi system that can be used to change the types of arcs in order to match the relationship types when the node types are changed.

In the Rigi system, each node can have a text file linked to it (such as source code). This file is specified by the *file attribute* for the node. A line position within the file is specified by the *lineno* attribute. Double-left-click on a node can edit the associated text file [8]. Through this way, the source code (DDL script) can be changed to coordinate the changes of nodes or arcs in the Rigi screen.

4.7.2 Displaying external schemas (views)

Usually, the end-users do not want to see more data than the data they actually need to see. In recognition of this problem, a DBMS provides another facility known as a *view mechanism*, which allows each user to have his or her own view of the database. The DDL allows *views* to be defined, where a *view* is a subset of the database. As well as reducing complexity by letting users see the data in the way they want to see it, *views* have several other benefits. *Views* provide a level of security, a mechanism to customize the appearance of the database, and *views* can present a consistent, unchanging picture of the structure of the database, even if the underlying database is changed.

Database *views* allow developers to present data in any number of "logical" combinations, hiding any details of their underlying physical storage. *Views* effectively *transform* the structure of one or more underlying tables into a more useful or more appropriate structure for the demands of a specific application.

Because of the important roles that the *views* play and the dynamic relationships among the base tables and the *views*, the database administrators need to track the *views* at all times. They need to know how the *views* are generated and which base tables as well as their columns are related to a particular *view*. Based on these facts, the visualization of the *views* is considered an important component of the Rigi DBRE process. The emphasis of this visualization process is on expressing the relationships between the *base tables* and the *views*. In other words, the Rigi DBRE tool provides database administrators with a simple way to find the resources (tables) from which the *views* are selected.

Figure 4.5 shows an example *view* on the screen of the Rigi DBRE tool representing the results of the visualization process. There is a *view* node named *members*. It is a *view* that is only concerned with the personal email information of the Rigi group. From this graphical representation, it is clear that this *view* is selected from the base table Rigi.People. In this case, there is no interest for all columns of this base table and the only information needed are the people's names and their email addresses. The *view* node is displayed with a special color (defined in the domain file: Rigicolor), which indicates that this node is a *view* node. The arcs that connect the base tables and the views also are displayed using a particular color (defined in the Rigicolor file) that indicates the arc type *selectFrom*.

The sample showed here is the simple case of *views*. More complex *views* selected from more than one tables can be easily visualized with the Rigi DBRE tool using the same method described here.

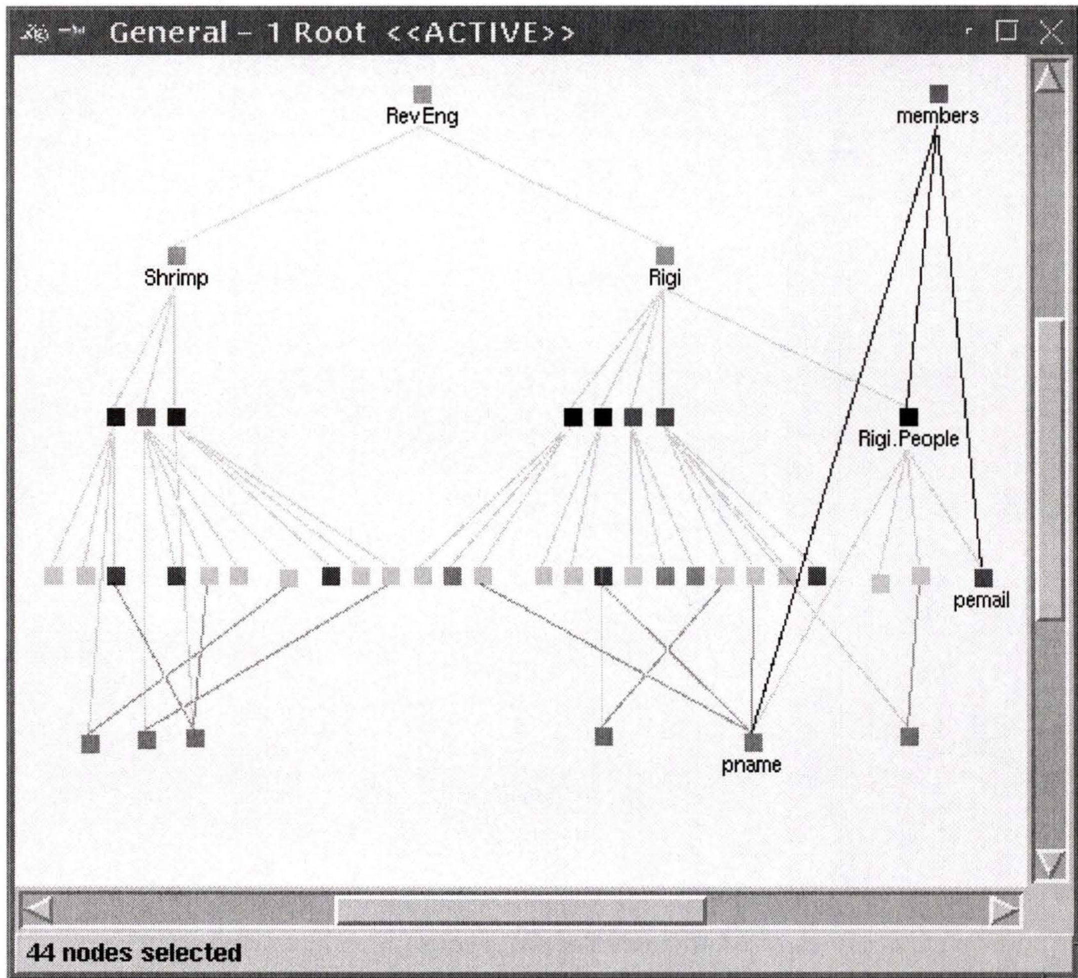


Figure 4.5 Visualization of a sample view

4.7.3 Visualizing data sizes for tables

So far, our visualization was based only on parsing DDL. To make the Rigi DBRE tool more useful, it should not be limited to the information produced from the semantic DDL

statements. Additional information other than DDL statements should be under consideration. In this section, another visualization process implemented by using DML is discussed.

One goal of understanding a software system using reverse engineering technology is to expose properties of this system for managing risk and deciding personnel assignments. For example, highly complex "central" components (as opposed to simpler "fringe" components) are best handled by experienced maintainers [2].

A relational database is a collection of tables, which contains rows and columns. Related tables build up a relational database. In other words, the management and maintenance of a relational database is mostly considered through its tables.

The size of a relational database heavily depends on the data size of each table. Database administrators should pay more attention to those tables that have a huge number of rows (much more than the average number) inside. For a large, static database, such tables are the "central" components for management. Even for non-static databases (the inside data is accessed and changed often), knowing the data sizes can also aid the maintenance activities, such as the maintenance of physical storage.

The data size of each table discussed here is a *relative data size*. In the Rigi DBRE environment, data sizes of tables are to be related with numbers of rows (the numbers of columns are ignored, assuming no dramatic difference between the numbers of columns for the tables in a database). In other words, for the approach proposed here, the data size of a table is presented using the number of the rows in this table. Furthermore, for making a distinction between "central" and "fringe" tables, knowing the exact number of the rows of each table is not necessary. On the contrary, the range, the number of the rows of

each table belongs in, is a much more useful message to the database management. For example, tables of a small size relational database can be divided into three groups: group *A*, group *B* and group *C*. For group *A*, the number of the rows of each table is less than *number1*. The tables in this group are represented as a normal size table in the graphical views generated by the Rigi DBRE tool. For group *B*, the number of rows of each table in this group is between *number1* and *number2*. Each of these tables has more data (rows) than the tables in group *A* and these tables are represented as mid size tables in the graphical views generated by the Rigi DBRE tool. For group *C*, the number of rows of each table in this group is greater than number2 and they are presented as large size tables in the graphical views. *Number1* and *number2* are numbers and could be customizable. In that case, the data size of each table being a main factor for managing risk and personnel, the consideration order should be *C*, *B*, and *A*. Practically, the partitions of the tables of a database like the groups *A*, *B* and *C* can be decided by the requirements of users.

The visualization of the data size of each table provides a resourceful way for database administrators to acquire the needed information. This visualization process is implemented using colors. In the example mentioned above, two special node types are created and added into the SQL domain file to separate the tables with different data sizes. And, as mentioned previously, the different data sizes mean different ranges of row numbers. The two added special node types are *Midtable* and *Heavytable* and each of these nodes is displayed with a specific color that is different from the color of the ordinary table node (tables in group *A*). Together, there are three node types with three different colors to represent all tables of a database in the generated graphical views. The table nodes with different colors might show different levels of importance to the

database administrators for managing this database. Thus, with these visual results, database administrators and maintainers can find the "central" tables immediately.

Figure 4.6 illustrates the proposed approach for visualizing the data size of each table.

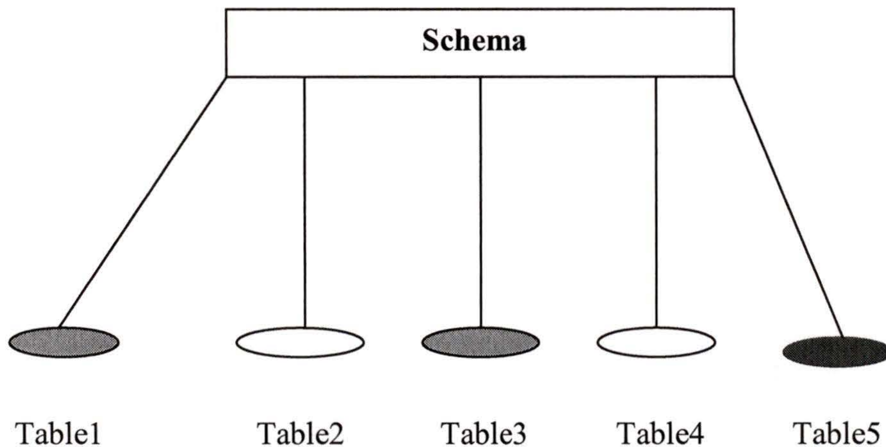


Figure 4.6 Tables with different sizes

The diagram shows that Table2 and Table4 are in the same range of row numbers. For the example database, they have less than *number1* rows. This diagram also shows that Table1 and Table3 are in the same range of row numbers; the number of the rows of this level is between *number1* and *number2*. Table5 is in the highest range of row number; the number of the rows for this range is greater than number2. Thus, the database administrators might obtain a clue from this picture: Table5 is the central component of this database, and is relatively more important to the database management.

The resulting view of the data size visualization is presented in Figure 4.3

Chapter 5

A case study

5.1 The sample System

Hopefully the discussion in the previous chapters about the Rigi DBRE tool should give the impression that the Rigi DBRE system is a useful DBRE tool for aiding the understanding of relational databases. To demonstrate this further, the Rigi DBRE tool was tested on a real industrial database called MERCK. The objective of this case study is to visualize the database structure and the logical schema that was extracted from the MERCK source code as faithfully as possible. This case study was based on the MERCK's DDL source code only, and no data and application code were available. As one would expect for a case study, not all the problems nor all the techniques and reasoning are illustrated.

MERCK is a relational database of an international enterprise that produces a great variety of drugs and other chemical products. The original version of MERCK was developed as an information system to maintain the company's product catalog. Subsequently, the functionality was extended to support the maintenance of information about documents related to stored products. MERCK has become the central source for

information about the products and documents of this company. The MERCK system has evolved over more than ten years and has been subject to many modifications. Old hard- and software platforms and the lack of consistent technical documentation result in serious problems in maintaining the database system [28]. It consists of 85 tables and 347 attributes that are declared by about 1,200 lines of SQL source code. Certainly, it is not an easy task to analyze such a database's structure and logical schema based on studying source code.

But an attempt is made to demonstrate the benefits of using a visualization tool. Observations are made using the visual representations of the Rigi DBRE tool that otherwise could be difficult to deduce from the textual representation. This case study was done in two steps: *DDL code extraction* and *View refinement*. The *DDL code extraction* operation is carried out using the *SQL* parser of Rigi. The *view refinement* operation is implemented using the visualization functions of Rigi, such as filtering and collapsing.

The details of the implementation of the Rigi DBRE processes have already been described in Chapter 4. Hence, the focus of this chapter is on presenting the resulting visual representations of MERCK.

5.2 The visual representations

Figure 5.1 displays the original graphical view of the MERCK database generated by the Rigi DBRE tool. This view represents the structural and design information extracted from the MERCK DDL source code. The nodes and arcs represent the objects of the MERCK database as well as the relationships among these objects. Using the

visualization techniques provided by the Rigi DBRE tool, this first-cut graphical view can be simplified easily.

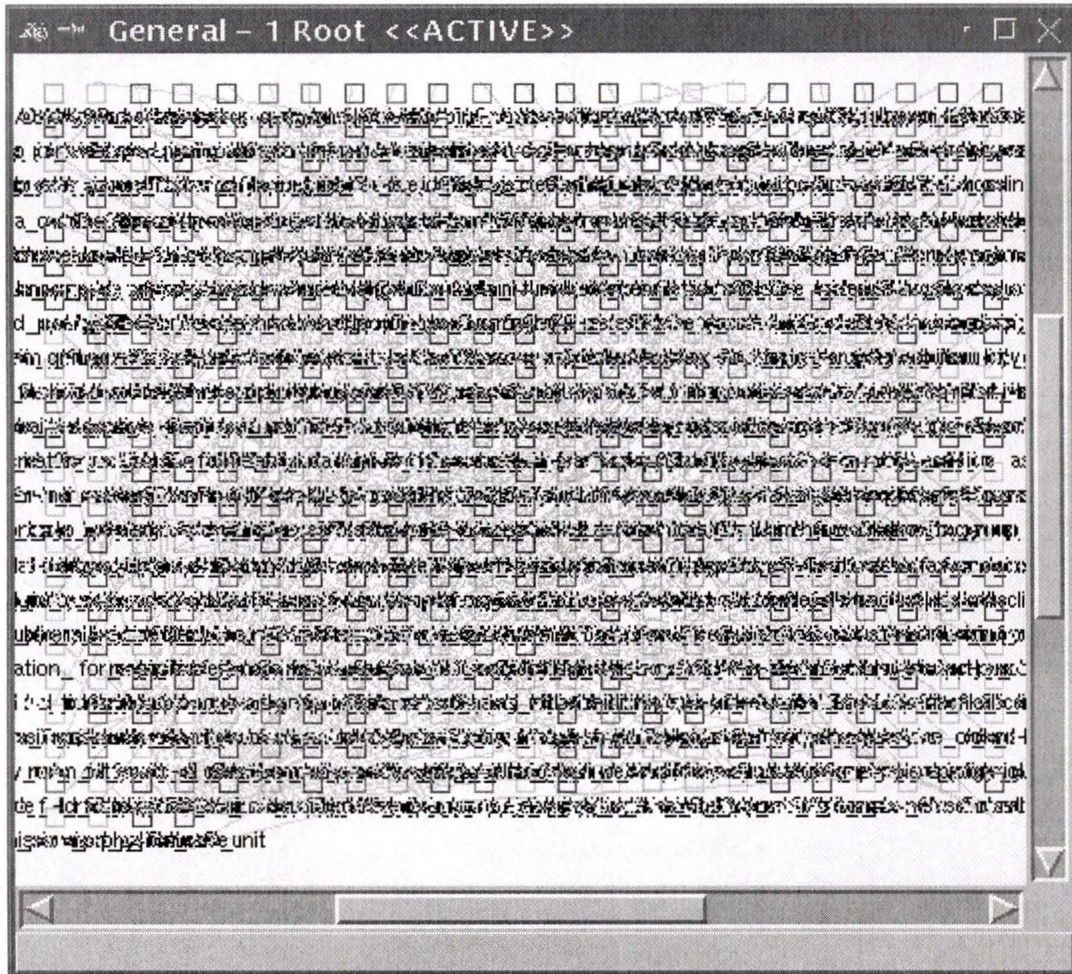


Figure 5.1 First-cut view of the MERCK database

As mentioned in previous chapters, the Rigi DBRE tool provides two ways to help in the understanding process: *structure visualization* and *logical schema analysis*. Compared to the visualization of structure, the logical schema analysis of the Rigi DBRE tool focuses on analyzing the detailed design information in a table, such as definitions of

the columns and the constraints. The details of schema analysis have already been discussed in the previous chapter and a small database example was analyzed to present the detailed information of tables visually. A better way to understand a middle or large size database, such as MERCK, is to first understand its structure, then the detailed design information. Thus, this case study, due to the size of MERCK, first focuses on structure visualization in order to facilitate database understanding.

A relational database is a collection of tables. In other words, the structure of a relational database is about the tables and their relationships. After filtering some details, such as columns and constraints, the original picture is simplified as depicted in Figure 5.2.

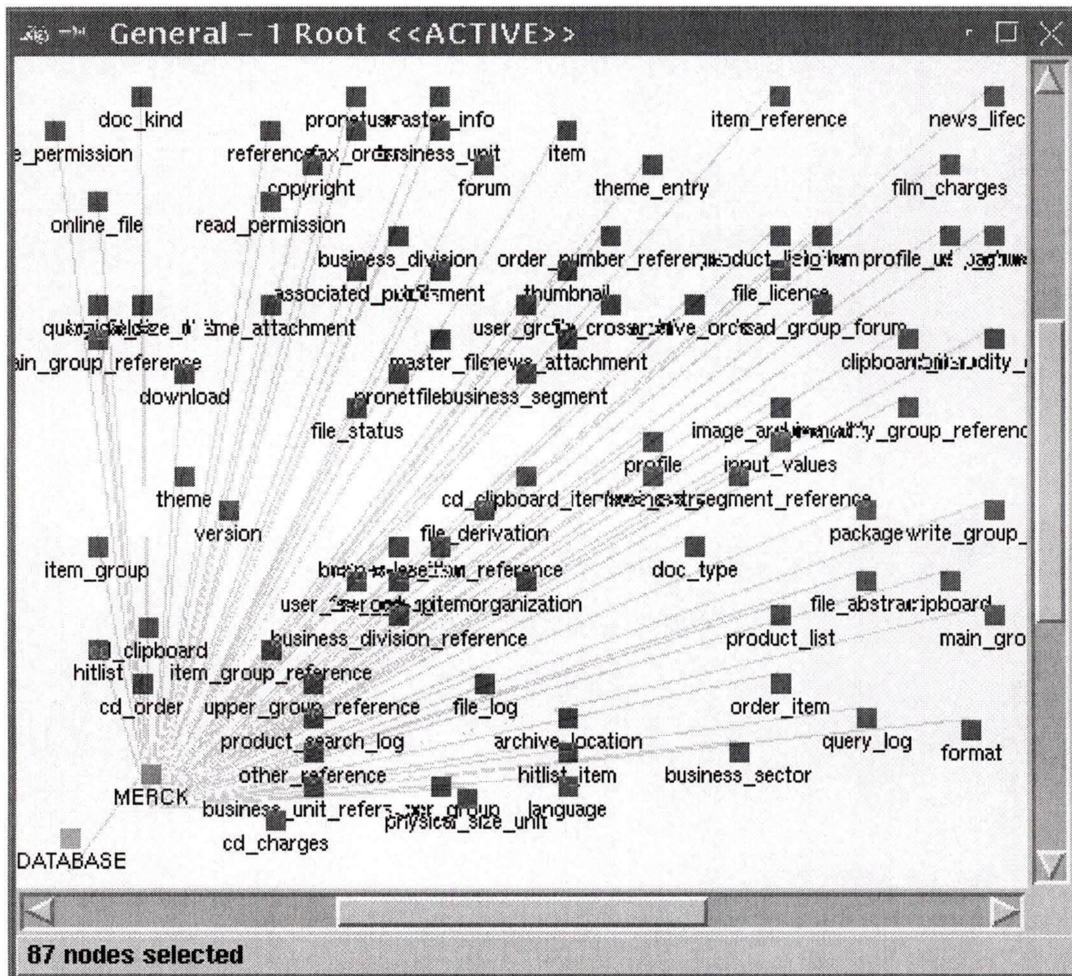


Figure 5.2 Tables in the MERCK database

Figure 5.2 shows all the tables in MERCK. Although this graph has been simplified significantly compared to the original graph, it is still too complicated to provide much meaningful architectural information. To solve this problem, the database system needs to be decomposed into subsystems based on the relationships between the tables. The operation involved in this process is the collapsing of related tables into groups.

The collapsing process can be done manually or through an RCL script. An RCL script, called *Collapse_Tables* was developed to automate the collapsing process. The

automated collapsing process is necessary because it is difficult to collapse tables manually if there are numerous (more than 200) tables represented in a single window. The developed RCL script is a single batch script that includes two simple procedures. The first procedure asks the user to input the names of tables, and the second procedure collapses those tables into a subsystem. The RCL script was written by using the Rigi Command Library commands. The following Figure 5.3 displays the result after the collapse operation.

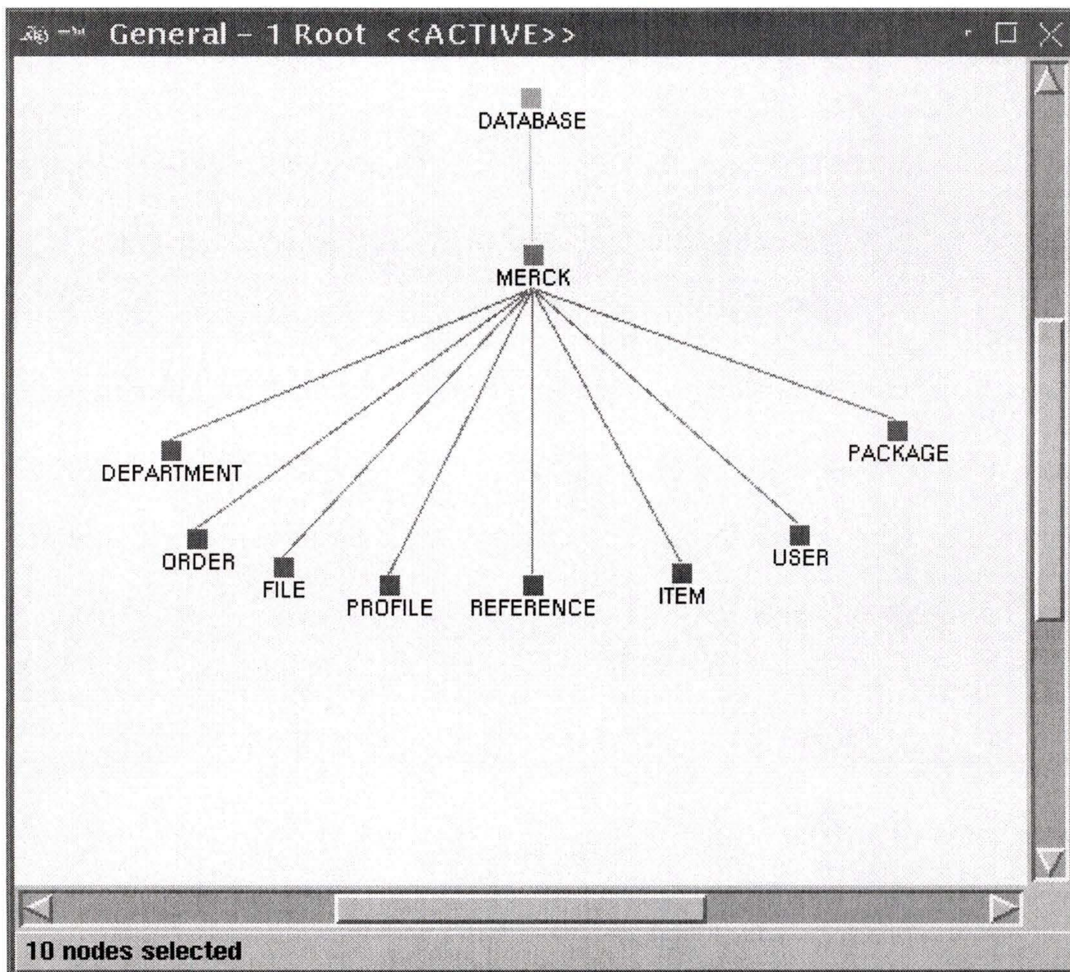


Figure 5.3 Recovered structure of the MERCK database

As can be seen in the display above, the mess is cleared up and the whole database is divided into eight subsystems. After the structure of MERCK is recovered, if users still want to know the detailed design information of MERCK, clicking on a subsystem node or a table node would bring them to a detailed view (logical schema) of the subsystem or table. Figure 5.4 shows the details of subsystem ITEM, which consists of five tables and each of them contains columns and constraints presented.

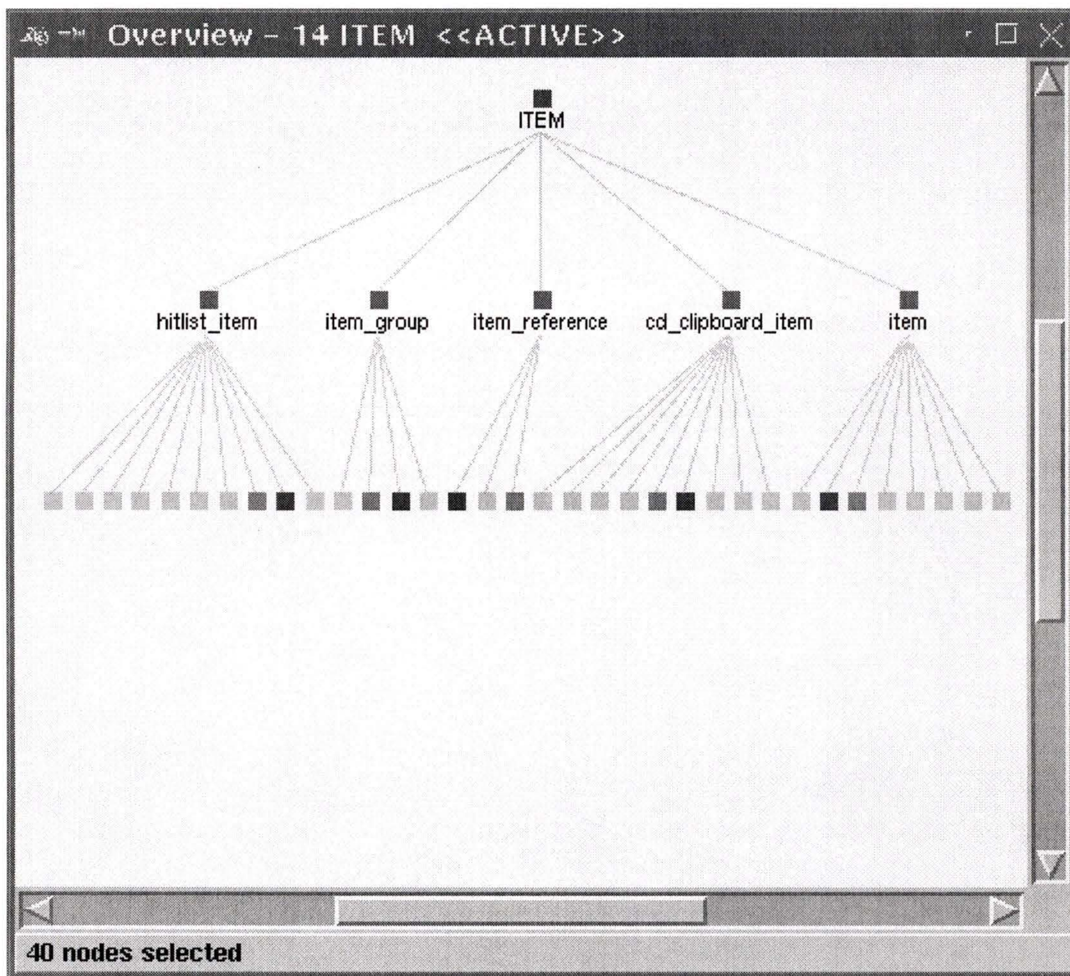


Figure 5.4 Detailed design information of the ITEM subsystem

5.3 Summary

As expected, this case study showed that the Rigi DBRE tool is a useful tool for understanding relational databases. Compared with hours and hours of time-consuming source code study, database administrators can obtain the graphical views of the analyzed database shown in this chapter in twenty minutes. Through this case study, it has been shown that the understanding process is simplified using the Rigi DBRE tool. According to the database design methodology described in Chapter 2 and the DBRE process described in Chapter 3, the extracted structure information and logical design information are shown to be critical components in the understanding of a relational database.

Chapter 6

Conclusions

The objectives of this thesis were to develop a process for a better understanding of relational databases and to find a practical approach for relational database reverse engineering.

Since DBRE is a complex process, it cannot be successful without the support of adequate tools. In this thesis, a semi-automatic relational Database Reverse Engineering tool was presented. This DBRE tool provides users with invaluable help in carrying out DBRE more effectively. As a result, this thesis is representative of the kind of insight you can obtain with DBRE tools. This thesis explores how graph-oriented visualization can be used to support DBRE. Another benefit of this approach is to leverage the Rigi system to build a DBRE tool rather than building a DBRE tool from scratch.

The contribution of this thesis is to overcome the limitations found in existing DBRE tools by combining the main DBRE facilities into one: *structure visualization* and *schema visualization*.

6.1 Summary

The goal of this thesis is to provide a DBRE environment for relational database engineers to support understanding, managing, and evolving relational databases during their daily work.

This DBRE environment is implemented based on the current version of the Rigi environment that is a tool used to understand large information spaces. To achieve the goal of visualizing relational databases, the features of the Rigi system are explored for extracting, organizing, and abstracting components while representing them visually. The work is done mainly with two components of the Rigi system: the Rigi parsing system and the graphical editor, *rigiedit*.

The approach followed covers several different aspects of visualizing relational databases. The visualization process of Rigi DBRE is concerned with both *structure analysis* and *schema analysis.* This tool deals with not only the overall structure view, (through *structure analysis*), but also the detailed information about the analyzed database (through *schema analysis* as well as *data size analysis*). Both can definitely facilitate understanding as well as managing of relational databases. Furthermore, reengineering can also be achieved on a generated logical schema in the Rigi DBRE environment.

Specifically, this tool supports understanding by visualizing the overall hierarchical structures of relational databases. Database administrators control the performance of the entire database system. Their primary responsibilities are centered around developing and maintaining the database system. They really need a picture, which exhibits the database components and the relationships among them. The Rigi DBRE tool achieves this goal by

providing users with a global view, the hierarchical structure of the analyzed database system to layout the components of the relational database. More precisely, this generated hierarchical structure includes database components, such as schemas, tables, and columns as well as the relationships among them.

Many users can share the information of a database, but each user requires a different perspective of the accessed database. In fact, they only have an interest in the particular parts of a database. These are the views. The relationships between the base tables and the views are important to the database administrators for managing the database. In other words, they need to trace the views to know from where the views are selected. Using the Rigi DBRE tool, the views are traced and presented with their base tables.

For the schema analysis, a logical schema is represented based on the generated hierarchical structure with *primary keys*, *foreign keys*, and the *NOT NULL* constraint. These constraints should be enforced by the DBMS, and be known by the database administrators. The presented logical schema can also be redesigned through the functions provided by the Rigi system.

In certain cases, the data size of each table is also an important factor to aid the maintenance activities. The Rigi DBRE tool can display the relative data size of each table based on the number of rows in that table.

To sum up, the Rigi DBRE tool provides an easy way to manage and control the structure of a relational database. The graphical representations of tables and the relationships among them (rather than DDL code) simplify the task of maintaining the database. The Rigi DBRE tool overcomes the limitations of the existing DBRE tools. The

proposed structural logical schema analysis develops a new way to get the structure and logical schema of a subject database system simultaneously.

6.2 Future work

While working on several important issues of DBRE, some interesting research areas were uncovered. The current implementation of the Rigi DBRE tool is an attempt to show that Rigi can be extended to an environment to perform DBRE. Certainly, it has a number of shortcomings. But, there are some possible directions for future work

First of all, it is not a simple task to implement a mature parser for any language, even for SQL, a rather simple language. The current version of this parser is not perfect. But, employing it more will help to improve it.

Some constraints were not discussed here: for example, *CHECK*. *CHECK* constraint is particularly useful in that it can take any valid expression as an argument. The *CHECK* clause refuses to accept any entries that fall outside its range. For future work, more constraints should be considered and visualized with the Rigi DBRE tool to support the complete visualization of logical schemas of relational databases.

Another important issue is that more visualization functions that represent the results other than the parsed SQL information need to be added to the Rigi DBRE tool. A database is valuable only if you are reasonably sure that the data it contains is correct. Integrity refers to the validity and consistency of stored data. Integrity is usually expressed in terms of constraints which are consistency rules that the database is not permitted to violate [14]. As compared with the most recent database management systems that furnish a means to ensure that both the data and the change to the data follow certain rules, unfortunately many legacy commercial database systems do not fully

support these constraints. A future version of the Rigi DBRE tool could consider such constraints and ensure the data integrity (e.g., checking inserted data to find data or data rows that are threats to data integrity). This would be helpful to the maintenance and evolution of legacy database systems.

Another valuable area of this research is the analysis of embedded SQL. The current work of the Rigi DBRE tool is implemented by parsing unitary SQL statements. However, for the vast majority of applications, SQL is *embedded* in a high-level language. In such a case, SQL statements are dropped right into the middle of a procedural program, wherever they are needed. The Rigi DBRE tool might employ DBRE on embedded SQL in many ways. For example, for the current version of the Rigi system, several parsers of different procedural programing languages, such as C and COBOL exist. Fully utilizing these parsers to analyze the embedded SQL with different host-languages could be implemented for the Rigi DBRE tool. First, the preprocessors of different host-languages could be used to turn the SQL code into calls of the host-language routines, for example, the C routines. Then, the host-language parsers can be used to parse the pure source code of the host-language. Second, the host-language parsers could be modified to recognize both the SQL statements and the host-language code in order to parse the embedded SQL application source code. After parsing the embedded SQL RSF files are generated and loaded into the *rigiedit*, and the embedded SQL application code could be represented. Furthermore, from the generated graphical views, there are some special functions, calls, and data nodes and arcs that are only related to the SQL statements embedded in the host-language source code. As a result, Rigi Command Library (RCL) scripts can be written to separate the nodes and arcs of the

graphical views into two subsystems. One includes only those nodes and arcs generated from the SQL statements, namely the SQL subsystem. Another includes only those nodes and arcs generated from the host-language source code, namely, the C subsystem. After such separation, if modification happens on the database that is accessed by the embedded SQL application, only the SQL subsystem would be affected. In other words, separating the embedded SQL code from the embedded SQL applications could be implemented with the Rigi DBRE tool. This would be useful for database reverse engineering and reengineering, such as, migrating the current legacy database systems to modern database systems. Because the SQL code and host-language code could be split, database reverse engineering or reengineering would only be concerned with the SQL code part.

Finally, in the current reengineering solution using the Rigi DBRE tool, reengineers have to change the Rigi graph and the DDL source code. It would be even better, if the reengineers could just change the graph and the source code is changed accordingly. Basically, this boils down to adding an *unparser* to the environment. The *unparser* has the job of converting an abstract syntax tree back into its original source form. This seemingly simple task is actually non-trivial, due to the complications of operator precedence and parentheses. In other words, the *unparser* can regenerate and update the DDL source code from the graph.

Except in simple or accidentally favorable situations, reverse engineering a relational database from DDL source code is a complex task that still needs in-depth research [15]. As expected for the first version of this DBRE tool and limited experience using the tool,

not all questions and problems of relational database reverse engineering are investigated. Hence, there is a real need for more case studies.

Bibliography

[1] John R. Levine; Tony Mason and Doug Brown. *Lex & Yacc,* O'REILLY, 1995.

[2] Müller, H.A.; K. Wong; and S.R. Tilley. "Understanding Software Systems Using Reverse Engineering Technology," In V.S. Alagar and R. Missaoui (eds). *Object-Oriented Technology for Database and Software Systems,* World Scientific, pages 240-252, 1995.

[3] Xia Lin. "Visualization for the Document Space," In *Proceeding of the IEEE Visualization Conference*, pages 274-281, Boston, 1992.

[4] Jens H. Jahnke and Melanie Heitbreder. "Design Recovery of Legacy Database Applications based on Possibilistic Reasoning," In *Proceeding of International Conference on Fuzzy Systems* (FUZZ'98), Anchorage, US, 1998.

[5] Jens H. Jahnke and Joerg Wadsack. "The Varlet Analyst: Employing Imperfect Knowledge in Database Reverse Engineering Tools," *Proc. of 3rd International Workshopon Intelligent Software Engineering* (WISE-3), Limerick, Ireland, 2000.

[6] J-L. Hainaut; J. Henrard; D. Roland; V. Englebert and J-M. Hick. "Structure Elicitation in Database Reverse Engineering," In *Proceeding of the IEEE WCRE*, pages 131-140, 1996.

[7] J-L. Hainaut; V. Englebert; J. Henrard; J-M, Hick and D. Roland. "Requirements for Information System Reverse Engineering Support," In *Proceeding of the IEEE WCRE*, pages136-145, 1995.

[8] Kenny Wong. *Rigi User's Manual, Version 5.4.3.* 1996.

[9] Jacqueline M. Antis; Stephen G. Erik and John D. Pyrce. "Visualizing the Structure of Large Relational Databases," In *IEEE Software*, Vol.13, pages 72-80, January 1996.

[10] D. Harel. "On Visual Formalisms," In *Communications of the ACM*, Vol.31, pages 514-530, 1988.

[11] Susan Elliott Sim; Charles L. A. Clarke; Richard C Holt and Anthony M. Cox. "Browsing and Searching Software Architectures, In *Proceedings of International Conference on Software Maintenance* (ICSM'99)," Oxford, England, pages 381-390, September 1999.

[12] Hausi A. Müller; Mehmet A. Orgun; Scott R. Tilley and James S. Uhl. "A Reverse-engineering Approach to Subsystem Structure Identification," In *Software Maintenance*: *Research and Practice*, Vol.5, pages 181-204, 1993.

[13] Fikas, S., F. "Automating the transformational development of software," *IEEE TSE*, Vol. SE-11, pp1268-1277, 1985.

[14] Thomas M. Connolly and Carolyn E. Begg. *Database Systems — a Practical Approach to Design, Implementation, and Management,* ADDISON-WESLEY, 1998.

[15] J-L. Hainaut; M. Chandelon; C. Tonneau and M. Joris. "Contribution to a Theory of Database Reverse Engineering," In *Proceeding of the IEEE WCRE*, Baltimore, pages 161-170, May 1993.

[16] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*, ADDISON-WESLEY, 1994.

[17] R. Nigel Horspool. *The Berkeley UNIX Environment - Second Edition*, Prentice-Hall Canada Inc., Scarborough, Ontario, 1992.

[18] J.-M. Hick; V. Englebert; J. Henrard; D. Roland and J.-L. Hainaut. "The DB-MAIN Database Engineering CASE Tool Functions Overview," *DB-MAIN Technical manual, public*. Institut d'informatique, FUNDP, December 1999.

[19] Scott R. Tilley; Kenny Wong; Margaret-Anne D. Storey and Hausi A. Müller. "Programmable Reverse Engineering," In *International Journal of Software Engineering and Knowledge Engineering*, Vol.4, pages 501-520, December 1994.

[20] Jean-Luc Hainaut. Database Reverse Engineering, University of Namur - Institut d'Informatique, FUNDP, December 1999.

[21] Paul Litwin. "Fundamentals of Relational Database Design," http://www.microsoft.com/TechNet/Access/technote

[22] Pascal Fabian. *SQL and Relational Basics*, Redwood City: M&T Books, 1990.

[23] Joseph R. Kiniry and Elaine Cheong. "JPP: A Java Pre-Processor," *Caltech Technical Report* CS-TR-98-15, September 1998.

[24] M. E. Lesk and E. Schmidt. "Lex - A Lexical Analyzer Generator," http://www.cs.utexas.edu/users/novak/lexpaper.htm

[25] S.C. Johnson. "YACC—Yet Another Compiler-Compiler," *Computer Science Technical Report* 1975, Bell Laboratories, Murray Hill, NJ.

[26] Bell Telephone Laboratories. *Unix Programmer's Manual – Seventh Edition Volume*, January 1979.

[27] J. H. Jahnke. "Managing Uncertainty and Inconsistency in Database Reengineering Processes," Ph.D. Dissertation, University of Paderborn, Germany, 1999.

[28] George Baklarz and Bill Wang. *DB2 Universal Database v7.1 for Unix, Linux, Windows and OS/2,* Prentice Hall PTR, Upper Saddle River, NJ.

Appendix

```
*******************************sql.l file*********************************
%{
#include "sql.h"
#include <string.h>

int lineno = 1;
void yyerror(char *s);

        /* macro to save the text of a SQL token */
#define SV save_str(yytext)

        /* macro to save the text and return a token */
#define TOK(name) { SV;return name; }
%}
%s SQL
%%

EXEC[ \t]+SQL   { BEGIN SQL; start_save(); }


        /* literal keyword tokens */

<SQL>ALL                TOK(ALL)
<SQL>AND                TOK(AND)
<SQL>AVG                TOK(AMMSC)
<SQL>MIN                TOK(AMMSC)
<SQL>MAX                TOK(AMMSC)
<SQL>SUM                TOK(AMMSC)
<SQL>COUNT              TOK(AMMSC)
<SQL>ANY                TOK(ANY)
<SQL>AS                         TOK(AS)
<SQL>ASC                TOK(ASC)
<SQL>AUTHORIZATION              TOK(AUTHORIZATION)
<SQL>BETWEEN                    TOK(BETWEEN)
<SQL>BY                         TOK(BY)
<SQL>CHAR(ACTER)?  TOK(CHARACTER)
<SQL>CHECK              TOK(CHECK)
<SQL>CLOSE              TOK(CLOSE)
<SQL>COMMIT             TOK(COMMIT)
<SQL>CONTINUE                   TOK(CONTINUE)
<SQL>CREATE             TOK(CREATE)
<SQL>CURRENT                    TOK(CURRENT)
<SQL>CURSOR             TOK(CURSOR)
<SQL>DECIMAL                    TOK(DECIMAL)
<SQL>DECLARE                    TOK(DECLARE)
<SQL>DEFAULT                    TOK(DEFAULT)
<SQL>DELETE             TOK(DELETE)
<SQL>DESC               TOK(DESC)
<SQL>DISTINCT                   TOK(DISTINCT)
<SQL>DOUBLE             TOK(DOUBLE)
<SQL>ESCAPE             TOK(ESCAPE)
```

```
<SQL>EXISTS          TOK(EXISTS)
<SQL>FETCH           TOK(FETCH)
<SQL>FLOAT           TOK(FLOAT)
<SQL>FOR             TOK(FOR)
<SQL>FOREIGN                 TOK(FOREIGN)
<SQL>FOUND           TOK(FOUND)
<SQL>FROM            TOK(FROM)
<SQL>GO[ \t]*TO              TOK(GOTO)
<SQL>GRANT           TOK(GRANT)
<SQL>GROUP           TOK(GROUP)
<SQL>HAVING          TOK(HAVING)
<SQL>IN                      TOK(IN)
<SQL>INDICATOR               TOK(INDICATOR)
<SQL>INSERT          TOK(INSERT)
<SQL>INT(EGER)?              TOK(INTEGER)
<SQL>INTO            TOK(INTO)
<SQL>IS                      TOK(IS)
<SQL>KEY             TOK(KEY)
<SQL>LANGUAGE                TOK(LANGUAGE)
<SQL>LIKE            TOK(LIKE)
<SQL>NOT             TOK(NOT)
<SQL>NULL            TOK(NULLX)
<SQL>NUMERIC                 TOK(NUMERIC)
<SQL>OF                      TOK(OF)
<SQL>ON                      TOK(ON)
<SQL>OPEN            TOK(OPEN)
<SQL>OPTION          TOK(OPTION)
<SQL>OR                      TOK(OR)
<SQL>ORDER           TOK(ORDER)
<SQL>PRECISION               TOK(PRECISION)
<SQL>PRIMARY                 TOK(PRIMARY)
<SQL>PRIVILEGES              TOK(PRIVILEGES)
<SQL>PROCEDURE               TOK(PROCEDURE)
<SQL>PUBLIC          TOK(PUBLIC)
<SQL>REAL            TOK(REAL)
<SQL>REFERENCES              TOK(REFERENCES)
<SQL>ROLLBACK                TOK(ROLLBACK)
<SQL>SCHEMA          TOK(SCHEMA)
<SQL>SELECT          TOK(SELECT)
<SQL>SET             TOK(SET)
<SQL>SMALLINT                TOK(SMALLINT)
<SQL>SOME            TOK(SOME)
<SQL>SQLCODE                 TOK(SQLCODE)
<SQL>TABLE           TOK(TABLE)
<SQL>TO                      TOK(TO)
<SQL>UNION           TOK(UNION)
<SQL>UNIQUE          TOK(UNIQUE)
<SQL>UPDATE          TOK(UPDATE)
<SQL>USER            TOK(USER)
<SQL>VALUES          TOK(VALUES)
<SQL>VIEW            TOK(VIEW)
<SQL>WHENEVER                TOK(WHENEVER)
<SQL>WHERE           TOK(WHERE)
<SQL>WITH            TOK(WITH)
<SQL>WORK            TOK(WORK)
```

```
	/* punctuation */

<SQL>"="	|
<SQL>"<>"	|
<SQL>"<"	|
<SQL>">"	|
<SQL>"<="	|
<SQL>">="		TOK(COMPARISON)

<SQL>[-+*/(),.;]	TOK(yytext[0])


	/* names */
<SQL>[A-Za-z][A-Za-z0-9_]*	TOK(NAME)

	/* parameters */
<SQL>":"[A-Za-z][A-Za-z0-9_]*	{
			save_param(yytext+1);
			return PARAMETER;
		}

	/* numbers */

<SQL>[0-9]+	|
<SQL>[0-9]+"."[0-9]* |
<SQL>"."[0-9]*		TOK(INTNUM)

<SQL>[0-9]+[eE][+-]?[0-9]+	|
<SQL>[0-9]+"."[0-9]*[eE][+-]?[0-9]+ |
<SQL>"."[0-9]*[eE][+-]?[0-9]+	TOK(APPROXNUM)

	/* strings */

<SQL>'[^'\n]*'	{
		int c = input();

		unput(c);	/* just peeking */
		if(c != '\'') {
			SV;return STRING;
		} else
			yymore();
	}

<SQL>'[^'\n]*$	{	yyerror("Unterminated string"); }

<SQL>\n			{ save_str(" ");lineno++; }
\n		{ lineno++; ECHO; }

<SQL>[ \t\r]+	save_str(" ");	/* white space */

<SQL>"--".*	;	/* comment */

.		ECHO;	/* random non-SQL text */
%%

void
```

```
yyerror(char *s)
{
	printf("%d: %s at %s\n", lineno, s, yytext);
}

main(int ac, char **av)
{
	if(ac > 1 && (yyin = fopen(av[1], "r")) == NULL) {
		perror(av[1]);
		exit(1);
	}

	if(!yyparse())
		fprintf(stderr, "Embedded SQL parse worked\n");
	else
		fprintf(stderr, "Embedded SQL parse failed\n");
} /* main */

/* leave SQL lexing mode */
un_sql()
{
	BEGIN INITIAL;
} /* un_sql */
```

**********************************sql.y file**

```
/* symbolic tokens */

%union {
	int intval;
	double floatval;
	char *strval;
	int subtok;
}

%token NAME
%token STRING
%token INTNUM APPROXNUM

	/* operators */

%left OR
%left AND
%left NOT
%left <subtok> COMPARISON /* = <> < > <= >= */
%left '+' '-'
%left '*' '/'
%nonassoc UMINUS

	/* literal keyword tokens */

%token ALL AMMSC ANY AS ASC AUTHORIZATION BETWEEN BY
%token CHARACTER CHECK CLOSE COMMIT CONTINUE CREATE CURRENT
%token CURSOR DECIMAL DECLARE DEFAULT DELETE DESC DISTINCT DOUBLE
%token ESCAPE EXISTS FETCH FLOAT FOR FOREIGN FOUND FROM GOTO
%token GRANT GROUP HAVING IN INDICATOR INSERT INTEGER INTO
%token IS KEY LANGUAGE LIKE NULLX NUMERIC OF ON OPEN OPTION
%token ORDER PARAMETER PRECISION PRIMARY PRIVILEGES PROCEDURE
%token PUBLIC REAL REFERENCES ROLLBACK SCHEMA SELECT SET
%token SMALLINT SOME SQLCODE SQLERROR TABLE TO UNION
%token UNIQUE UPDATE USER VALUES VIEW WHENEVER WHERE WITH WORK

%%

sql_list:
		sql ';'	{ end_sql(); }
	|	sql_list sql ';' { end_sql(); }
	;


	/* schema definition language */
sql:		schema
	;

schema:
		CREATE SCHEMA AUTHORIZATION user opt_schema_element_list
	;
```

```
opt_schema_element_list:
                /* empty */
        |       schema_element_list
        ;

schema_element_list:
                schema_element
        |       schema_element_list schema_element
        ;

schema_element:
                base_table_def
        |       view_def
        |       privilege_def
        ;

base_table_def:
                CREATE TABLE table '(' base_table_element_commalist ')'
        ;

base_table_element_commalist:
                base_table_element
        |       base_table_element_commalist ',' base_table_element
        ;

base_table_element:
                column_def
        |       table_constraint_def
        ;

column_def:
                column data_type column_def_opt_list
        ;

column_def_opt_list:
                /* empty */
        |       column_def_opt_list column_def_opt
        ;

column_def_opt:
                NOT NULLX
        |       NOT NULLX UNIQUE
        |       NOT NULLX PRIMARY KEY
        |       DEFAULT literal
        |       DEFAULT NULLX
        |       DEFAULT USER
        |       CHECK '(' search_condition ')'
        |       REFERENCES table
        |       REFERENCES table '(' column_commalist ')'
        ;

table_constraint_def:
                UNIQUE '(' column_commalist ')'
        |       PRIMARY KEY '(' column_commalist ')'
        |       FOREIGN KEY '(' column_commalist ')'
                        REFERENCES table
```

```
	|	FOREIGN KEY '(' column_commalist ')'
			REFERENCES table '(' column_commalist ')'
	|	CHECK '(' search_condition ')'
	;

column_commalist:
		column
	|	column_commalist ',' column
	;

view_def:
		CREATE VIEW table opt_column_commalist
		AS query_spec opt_with_check_option
	;

opt_with_check_option:
		/* empty */
	|	WITH CHECK OPTION
	;

opt_column_commalist:
		/* empty */
	|	'(' column_commalist ')'
	;

privilege_def:
		GRANT privileges ON table TO grantee_commalist
		opt_with_grant_option
	;

opt_with_grant_option:
		/* empty */
	|	WITH GRANT OPTION
	;

privileges:
		ALL PRIVILEGES
	|	ALL
	|	operation_commalist
	;

operation_commalist:
		operation
	|	operation_commalist ',' operation
	;

operation:
		SELECT
	|	INSERT
	|	DELETE
	|	UPDATE opt_column_commalist
	|	REFERENCES opt_column_commalist
	;


grantee_commalist:
```

```
		grantee
	|	grantee_commalist ',' grantee
	;

grantee:
		PUBLIC
	|	user
	;

	/* cursor definition */
sql:
		cursor_def
	;


cursor_def:
		DECLARE cursor CURSOR FOR query_exp opt_order_by_clause
	;

opt_order_by_clause:
		/* empty */
	|	ORDER BY ordering_spec_commalist
	;

ordering_spec_commalist:
		ordering_spec
	|	ordering_spec_commalist ',' ordering_spec
	;

ordering_spec:
		INTNUM opt_asc_desc
	|	column_ref opt_asc_desc
	;

opt_asc_desc:
		/* empty */
	|	ASC
	|	DESC
	;

	/* manipulative statements */

sql:		manipulative_statement
	;

manipulative_statement:
		close_statement
	|	commit_statement
	|	delete_statement_positioned
	|	delete_statement_searched
	|	fetch_statement
	|	insert_statement
	|	open_statement
	|	rollback_statement
	|	select_statement
	|	update_statement_positioned
```

```
	|	update_statement_searched
	;

close_statement:
		CLOSE cursor
	;

commit_statement:
		COMMIT WORK
	;

delete_statement_positioned:
		DELETE FROM table WHERE CURRENT OF cursor
	;

delete_statement_searched:
		DELETE FROM table opt_where_clause
	;

fetch_statement:
		FETCH cursor INTO target_commalist
	;

insert_statement:
		INSERT INTO table opt_column_commalist values_or_query_spec
	;

values_or_query_spec:
		VALUES '(' insert_atom_commalist ')'
	|	query_spec
	;

insert_atom_commalist:
		insert_atom
	|	insert_atom_commalist ',' insert_atom
	;

insert_atom:
		atom
	|	NULLX
	;

open_statement:
		OPEN cursor
	;

rollback_statement:
		ROLLBACK WORK
	;

select_statement:
		SELECT opt_all_distinct selection
		INTO target_commalist
		table_exp
	;
```

```
opt_all_distinct:
                    /* empty */
          |         ALL
          |         DISTINCT
          ;

update_statement_positioned:
                    UPDATE table SET assignment_commalist
                    WHERE CURRENT OF cursor
          ;

assignment_commalist:
          |         assignment
          |         assignment_commalist ',' assignment
          ;

assignment:
                    column '=' scalar_exp
          |         column '=' NULLX
          ;

update_statement_searched:
                    UPDATE table SET assignment_commalist opt_where_clause
          ;

target_commalist:
                    target
          |         target_commalist ',' target
          ;

target:
                    parameter_ref
          ;

opt_where_clause:
                    /* empty */
          |         where_clause
          ;

          /* query expressions */

query_exp:
                    query_term
          |         query_exp UNION query_term
          |         query_exp UNION ALL query_term
          ;

query_term:
                    query_spec
          |         '(' query_exp ')'
          ;

query_spec:
                    SELECT opt_all_distinct selection table_exp
          ;
```

```
selection:
		scalar_exp_commalist
	|	'*'
	;

table_exp:
		from_clause
		opt_where_clause
		opt_group_by_clause
		opt_having_clause
	;

from_clause:
		FROM table_ref_commalist
	;

table_ref_commalist:
		table_ref
	|	table_ref_commalist ',' table_ref
	;

table_ref:
		table
	|	table range_variable
	;

where_clause:
		WHERE search_condition
	;

opt_group_by_clause:
		/* empty */
	|	GROUP BY column_ref_commalist
	;

column_ref_commalist:
		column_ref
	|	column_ref_commalist ',' column_ref
	;

opt_having_clause:
		/* empty */
	|	HAVING search_condition
	;

	/* search conditions */

search_condition:
	|	search_condition OR search_condition
	|	search_condition AND search_condition
	|	NOT search_condition
	|	'(' search_condition ')'
	|	predicate
	;

predicate:
```

```
                comparison_predicate
        |       between_predicate
        |       like_predicate
        |       test_for_null
        |       in_predicate
        |       all_or_any_predicate
        |       existence_test
        ;

comparison_predicate:
                scalar_exp COMPARISON scalar_exp
        |       scalar_exp COMPARISON subquery
        ;

between_predicate:
                scalar_exp NOT BETWEEN scalar_exp AND scalar_exp
        |       scalar_exp BETWEEN scalar_exp AND scalar_exp
        ;

like_predicate:
                scalar_exp NOT LIKE atom opt_escape
        |       scalar_exp LIKE atom opt_escape
        ;

opt_escape:
                /* empty */
        |       ESCAPE atom
        ;

test_for_null:
                column_ref IS NOT NULLX
        |       column_ref IS NULLX
        ;

in_predicate:
                scalar_exp NOT IN '(' subquery ')'
        |       scalar_exp IN '(' subquery ')'
        |       scalar_exp NOT IN '(' atom_commalist ')'
        |       scalar_exp IN '(' atom_commalist ')'
        ;

atom_commalist:
                atom
        |       atom_commalist ',' atom
        ;

all_or_any_predicate:
                scalar_exp COMPARISON any_all_some subquery
        ;

any_all_some:
                ANY
        |       ALL
        |       SOME
        ;
```

```
existence_test:
                EXISTS subquery
        ;

subquery:
                '(' SELECT opt_all_distinct selection table_exp ')'
        ;

        /* scalar expressions */

scalar_exp:
                scalar_exp '+' scalar_exp
        |       scalar_exp '-' scalar_exp
        |       scalar_exp '*' scalar_exp
        |       scalar_exp '/' scalar_exp
        |       '+' scalar_exp %prec UMINUS
        |       '-' scalar_exp %prec UMINUS
        |       atom
        |       column_ref
        |       function_ref
        |       '(' scalar_exp ')'
        ;

scalar_exp_commalist:
                scalar_exp
        |       scalar_exp_commalist ',' scalar_exp
        ;

atom:
                parameter_ref
        |       literal
        |       USER
        ;

parameter_ref:
                parameter
        |       parameter parameter
        |       parameter INDICATOR parameter
        ;

function_ref:
                AMMSC '(' '*' ')'
        |       AMMSC '(' DISTINCT column_ref ')'
        |       AMMSC '(' ALL scalar_exp ')'
        |       AMMSC '(' scalar_exp ')'
        ;

literal:
                STRING
        |       INTNUM
        |       APPROXNUM
        ;

        /* miscellaneous */

table:
```

```
		NAME
	|	NAME '.' NAME
	;

column_ref:
		NAME
	|	NAME '.' NAME /* needs semantics */
	|	NAME '.' NAME '.' NAME
	;

		/* data types */

data_type:
		CHARACTER
	|	CHARACTER '(' INTNUM ')'
	|	NUMERIC
	|	NUMERIC '(' INTNUM ')'
	|	NUMERIC '(' INTNUM ',' INTNUM ')'
	|	DECIMAL
	|	DECIMAL '(' INTNUM ')'
	|	DECIMAL '(' INTNUM ',' INTNUM ')'
	|	INTEGER
	|	SMALLINT
	|	FLOAT
	|	FLOAT '(' INTNUM ')'
	|	REAL
	|	DOUBLE PRECISION
	;

	/* the various things you can name */

column:		NAME
	;

cursor:		NAME
	;

parameter:
		PARAMETER	/* :name handled in parser */
	;

range_variable:	NAME
	;

user:		NAME
	;

	/* embedded condition things */
sql:		WHENEVER NOT FOUND when_action
	|	WHENEVER SQLERROR when_action
	;

when_action:	GOTO NAME
	|	CONTINUE
	;
%%
```

**************************************sql.h file**

```
#ifndef SQLDDL
#define SQLDDL
#include "ProgresDB.h"

#ifdef  true
#undef  true
#endif
#define true 1

#ifdef  false
#undef  false
#endif
#define false 0

#define boolean int

struct column {
  char        *name ;
  char        *type ;
  int         length1 ;
  int         length2 ;
  boolean      nn;
  boolean      pk;
  DBid        id;
  struct column *next;
};

struct fkey {
  DBid        ind;
  char       *table;
  struct column  *rcolumn;
  struct column  *column;
  struct fkey *next;
};

struct ckey {
  DBid        ind;
  char       *table;
  struct column  *column;
  struct ckey *next;
};

struct table {
  char        *name;
  struct column    *primkey;
  DBid        id;
  DBid        pkid;
  struct column *col;
  struct fkey   *fkeys;
  struct table  *next;
};

struct table  *tablist;
```

```
struct table  *tabpointer;

struct fkey   *fkeylist;
struct fkey   *fkeypointer;

struct ckey   *ckeylist;
struct ckey   *ckeypointer;

struct column *collist;
struct column *colpointer;

struct column *col1;
struct column *col2;
struct column *colp1;
struct column *colp2;
struct column *primkeylist;

boolean reference   = false ;
boolean firstcol    = true  ;
boolean NotNull     = false ;
boolean primkey     = false ;

char *scheme ;

int    type_length1 = 0 ;
int    type_length2 = 0 ;

extern void yyerror( char * buf);
extern int yylex ( );
void AddColToColList(char *buf,boolean changecol);
void AddColsToFKey(char *buf, struct column *col1, struct column *col2);
void InsertColInFKey(struct column *col1);
void add_column(char *buf1,char *buf2, int len1, int len2,boolean nn);
void add_table(char *buf, struct fkey *fkeylist, struct column *primkeylist);
void AddColToCKeyList(char *buf, struct column *col1);
void ExptoProgres( int argc, char ** argv, char ** envp);
void clear_all(void);

#endif
```

VITA

Surname: Du	Given Name: Ming
Place of Birth: Beijing, China	
Educational Institutions Attended:	
University of Victoria, Canada	1997 to 2002
Beijing Information Technology Institute, China	1979 to 1983
Degrees Awarded:	
B.S. in Computer Science	1983

Partial Copyright License

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

Relational Database Reverse Engineering using the Rigi System

Author:

Ming Du
February, 2002